Polymorphic set-theoretic types for functional languages

Tommaso Petrucciani

Joint Ph.D. Thesis

Università di Genova

Dipartimento di Informatica,
Bioingegneria, Robotica
e Ingegneria dei Sistemi

Ph.D. Thesis
in Computer Science
and Systems Engineering
(Computer Science Curriculum)

Université Sorbonne Paris Cité
Université Paris Diderot

École Doctorale de Sciences
Mathématiques de Paris Centre

Ph.D. Thesis
in Computer Science

Polymorphic set-theoretic types for functional languages

Tommaso Petrucciani

March 2019

Université Sorbonne
Paris Cité

université
PARIS
DIDEROT
PARIS 7

Thèse de doctorat
de l'Università di Genova
et de l'Université Sorbonne Paris Cité

Préparée à l'Université Paris Diderot
ED 386 – Sciences Mathématiques de Paris Centre
Institut de Recherche en Informatique Fondamentale

Polymorphic set-theoretic types for functional languages

par

Tommaso Petrucciani

Thèse de doctorat en Informatique

Dirigée par Giuseppe Castagna

Présentée et soutenue publiquement à Gênes (Italie)
le 14 mars 2019 devant le jury composé de

Directeur de thèse	Giuseppe Castagna Directeur de recherche, CNRS
Rapporteur	Mariangiola Dezani Professeur émérite, Università di Torino
Président du jury et rapporteur	Alan Mycroft Professeur, University of Cambridge
Rapporteur	Sam Tobin-Hochstadt Maître de conférence, Indiana University
Co-directeur de thèse	Elena Zucca Professeur, Università di Genova

Joint Ph.D. Thesis

Ph.D. Thesis in Computer Science and Systems Engineering (S.S.D. INF/01)
Dipartimento di Informatica, Bioingegneria,
Robotica e Ingegneria dei Sistemi
Università di Genova

Ph.D. Thesis in Computer Science
École Doctorale 386 – Sciences Mathématiques de Paris Centre
Université Sorbonne Paris Cité – Université Paris Diderot

Candidate
Tommaso Petrucciani
Tommaso.Petrucciani@dibris.unige.it

Title
Polymorphic set-theoretic types for functional languages

Advisors
Giuseppe Castagna
IRIF, CNRS, Université Paris Diderot
Giuseppe.Castagna@irif.fr

Elena Zucca
DIBRIS, Università di Genova
Elena.Zucca@unige.it

External Reviewers
Mariangiola Dezani
Dipartimento di Informatica, Università di Torino
dezani@di.unito.it

Alan Mycroft
Computer Laboratory, University of Cambridge
Alan.Mycroft@cl.cam.ac.uk

Sam Tobin-Hochstadt
School of Informatics, Computing, and Engineering, Indiana University
samth@cs.indiana.edu

Location
DIBRIS, Univ. di Genova
Via Opera Pia, 13
I-16145 Genova, Italy

Submitted On
March 2019

Abstract

TITLE Polymorphic set-theoretic types for functional languages

KEYWORDS type systems, subtyping, type inference, gradual typing, non-strict semantics

We study *set-theoretic types*: types that include union, intersection, and negation connectives. Set-theoretic types, coupled with a suitable subtyping relation, are useful to type several programming language constructs – including conditional branching, pattern matching, and function overloading – very precisely. We define subtyping following the *semantic subtyping* approach, which interprets types as sets and defines subtyping as set inclusion. Our set-theoretic types are *polymorphic*, that is, they contain type variables to allow parametric polymorphism.

We extend previous work on set-theoretic types and semantic subtyping by showing how to adapt them to new settings and apply them to type various features of functional languages. More precisely, we integrate semantic subtyping with three important language features.

In Part I we study implicitly typed languages with let-polymorphism and type inference (previous work on semantic subtyping focused on explicitly typed languages). We describe an implicitly typed λ-calculus and a declarative type system for which we prove soundness. We study type inference and prove results of soundness and completeness. Then, we show how to make type inference more precise when programs are partially annotated with types.

In Part II we study gradual typing. We describe a new approach to add gradual typing to a static type system; the novelty is that we give a declarative presentation of the type system, while previous work considered algorithmic presentations. We first illustrate the approach on a Hindley-Milner type system without subtyping. We describe declarative typing, compilation to a cast language, and sound and complete type inference. Then, we add set-theoretic types, defining a subtyping relation on set-theoretic gradual types, and we describe sound type inference for the extended system.

In Part III we consider non-strict semantics. The existing semantic subtyping systems are designed for call-by-value languages and are unsound for non-strict semantics. We adapt them to obtain soundness for call-by-need. To do so, we introduce an explicit representation for divergence in the types, allowing the type system to distinguish the expressions that are already evaluated from those that are computations which might diverge.

Résumé

TITRE Types ensemblistes polymorphes pour les langages fonctionnels

MOTS-CLÉS systèmes de types, sous-typage, inférence de types, typage graduel, sémantiques non-strictes

Cette thèse porte sur l'étude des *types ensemblistes* : des types qui contiennent des connecteurs d'union, d'intersection et de négation. Les types ensemblistes permettent de typer de manière très précise plusieurs constructions des langages de programmation (comme par exemple les branches conditionnelles, le filtrage par motif et la surcharge des fonctions) lorsqu'ils sont utilisés avec une notion appropriée de sous-typage. Pour définir celle-ci, nous utilisons l'approche du *sous-typage sémantique*, dans laquelle les types sont interprétés comme des ensembles, et où le sous-typage est défini comme l'inclusion ensembliste. Dans la plupart de cette thèse, les types ensemblistes sont *polymorphes*, dans le sens où ils contiennent des variables de type pour permettre le polymorphisme paramétrique.

La thèse étend les travaux précédents sur les types ensemblistes et le sous-typage sémantique en montrant comment les adapter à de nouveaux contextes et comment les utiliser pour typer plusieurs aspects des langages fonctionnels. Elle se compose de trois parties.

La première partie porte sur une étude des langages typés de manière implicite avec polymorphisme du let et inférence de types (contrairement aux travaux précédents sur le sous-typage sémantique qui étudiaient des langages typés explicitement). Nous y décrivons un λ-calcul typé implicitement avec un système de types dont nous démontrons la correction. De même, nous y étudions l'inférence de types dont nous démontrons la correction et la complétude. Enfin, nous montrons comment rendre l'inférence plus précise quand les programmes sont partiellement annotés avec des types.

La deuxième partie décrit une nouvelle approche permettant d'étendre un système de types statique avec du typage graduel; l'originalité venant du fait que nous décrivons le système de types de façon déclarative, lorsque les systèmes existants proposent des descriptions algorithmiques. Nous illustrons cette approche en ajoutant le typage graduel à un système de types à la Hindley-Milner sans sous-typage. Nous décrivons pour cela un système de types déclaratif, un processus de compilation vers un langage avec vérifications de type dynamiques (ou "casts"), et nous présentons un système d'inférence de types correct et complet. Ensuite, nous y ajoutons les types ensemblistes, en définissant une relation de sous-typage sur les types graduel ensemblistes, puis en présentant un système d'inférence de types correct pour le système étendu.

La troisième partie porte sur l'étude des sémantiques non-strictes. Les systèmes existants qui utilisent le sous-typage sémantique ont été développés pour des langages avec appel par valeur et ne sont pas sûrs pour des sémantiques non-strictes. Nous montrons ici comment les adapter pour garantir leur sûreté en appel par nécessité. Pour faire ça, nous introduisons dans les types une représentation explicite de la divergence, afin que le système des types puisse distinguer les expressions qui ne demandent pas d'évaluation de celles qui la demandent et pourraient ainsi diverger.

Résumé substantiel

Cette thèse porte sur l'étude des *types ensemblistes* avec *sous-typage sémantique* et de leur utilisation pour typer plusieurs aspects des langages de programmation fonctionnels. En particulier, nous considérons le typage implicite et l'inférence des types, le polymorphisme du let, le typage graduel et les sémantiques non-strictes.

Les types ensemblistes permettent de typer de manière très précise plusieurs constructions des langages de programmation : par exemple, les branches conditionnelles, le filtrage par motif et la surcharge des fonctions. Cependant, pour utiliser ces types efficacement, il faut définir une notion appropriée de sous-typage. Nous suivons l'approche du *sous-typage sémantique* : nous définissons une interprétation $[\![\cdot]\!]$ des types comme des ensembles et nous utilisons celle-ci pour définir le sous-typage entre les types comme l'inclusion ensembliste de leurs interprétations. Dans la plupart de la thèse, les types ensemblistes sont *polymorphes*, dans le sens où ils contiennent des variables de type permettant le polymorphisme paramétrique.

Dans cette thèse, nous montrons comment étendre les travaux précédents sur les types ensemblistes pour les adapter à de nouveaux contextes et langages. Nous tâchons d'y montrer que le sous-typage sémantique est une approche efficace pour définir le sous-typage dans les systèmes considérés. En particulier, nous montrons comment réutiliser directement certains des résultats existants sur le sous-typage sémantique (notamment ceux qui concernent la procédure de décision) dans différents contextes.

La thèse se compose de trois parties.

Typage implicite et inférence de types

La première partie porte sur une étude des langages typés de manière implicite avec polymorphisme du let et inférence de types. Les travaux précédents sur le sous-typage sémantique étudiaient des langages où les fonctions étaient annotées explicitement avec leurs types (Frisch, Castagna et Benzaken, 2008 ; Castagna et al., 2014) et considéraient au plus l'inférence de types locale pour l'instantiation des fonctions polymorphes (Castagna et al., 2015b).

Nous étudions un λ-calcul étendu avec des constantes, des paires, une construction de "typecase" (pour modéliser la sélection de type durant l'execution et le filtrage par motif), ainsi que des déclarations let.

Nous décrivons un système de types pour ce langage : un système à la Hindley-Milner étendu avec les deux règles structurelles suivantes pour la subsomption et pour l'introduction des types intersection.

$$[\mathrm{T}_{\leq}]\ \frac{\Gamma \vdash e : t'}{\Gamma \vdash e : t}\ t' \leq t \qquad\qquad [\mathrm{T}_{\wedge}]\ \frac{\Gamma \vdash e : t_1 \qquad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2}$$

Le système est simple à décrire ; la difficulté est dans la preuve de correction par rapport à la sémantique. À cause de la présence de la règle $[\mathrm{T}_\wedge]$ et des types négation, pour que la réduction du sujet soit valable, nous devons étendre le système avec une règle pour dériver des types négation pour les fonctions, pour avoir, par exemple, $\vdash \lambda x.\, x : \neg(\mathsf{Int} \to \mathsf{Bool})$. Cette difficulté a déjà été résolue dans des travaux précédents (Frisch, Castagna et Benzaken, 2008), mais ici elle demande une solution différente car les fonctions ne sont pas annotées. Nous développons cette solution et la preuve de correction pour le système étendu, qui implique aussi la correction pour le système original.

Ensuite, nous étudions l'inférence de types en définissant un algorithme d'inférence de types fondé sur la génération et la résolution de contraintes. Nous utilisons des contraintes similaires à celles de Pottier et Rémy (2005) ; tandis que la résolution de contraintes réutilise l'algorithme de *tallying* de Castagna et al. (2015b). Nous prouvons que l'inférence est correcte par rapport au système de types, et complète par rapport à la restriction du système sans la règle $[\mathrm{T}_\wedge]$. Nous ne comparons pas l'inférence directement au système de types original, mais à un système différent – fondé sur les "règles de typage reformulées" de Dolan et Mycroft (2017) – dont nous montrons l'équivalence avec le système original. Ce système différent gère la généralisation du `let` d'une manière qui est plus adaptée à la comparaison à l'algorithme d'inférence.

Ensuite, nous ajoutons des annotations de type au langage et nous montrons comment l'inférence peut les utiliser pour calculer des types plus précis (notamment, des types intersection pour les fonctions). Finalement, nous présentons comment éteindre le langage avec des fonctionnalités ultérieures : le filtrage par motif, les variants polymorphes à la OCaml, et les enregistrements.

Typage graduel

La deuxième partie porte sur l'étude du *typage graduel*, une approche qui permet de faire coexister dans un même langage le typage statique et le typage dynamique (Siek et Taha, 2006). On fait cela en introduisant un type *inconnu*, noté ?, et en assouplissant le système de types pour que les expressions de ce type ? soient utilisables dans tout contexte. Les programmes ne sont donc controlés statiquement que en partie ; pour garantir la sûreté de l'exécution, il sont ensuite compilés vers un langage avec vérifications de type dynamiques. Un résultat de correction garantit alors que l'exécution d'un programme bien typé produit une valeur ou bien échoue dans l'évaluation d'une des ces vérifications, mais ne peut pas échouer pour d'autres raisons.

Nos apports à l'étude du typage graduel sont la description d'une nouvelle approche permettant d'ajouter le typage graduel à un système statique existant, et le développement de cette approche pour des systèmes aussi bien avec que sans sous-typage.

D'abord, nous ajoutons le typage graduel à un système à la Hindley-Milner sans sous-typage. La nouveauté de notre approche est que nous définissons le système de types graduel en ajoutant une seule règle au système statique : une règle structurelle qui utilise la relation de *précision* déjà connue dans la

littérature sur le typage graduel. En revanche, les systèmes existants pour le typage graduel utilisent une notion de cohérence (*consistency*) qui ne peut pas être utilisée dans une règle structurelle car elle n'est pas transitive. La différence entre notre système et ceux des travaux précédents est donc similaire à celle entre les descriptions *déclaratives* (c'est-à-dire, avec des règles structurelles) et *algorithmiques* (sans ces règles) des systèmes avec sous-typage.

Nous définissons ensuite le langage avec vérifications de type dynamiques et nous décrivons la compilation vers celui-ci : chaque utilisation de la règle structurelle pour la précision correspond à l'insertion d'une vérification de type dans le programme compilé. Nous décrivons l'inférence de types et nous en démontrons la correction et la complétude. Nous montrons que, pour la résolution des contraintes d'inférence, nous pouvons réutiliser l'unification en traduisant les types graduels dans des types statiques (en remplaçant les occurrences de ? par des variables de type).

Nous ajoutons ensuite du sous-typage au système précédent. Ajouter le sous-typage sémantique au système de types revient à ajouter une règle de subsomption : cependant, cette règle doit utiliser une relation de sous-typage sur les types graduels. Cette relation ne peut pas être définie directement en étendant l'interprétation ensembliste $[\![\cdot]\!]$ aux types graduels, car le type dynamique ? ne peut pas être interprété comme un ensemble. Pour palier à ce problème, nous traduisons les types graduels dans des types ensemblistes polymorphes, cette fois aussi en remplaçant les occurrences de ? par des variables de type (en faisant attention à l'interaction de ? avec les types négation). Nous étendons l'inférence de types au sous-typage et nous prouvons qu'elle est correcte (mais pas complète).

Le typage graduel est une technique essentielle pour ajouter une forme de typage statique à des langages qui étaient auparavant typés dynamiquement. Ce travail est donc un pas vers l'objectif de rendre les types ensemblistes avec sous-typage sémantique un outil efficace pour typer ces langages.

Langages non-stricts

La troisième partie montre comment adapter les systèmes avec types ensemblistes à des langages avec sémantiques non-strictes. Les systèmes existants qui utilisent le sous-typage sémantique ont été développés pour des langages avec appel par valeur. Il ne sont pas sûrs pour des sémantiques non-strictes, à cause de la manière dont le sous-typage traite le type minimum (noté $\mathbb{0}$). Ce type correspond à l'ensemble vide des valeurs et il ne peut être dérivé que pour les expressions qui sont sûrement divergentes. Certaines des équivalences satisfaites par le sous-typage sémantique utilisant ce type ne sont pas appropriées pour les sémantiques non-strictes. Par exemple, les deux types $\mathbb{0} \times \mathsf{Int}$ et $\mathbb{0} \times \mathsf{Bool}$ sont considérés équivalents : en effet, dans un langage avec appel par valeur, aucun des deux ne contient une valeur (étant donné qu'il n'y a pas de valeurs de type $\mathbb{0}$ et qu'une valeur dans un type produit est une paire de valeurs). Dans un langage non-strict, on ne peut pas identifier ces deux types parce qu'ils peuvent être distingués : les projections des paires peuvent être

évaluées même si une composante de la paire diverge.

Pour recouvrer la correction, nous ne changeons pas le sous-typage sémantique dans ses fondements car cela nous empêcherait de réutiliser beaucoup des résultats existants (notamment ceux qui concernent l'algorithme de décision du sous-typage). Par contre, nous ajoutons un nouveau type $\bot$ pour representer la divergence : ce type nous permet de distinguer au niveau des types les expressions qui terminent de celles qui pourraient diverger. Nous modifions les règles de typage pour prendre en compte la divergence, avec une forte approximation : nous supposons que toute expression qui demande une évaluation pourrait éventuellement diverger.

Nous décrivons ce système de types pour un λ-calcul typé de manière explicite qui est assez proche au langage étudié par Frisch, Castagna et Benzaken (2008), mais qui est évalué en appel par nécessité. La choix de l'appel par nécessité (au lieu de l'appel par nom) est motivé par la présence des types union, qui exigeraient une règle complexe de disjunction de l'union pour garantir la réduction du sujet (et qui, si on étendait le langage avec des constructions non déterministes, feraient en fait échouer la réduction du sujet). Nous prouvons que le système de types obtenu est correct. La relation de sous-typage maintient beaucoup des propriétés du sous-typage sémantique pour langages stricts : en particulier, elle permet le même usage des types intersection pour typer les fonctions surchargées.

Contents

Introduction **23**

1 Introduction **25**
1.1 Background and motivations . . . 25
1.1.1 Set-theoretic types . . . 26
1.1.2 Subtyping on set-theoretic types . . . 29
1.1.3 Semantic subtyping . . . 30
1.2 Overview and contributions . . . 31
1.2.1 Implicit typing and type inference . . . 32
1.2.2 Gradual typing . . . 32
1.2.3 Non-strict languages . . . 33
1.3 Relationship with published or submitted work . . . 34
1.4 Outline . . . 35
1.5 Notational conventions . . . 36

2 Background **39**
2.1 Introduction . . . 39
2.1.1 Semantic subtyping for first-order languages . . . 41
2.1.2 Adding arrow types . . . 41
2.1.3 Adding type variables . . . 44
2.2 Types . . . 46
2.2.1 Type substitutions . . . 48
2.3 Semantic subtyping . . . 49
2.4 Study of the subtyping relation . . . 50
2.4.1 Defining subtyping using quantification . . . 50
2.4.2 Subtyping and type substitutions . . . 53
2.4.3 Decomposition of subtyping on arrow types . . . 55

I Implicit typing and type inference **59**

3 An implicitly typed language with set-theoretic types **61**
3.1 Language syntax and semantics . . . 61
3.1.1 Syntax . . . 61
3.1.2 Semantics . . . 62
3.2 Type system . . . 63
3.3 Type soundness . . . 66
3.3.1 Why subject reduction does not hold . . . 67
3.3.2 Negation types for functions . . . 68
3.3.3 Deriving negations of arrow types . . . 70

3.3.4 Substitution and weakening properties 72
3.3.5 Inversion of the typing relation 76
3.3.6 Relating ground types and sets of values 78
3.3.7 Progress, subject reduction, and soundness 80

4 Type inference **87**
4.1 The reformulated type system 89
4.1.1 The problem with generalization 89
4.1.2 Definition of the reformulated type system 91
4.1.3 Relating the systems $\mathcal{T}^{\mathrm{i}}$ and $\mathcal{T}^{\mathrm{r}}$ 93
4.1.4 Inversion for the type system $\mathcal{T}^{\mathrm{r}\setminus\wedge}$ 100
4.2 Constraints and constraint generation 101
4.2.1 Constraints and constraint satisfaction 101
4.2.2 Constraint generation 102
4.2.3 Relating typing with constraint satisfaction 104
4.2.4 Properties of structured-constraint satisfaction 107
4.3 Constraint solving 108
4.3.1 Type-constraint solving by tallying 108
4.3.2 Structured-constraint simplification 110
4.4 Results and discussion 115
4.4.1 Non-determinism and lack of principal solutions . . . 116

5 Adding type annotations **119**
5.1 Language syntax and type system 119
5.1.1 Syntax 119
5.1.2 Reformulated type system 120
5.2 Constraints and constraint solving 122
5.2.1 Constraints and constraint satisfaction 122
5.2.2 Constraint generation 123
5.2.3 Constraint solving 126
5.3 Results and discussion 128
5.3.1 Towards a stronger completeness result 129

6 Language extensions **131**
6.1 Binding typecase and pattern matching 131
6.1.1 Binding typecase 131
6.1.2 Pattern matching 132
6.2 Polymorphic variants 133
6.3 Records 134
6.3.1 Polymorphic typing of record operations 135

7 Discussion **137**
7.1 Related work 137
7.2 Future work 140

II Gradual typing **143**

8 Introduction **145**
8.1 Gradual typing with polymorphic set-theoretic types 145
8.2 Our approach 147
8.3 Overview 149

9 Gradual typing for Hindley-Milner systems **151**
9.1 Source language 151
9.1.1 Types and expressions 151
9.1.2 Type system 152
9.1.3 Static gradual guarantee 155
9.1.4 Relationship with standard gradual type systems . . . 156
9.2 Cast language 158
9.2.1 Syntax 158
9.2.2 Type system 159
9.2.3 Semantics 160
9.2.4 Compilation 161
9.3 Type inference 162
9.3.1 Type constraints and solutions 163
9.3.2 Type-constraint solving 164
9.3.3 Structured constraints and constraint generation . . . 166
9.3.4 Constraint solving 167
9.3.5 Soundness of type inference 171
9.3.6 Completeness of type inference 173
9.3.7 An example of type inference 174
9.4 Adding subtyping 175
9.4.1 Declarative system 176
9.4.2 Type inference 176

10 Gradual typing for set-theoretic types **179**
10.1 Type frames, static types, and gradual types 179
10.1.1 Subtyping on type frames and static types 181
10.1.2 Materialization 181
10.2 Subtyping on gradual set-theoretic types 181
10.2.1 Polarity, parity, and variance 182
10.2.2 Subtyping using polarized discriminations 183
10.2.3 Avoiding existential quantification 184
10.2.4 Equivalence of the different characterizations of subtyping 186
10.2.5 Properties of subtyping 190
10.3 Source and cast languages 193
10.3.1 Syntax and typing 193
10.3.2 Semantics 193
10.4 Type inference 194
10.4.1 Type constraints and solutions 194

10.4.2 Type-constraint solving . . . 194
10.4.3 Structured constraints, generation, and simplification . 197
10.4.4 Soundness of type inference . . . 197

11 Discussion **201**
11.1 Related work . . . 201
11.2 Future work . . . 203

III Non-strict languages **205**

12 Introduction **207**
12.1 Semantic subtyping for non-strict languages . . . 207
12.2 Our approach . . . 208
12.3 Contributions . . . 211
12.4 Related work . . . 211

13 A call-by-need language with set-theoretic types **213**
13.1 Types and subtyping . . . 213
13.1.1 Properties of subtyping . . . 215
13.2 Language syntax and semantics . . . 216
13.2.1 Source language . . . 217
13.2.2 Internal language . . . 218
13.2.3 Semantics . . . 218
13.3 Type system . . . 221
13.3.1 Type system of the source language . . . 222
13.3.2 Type system of the internal language . . . 223
13.4 Proving type soundness . . . 225
13.4.1 Call-by-name and call-by-need . . . 226
13.4.2 Proving subject reduction: challenges . . . 227
13.4.3 Decompositions of product types . . . 228
13.4.4 Additional results . . . 230
13.4.5 Progress and subject reduction . . . 232

14 Discussion **233**
14.1 On the interpretation of types . . . 233
14.2 Future work . . . 237

Conclusion **239**

15 Conclusion **241**
15.1 Future work . . . 242

Appendices **245**

A Additional proofs **247**
Implicit typing and type inference 247
Adding type annotations 247
Gradual typing 251
Gradual typing for Hindley-Milner systems 251
Gradual typing for set-theoretic types 269
Non-strict languages 291
A call-by-need language with set-theoretic types 291
Discussion 307

B Semantics of the cast languages **313**
B.1 Semantics of the cast language without subtyping 313
B.1.1 Adding subtyping 314
B.2 Semantics of the cast language with set-theoretic types 315
B.2.1 Defining cast application and projection operators . . 319

Bibliography **323**

List of figures

3.1 Reduction rules 63
3.2 $\mathcal{T}$: Typing rules 65
3.3 $\mathcal{T}^{\mathrm{n}}$: Size-indexed typing rules 71

4.1 $\mathcal{T}^{\mathrm{i}}$: Typing rules 88
4.2 $\mathcal{T}^{\mathrm{r}}$: Reformulated typing rules 92
4.3 $\mathcal{T}^{\mathrm{ri}}$: Reformulated typing rules with explicit instantiations . . 96
4.4 $\mathcal{C}^{\mathrm{sat}}$: Constraint satisfaction rules 103
4.5 Constraint generation 103
4.6 $\mathcal{C}^{\mathrm{sim}}$: Constraint simplification rules 109

5.1 $\mathcal{T}^{\mathrm{ra}}$: Reformulated typing rules (with type annotations) 121
5.2 $\mathcal{C}^{\mathrm{sata}}$: Constraint satisfaction rules (with type annotations) . . 122
5.3 Constraint generation (with type annotations) 124
5.4 $\mathcal{C}^{\mathrm{sima}}$: Constraint simplification rules (with type annotations) . 127

6.1 Semantics of patterns 132
6.2 Environment typing for patterns 133

9.1 $\mathcal{T}_?$: Typing rules of the source language 153
9.2 Lifting of the materialization relation to expressions 155
9.3 Monomorphic restriction of the implicative fragment of $\mathcal{T}_?$ . . 157
9.4 Polymorphic restriction of the implicative fragment of $\mathcal{T}_?$ . . . 158
9.5 $\mathcal{T}_?^{\langle\rangle}$: Typing rules of the cast language 160
9.6 $\mathcal{T}_?^{\rightsquigarrow}$: Compilation from the source language to the cast language 162
9.7 Constraint generation 167
9.8 $\mathcal{C}_?^{\mathrm{sim}}$: Constraint simplification rules 168
9.9 Algorithmic compilation 170

13.1 Reduction rules 220
13.2 $\mathcal{T}_\perp^{\mathrm{s}}$: Typing rules of the source language 222
13.3 $\mathcal{T}_\perp^{\mathrm{i}}$: Typing rules of the internal language 224

B.1 Reduction rules of the cast language without subtyping 314
B.2 Reduction rules of the cast language with set-theoretic types . 317

List of inference systems

We list here the main inference systems used throughout this thesis to define type systems and constraint-based type inference. For each system, we give its name (e.g., $\mathcal{T}$ or $\mathcal{T}^{\lambda\neg}$) and the shape of its judgments (e.g., $\Gamma \vdash e : t$), and we point to where it is defined.

Part I

$\mathcal{T}$	$\Gamma \vdash e : t$	Figure 3.2 (p. 65)
$\mathcal{T}^{\mathrm{n}}$	$\Gamma \vdash_n e : t$	Definition 3.8 (p. 70), Figure 3.3 (p. 71)
$\mathcal{T}^{\lambda\neg}$	$\Gamma \vdash e : t$	Definition 3.11 (p. 72), Figure 3.2 (p. 65)
$\mathcal{T}^{\mathrm{i}}$	$\Gamma \vdash e : t$	Figure 4.1 (p. 88)
$\mathcal{T}^{\mathrm{i}\setminus\wedge}$	restriction of $\mathcal{T}^{\mathrm{i}}$ without the rule $[\mathrm{T}_\wedge]$	
$\mathcal{T}^{\mathrm{r}}$	$P; M \Vdash e : t$	Definition 4.2 (p. 91), Figure 4.2 (p. 92)
$\mathcal{T}^{\mathrm{r}\setminus\wedge}$	restriction of $\mathcal{T}^{\mathrm{r}}$ without the rule $[\mathrm{T}^{\mathrm{r}}_\wedge]$	
$\mathcal{T}^{\mathrm{ri}}$	$P; M \Vdash e : t \mid \mathcal{I}$	Definition 4.7 (p. 95), Figure 4.3 (p. 96)
$\mathcal{C}^{\mathrm{sat}}$	$P; M; \sigma \Vdash C$	Definition 4.19 (p. 102), Figure 4.4 (p. 103)
$\mathcal{C}^{\mathrm{sim}}$	$P \vdash C \rightsquigarrow D \mid M \mid \vec{\alpha}$	Definition 4.26 (p. 110), Figure 4.6 (p. 109)
$\mathcal{T}^{\mathrm{ra}}$	$P; M; \Delta \Vdash \mathbf{e} : t$	Section 5.1.2 (p. 120), Figure 5.1 (p. 121)
$\mathcal{T}^{\mathrm{ra}\setminus\wedge}$	restriction of $\mathcal{T}^{\mathrm{ra}}$ without the rule $[\mathrm{T}^{\mathrm{ra}}_\wedge]$	
$\mathcal{C}^{\mathrm{sata}}$	$P; M; \Delta; \sigma \Vdash C$	Section 5.2.1 (p. 122), Figure 5.2 (p. 122)
$\mathcal{C}^{\mathrm{sima}}$	$P; \Delta \vdash C \rightsquigarrow D \mid M \mid \vec{\alpha}$	Section 5.2.3 (p. 126), Figure 5.4 (p. 127)

Part II

$\mathcal{T}_?$	$\Gamma \vdash e : \tau$	Section 9.1.2 (p. 152), Figure 9.1 (p. 153)
$\mathcal{T}_?^{\langle\rangle}$	$\Gamma \vdash E : \tau$	Section 9.2.2 (p. 159), Figure 9.5 (p. 160)
$\mathcal{T}_?^{\rightsquigarrow}$	$\Gamma \vdash e \rightsquigarrow E : \tau$	Section 9.2.4 (p. 161), Figure 9.6 (p. 162)
$\mathcal{C}_?^{\mathrm{sim}}$	$\Gamma; \Delta \vdash C \rightsquigarrow D \mid \vec{\alpha}$	Section 9.3.4 (p. 167), Figure 9.8 (p. 168)

Part III

$\mathcal{T}_\perp^{\mathrm{s}}$	$\Gamma \vdash \mathbf{e} : t$	Section 13.3.1 (p. 222), Figure 13.2 (p. 222)
$\mathcal{T}_\perp^{\mathrm{i}}$	$\Gamma \vdash e : t$	Section 13.3.2 (p. 223), Figure 13.3 (p. 224)

Introduction

1 Introduction

In this thesis, we study *set-theoretic types*: types that include union, intersection, and negation connectives. Set-theoretic types can be used to type several language constructs – including conditional branching, pattern matching, and function overloading – very precisely when coupled with a suitable subtyping relation. We define subtyping following the *semantic subtyping* approach of Frisch, Castagna, and Benzaken (2008).

Set-theoretic types and semantic subtyping have been adapted to various settings and language features over time. In this thesis, we continue along this path by showing how to use set-theoretic types to design type systems for different functional languages: implicitly typed languages with type inference, gradually typed languages, and non-strict languages.

1.1 Background and motivations

Much research on type systems for programming languages tries to devise systems that are more accurate in characterizing the behaviour and properties of programs, so that type checkers can recognize more kinds of errors while rejecting fewer correct programs. *Polymorphism* is a major ingredient towards this goal. In a polymorphic type system, expressions may have more than one type; these types express how they behave in different contexts or describe them more or less precisely. We often distinguish three forms of polymorphism, as follows.

Parametric polymorphism: describing code that can act uniformly on any type, using type variables that can be instantiated with any type (e.g., typing the identity function as $\forall\alpha.\,\alpha \to \alpha$).

Ad-hoc polymorphism: allowing code that can act on more than one type, possibly with different behaviour in each case, as in function overloading (e.g., allowing "+" to have both types $\mathsf{Int}\times\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Real}\times\mathsf{Real} \to \mathsf{Real}$, corresponding to different implementations).

Subtype polymorphism: creating a hierarchy of more or less precise types for the same code (e.g., typing 3 as both Int and Real, with $\mathsf{Int} \leq \mathsf{Real}$).

All three forms feature prominently in this thesis. Subtype polymorphism is fundamental for set-theoretic types and is used throughout all of the thesis except for Chapter 9. The systems of Parts I and II feature parametric polymorphism; we consider let-polymorphism in the style of ML – also called *prenex* polymorphism – and not the first-class polymorphism of System F. Intersection types and the typecase construct allow ad-hoc polymorphism in the systems of Parts I and III.

1.1.1 Set-theoretic types

Set-theoretic types include *union types* $t_1 \vee t_2$, *intersection types* $t_1 \wedge t_2$, and *negation types* $\neg t$. Intuitively:

- $t_1 \vee t_2$ is the type of values that are *either* of type t_1 *or* of type t_2;
- $t_1 \wedge t_2$ is the type of values that are *both* of type t_1 *and* of type t_2;
- $\neg t$ is the type of values that are *not* of type t.

We speak of *polymorphic set-theoretic types* when set-theoretic types include type variables to allow prenex parametric polymorphism (as in Parts I and II).

These types allow us to type several features and idioms of programming languages effectively. We illustrate this with some examples.

UNION TYPES: The simplest use cases for union types include branching constructs. In a language with union types, we can type precisely conditionals that return results of different types: for instance, if e then 3 else true has type Int $\vee$ Bool (provided that e has type Bool). Without union types, it could have an approximated type (e.g., a top type) or be ill-typed. Similarly, we can use union types for structures like lists that mix different types: for instance, typing [1, false, "string"] as List(Int $\vee$ Bool $\vee$ String).

This makes union types invaluable to design type systems for previously untyped languages: witness for example their inclusion in Typed Racket (Tobin-Hochstadt and Felleisen, 2008) and in TypeScript (Microsoft, 2018) and Flow (Facebook, 2018), both of which extend JavaScript with static type checking.

FUNCTION OVERLOADING: We can use intersection types to assign more than one type to an expression. This is particularly relevant for functions. For example, the identity function can be typed as (Int $\to$ Int) $\wedge$ (Bool $\to$ Bool): this means it has both types Int $\to$ Int and Bool $\to$ Bool, because it maps integers to integers and Booleans to Booleans. This type describes uniform behaviour over two different argument types, which can also be described using parametric polymorphism. However, intersection types let us express ad-hoc polymorphism if coupled with some mechanism that allows functions to test the type of their argument. For example, the function $\lambda x.\, x \in \mathsf{Int}\ ?\ (x + 1) : \neg x$ checks whether its argument x is an Int and returns the successor of x in that case, its negation otherwise. The function can be applied to integers, returning their successor, and to Booleans, returning their negation. This behaviour can be described by the same type (Int $\to$ Int) $\wedge$ (Bool $\to$ Bool) but does not correspond to parametric behaviour.

A function of type $(t_1 \to t_1') \wedge (t_2 \to t_2')$ can be applied safely to any argument of type $t_1 \vee t_2$, since it is defined on both t_1 and t_2. We know that the result will always have type $t_1' \vee t_2'$. However, if we know the type of the argument more precisely, we can predict the type of the result more precisely: for example, if the argument is of type t_1, then the result will be of type t_1'.

We have said that the type (Int $\to$ Int) $\wedge$ (Bool $\to$ Bool) can be assigned to the identity function and expresses parametric behaviour. In this respect,

we can see intersection types as a finitary form of parametric polymorphism; however, they are not constrained to represent uniform behaviour, as our other example illustrates. Conversely, we could see a polymorphic type (or type scheme) $\forall\alpha.\, \alpha \to \alpha$ as an infinite intersection (intuitively, $\bigwedge_{t \in \mathsf{Type}} t \to t$, where Type is the set of all types), but infinite intersections do not actually exist in our types.

OCCURRENCE TYPING: *Occurrence typing* or *flow typing* (Tobin-Hochstadt and Felleisen, 2010; Pearce, 2013; Chaudhuri et al., 2017) allows the type of a variable to be made more precise in the branches of conditionals. For example, if x is of type Int ∨ Bool, then to type an expression $x \in \mathsf{Int}\ ?\ e_1 : e_2$ we can assume that the occurrences of x in e_1 have type Int and those in e_2 have type Bool, because the first branch will only be reached if x is an Int and the second if it is not an Int (and is therefore a Bool). Intersection and negation types are useful to describe this type discipline. If we test for the type Int as in our example, then we can assign to x the type Int if the test succeeds and ¬Int if it fails. Using intersections, we can add this information to what we had already, so the type of x is (Int ∨ Bool) ∧ Int (which should be equal to Int) in the first branch and (Int ∨ Bool) ∧ ¬Int (which should be equal to Bool) in the second branch.

This method of refining types according to conditionals is important in type systems for dynamic languages and in those that enforce null safety: some examples include Ceylon (King, 2017), Flow, Kotlin (JetBrains, 2018), Typed Racket, TypeScript, and Whiley (Pearce and Groves, 2013). In particular, Ceylon relies on intersection types (King, 2017; Muehlboeck and Tate, 2018) and Whiley on both intersection and negation types (Pearce, 2013).

ENCODING DISJOINT UNION TYPES: Disjoint union types (also known as variant or sum types) are an important feature of functional programming languages. They can be encoded using union types and product (or record, or object) types. It is also useful to have *singleton types*, that is, types that correspond to a single value: for example, two types true and false for the respective constants, both subtypes of the Boolean type (which we can see as the union true ∨ false).

For instance, consider this example in Flow.[1]

```
type Success = { success: true, value: boolean }
type Failed = { success: false, error: string }
type Response = Success | Failed

function handleResponse(response: Response) {
  if (response.success) { var value: boolean = response.value }
  else { var error: string = response.error }
}
```

The type Response is the union (denoted by "|") of two object types: both have a Boolean field success, but the types state that success must be true for objects

1 From the documentation of Flow, available at https://flow.org/en/docs/types/unions.

of type Success and false for objects of type Failure. An analogous type could be declared in OCaml as type response = Success of bool | Failed of string. Occurrence typing is used to distinguish the two cases, like pattern matching could do in ML: if response.success is true, then response must be of type Success; if it is false, response must be of type Failure.

TYPING PATTERN MATCHING: Pattern matching is widely used in functional programming. However, using pattern matching in ML-like languages, we can write functions that cannot be given an exact domain in the type system. For instance, the OCaml code let f = function 0 ⟶ true | 1 ⟶ false defines a function that can only be applied to the integers 0 and 1, but OCaml infers the unsafe type int ⟶ bool (albeit with a warning that pattern matching is not exhaustive). The precise domain cannot be expressed in OCaml. Using set-theoretic types and singleton types, we can express it precisely as $0 \vee 1$. Intersection and negation types are also useful, as for occurrence typing, to describe the types of variables in the patterns.

ENCODING BOUNDED POLYMORPHISM: Using union and intersection types, we can encode bounded polymorphism as unbounded polymorphism. For example, a type scheme with bounded polymorphism is $\forall(\alpha \leq t).\, \alpha \to \alpha$: it describes functions that can be applied to arguments of any subtype of t and that return a result of the same type as the argument. Using intersection types, we can write $\forall\alpha.\, (\alpha \wedge t) \to (\alpha \wedge t)$, writing the bound on the occurrences of the type variable and not on the quantifier. Analogously, we can use union types to represent lower bounds: in general, a bound $t' \leq \alpha \leq t$ on a type can be eliminated by replacing every occurrence of α in the type with $(\alpha \wedge t) \vee t'$.

NEGATION TYPES: Assume that x has type Int ∨ Bool; to type the typecase $x \in \mathsf{Int}\ ?\ e_1 : e_2$, we can assume that the occurrences of x in e_2 have type $(\mathsf{Int} \vee \mathsf{Bool}) \wedge \neg\mathsf{Int}$ (which should be Bool). We express this using negation types. To avoid introducing negation in types, instead, we could use a meta-operation of type difference, written $t_1 \setminus t_2$, such that $(\mathsf{Int} \vee \mathsf{Bool}) \setminus \mathsf{Int} = \mathsf{Bool}$. However, sometimes we would not be able to express the result of type difference precisely: for example, $\alpha \setminus \mathsf{Int}$ could not be expressed as a type. Using negation types, instead, difference is just a shorthand for intersection with the negation type: $t_1 \setminus t_2 \stackrel{\text{def}}{=} t_1 \wedge \neg t_2$. Consider for instance a function $\lambda x.\, x \in \mathsf{Int}\ ?\ (x + 1) : x$. It can act on arguments of any type, computing the successor of integers and leaving other arguments unchanged. Using intersection and difference types, plus parametric polymorphism, we can type it as $\forall\alpha.\, (\mathsf{Int} \to \mathsf{Int}) \wedge (\alpha \setminus \mathsf{Int} \to \alpha \setminus \mathsf{Int})$, which expresses its behaviour precisely.

Castagna et al. (2015b, app. A) present a compelling example of the use of polymorphic set-theoretic types to type the function to insert a new node in a red-black tree. The types enforce three out of the four invariants of red-black trees,[2] requiring only the addition of type annotations to the code and no other

2 Specifically, that the root of the tree is black, that the leaves of the tree are black, and that

change to a standard implementation (due to Okasaki, 1998). The type of the balancing function is

$$\forall\alpha,\beta.\ \big(\mathsf{Unbalanced}(\alpha) \to \mathsf{Rtree}(\alpha)\big) \wedge \big(\beta\backslash\mathsf{Unbalanced}(\alpha) \to \beta\backslash\mathsf{Unbalanced}(\alpha)\big)$$

and uses difference types like our example above: it maps unbalanced binary trees (of elements of type α) to red-rooted balanced trees, and it leaves any other argument unchanged.

1.1.2 Subtyping on set-theoretic types

We have given examples of the use of set-theoretic types, but up to now we have glossed over exactly how a type checker should treat them. It is essential to define a suitable notion of *subtyping* on these types. The informal description we have given suggests that certain properties should hold. In particular, we expect union and intersection types to satisfy commutative and distributive properties. Moreover, we expect, for example,

$$(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool}) \leq (\mathsf{Int} \vee \mathsf{Bool}) \to (\mathsf{Int} \vee \mathsf{Bool})$$

to hold to have the typing of functions with typecases work as we sketched. To model occurrence typing, we want $(\mathsf{Int} \vee \mathsf{Bool}) \wedge \mathsf{Int}$ to be equivalent to Int and $(\mathsf{Int} \vee \mathsf{Bool}) \wedge \neg\mathsf{Int}$ to be equivalent to Bool.

Arguably, it is intuitive to view types and subtyping in terms of sets and set inclusion, especially to describe set-theoretic types.[3] We can see a type as the set of the values of that type in the language we consider. Then, we expect t_1 to be a subtype of t_2 if every value of type t_1 is also of type t_2, that is, if the set of values denoted by t_1 is included in that denoted by t_2. In this view, union and intersection types correspond naturally to union and intersections of sets; negation corresponds to complementation with respect to the set of all values.

However, most systems reason on subtyping using rules that are sound but not complete with respect to this model: that is, they do not allow $t_1 \leq t_2$ in some cases in which every value of type t_1 is in fact a value of type t_2. Incompleteness is not necessarily a problem, but it can result in unintuitive behaviour. We show two examples below.

LACK OF DISTRIBUTIVITY: Consider this code in Flow.[4]

```
type A = { a: number }
type B = { kind: "b", b: number }
type C = { kind: "c", c: number }

type T = (A & B) | (A & C)
function f (x: T) { return (x.kind === "b") ? x.b : x.c }
```

no red node has a red child; the missing invariant is that every path from the root to a leaf should contain the same number of black nodes.

3 For instance, this model is used to explain subtyping in the online documentation of Flow at https://flow.org/en/docs/lang/subtypes.

4 Adapted from the StackOverflow question at https://stackoverflow.com/questions/44635326.

The first three lines declare three object types; in B and C, "b" and "c" are the singleton types of the corresponding strings. The type T is defined as the union of two intersection types (Flow denotes intersection by "&").

The function f is well typed: as in handleResponse before, occurrence typing recognizes that x is of type A & B in the branch x.b and of type A & C in the branch x.c. However, if we replace the definition of T to be type T = A & (B | C), the code is rejected by the type checker of Flow. Occurrence typing does not work because T is no longer explicitly a union type. Flow considers (A & B) | (A & C) a subtype of A & (B | C): indeed, this can proven just by assuming that unions and intersections are respectively joins and meets for subtyping. But subtyping does not hold in the other direction, because Flow does not consider distributivity.

UNION AND PRODUCT TYPES: Apart from distributivity laws, we could also expect interaction between union and intersection types and various type constructors. Consider product types; we might expect the two types $(t_1 \times t) \vee (t_2 \times t)$ and $(t_1 \vee t_2) \times t$ to be equivalent: intuitively, both of them describe the pairs whose first component is either in t_1 or in t_2 and whose second component is in t. But this reasoning is not always reflected in the behaviour of type checkers.

For example, consider this code in Typed Racket (similar examples can be written in Flow or TypeScript).

```
(define-type U-of-Pair (U (Pair Integer Boolean) (Pair String Boolean)))
(define-type Pair-of-U (Pair (U Integer String) Boolean))

(define f (lambda ([x : U-of-Pair]) x))
(define x (ann (cons 3 #f) Pair-of-U))
(f x)
```

We define two type abbreviations. In Typed Racket, U denotes a union type and Pair a product type, so U-of-Pair is (Integer × Boolean) ∨ (String × Boolean), and Pair-of-U is (Integer ∨ String) × Boolean. The two types are not considered equivalent. To show it, we define a function f whose domain is U-of-Pair (for simplicity, we take the identity function) and try to apply it to an argument x of type Pair-of-U; to define x, we use an explicit type annotation (ann) to mark the pair (cons 3 #f) as having type Pair-of-U. The application is rejected. If we exchange the two type annotations, instead, it is accepted: the type checker considers U-of-Pair a subtype of Pair-of-U, but not the reverse.

1.1.3 Semantic subtyping

To define subtyping for set-theoretic types, we use the *semantic subtyping* approach, following Frisch, Castagna, and Benzaken (2008) and later work. We give a detailed introduction to this approach in Chapter 2. In brief, using semantic subtyping means that we interpret types as sets and define subtyping as set inclusion. Therefore, we take the intuitive view of subtyping that we have discussed and use it as the actual definition of subtyping, except that, as

we will explain, we cannot interpret types directly as sets of values, but we must find an alternative interpretation that induces the subtyping relation we want.

An advantage of semantic subtyping is that the interpretation of types serves as a simple specification of the behaviour of a subtyping algorithm derived from it. Properties such as distributivity of intersections over unions and the equivalence of product types above can be verified easily on the interpretation that we will describe. If the interpretation and the language match well enough, subtyping can be complete with respect to the intuitive interpretation of types as sets of values. While we will not have such a result in this work, we will have some partial results of this kind. For instance, in the system of Part I we will prove that the values in a type $t_1 \vee t_2$ are exactly those either in t_1 or in t_2, provided that t_1 and t_2 are ground (i.e., without type variables).

Semantic subtyping was first developed for domain-specific languages for XML processing with the work on XDuce by Hosoya and Pierce (2003). It has been extended to consider higher-order functions (Benzaken, Castagna, and Frisch, 2003; Frisch, Castagna, and Benzaken, 2008) and parametric polymorphism (Castagna and Xu, 2011; Gesbert, Genevès, and Layaïda, 2011; Castagna et al., 2014, 2015b). This approach has also been used in different settings including object-oriented languages (Dardha, Gorla, and Varacca, 2013; Ancona and Corradi, 2016), XML and NoSQL query languages (Benzaken et al., 2013; Castagna et al., 2015a), and process calculi (Castagna, De Nicola, and Varacca, 2008). However, its interaction with many other language features remains unexplored.

1.2 Overview and contributions

In this thesis we study how to use set-theoretic types with semantic subtyping to type different features of functional programming languages. Specifically, we consider implicit typing and type inference, let-polymorphism, gradual typing, and non-strict semantics.

We argue that set-theoretic types allow us to obtain rich type systems for these different settings and language features. We also argue that semantic subtyping is an effective approach to define subtyping in such systems. In particular, in all this work we show that we can reuse directly many of the previous results on semantic subtyping – notably, the algorithms to decide subtyping and to solve subtyping constraints – even in these different settings; however, we will also point out adaptations that should be made in order to continue this work and improve on its results. While we do not prove that subtyping is complete with respect to an interpretation of types as sets of values, using semantic subtyping we still obtain an expressive subtyping relation which satisfies the properties we need to obtain the type discipline that we have sketched.

The thesis is organized in three parts in which we consider different language features. We introduce each of these in the next three subsections.

1.2.1 Implicit typing and type inference

In Part I we study how to use polymorphic set-theoretic types for implicitly typed languages with let-polymorphism and type inference. In contrast, previous work on semantic subtyping studied languages where functions are explicitly annotated with their type (Frisch, Castagna, and Benzaken, 2008; Castagna et al., 2014) and considered at most local type inference to infer the instantiations of polymorphic functions (Castagna et al., 2015b).

The language we study is a call-by-value λ-calculus with constants, pairs, a typecase construct (to model runtime type dispatch and pattern matching), and let binders.

We describe a type system for this language: a standard Hindley-Milner system extended with the following structural rules for subsumption and intersection introduction.

$$[\mathrm{T}_{\leq}]\ \frac{\Gamma \vdash e : t'}{\Gamma \vdash e : t}\ t' \leq t \qquad\qquad [\mathrm{T}_{\wedge}]\ \frac{\Gamma \vdash e : t_1 \qquad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2}$$

The system is straightforward to describe. However, the proof of soundness with respect to the semantics is challenging because of the presence of $[\mathrm{T}_{\wedge}]$ and of negation types. To ensure subject reduction, we must extend the system with a rule to derive negation types for functions, in order, for example, to have $\vdash \lambda x.\, x : \neg(\mathsf{Int} \to \mathsf{Bool})$. This difficulty is already solved for previous work (Frisch, Castagna, and Benzaken, 2008), but here it is more challenging and requires a different solution because functions are not annotated. We develop this solution and the proof of soundness for the extended system; this implies soundness also for the simpler system without that rule.

We then study type inference, defining a type inference algorithm based on constraint generation and solving. The constraints we use are similar to those of Pottier and Rémy (2005); constraint solving reuses the *tallying* algorithm of Castagna et al. (2015b). We prove that inference is sound with respect to the type system and complete with respect to the restriction of the system without the rule $[\mathrm{T}_{\wedge}]$. We do not relate inference to the original type system directly, but to a different one – closely based on the "reformulated typing rules" of Dolan and Mycroft (2017) – which we show to be equivalent to the original. This different system handles generalization for let in a way that is more convenient to relate to the inference algorithm.

Then, we add type annotations to the language and show how inference can use them to compute more precise types (notably, intersection types for functions). Finally, we outline how to extend the language with additional features: pattern matching, OCaml-style polymorphic variants, and records.

1.2.2 Gradual typing

Part II studies *gradual typing*, an approach that allows static and dynamic typing to coexist in the same language (Siek and Taha, 2006). This is achieved by introducing an *unknown* type, written "?", and by relaxing the type system

allowing expressions of type ? to be used in any context. Therefore, programs are type checked statically only in part; to ensure safe execution, they are compiled to a *cast language* with runtime type tests. Soundness ensures that well-typed programs produce a value, diverge, or fail because of such tests, but cannot go wrong otherwise.

Our contributions are the description of a new approach to make a static type system gradual and its development for type systems both without and with subtyping.

We first add gradual typing to a standard Hindley-Milner type system. The novelty is that we define a gradual type system by adding a single rule to the static system: a subsumption-like structural rule using the *precision* relation from gradual typing literature. In contrast, the existing systems for gradual typing rely on the *consistency* relation, which cannot be used in a structural rule because it is not transitive: therefore, they embed checks for consistency in several rules. The difference between our system and the existing ones thus mirrors that between *declarative* (i.e., with structural rules) and *algorithmic* (without them) type systems with subtyping. We define a cast language with a standard semantics and describe compilation to it: each use of the structural rule for precision corresponds to the insertion of a cast in the compiled program. We describe type inference for the system and prove it sound and complete. We show that, for constraint solving, we can use unification by translating gradual types to static types, changing occurrences of ? to type variables.

Then, we add subtyping. Adding semantic subtyping to the type system amounts to adding a subsumption rule, but this rule must use a subtyping relation on gradual types. This cannot be defined directly by extending the interpretation $[\![\cdot]\!]$ to gradual types: the dynamic type ? cannot be given a set-theoretic interpretation. Rather, we translate gradual types to polymorphic set-theoretic types, again by changing occurrences of the dynamic type ? to type variables (some care is needed for negation). We extend type inference to subtyping and prove it sound (but not complete).

Gradual typing has emerged as an essential technique to add static typing to previously untyped languages. Therefore, this work is a step towards making set-theoretic types with semantic subtyping a viable tool to type such languages.

1.2.3 Non-strict languages

In Part III we show how to adapt set-theoretic type systems for non-strict languages. The existing type systems using semantic subtyping are designed for call-by-value languages. They are unsound for non-strict semantics because of how subtyping deals with the bottom type $\mathbb{0}$. This type corresponds to the empty set of values and can be assigned soundly only to expressions that can be proven to diverge. Some of the laws satisfied by semantic subtyping are inappropriate for non-strict semantics. For instance, the types $\mathbb{0} \times \mathsf{Int}$ and $\mathbb{0} \times \mathsf{Bool}$ are considered equivalent: indeed, in a call-by-value language, none contains any value (a value in a product type must be pair of values, and there

are no values in $\mathbb{0}$). In a non-strict language, it is unsound to identify them because they can be distinguished: projections of pairs can be evaluated even if a component of the pair diverges.

To obtain soundness, we do not change semantic subtyping essentially: doing so would require modification of many previous results, including those related to the algorithm to check subtyping. Instead, we add a new type $\bot$ to represent divergence: this allows us to distinguish terminating and possibly diverging expressions at the type level. We modify the typing rules to track divergence, with a very coarse approximation (they treat every expression that requires any evaluation as possibly diverging).

We describe this type system for an explicitly typed λ-calculus closely based on the language considered by Frisch, Castagna, and Benzaken (2008), but with a call-by-need semantics. The choice of call-by-need is motivated by the presence of union types, which would require a complex union disjunction rule to have subject reduction hold (and would make subject reduction fail outright if the language included non-deterministic constructs). We prove that the type system is sound. The subtyping relation (mostly) maintains the behaviour of call-by-value semantic subtyping, allowing, for instance, the same use of intersection types to type overloaded functions.

1.3 Relationship with published or submitted work

The contents of Part I originate from the work on typing polymorphic variants presented at *ICFP 2016* (Castagna, Petrucciani, and Nguyễn, 2016).[5] However, they have been greatly reworked. In particular, the soundness proof for the type system in Chapter 3 is new: the system of the cited paper did not include the rule $[\mathrm{T}_\wedge]$ and therefore admitted a simpler proof. Moreover, type inference has been overhauled to correct a problem in the original proof of completeness and to improve the description of constraint solving. The material in Chapter 5 is also new.

The material in Part II has been presented at *POPL 2019*. It is joint work with Giuseppe Castagna, Victor Lanvin, and Jeremy Siek. In this presentation, I concentrate on declarative typing and type inference, which are the parts of the paper on which I have worked more directly, and which are closer to the rest of the thesis. The operational semantics of the cast language is discussed only cursorily (its full definition is in Appendix B). The difficulties we met in defining this semantics are outside the main scope of this thesis: in particular, the semantics is driven by type information, whereas in the rest of thesis we concentrate on designing type systems for semantics that do not depend on static types.

The material in Part III is currently under submission for publication in the post-proceedings of *TYPES 2018*. It is joint work with Giuseppe Castagna, Davide Ancona, and Elena Zucca.

The results in Parts II and III have both been presented at *TYPES 2018*.

5 A prototype implementation of the type inference algorithm described in the cited work is available at http://www.cduce.org/ocaml.

1.4 Outline

Chapter 2 introduces the semantic subtyping approach, recapitulating the previous work that constitutes the starting point for this thesis. We define set-theoretic types and the subtyping relation on them, and we prove several properties of subtyping.

The greater portion of the thesis is structured in three parts.

PART I We study how to use polymorphic set-theoretic types for implicitly typed languages with type inference.

Chapter 3 We describe the syntax and semantics of an implicitly typed λ-calculus. We define a type system for it and prove it sound.

Chapter 4 We show how to perform type inference for the system of the previous chapter, and prove results of soundness and completeness.

Chapter 5 We describe how to make type inference more precise when programs contain some type annotations.

Chapter 6 We sketch how to extend the language with additional features including pattern matching, polymorphic variant types, and records.

Chapter 7 We discuss the results we have obtained in this part, their relationship with previous work, and possible directions for future research.

PART II We describe our approach to gradual typing and how to combine gradual typing with polymorphic set-theoretic types.

Chapter 8 We motivate the work by describing the kind of type discipline which the combination of gradual typing, polymorphic set-theoretic types, and type inference can provide. Then, we introduce our approach and methods.

Chapter 9 We describe a gradual type system for an ML-like language with let-polymorphism but no subtyping. We describe the source language and its type system, the cast language with its type system and the compilation procedure, and the type inference algorithm.

Chapter 10 We show how to extend our approach to set-theoretic types, notably by defining a subtyping relation on gradual set-theoretic types.

Chapter 11 We conclude by discussing our results, their relation to previous work, and some objectives to work towards in the future.

PART III We show how to adapt set-theoretic type systems to languages with non-strict semantics.

Chapter 12 We explain why standard systems with semantic subtyping are unsound for non-strict languages, and we introduce our approach to achieve soundness.

Chapter 13 We describe our results: we define a call-by-need λ-calculus and a type system for it featuring set-theoretic types; we prove soundness of the type system.

Chapter 14 We discuss the results of the previous chapter and present directions for future work. In particular, we show how we could work towards an alternative interpretation of types.

Finally, in Chapter 15, we summarize the results in the thesis and the main directions for future work.

Two appendices complete the thesis. Appendix A includes all the proofs omitted from the main text. We leave many of the proofs of Part I in the text because they illustrate the techniques we use; in contrast, in Parts II and III we omit most of them since they usually rely on similar techniques. Appendix B defines the operational semantics of the cast calculi in Part II, which we do not give in the main text because we concentrate on typing.

1.5 Notational conventions

POWERSET: Given a set A, we denote by $\mathcal{P}(A)$ the *powerset* of A (i.e., the set of all sets A' such that $A' \subseteq A$). We denote by $\mathcal{P}_{\text{fin}}(A)$ the *finite powerset* of A (i.e., the set of all *finite* sets A' such that $A' \subseteq A$).

VECTORS: We write vectors (or tuples) using a superscript arrow ($\vec{\cdot}$). For instance, we write vectors of types t as $\vec{t}$. When we write a vector of type variables ($\vec{\alpha}$, $\vec{\beta}$, $\vec{\gamma}$ and, in Part II, also $\vec{X}$, $\vec{Y}$, $\vec{A}$) we always assume that they are all distinct. Therefore, we often convert implicitly between vectors and sets of type variables. We sometimes use an overline to indicate sets: for instance, $\overline{\alpha}$ for sets of α type variables.

DISJOINTNESS: We use $\sharp$ to indicate disjointness of sets of type variables: when A and B are sets of type variables, we write $A \sharp B$ for $A \cap B = \emptyset$.

We use this notation also with other terms in place of sets of type variables; in this case we refer to the type variables in the term. For instance, this term can be a type, a type scheme (i.e., a type with some quantified variables), a type environment (i.e., a mapping from expression variables to type schemes), or a type substitution (i.e., a mapping from type variables to types). When we write a type, a type scheme, or a type environment, we take the set of the type variables in it (written $\mathsf{var}(\cdot)$ elsewhere, but left implicit when using $\sharp$). When we write a type substitution, we refer to both the variables in its domain and those in the types in its range ($\mathsf{dom}(\cdot) \cup \mathsf{var}(\cdot)$, where $\mathsf{var}(\cdot)$ denotes the variables appearing in the types in the range). When more than one term appears on one side of the symbol $\sharp$, we take the union of the sets.

For instance: we write $\alpha \sharp \vec{\alpha}, \vec{\beta}$ to mean $\{\alpha\} \cap (\vec{\alpha} \cup \vec{\beta}) = \emptyset$ (treating vectors of variables as sets by the convention above); we write $\alpha \sharp t$ to mean that α does not occur in t; we write $\vec{\alpha} \sharp \sigma$ (when σ is a type substitution) to mean that the variables in $\vec{\alpha}$ are not instantiated and are not introduced by σ.

In Part II, we distinguish two kinds of variables in types, type variables and frame variables: we use this notation for both.

STATEMENTS AND PROOFS: We sometimes write statements in a condensed form using braces for conjunction and implicitly quantifying universally over all variables that are not quantified explicitly. For example, we write

$$\left.\begin{matrix} P_1(X) \\ P_2(X,Y) \end{matrix}\right\} \implies \exists Z.\, \begin{cases} Q_1(X,Z) \\ Q_2(X,Y,Z) \end{cases}$$

(where the P_i and the Q_i are already defined predicates) to mean

$$\forall X, Y.\, \Big(\big(P_1(X) \land P_2(X,Y)\big) \implies \exists Z.\, \big(Q_1(X,Z) \land Q_2(X,Y,Z)\big) \Big)\,.$$

In proofs, we sometimes use circled letters (Ⓐ, Ⓑ, Ⓒ, …) to refer to parts of the hypotheses or to intermediate results in a proof.

We write IH to abbreviate "induction hypothesis" in the proofs.

2 Background

This chapter introduces the background needed for the rest of the work: the theory of semantic subtyping for polymorphic set-theoretic types. Most of the definitions and results presented here come from the work of Frisch, Castagna, and Benzaken (2008), Castagna and Xu (2011), and Gesbert, Genevès, and Layaïda (2015).

In the three parts of the thesis, we will rely extensively on these results. In Part I, we use them directly. In Parts II and III, we will make some adaptations and develop more results, but most of the material here will only need slight modifications.

CHAPTER OUTLINE:

Section 2.1 We give a general introduction to semantic subtyping.

Section 2.2 We define the language of types that we will use.

Section 2.3 We define the subtyping relation.

Section 2.4 We study some properties of subtyping. To do so, we also introduce an alternative definition of subtyping and prove it equivalent to that of Section 2.3.

2.1 Introduction

In the previous chapter, we have given examples of why union, intersection, and negation types – that we collectively refer to as *set-theoretic types* – are useful to type programming languages. To add them to a type system, though, we should define a suitable notion of subtyping on them.

Arguably, when reasoning on types in a programming language, it is intuitive to view a type as representing a set of values of the language. Then, set-theoretic types have a natural interpretation as the corresponding set-theoretic notions (negation being complementation with respect to the set of all values). Following this view, we want subtyping to satisfy natural distribution laws. For example, it should treat $(t_1 \times t) \vee (t_2 \times t)$ and $(t_1 \vee t_2) \times t$ as equivalent, since they correspond to the same set of pair values. Likewise, $(t \to t_1) \wedge (t \to t_2)$ and $t \to (t_1 \wedge t_2)$ should be equivalent, since they identify the same set of functions.

Subtyping is often defined by axiomatizing it in a system of inference rules. However, a system would need many rules to capture the properties we want. As a result, it could be complex to work with and lack intelligibility. An alternative way to define subtyping is to build a model of the language and interpret types as subsets of the model; then, subtyping is defined as inclusion

between the sets denoted by the types. The difficulty is to find a suitable denotational model of the language.

Semantic subtyping as presented here takes a middle ground between these two possibilities. Subtyping is defined using a set-theoretic interpretation of types and not by axiomatizing it in a deduction system. However, this interpretation is not part of a full-fledged denotational model of the language: it is only used to define subtyping. It is, indeed, an interpretation of *types* and not necessarily connected to an interpretation of the terms of the language. In principle, we can interpret types into sets in any way that induces a subtyping relation with the properties we want. Of course, the interpretation will have to be somehow connected to the actual meaning of types in the language, if we want subtyping to behave correctly (e.g., to ensure type soundness for the type system that uses it). A better correspondence could yield a more precise subtyping relation (one that accepts more programs, while remaining sound). However, it is not necessary to be able to prove a formal connection between the interpretation of types and any semantic notion of the language.

This is the essence of the semantic subtyping approach. To define subtyping, we fix some set Domain as our *domain of interpretation* of types. Domain should represent, at least in some intuitive sense, the set of values in the language. Then, we define an interpretation function $[\![\cdot]\!] \colon \mathsf{Type} \to \mathcal{P}(\mathsf{Domain})$ which maps types into subsets of Domain. Finally, we define the subtyping relation $\leq$ as $t_1 \leq t_2 \stackrel{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!]$.

Types will include some type constructors and the set-theoretic type connectives (union $\vee$, intersection $\wedge$, and negation $\neg$) plus the bottom type $\mathbb{0}$ and the top type $\mathbb{1}$:

$$t ::= \cdots \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \,,$$

leaving the type constructors unspecified for now. We will allow types to be recursive, not by using explicit binders but by interpreting the grammar coinductively (with restrictions of regularity and contractivity). We want the interpretation to satisfy

$$\begin{aligned} [\![t_1 \vee t_2]\!] &= [\![t_1]\!] \cup [\![t_2]\!] & [\![\mathbb{0}]\!] &= \varnothing \\ [\![t_1 \wedge t_2]\!] &= [\![t_1]\!] \cap [\![t_2]\!] & [\![\mathbb{1}]\!] &= \mathsf{Domain} \\ [\![\neg t]\!] &= \mathsf{Domain} \setminus [\![t]\!] \end{aligned}$$

to ensure that subtyping indeed treats set-theoretic types set-theoretically. (Actually, this interpretation means that we can treat some forms as derived: in our formalization, we define $t_1 \wedge t_2 \stackrel{\text{def}}{=} \neg(\neg t_1 \vee \neg t_2)$ and $\mathbb{1} \stackrel{\text{def}}{=} \neg\mathbb{0}$). Since these interpretations are fixed, defining $[\![\cdot]\!]$ consists essentially in defining the interpretation of type constructors: we will discuss this below.

Once we have defined $[\![\cdot]\!]$, we can use it to define subtyping as set containment. To use the subtyping relation in a type system we must do more, of course. We must prove some properties of subtyping, at least those that we need to show type soundness for the system. To implement the system in a practical type checker, we must find an algorithm to check subtyping. An advantage of this approach is that many properties are simple to derive

(transitivity, for instance, holds trivially). To find an algorithm, we can rely on set-theoretic calculations on the interpretation of types. Note in passing that, using the notation $t_1 \setminus t_2$ for $t_1 \wedge \neg t_2$, we have $[\![t_1]\!] \subseteq [\![t_2]\!]$ if and only if $[\![t_1 \setminus t_2]\!] = \varnothing$. Therefore, checking subtyping is equivalent to checking emptiness of types.

We do not discuss here the algorithmic problem of deciding subtyping. Rather, we continue the introduction by explaining how the interpretation of types is defined in previous work.

2.1.1 Semantic subtyping for first-order languages

The starting point for this approach was the work on the XML processing language XDuce (Hosoya, Vouillon, and Pierce, 2005). The authors show that subtyping can be defined semantically without building a full model of the language: a model of the types is enough, and it can be obtained by interpreting types as sets of values of the language.

The language they study is monomorphic and first-order. Rephrasing this outside the context of XML, let us take a language which does not include higher-order functions. Values are constants or pairs of values: $v ::= c \mid (v, v)$. Types include base types b for constants, product types $t_1 \times t_2$, and set-theoretic types; they can also be recursive. In this setting, we can interpret a type as the set of values of that type in the language: we interpret each base type into the appropriate set of constants – e.g., $[\![\mathsf{Bool}]\!] = \{\mathsf{true}, \mathsf{false}\}$ – and we define $[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$. We use $[\![\cdot]\!]$ to define the subtyping relation as set inclusion; then, the relation can be used in a type system for the language.

Hosoya, Vouillon, and Pierce study this setting, noting that the obtained subtyping relation reduces to the inclusion problem of tree automata; they develop a practical algorithm to decide it.

2.1.2 Adding arrow types

Frisch, Castagna, and Benzaken (2008) extend the previous approach to higher-order languages where types include arrow types $t_1 \to t_2$. This requires a major change in the approach. We can no longer interpret types directly as sets of values, because of a problem of circularity. Assume that we want to interpret a type as the set of the values of that type in the language. Then we should define $[\![t_1 \to t_2]\!] = \{\, \lambda x.\, e \mid \vdash \lambda x.\, e : t_1 \to t_2 \,\}$: but the definition of the typing relation $\vdash e : t$ relies itself on the definition of subtyping, which is what we are trying to define using the interpretation of types. If values are only constants or pairs, the approach works because the typing relation restricted to values is straightforward. The typing of functions, instead, is more difficult because it involves the typing of function bodies, which are arbitrary expressions.

So, we cannot have λ-abstractions in `Domain` because, at this stage, we do not yet know how to associate types to them. But, as we have said, we do not need `Domain` to be exactly the set of syntactic values. Indeed, we do not care at all about what the elements in a set $[\![t]\!]$ are: we just care about how those

in two sets $[\![t_1]\!]$ and $[\![t_2]\!]$ are related, because we use the interpretation only to define subtyping as set inclusion.

We can try to see functions *extensionally*, as graphs. Then, we could interpret arrow types like this:

$$[\![t_1 \to t_2]\!] = \{\, R \subseteq \mathsf{Domain}^2 \mid \forall (d, d') \in R.\ d \in [\![t_1]\!] \implies d' \in [\![t_2]\!] \,\}\ .$$

Intuitively, a relation $R = \{\, (d_i, d_i') \mid i \in I \,\}$ represents a function that maps each d_i to the corresponding d_i' and diverges on elements that do not appear in its domain $\{\, d_i \mid i \in I \,\}$. The relations in $[\![t_1 \to t_2]\!]$ must map elements of $[\![t_1]\!]$ to elements of $[\![t_2]\!]$, but they are not required to map all of them (since they can be partial), and they can also map elements outside $[\![t_1]\!]$ without restrictions. We do not demand functionality – that is, we allow a relation to contain two pairs (d, d_1) and (d, d_2) with $d_1 \neq d_2$ – because we assume that the functions in our language could be non-deterministic.

How should we define Domain to use this interpretation? The domain should include constants, pairs, and relations: it should satisfy the equation

$$\mathsf{Domain} = \mathsf{Const} \uplus \mathsf{Domain}^2 \uplus \mathcal{P}(\mathsf{Domain}^2)\ ,$$

where Const is the set of language constants, $\uplus$ denotes the disjoint union, and $\mathcal{P}(\cdot)$ the powerset. But no such set can exist: the cardinality of $\mathcal{P}(\mathsf{Domain}^2)$ is always strictly greater than that of Domain.

To solve this difficulty, Frisch, Castagna, and Benzaken propose to use *finite* relations only. Considering the restriction of the powerset to finite sets, the equation above becomes satisfiable: we can define domain elements as the finite terms d given by $d ::= c \mid (d, d) \mid \{(d, d), \ldots, (d, d)\}$ (where $c \in \mathsf{Const}$). Of course, finite relations are not a faithful representation of the functions in languages in which, presumably, functions can be defined on an infinite domain. For example, the set $[\![\mathsf{Int} \to \mathsf{Int}]\!]$ no longer contains the successor function on integers; however, it contains all its finite approximations. This restriction is not a problem for subtyping, because it does not affect set inclusion: note that $\mathcal{P}(A) \subseteq \mathcal{P}(B) \iff A \subseteq B \iff \mathcal{P}_{\text{fin}}(A) \subseteq \mathcal{P}_{\text{fin}}(B)$ holds for any two sets A and B (where $\mathcal{P}_{\text{fin}}$ denotes the restriction of the powerset to finite sets). Frisch, Castagna, and Benzaken use their notions of *extensional interpretation* and of *model* to argue more precisely that using finite relations does not compromise subtyping.

Taking the restriction to finite sets, we can indeed define Domain as we have said and define the interpretation as

$$\begin{aligned}
[\![\mathsf{Bool}]\!] &= \{\mathsf{true}, \mathsf{false}\} \quad \text{(and similarly for other base types)} \\
[\![t_1 \times t_2]\!] &= [\![t_1]\!] \times [\![t_2]\!] \\
[\![t_1 \to t_2]\!] &= \{\, R \in \mathcal{P}_{\text{fin}}(\mathsf{Domain}^2) \mid \forall (d, d') \in R.\ d \in [\![t_1]\!] \implies d' \in [\![t_2]\!] \,\}
\end{aligned}$$

plus the already given definitions on type connectives, $\mathbb{0}$, and $\mathbb{1}$.

There is one further problem. With this definition, we have $t_1 \to t_2 \leq \mathbb{1} \to \mathbb{1}$ for any two types t_1 and t_2. This means that any λ-abstraction that is well typed (with some type $t_1 \to t_2$) can be applied to any argument whatsoever (by subsuming $t_1 \to t_2$ to $\mathbb{1} \to \mathbb{1}$). This is unsound in a language with constants:

for instance, $(\lambda x.\, x\ 3)$ true has type $\mathbb{1}$, but it reduces to the stuck term true 3. The solution is to allow a new element Ω, representing a runtime type error, to occur in the second components of pairs in relations, while not being in Domain. That is, we define Domain as the set of finite terms d given by $d ::= c \mid (d, d) \mid \{(d, d_\Omega), \ldots, (d, d_\Omega)\}$, where $d_\Omega ::= d \mid \Omega$. Intuitively, a pair (d, Ω) in a relation means that the function crashes on the input d. With this change, $\mathbb{1} \to \mathbb{1}$ is no longer a supertype of all arrows, but only of those of the form $\mathbb{1} \to t$. For example, $\mathsf{Int} \to \mathsf{Int} \leq \mathbb{1} \to \mathbb{1}$ no longer holds, because the relations in $\mathsf{Int} \to \mathsf{Int}$ can contain the pair (true, Ω), since $\mathsf{true} \notin [\![\mathsf{Int}]\!]$, while the relations in $\mathbb{1} \to \mathbb{1}$ cannot.

This change allows us to define a subtyping relation which has the correct properties to be used in a sound type system. It is also decidable: Frisch, Castagna, and Benzaken (2008) describe an algorithm, and there are several optimizations to it used in the implementation of ℂDuce, which relies on this subtyping relation.

An important result of Frisch, Castagna, and Benzaken (2008) is that – for their interpretation, language, and type system – they show a close correspondence between the interpretation of types and the sets of values in a type. As we have said, types cannot be directly interpreted as sets of values because of a problem of circularity. However, once we have an interpretation $[\![\cdot]\!]$, defined as above, we can define the subtyping relation and, using it, the type system. Then, we can define the interpretation we wanted at first: $[\![t]\!]^{\mathcal{V}} \stackrel{\text{def}}{=} \{\, v \mid \vdash v : t \,\}$. Frisch, Castagna, and Benzaken prove $\forall t_1, t_2.\ [\![t_1]\!] \subseteq [\![t_2]\!] \iff [\![t_1]\!]^{\mathcal{V}} \subseteq [\![t_2]\!]^{\mathcal{V}}$. Showing the result above implies that, once the type system is defined, we can indeed reason on subtyping by reasoning on inclusion between sets of values.

This result is useful in practice: when type checking fails because a subtyping judgment $t_1 \leq t_2$ does not hold, we know that there exists a value v such that $\vdash v : t_1$ holds while $\vdash v : t_2$ does not. This value v can be shown as a witness to the unsoundness of the program while reporting the error.[1] Moreover, at a more foundational level, the result nicely formalizes the intuition that types statically approximate computations: a type t corresponds to the set of all possible values of expressions of type t.

This is a very brief introduction to the work of Frisch, Castagna, and Benzaken (2008). In particular, we have described how to find a specific interpretation that induces a suitable subtyping relation. The authors instead identify more general properties that an interpretation should satisfy, using their notions of *extensional interpretation* and of *model* to capture these properties and to argue that the restriction to finite relations does not pose problems for subtyping. We refer the interested reader to their work for more details on this.

1 In case of a type error, the ℂDuce compiler shows to the programmer a default value for the type $t_1 \wedge \neg t_2$. Some heuristics are used to build a value in which only the part relevant to the error is detailed.

2.1.3 Adding type variables

The next step is to allow types to contain type variables. We need to do so to use the subtyping relation for type systems with prenex parametric polymorphism. (Types with quantifiers that bind type variables, to use for first-class polymorphism, have not been studied yet in semantic subtyping.)

In syntactic subtyping, we would normally expect a type variable α to be treated similarly to an abstract type: it should be unrelated to any other type except by trivial rules (e.g., $\alpha \leq \mathbb{1}$ if $\mathbb{1}$ is the top type) and by reflexivity (e.g., $\alpha \leq \alpha \vee t$ holds because $\alpha \leq \alpha$). To achieve soundness, we should ensure that if $t_1 \leq t_2$ holds, then $t_1\sigma \leq t_2\sigma$ holds for any type substitution σ.

In semantic subtyping, we can proceed as follows. We add type variables α, drawn from a set TVar, to the grammar of types. We parameterize the interpretation of types making it depend on an *assignment* which gives meaning to the type variables. An assignment is a function $\eta \colon \mathsf{TVar} \to \mathcal{P}(\mathsf{Domain})$ which maps type variables to subsets of Domain. The interpretation is now a function $[\![\cdot]\!] \colon \mathsf{Type} \to \big(\mathsf{TVar} \to \mathcal{P}(\mathsf{Domain})\big) \to \mathcal{P}(\mathsf{Domain})$. We define $[\![\alpha]\!]\eta$ as $\eta(\alpha)$. Ground types, instead, have the same interpretation in every η: for instance, $[\![\mathsf{Bool}]\!]\eta = \{\mathsf{true}, \mathsf{false}\}$. Subtyping is defined as

$$t_1 \leq t_2 \stackrel{\text{def}}{\iff} \forall \eta \colon \mathsf{TVar} \to \mathcal{P}(\mathsf{Domain}).\ [\![t_1]\!]\eta \subseteq [\![t_2]\!]\eta\ .$$

This ensures that $t_1 \leq t_2$ implies $t_1\sigma \leq t_2\sigma$ for every σ.

Hosoya, Frisch, and Castagna (2009) and Castagna and Xu (2011) discuss two problems of this approach. One is algorithmic: the relation is not known to be decidable, and it is conjectured to be NEXPTIME-complete if it is, with no practical algorithm known. In particular, it seems difficult to reuse the algorithms for monomorphic subtyping to decide it.

The other problem is that, arguably, the behaviour of subtyping does not match one's intuitive expectations, and it does not match the behaviour of syntactic subtyping. The problematic example of Castagna and Xu (2011) is

$$t \times \alpha \leq (t \times \neg t) \vee (\alpha \times t)$$

where t is some ground type (so its interpretation is the same for every η). One could expect this judgment not to hold, because the type variable α appears in unrelated positions (in the second component on the left, in the first one on the right). According to this definition, instead, the judgment holds if and only if t is a singleton type (that is, if $[\![t]\!]\eta$ is a singleton for every η). The judgment is equivalent to

$$\forall \eta \colon \mathsf{TVar} \to \mathcal{P}(\mathsf{Domain}).\ [\![t]\!]\eta \times \eta(\alpha) \subseteq \big([\![t]\!]\eta \times (\mathsf{Domain} \setminus [\![t]\!]\eta)\big) \cup \big(\eta(\alpha) \times [\![t]\!]\eta\big)\,.$$

If, for some d, we have $\forall \eta.\ [\![t]\!]\eta = \{d\}$, then the judgment is equivalent to

$$\forall \eta \colon \mathsf{TVar} \to \mathcal{P}(\mathsf{Domain}).\ \{d\} \times \eta(\alpha) \subseteq \big(\{d\} \times (\mathsf{Domain} \setminus \{d\})\big) \cup \big(\eta(\alpha) \times \{d\}\big)\,,$$

which is true because, for every η, either $d \in \eta(\alpha)$ or $\eta(\alpha) \subseteq \mathsf{Domain} \setminus \{d\}$. In contrast, is t is not a singleton, taking $\eta(\alpha)$ to be a proper subset of $[\![t]\!]\eta$ disproves the containment.

Castagna and Xu (2011) argue that we should only consider interpretations where judgments such as the above do not hold. This should ensure that subtyping on type variables behaves closer to the expectations for parametric polymorphism, so that a type variable can occur on the right-hand side of a subtyping judgment only if it occurs in a corresponding position on the left-hand side.

Castagna and Xu propose *convexity* as a general property of interpretations of types that avoid this problematic behaviour. An interpretation $[\![\cdot]\!]$ is *convex* if, for every finite set of types $\{t_1, \ldots, t_n\}$, it satisfies

$$\forall \eta.\ \big([\![t_1]\!]\eta = \varnothing \text{ or } \ldots \text{ or } [\![t_n]\!]\eta = \varnothing\big) \\ \iff (\forall \eta.\ [\![t_1]\!]\eta = \varnothing) \text{ or } \ldots \text{ or } (\forall \eta.\ [\![t_n]\!]\eta = \varnothing)\,.$$

An interpretation where there are ground singleton types (i.e., types t such that $\exists d.\, \forall \eta.\, [\![t]\!]\eta = \{d\}$) is not convex because $\forall \eta.\, ([\![t \wedge \alpha]\!] = \varnothing \text{ or } [\![t \wedge \neg\alpha]\!] = \varnothing)$ is true if and only if t is a singleton, while $\forall \eta.\, [\![t \wedge \alpha]\!] = \varnothing$ and $\forall \eta.\, [\![t \wedge \neg\alpha]\!] = \varnothing$ never hold.

To achieve convexity, Castagna and Xu suggest to interpret all ground types into infinite sets. This loses in part the intuitive set-theoretic meaning of types: for example, Bool cannot be interpreted as $\{\mathsf{true}, \mathsf{false}\}$. However, it seems sufficient to ensure convexity and this, in turn, to have a subtyping relation that avoids problematic judgments such as that shown above and that is easier to compute by extending the previous work on monomorphic semantic subtyping.

We can define Domain as the set of the finite terms d generated by $d ::= c^L \mid (d, d)^L \mid \{(d, d_\Omega), \ldots, (d, d_\Omega)\}^L$, where L is a label drawn from some countable set Label. The interpretation of base types contain constants with every possible labelling, and likewise for products and arrow: for instance,

$$[\![\mathsf{Bool}]\!]\eta = \{\, c^L \mid c \in \{\mathsf{true}, \mathsf{false}\}, L \in \mathsf{Label} \,\} \\ [\![t_1 \times t_2]\!]\eta = \{\, (d_1, d_2)^L \mid (d_1, d_2) \in [\![t_1]\!]\eta \times [\![t_2]\!]\eta, L \in \mathsf{Label} \,\}\,.$$

Gesbert, Genevès, and Layaïda (2011, 2015) study polymorphic semantic subtyping to give an algorithm to decide it using a logical solver. They adopt the idea of interpreting ground types into infinite sets. They also show how we can avoid using quantification and give a fixed interpretation to type variables. Indeed, assume that labels are finite sets of type variables, that is, $\mathsf{Label} = \mathcal{P}_{\text{fin}}(\mathsf{TVar})$. Then, we can define subtyping in two different ways:

- by defining subtyping using quantification, having $[\![\cdot]\!]$ depend on an assignment η, and having $[\![\alpha]\!]\eta = \eta(\alpha)$;
- by defining subtyping as set containment of the interpretations, with $[\![\cdot]\!]$ mapping types to sets of values (without using an assignment), defining $[\![\alpha]\!] = \{\, d \in \mathsf{Domain} \mid \alpha \in \mathsf{tags}(d) \,\}$, where $\mathsf{tags}(d)$ denotes the top-level label of d.

It can be shown that the two definitions produce the same relation (as we will see in Section 2.4.1). The latter interpretation is arguably less intuitive, but it is very convenient to work with because it interprets types directly as sets.

In the rest of the chapter, we will define types and subtyping formally using the approach of Gesbert, Genevès, and Layaïda (2015) to interpret type variables without quantification; then, we will introduce the interpretation with quantification and prove the equivalence. Like Gesbert, Genevès, and Layaïda, we fix for simplicity a specific interpretation of types; in contrast, Castagna and Xu study more in general the properties a suitable interpretation should satisfy, but there are some inconsistencies in their technical development of this more general theory.

2.2 Types

Types should include *type variables* and *base types* (which are the types of language constants). Therefore, we assume that there exist three sets TVar, Const, and Base: for these, we use the metavariables listed below.

$\mathsf{TVar} \ni \alpha, \beta, \gamma$	type variables
$\mathsf{Const} \ni c$	language constants
$\mathsf{Base} \ni b$	base types

We assume that TVar is countably infinite and disjoint from Base. We also assume that there exist two functions

$$b_{(\cdot)} \colon \mathsf{Const} \to \mathsf{Base} \qquad \mathbb{B}(\cdot) \colon \mathsf{Base} \to \mathcal{P}(\mathsf{Const})$$

which map constants to base types and base types to sets of constants. Given a constant c, the base type b_c is its most precise type. Given a base type b, the constants in $\mathbb{B}(b)$ are all the language constants that can be given type b.

We assume that Base includes *singleton types* for each constant and therefore that $\mathbb{B}(b_c) = \{c\}$ for every $c \in \mathsf{Const}$. We also assume that there exists a base type $\mathbb{1}_\mathsf{B} \in \mathsf{Base}$ such that $\mathbb{B}(\mathbb{1}_\mathsf{B}) = \mathsf{Const}$. Singleton types and $\mathbb{1}_\mathsf{B}$ are not strictly necessary in the theory, but they simplify parts of the technical development. Singleton types are also useful for typing,[2] and, in our system, to be able to represent pattern matching using typecase constructs.

EXAMPLE: As an example, we could take the following definitions

$$\mathsf{Const} = \{\mathsf{true}, \mathsf{false}\} \cup \mathbb{Z} \qquad \mathsf{Base} = \mathsf{Const} \cup \{\mathsf{Bool}, \mathsf{Int}, \mathbb{1}_\mathsf{B}\}$$

$$b_c = c \qquad \mathbb{B}(b) = \begin{cases} \{c\} & \text{if } b = c \\ \{\mathsf{true}, \mathsf{false}\} & \text{if } b = \mathsf{Bool} \\ \mathbb{Z} & \text{if } b = \mathsf{Int} \\ \mathsf{Const} & \text{if } b = \mathbb{1}_\mathsf{B} \end{cases}$$

(we represent the singleton type of each constant by the constant itself). □

Assuming any suitable definition of TVar and Base, we define types as follows.

2 They are used, for instance, in Typed Racket, TypeScript, and Flow to be able to type check some idioms of dynamic programming.

2.1 DEFINITION (Types): The set Type of *types* is the set of terms t generated coinductively by the following grammar

$$
\begin{array}{lll}
t ::= \alpha & & \text{type variable} \\
\mid b & & \text{base} \\
\mid t \times t & & \text{product} \\
\mid t \to t & & \text{arrow} \\
\mid t \vee t & & \text{union} \\
\mid \neg t & & \text{negation} \\
\mid \mathbb{0} & & \text{empty}
\end{array}
$$

(where α ranges over TVar and b over Base) and that satisfy the following two conditions:

(regularity) the term has finitely many distinct subterms;

(contractivity) every infinite path in the term contains infinitely many occurrences of the $\times$ or $\to$ constructors. □

The only primitive set-theoretic connectives in types are union and negation, but we introduce the following abbreviations.

$$
\begin{array}{rcll}
t_1 \wedge t_2 & \stackrel{\text{def}}{=} & \neg(\neg t_1 \vee \neg t_2) & \text{intersection} \\
t_1 \setminus t_2 & \stackrel{\text{def}}{=} & t_1 \wedge (\neg t_2) & \text{difference} \\
\mathbb{1} & \stackrel{\text{def}}{=} & \neg\mathbb{0} & \text{any}
\end{array}
$$

We refer to b, $\times$ and $\to$ as *type constructors* and to $\vee$, $\neg$, $\wedge$, and $\setminus$ as *type connectives*.

Note that types are defined *coinductively* rather than inductively, so they can be infinite trees (subject to the conditions of regularity and contractivity). This is a way to have equi-recursive types, alternative (but equivalent) to using explicit binders for recursion.

The purpose of the regularity condition imposed on types is simply to ensure the decidability of the subtyping relation. Contractivity, instead, is fundamental to exclude terms which do not have a meaningful interpretation as types or sets of values: for instance, the trees satisfying the equations $t = t \vee t$ (which gives no information on which values are in it) or $t = \neg t$ (which cannot represent any set of values).

Contractivity also ensures that the binary relation $\triangleright \subseteq \mathsf{Type}^2$ defined by $t_1 \vee t_2 \triangleright t_i$ and $\neg t \triangleright t$ is Noetherian (that is, strongly normalizing). This gives an induction principle on types that we will use without explicit reference to the relation $\triangleright$. This induction principle allows us to apply the induction hypothesis below type connectives (union and negation), but not below type constructors. As a consequence of contractivity, types cannot contain infinite unions or intersections.

Given a type t, we write $\mathsf{var}(t)$ for the set of type variables occurring in it.

The following equalities hold.

$$\begin{aligned} \mathsf{var}(\alpha) &= \{\alpha\} & \mathsf{var}(b) &= \varnothing \\ \mathsf{var}(t_1 \times t_2) &= \mathsf{var}(t_1) \cup \mathsf{var}(t_2) & \mathsf{var}(t_1 \to t_2) &= \mathsf{var}(t_1) \cup \mathsf{var}(t_2) \\ \mathsf{var}(t_1 \vee t_2) &= \mathsf{var}(t_1) \cup \mathsf{var}(t_2) & \mathsf{var}(\neg t) &= \mathsf{var}(t) \\ \mathsf{var}(\mathbb{0}) &= \varnothing \end{aligned}$$

Note that these equalities cannot be taken directly as an inductive definition of $\mathsf{var}(\cdot)$, because types are defined coinductively. We say that a type t is *ground* or *closed* if $\mathsf{var}(t) = \varnothing$.

2.2.1 Type substitutions

The description of polymorphic typing and type inference relies on type substitutions. We give a standard definition here.

2.2 DEFINITION: A *type substitution* σ is a mapping from type variables to types which is the identity everywhere except on a finite set of type variables, the *domain* $\mathsf{dom}(\sigma) = \{\, \alpha \in \mathsf{TVar} \mid \sigma(\alpha) \neq \alpha \,\}$ of the type substitution.

We write $t\sigma$ for the application of the type substitution σ to the type t. □

The application of a type substitution satisfies the following equalities.

$$\begin{aligned} \alpha\sigma &= \sigma(\alpha) & b\sigma &= b \\ (t_1 \times t_2)\sigma &= (t_1\sigma) \times (t_2\sigma) & (t_1 \to t_2)\sigma &= (t_1\sigma) \to (t_2\sigma) \\ (t_1 \vee t_2)\sigma &= (t_1\sigma) \vee (t_2\sigma) & (\neg t)\sigma &= \neg(t\sigma) \\ \mathbb{0}\sigma &= \mathbb{0} \end{aligned}$$

We extend the application of a type substitution to vectors of types by defining it pointwise. We use the notation $[\vec{t}/\vec{\alpha}]$ to denote the substitution σ such that $\mathsf{dom}(\sigma) \subseteq \vec{\alpha}$ and $\vec{\alpha}\sigma = \vec{t}$. We write $[\,]$ to denote the empty (or identity) substitution.

We define $\mathsf{var}(\sigma)$ to be the set $\bigcup_{\alpha \in \mathsf{dom}(\sigma)} \mathsf{var}(\alpha\sigma)$.

We write $\sigma_1 \cup \sigma_2$ for the union of disjoint type substitutions (i.e., substitutions with disjoint domains), defined by:

$$(\sigma_1 \cup \sigma_2)(\alpha) \stackrel{\text{def}}{=} \begin{cases} \sigma_1(\alpha) & \text{if } \alpha \in \mathsf{dom}(\sigma_1) \\ \sigma_2(\alpha) & \text{if } \alpha \in \mathsf{dom}(\sigma_2) \\ \alpha & \text{if } \alpha \notin \mathsf{dom}(\sigma_1 \cup \sigma_2) \end{cases}$$

We write $\sigma_2 \circ \sigma_1$ to denote the composition of type substitutions, defined by $(\sigma_2 \circ \sigma_1)(\alpha) \stackrel{\text{def}}{=} \alpha\sigma_1\sigma_2$. We use the notations $\sigma|_{\vec{\alpha}}$ and $\sigma|_{\setminus\vec{\alpha}}$ to denote restrictions of type substitutions. These are defined as follows.

$$(\sigma|_{\vec{\alpha}})(\alpha) \stackrel{\text{def}}{=} \begin{cases} \sigma(\alpha) & \text{if } \alpha \in \vec{\alpha} \\ \alpha & \text{otherwise} \end{cases} \qquad \sigma|_{\setminus\vec{\alpha}} \stackrel{\text{def}}{=} \sigma|_{\mathsf{dom}(\sigma)\setminus\vec{\alpha}}$$

2.3 Semantic subtyping

As anticipated, in semantic subtyping we interpret types as subsets of an *interpretation domain*. This domain corresponds intuitively to the sets of values of a language, but it represents functions as finite relations and uses a labelling technique to interpret type variables.

In the following definition, we pick a distinguished symbol Ω (which is not in Const) to represent type errors.

2.3 DEFINITION: The *interpretation domain* Domain is the set of finite terms d generated inductively by the following grammar

$$\begin{aligned} d &::= c^L \mid (d, d)^L \mid \{(d, d_\Omega), \ldots, (d, d_\Omega)\}^L \\ d_\Omega &::= d \mid \Omega \end{aligned}$$

where c ranges over Const and L over $\mathcal{P}_{\text{fin}}(\mathsf{TVar})$. □

We have described the reasoning behind this definition in Section 2.1. The use of finite sets of type variables, in particular, is meant to be able to interpret type variables without using quantification. For this purpose, we define a function tags on domain elements as

$$\mathsf{tags}(c^L) = \mathsf{tags}((d_1, d_2)^L) = \mathsf{tags}(\{\, (d^i, d^i_\Omega) \mid i \in I \,\}^L) = L\,,$$

that is, $\mathsf{tags}(d)$ is the outermost set of type variables labelling d.

Having defined the domain, we now define the interpretation of types, which is a function mapping each type to a subset of Domain. We want to define the interpretation $[\![t]\!]$ of a type t so that it satisfies the following equalities:

$$\begin{aligned} [\![\alpha]\!] &= \{\, d \mid \alpha \in \mathsf{tags}(d) \,\} \\ [\![b]\!] &= \{\, c^L \mid c \in \mathbb{B}(b) \,\} \\ [\![t_1 \times t_2]\!] &= \{\, (d_1, d_2)^L \mid d_1 \in [\![t_1]\!] \wedge d_2 \in [\![t_2]\!] \,\} \\ [\![t_1 \to t_2]\!] &= \left\{\, \{\, (d^i, d^i_\Omega) \mid i \in I \,\}^L \;\middle|\; \forall i \in I.\, d^i \in [\![t_1]\!] \implies d^i_\Omega \in [\![t_2]\!] \right\} \\ [\![t_1 \vee t_2]\!] &= [\![t_1]\!] \cup [\![t_2]\!] \\ [\![\neg t]\!] &= \mathsf{Domain} \setminus [\![t]\!] \\ [\![\mathbb{0}]\!] &= \varnothing \end{aligned}$$

If types were defined inductively, we could take these equalities as an inductive definition of $[\![\cdot]\!]$. Since they are defined coinductively, instead, we give the following definition, which satisfies these equalities and relies on the aforementioned induction principle on Type and on structural induction on Domain.

2.4 DEFINITION (Set-theoretic interpretation of types): We define a binary predicate $(d : t)$, where $d \in \mathsf{Domain}$ and $t \in \mathsf{Type}$, by induction on the pair (d, t)

ordered lexicographically. The predicate is defined as follows:

$$
\begin{aligned}
(d\colon \alpha) &= \alpha \in \mathsf{tags}(d) \\
(c^L\colon b) &= c \in \mathbb{B}(b) \\
((d_1, d_2)^L\colon t_1 \times t_2) &= (d_1\colon t_1) \wedge (d_2\colon t_2) \\
(\{\,(d^i, d^i_\Omega) \mid i \in I\,\}^L\colon t_1 \to t_2) &= \forall i \in I.\, (d^i\colon t_1) \implies (d^i_\Omega \neq \Omega) \wedge (d^i_\Omega\colon t_2) \\
(d\colon t_1 \vee t_2) &= (d\colon t_1) \vee (d\colon t_2) \\
(d\colon \neg t) &= \neg(d\colon t) \\
(d\colon t) &= \text{false} \qquad\qquad \text{otherwise}
\end{aligned}
$$

We define the *set-theoretic interpretation* $[\![\,\cdot\,]\!]\colon \mathsf{Type} \to \mathcal{P}(\mathsf{Domain})$ as

$$[\![t]\!] = \{\, d \in \mathsf{Domain} \mid (d\colon t)\,\}\,. \qquad \square$$

Finally, we define the subtyping preorder and its associated equivalence relation as follows.

2.5 DEFINITION (Subtyping): We define the *subtyping* relation $\leq$ and the *subtype equivalence* relation $\simeq$ on types as:

$$
\begin{aligned}
t_1 \leq t_2 &\stackrel{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!] \\
t_1 \simeq t_2 &\stackrel{\text{def}}{\iff} (t_1 \leq t_2) \wedge (t_2 \leq t_1)\,. \qquad \square
\end{aligned}
$$

2.4 Study of the subtyping relation

In this section, we study the properties of the subtyping relation and we prove the main results we need in the rest of the work. First, we give an alternative definition of subtyping and show that it is equivalent to that of Definition 2.5. Then, we use this alternative definition to prove that type substitutions preserve subtyping. Finally, we study subtyping judgments of a particular form ($t_1 \leq t_2$ where t_1 and t_2 are unions or intersections of arrow types) to derive properties that we need in the proofs of soundness.

Previous work (mainly Frisch, Castagna, and Benzaken, 2008; Castagna and Xu, 2011), contains other results which are used to describe subtyping algorithmically: for instance, they prove that types can always be put in a disjunctive normal form and study subtyping judgments on unions and intersections of product types. We do not treat these results here because we do not need them for most of the work, though we will introduce some of them in Part III.

2.4.1 Defining subtyping using quantification

The subtyping relation that we have just defined is simple to describe and to work with. Arguably, though, it would be more intuitive for subtyping on polymorphic types to be based on quantification, as introduced in Section 2.1.3. Gesbert, Genevès, and Layaïda (2015) give an alternative definition using quantification, for comparison with the system of Castagna and Xu (2011). We

describe this definition and report their proof of equivalence. Apart from its interest as a different characterization, this definition is useful to prove that subtyping is preserved by type substitutions.

In this definition, the interpretation domain is the same as before. However, the interpretation of a polymorphic type depends on the meaning we give to the type variables in it. This meaning is given by an *assignment*, which is a function $\eta\colon \mathsf{TVar} \to \mathcal{P}(\mathsf{Domain})$ that maps type variables to subsets of Domain.

We give alternative definitions of $(d\colon t)$, $[\![t]\!]$, and $\leq$ based on quantification (we mark them with a superscript $^{\mathsf{q}}$ to distinguish them from the previous definitions).

2.6 DEFINITION: We define a ternary predicate $(d :_\eta t)^{\mathsf{q}}$, where $d \in \mathsf{Domain}$, $t \in \mathsf{Type}$, and $\eta\colon \mathsf{TVar} \to \mathcal{P}(\mathsf{Domain})$, by induction on the pair (d, t) ordered lexicographically. The predicate is defined as follows:

$$\begin{aligned}
(d :_\eta \alpha)^{\mathsf{q}} &= d \in \eta(\alpha) \\
(c^L :_\eta b)^{\mathsf{q}} &= c \in \mathbb{B}(b) \\
((d_1, d_2)^L :_\eta t_1 \times t_2)^{\mathsf{q}} &= (d_1 :_\eta t_1)^{\mathsf{q}} \wedge (d_2 :_\eta t_2)^{\mathsf{q}} \\
(\{\, (d^i, d^i_\Omega) \mid i \in I \,\}^L :_\eta t_1 \to t_2)^{\mathsf{q}} &= \forall i{\in}I.\ (d^i :_\eta t_1)^{\mathsf{q}} \implies (d^i_\Omega \neq \Omega) \wedge (d^i_\Omega :_\eta t_2)^{\mathsf{q}} \\
(d :_\eta t_1 \vee t_2)^{\mathsf{q}} &= (d :_\eta t_1)^{\mathsf{q}} \vee (d :_\eta t_2)^{\mathsf{q}} \\
(d :_\eta \neg t)^{\mathsf{q}} &= \neg (d :_\eta t)^{\mathsf{q}} \\
(d :_\eta t)^{\mathsf{q}} &= \mathsf{false} \qquad \text{otherwise}
\end{aligned}$$

The interpretation $[\![t]\!]^{\mathsf{q}}\eta$ of a type t with respect to an assignment η is

$$[\![t]\!]^{\mathsf{q}}\eta \stackrel{\text{def}}{=} \{\, d \in \mathsf{Domain} \mid (d :_\eta t)^{\mathsf{q}} \,\} .$$

The quantification-based subtyping relation $\leq^{\mathsf{q}}$ is given by

$$t_1 \leq^{\mathsf{q}} t_2 \iff \forall \eta\colon \mathsf{TVar} \to \mathcal{P}(\mathsf{Domain}).\ [\![t_1]\!]^{\mathsf{q}}\eta \subseteq [\![t_2]\!]^{\mathsf{q}}\eta\ . \qquad \square$$

The two subtyping relations $\leq$ and $\leq^{\mathsf{q}}$ actually coincide. First, note that the interpretation function $[\![\cdot]\!]$ can be obtained from $[\![\cdot]\!]^{\mathsf{q}}$ by using the *canonical assignment* $\hat{\eta}\colon \mathsf{TVar} \to \mathcal{P}(\mathsf{Domain})$ defined by $\hat{\eta}(\alpha) = \{\, d \mid \alpha \in \mathsf{tags}(d) \,\}$.

2.7 LEMMA: For every type t, $[\![t]\!] = [\![t]\!]^{\mathsf{q}}\hat{\eta}$. $\square$

Proof: The statement is equivalent to

$$\forall t \in \mathsf{Type}.\ \forall d \in \mathsf{Domain}.\quad (d\colon t) \iff (d :_{\hat{\eta}} t)^{\mathsf{q}}$$

which can be shown by induction on the pair (d, t). $\square$

This already shows that $t_1 \leq^{\mathsf{q}} t_2$ implies $t_1 \leq t_2$: if $t_1 \leq^{\mathsf{q}} t_2$, then, for every $\eta\colon \mathsf{TVar} \to \mathcal{P}(\mathsf{Domain})$, we have $[\![t_1]\!]^{\mathsf{q}}\eta \subseteq [\![t_2]\!]^{\mathsf{q}}\eta$; therefore, $[\![t_1]\!]^{\mathsf{q}}\hat{\eta} \subseteq [\![t_2]\!]^{\mathsf{q}}\hat{\eta}$ and $[\![t_1]\!] \subseteq [\![t_2]\!]$. We use the following lemma (due to Gesbert, Genevès, and Layaïda, 2015) to prove the other implication.

2.8 LEMMA: Let V be a finite subset of TVar. Let $T = \{\, t \in \mathsf{Type} \mid \mathsf{var}(t) \subseteq V \,\}$. Then, for every $t \in T$,

$$[\![t]\!]^{\mathsf{q}}\hat{\eta} = \varnothing \implies \forall \eta\colon \mathsf{TVar} \to \mathcal{P}(\mathsf{Domain}).\ [\![t]\!]^{\mathsf{q}}\eta = \varnothing\,. \qquad \square$$

Proof: For an arbitrary V (and T defined from V), we prove the statement by contraposition, proving

$$\forall t \in T.\quad \left(\exists \eta \in \mathcal{P}(\mathsf{Domain})^{\mathsf{TVar}}.\ [\![t]\!]^{\mathsf{q}}\eta \neq \varnothing\right) \implies [\![t]\!]^{\mathsf{q}}\hat{\eta} \neq \varnothing\,.$$

by proving for arbitrary η the stronger statement

$$\forall t \in T.\, \forall d \in \mathsf{Domain}.\quad (d :_\eta t)^{\mathsf{q}} \iff (F_V^\eta(d) :_{\hat{\eta}} t)^{\mathsf{q}}\,,$$

where the function F_V^η is defined as follows:

$$F_V^\eta(d) = \begin{cases} c^{\ell_V^\eta(d)} & \text{if } d = c^L \\ (F_V^\eta(d_1), F_V^\eta(d_2))^{\ell_V^\eta(d)} & \text{if } d = (d_1, d_2)^L \\ \{\, (F_\Omega^\eta(d^i), F_\Omega^\eta(d_\Omega^i)) \mid i \in I \,\}^{\ell_V^\eta(d)} & \text{if } d = \{\, (d^i, d_\Omega^i) \mid i \in I \,\}^L \end{cases}$$

$$\ell_V^\eta(d) = \{\, \alpha \in V \mid d \in \eta(\alpha) \,\}$$

The function F_V^η transforms domain elements by changing the labels L recursively. Each label is changed according to η. The requirement that V be finite ensures that the new labels are always finite (only finite subsets of TVar are allowed as labels).

The proof is by induction on the pair (d, t) ordered lexicographically.

Case: $t = \alpha$

We have

$$\begin{aligned} (d :_\eta \alpha)^{\mathsf{q}} &\iff d \in \eta(\alpha) \\ (F_V^\eta(d) :_{\hat{\eta}} \alpha)^{\mathsf{q}} &\iff F_V^\eta(d) \in \hat{\eta}(\alpha) \iff \alpha \in \mathsf{tags}(F_V^\eta(d)) \\ &\iff \alpha \in \ell_V^\eta(d) \iff (\alpha \in V) \wedge (d \in \eta(\alpha)) \end{aligned}$$

which is the result we need, since $\alpha \in T$ implies $\alpha \in V$.

Case: $t = b$

If d is not of the form c^L, then both $(d :_\eta b)^{\mathsf{q}}$ and $(F_V^\eta(d) :_{\hat{\eta}} b)^{\mathsf{q}}$ do not hold. If $d = c^L$, then $(c^L :_\eta b)^{\mathsf{q}} \iff c \in \mathbb{B}(b) \iff (F_V^\eta(c^L) :_\eta b)^{\mathsf{q}}$.

Case: $t = t_1 \times t_2$

As in the previous case, the equivalence is straightforward unless d is of the form $(d_1, d_2)^L$. In that case, we have

$$\begin{aligned} ((d_1, d_2)^L :_\eta t_1 \times t_2)^{\mathsf{q}} &\iff (d_1 :_\eta t_1)^{\mathsf{q}} \wedge (d_2 :_\eta t_2)^{\mathsf{q}} \\ (F_V^\eta((d_1, d_2)^L) :_{\hat{\eta}} t_1 \times t_2)^{\mathsf{q}} &\iff (F_V^\eta(d_1) :_{\hat{\eta}} t_1)^{\mathsf{q}} \wedge (F_V^\eta(d_2) :_{\hat{\eta}} t_2)^{\mathsf{q}} \end{aligned}$$

and $(d_1 :_\eta t_1)^{\mathsf{q}} \iff (F_V^\eta(d_1) :_{\hat{\eta}} t_1)^{\mathsf{q}}$ and $(d_2 :_\eta t_2)^{\mathsf{q}} \iff (F_V^\eta(d_2) :_{\hat{\eta}} t_2)^{\mathsf{q}}$ hold by IH.

Case: $t = t_1 \to t_2$

As in the previous two cases, the interesting case is when d is of the form $\{ (d^i, d^i_\Omega) \mid i \in I \}^L$. In that case, we have

$$(d :_\eta t_1 \to t_2)^{\mathsf{q}} \iff \forall i \in I.\, (d^i :_\eta t_1)^{\mathsf{q}} \implies (d^i_\Omega :_\eta t_2)^{\mathsf{q}}$$
$$(F^\eta_V(d) :_{\hat\eta} t_1 \to t_2)^{\mathsf{q}} \iff \forall i \in I.\, (F^\eta_V(d^i) :_{\hat\eta} t_1)^{\mathsf{q}} \implies (F^\eta_V(d^i_\Omega) :_{\hat\eta} t_2)^{\mathsf{q}}$$

and the equivalence holds by IH.

Case: $t = t_1 \vee t_2$

We have

$$(d :_\eta t_1 \vee t_2)^{\mathsf{q}} \iff (d :_\eta t_1)^{\mathsf{q}} \vee (d :_\eta t_2)^{\mathsf{q}}$$
$$(F^\eta_V(d) :_{\hat\eta} t_1 \vee t_2)^{\mathsf{q}} \iff (F^\eta_V(d) :_{\hat\eta} t_1)^{\mathsf{q}} \vee (F^\eta_V(d) :_{\hat\eta} t_2)^{\mathsf{q}}$$

and both $(d :_\eta t_1)^{\mathsf{q}} \iff (F^\eta_V(d) :_{\hat\eta} t_1)^{\mathsf{q}}$ and $(d :_\eta t_2)^{\mathsf{q}} \iff (F^\eta_V(d) :_{\hat\eta} t_2)^{\mathsf{q}}$ hold by IH.

Case: $t = \neg t'$

We have

$$(d :_\eta \neg t')^{\mathsf{q}} \iff \neg (d :_\eta t')^{\mathsf{q}}$$
$$(F^\eta_V(d) :_{\hat\eta} \neg t')^{\mathsf{q}} \iff \neg (F^\eta_V(d) :_{\hat\eta} t')^{\mathsf{q}}$$

and $\neg (d :_\eta t')^{\mathsf{q}} \iff \neg (F^\eta_V(d) :_{\hat\eta} t')^{\mathsf{q}}$ holds by IH.

Case: $t = \mathbb{0}$

Straightforward because $(d :_\eta \mathbb{0})^{\mathsf{q}}$ never holds for any d and η. □

2.9 PROPOSITION: For all types t_1 and t_2, $t_1 \le t_2$ holds if and only if $t_1 \le^{\mathsf{q}} t_2$. □

Proof: We have (applying Lemma 2.8):

$$t_1 \le t_2 \iff [\![t_1 \setminus t_2]\!] = \varnothing \iff [\![t_1 \setminus t_2]\!]^{\mathsf{q}} \hat\eta = \varnothing$$
$$t_1 \le^{\mathsf{q}} t_2 \iff \forall \eta : \mathsf{TVar} \to \mathcal{P}(\mathsf{Domain}).\, [\![t_1 \setminus t_2]\!]^{\mathsf{q}} \eta = \varnothing$$

If $t_1 \le t_2$, we obtain $t_1 \le^{\mathsf{q}} t_2$ by applying Lemma 2.8 with $V = \mathsf{var}(t_1 \setminus t_2)$.
If $t_1 \le^{\mathsf{q}} t_2$, we obtain $t_1 \le t_2$ because $\hat\eta : \mathsf{TVar} \to \mathcal{P}(\mathsf{Domain})$. □

2.4.2 Subtyping and type substitutions

We now show that subtyping judgments are preserved if we apply a type substitution to both types. This result is needed to ensure soundness for polymorphic type systems. The proof is adapted from Castagna and Xu (2011) and relies on the definition of subtyping based on quantification.

2.10 LEMMA: For every t, σ, and η, if η' is defined by $\eta'(\alpha) = [\![\sigma(\alpha)]\!]^{\mathsf{q}} \eta$, then $[\![t\sigma]\!]^{\mathsf{q}} \eta = [\![t]\!]^{\mathsf{q}} \eta'$. □

Proof: Consider arbitrary σ and $\eta : \mathsf{TVar} \to \mathcal{P}(\mathsf{Domain})$. Let η' be defined from σ and η as in the statement. We can show

$$\forall t \in \mathsf{Type}.\ \forall d \in \mathsf{Domain}.\ (d :_{\eta} t\sigma)^{\mathsf{q}} \iff (d :_{\eta'} t)^{\mathsf{q}}$$

by induction on (d, t). All cases are straightforward. □

2.11 PROPOSITION: If $t_1 \leq t_2$, then $t_1\sigma \leq t_2\sigma$ for any type substitution σ. □

Proof: By definition of subtyping and by Proposition 2.9, from $t_1 \leq t_2$ we have $\forall \eta : \mathsf{TVar} \to \mathcal{P}(\mathsf{Domain}).\ [\![t_1 \setminus t_2]\!]^{\mathsf{q}}\eta = \varnothing$.

We show $\forall \eta : \mathsf{TVar} \to \mathcal{P}(\mathsf{Domain}).\ [\![(t_1 \setminus t_2)\sigma]\!]^{\mathsf{q}}\eta = \varnothing$. Consider an arbitrary η: we must show $[\![(t_1 \setminus t_2)]\!]^{\mathsf{q}}\eta = \varnothing$. Take η' defined by $\eta'(\alpha) = [\![\sigma(\alpha)]\!]^{\mathsf{q}}\eta$. By Lemma 2.10, we have $[\![(t_1 \setminus t_2)\sigma]\!]^{\mathsf{q}}\eta = [\![t_1 \setminus t_2]\!]^{\mathsf{q}}\eta'$. Since $\eta' : \mathsf{TVar} \to \mathcal{P}(\mathsf{Domain})$, we have $[\![t_1 \setminus t_2]\!]^{\mathsf{q}}\eta'$ and $[\![(t_1 \setminus t_2)\sigma]\!]^{\mathsf{q}}\eta$.

From $\forall \eta : \mathsf{TVar} \to \mathcal{P}(\mathsf{Domain}).\ [\![(t_1 \setminus t_2)\sigma]\!]^{\mathsf{q}}\eta = \varnothing$ we have $t_1\sigma \leq t_2\sigma$ by applying again Proposition 2.9. □

We now show that type substitutions that are equivalent (meaning that they map each type variable to equivalent types) map any given type to equivalent types. We first define subtype equivalence on type substitutions pointwise.

2.12 DEFINITION: Two type substitutions σ_1 and σ_2 are equivalent, written $\sigma_1 \simeq \sigma_2$, if, for every type variable α, we have $\alpha\sigma_1 \simeq \alpha\sigma_2$. □

2.13 LEMMA: If $\sigma_1 \simeq \sigma_2$, then $t\sigma_1 \simeq t\sigma_2$. □

Proof: Assuming $\sigma_1 \simeq \sigma_2$, we prove the result

$$\forall d, t.\ \ (d : t\sigma_1) \iff (d : t\sigma_2)\,,$$

which is equivalent to the statement.

The proof is by induction on the pair (d, t) ordered lexicographically.

Case: $t = \alpha$
We have $\alpha\sigma_1 \simeq \alpha\sigma_2$, therefore $(d : \alpha\sigma_1) \iff (d : \alpha\sigma_2)$.

Case: $t = b$ Straightforward because $t\sigma_1 = b = t\sigma_2$.

Case: $t = t_1 \times t_2$
We have

$$(d : t\sigma_1) \iff \exists d_1, d_2, L.\ \ d = (d_1, d_2)^L \text{ and } (d_1 : t_1\sigma_1) \text{ and } (d_2 : t_2\sigma_1)$$
$$(d : t\sigma_2) \iff \exists d_1, d_2, L.\ \ d = (d_1, d_2)^L \text{ and } (d_1 : t_1\sigma_2) \text{ and } (d_2 : t_2\sigma_2)$$

and we conclude by applying the IH to (d_1, t_1) and (d_2, t_2).

Case: $t = t_1 \to t_2$
Analogous to the previous case.

Case: $t = t_1 \vee t_2$
We have:

$$\begin{aligned}(d\colon t\sigma_1) &\iff (d\colon t_1\sigma_1) \text{ or } (d\colon t_2\sigma_1)\\ &\iff (d\colon t_1\sigma_2) \text{ or } (d\colon t_2\sigma_2)\\ &\iff (d\colon t\sigma_2)\,.\end{aligned}$$

Case: $t = \neg t'$
If $(d\colon t\sigma_1)$, then $\neg(d\colon t'\sigma_1)$. Then, by IH, $\neg(d\colon t'\sigma_2)$. Therefore, $(d\colon t\sigma_2)$.

Case: $t = \mathbb{0}$ Straightforward because $t\sigma_1 = \mathbb{0} = t\sigma_2$. □

2.4.3 Decomposition of subtyping on arrow types

The results in this section show how subtyping judgments involving unions and intersections of arrow types can be decomposed to subtyping judgments on subterms of these types. They are adapted from the work of Frisch (2004) and Frisch, Castagna, and Benzaken (2008).

2.14 LEMMA: Let X and Y be two sets and $(X_i)_{i\in I}$ and $(Y_i)_{i\in I}$ two finite families of sets. Then:

$$\Big(X \times Y\Big) \setminus \Big(\bigcup_{i\in I} X_i \times Y_i\Big) = \bigcup_{I'\subseteq I} \Big(X \setminus \bigcup_{i\in I'} X_i\Big) \times \Big(Y \setminus \bigcup_{i\in I\setminus I'} Y_i\Big) \qquad \square$$

Proof: Note that, for any four sets A, B, C, D, we have $(A \times B) \setminus (C \times D) = ((A \setminus C) \times B) \cup (A \times (C \setminus D))$.
We proceed by induction on $|I|$.

Case: $I = \varnothing$ Straightforward.

Case: $I = I' \uplus \{i_0\}$
We have

$$\begin{aligned}&\big(X \times Y\big) \setminus \big(\textstyle\bigcup_{i\in I} X_i \times Y_i\big)\\ &= \Big(\big(X \times Y\big) \setminus \big(X_{i_0} \times Y_{i_0}\big)\Big) \setminus \big(\textstyle\bigcup_{i\in I'} X_i \times Y_i\big)\\ &= \Big(\big((X \setminus X_{i_0}) \times Y\big) \cup \big(X \times (Y \setminus Y_{i_0})\big)\Big) \setminus \big(\textstyle\bigcup_{i\in I'} X_i \times Y_i\big)\\ &= \Big(\big((X \setminus X_{i_0}) \times Y\big) \setminus \big(\textstyle\bigcup_{i\in I'} X_i \times Y_i\big)\Big)\\ &\qquad \cup \Big(\big(X \times (Y \setminus Y_{i_0})\big) \setminus \big(\textstyle\bigcup_{i\in I'} X_i \times Y_i\big)\Big)\end{aligned}$$

and, by IH,

$$\begin{aligned}&= \Big(\textstyle\bigcup_{I''\subseteq I'} \big((X \setminus X_{i_0}) \setminus \bigcup_{i\in I''} X_i\big) \times \big(Y \setminus \bigcup_{i\in I'\setminus I''} Y_i\big)\Big)\\ &\qquad \cup \Big(\textstyle\bigcup_{I''\subseteq I'} \big(X \setminus \bigcup_{i\in I''} X_i\big) \times \big((Y \setminus Y_{i_0}) \setminus \bigcup_{i\in I'\setminus I''} Y_i\big)\Big)\end{aligned}$$

$$= \left(\bigcup_{I'' \subseteq I'} \left(X \setminus \bigcup_{i \in I'' \cup \{i_0\}} X_i \right) \times \left(Y \setminus \bigcup_{i \in I' \setminus I''} Y_i \right) \right)$$
$$\cup \left(\bigcup_{I'' \subseteq I'} \left(X \setminus \bigcup_{i \in I''} X_i \right) \times \left(Y \setminus \bigcup_{i \in (I' \cup \{i_0\}) \setminus I''} Y_i \right) \right)$$
$$= \bigcup_{I'' \subseteq I} \left(X \setminus \bigcup_{i \in I''} X_i \right) \times \left(Y \setminus \bigcup_{i \in I \setminus I''} Y_i \right) . \qquad \square$$

2.15 LEMMA: Let $(X_i)_{i \in P}$ and $(X_i)_{i \in N}$ be two finite families of sets, with P non-empty. Then:

$$\bigcap_{i \in P} \mathcal{P}_{\text{fin}}(X_i) \subseteq \bigcup_{j \in N} \mathcal{P}_{\text{fin}}(X_j) \iff \exists j_0 \in N. \bigcap_{i \in P} X_i \subseteq X_{j_0} \qquad \square$$

Proof: Note that $\bigcap_{i \in P} \mathcal{P}_{\text{fin}}(X_i) = \mathcal{P}_{\text{fin}}(\bigcap_{i \in P} X_i)$ and that, for all sets A and B, $A \subseteq B$ implies $\mathcal{P}_{\text{fin}}(A) \subseteq \mathcal{P}_{\text{fin}}(B)$. The implication $\impliedby$ is straightforward to prove using these two facts.

For the reverse implication, assume that $\bigcap_{i \in P} \mathcal{P}_{\text{fin}}(X_i) \subseteq \bigcup_{j \in N} \mathcal{P}_{\text{fin}}(X_j)$, that is, that $\mathcal{P}_{\text{fin}}(\bigcap_{i \in P} X_i) \subseteq \bigcup_{j \in N} \mathcal{P}_{\text{fin}}(X_j)$. Let $X = \bigcap_{i \in P} X_i$. By contradiction, assume that no $j \in N$ is such that $X \subseteq X_j$. Then, for every $j \in N$, there exists an $x_j \in X \setminus X_j$. Consider the finite set $\{ x_j \mid j \in N \}$. It is in $\mathcal{P}_{\text{fin}}(X)$ but not in $\bigcup_{j \in N} \mathcal{P}_{\text{fin}}(X_j)$, which disproves the hypothesis. $\square$

2.16 LEMMA: Let P, N be two finite sets of types of the form $t_1 \to t_2$, with P non-empty. Then:

$$\bigwedge_{t_1 \to t_2 \in P} t_1 \to t_2 \leq \bigvee_{t_1 \to t_2 \in N} t_1 \to t_2 \iff \exists (\hat{t}_1 \to \hat{t}_2) \in N.$$
$$\left(\hat{t}_1 \leq \bigvee_{t_1 \to t_2 \in P} t_1 \right) \wedge \left(\forall P' \subsetneq P. \left(\hat{t}_1 \leq \bigvee_{t_1 \to t_2 \in P'} t_1 \right) \vee \left(\bigwedge_{t_1 \to t_2 \in P \setminus P'} t_2 \leq \hat{t}_2 \right) \right)$$

$\square$

Proof: Writing D_1 for $\mathsf{Domain} \cup \{\Omega\}$, D_2 for $\mathsf{Domain} \times D_1$, and $\overline{A}^B$ for $B \setminus A$, note that

$$\begin{aligned} &[\![t_1 \to t_2]\!] \\ &= \left\{ \{ (d^i, d^i_\Omega) \mid i \in I \}^L \;\middle|\; \forall i \in I.\, d^i \in [\![t_1]\!] \implies d^i_\Omega \in [\![t_2]\!] \right\} \\ &= \left\{ \{ (d^i, d^i_\Omega) \mid i \in I \}^L \;\middle|\; \{ (d^i, d^i_\Omega) \mid i \in I \} \in \mathcal{P}_{\text{fin}}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}^{D_1}}^{D_2}) \right\} . \end{aligned}$$

We have

$$\begin{aligned} &\textstyle\bigwedge_{t_1 \to t_2 \in P} t_1 \to t_2 \leq \bigvee_{t_1 \to t_2 \in N} t_1 \to t_2 \\ \iff &\textstyle\bigcap_{t_1 \to t_2 \in P} [\![t_1 \to t_2]\!] \subseteq \bigcup_{t_1 \to t_2 \in N} [\![t_1 \to t_2]\!] \\ \iff &\textstyle\bigcap_{t_1 \to t_2 \in P} \mathcal{P}_{\text{fin}}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}^{D_1}}^{D_2}) \subseteq \bigcup_{t_1 \to t_2 \in N} \mathcal{P}_{\text{fin}}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}^{D_1}}^{D_2}) \end{aligned}$$

(we can ignore the sets of type variables labelling the domain elements, because arrow types always contain each relation with all possible labels).

By Lemma 2.15,

$$\begin{aligned}
&\iff \exists \hat{t}_1 \to \hat{t}_2 \in N.\ \ \overline{\bigcap_{t_1\to t_2\in P} [\![t_1]\!] \times \overline{[\![t_2]\!]}^{D_1}}^{D_2} \subseteq \overline{[\![\hat{t}_1]\!] \times \overline{[\![\hat{t}_2]\!]}^{D_1}}^{D_2} \\
&\iff \exists \hat{t}_1 \to \hat{t}_2 \in N.\ D_2 \setminus \left(\bigcup_{t_1\to t_2\in P} [\![t_1]\!] \times \overline{[\![t_2]\!]}^{D_1}\right) \subseteq D_2 \setminus \left([\![\hat{t}_1]\!] \times \overline{[\![\hat{t}_2]\!]}^{D_1}\right) \\
&\iff \exists \hat{t}_1 \to \hat{t}_2 \in N.\ [\![\hat{t}_1]\!] \times \overline{[\![\hat{t}_2]\!]}^{D_1} \subseteq \bigcup_{t_1\to t_2\in P} [\![t_1]\!] \times \overline{[\![t_2]\!]}^{D_1}
\end{aligned}$$

and, applying Lemma 2.14,

$$\begin{aligned}
&\iff \exists \hat{t}_1 \to \hat{t}_2 \in N.\ \forall P' \subseteq P. \\
&\qquad \left([\![\hat{t}_1]\!] \subseteq \bigcup_{t_1\to t_2\in P'} [\![t_1]\!]\right) \vee \left(\overline{[\![\hat{t}_2]\!]}^{D_1} \subseteq \bigcup_{t_1\to t_2\in P\setminus P'} \overline{[\![t_2]\!]}^{D_1}\right) \\
&\iff \exists \hat{t}_1 \to \hat{t}_2 \in N. \\
&\qquad \left(\left([\![\hat{t}_1]\!] \subseteq \bigcup_{t_1\to t_2\in P} [\![t_1]\!]\right) \vee \left(\overline{[\![\hat{t}_2]\!]}^{D_1} \subseteq \bigcup_{t_1\to t_2\in \varnothing} \overline{[\![t_2]\!]}^{D_1}\right)\right) \wedge \\
&\qquad \left(\forall P' \subsetneq P.\ \left([\![\hat{t}_1]\!] \subseteq \bigcup_{t_1\to t_2\in P'} [\![t_1]\!]\right) \vee \left(\overline{[\![\hat{t}_2]\!]}^{D_1} \subseteq \bigcup_{t_1\to t_2\in P\setminus P'} \overline{[\![t_2]\!]}^{D_1}\right)\right) \\
&\iff \exists \hat{t}_1 \to \hat{t}_2 \in N.\ \left([\![\hat{t}_1]\!] \subseteq \bigcup_{t_1\to t_2\in P} [\![t_1]\!]\right) \wedge \\
&\qquad \left(\forall P' \subsetneq P.\ \left([\![\hat{t}_1]\!] \subseteq \bigcup_{t_1\to t_2\in P'} [\![t_1]\!]\right) \vee \left(\overline{[\![\hat{t}_2]\!]}^{D_1} \subseteq \bigcup_{t_1\to t_2\in P\setminus P'} \overline{[\![t_2]\!]}^{D_1}\right)\right)
\end{aligned}$$

(because $\overline{[\![\hat{t}_2]\!]}^{D_1}$ is never empty)

$$\begin{aligned}
&\iff \exists \hat{t}_1 \to \hat{t}_2 \in N.\ \left([\![\hat{t}_1]\!] \subseteq \bigcup_{t_1\to t_2\in P} [\![t_1]\!]\right) \wedge \\
&\qquad \left(\forall P' \subsetneq P.\ \left([\![\hat{t}_1]\!] \subseteq \bigcup_{t_1\to t_2\in P'} [\![t_1]\!]\right) \vee \left(\bigcap_{t_1\to t_2\in P\setminus P'} [\![t_2]\!] \subseteq [\![\hat{t}_2]\!]\right)\right) \\
&\iff \exists \hat{t}_1 \to \hat{t}_2 \in N.\ \left(\hat{t}_1 \leq \bigvee_{t_1\to t_2\in P} t_1\right) \wedge \\
&\qquad \left(\forall P' \subsetneq P.\ \left(\hat{t}_1 \leq \bigvee_{t_1\to t_2\in P'} t_1\right) \vee \left(\bigwedge_{t_1\to t_2\in P\setminus P'} t_2 \leq \hat{t}_2\right)\right)
\end{aligned}$$

□

2.17 COROLLARY: Let P, N be two finite sets of types of the form $t_1 \to t_2$, with P non-empty. Then:

$$\bigwedge_{t_1\to t_2\in P} t_1 \to t_2 \leq \bigvee_{t_1\to t_2\in N} t_1 \to t_2 \iff \exists (\hat{t}_1 \to \hat{t}_2) \in N.\ \bigwedge_{t_1\to t_2\in P} t_1 \to t_2 \leq \hat{t}_1 \to \hat{t}_2\ .$$

□

Proof: Consequence of Lemma 2.16. □

Part I

Implicit typing and type inference

3 An implicitly typed language with set-theoretic types

This chapter describes the syntax, operational semantics, and type system of a language: a typed, call-by-value λ-calculus with some additional constructs, notably a form of runtime type testing. The language is typed implicitly and defined in Curry style: the operational semantics is defined independently of typing. The type system features set-theoretic types and semantic subtyping. We study the properties of the type system and establish a standard type soundness result. We do not study algorithmic type checking or type inference; the latter is the object of the following chapters.

Compared to previous work on semantic subtyping (notably Frisch, Castagna, and Benzaken, 2008; Castagna et al., 2014; Castagna et al., 2015b) the contribution here is in considering an implicitly typed language. In contrast, in previous work functions were always annotated with their type. Implicit typing and a restriction of the typecase construct allow us to give a very simple operational semantics. The explicitly typed language of Castagna et al. (2014), instead, has a more complex semantics since type information must be propagated during reduction and updated when polymorphic functions are instantiated.

Here as in previous work on semantic subtyping, we need to handle negation types carefully to ensure subject reduction. This is the main technical difficulty of this chapter: the problem was already studied by Frisch, Castagna, and Benzaken (2008), but the implicitly typed setting requires a different solution.

CHAPTER OUTLINE:

Section 3.1 We describe the syntax and the semantics of the language.

Section 3.2 We describe its type system.

Section 3.3 We develop the proof of soundness. The proof actually requires a modification of the type system – adding a rule to derive negation types for functions – which we introduce here and not in Section 3.2 in order to motivate it properly.

3.1 Language syntax and semantics

3.1.1 Syntax

The language is an untyped λ-calculus extended with a few constructs.

To define the syntax, we take an arbitrary, countable set EVar of *expression variables*, ranged over by x, y, z, ... We also consider the set Const of language constants used to define types in Section 2.2.

3.1 DEFINITION: The *expressions* of the language are the terms e defined inductively by the grammar

$$e ::= x \mid c \mid \lambda x.\, e \mid e\ e \mid (e, e) \mid \pi_i\ e \mid e \in \mathbf{t}\ ?\ e : e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e$$

where x ranges over EVar, c over Const, i over $\{1, 2\}$, and where $\mathbf{t}$ is a type in Type generated coinductively by the following grammar:

$$\mathbf{t} ::= b \mid \mathbf{t} \times \mathbf{t} \mid \mathbb{0} \to \mathbb{1} \mid \mathbf{t} \vee \mathbf{t} \mid \neg\mathbf{t} \mid \mathbb{0}\,. \qquad \square$$

As customary, expressions are considered up to α-renaming of bound variables. In $\lambda x.\, e$, x is bound in e. In $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, x is bound in e_2.

Expressions include the forms of the λ-calculus: variables x, λ-abstractions $\lambda x.\, e$, and applications $e\ e$. There are also constants c, pairs (e, e) and pair projections $\pi_i\ e$, the typecase expression $e \in \mathbf{t}\ ?\ e : e$, and the let construct $\mathsf{let}\ x = e\ \mathsf{in}\ e$.

A typecase $e_0 \in \mathbf{t}\ ?\ e_1 : e_2$ is a dynamic type test. It is evaluated by evaluating e_0 and then, if e_0 reduces to a value v (the syntax of values is given below), evaluating e_1 if v has type $\mathbf{t}$ or e_2 otherwise.

Typecases cannot test arbitrary types, since they use the restricted grammar for $\mathbf{t}$. There are two restrictions with respect to the types of Definition 2.1: types must be ground (α does not appear), and the only arrow type that can appear is $\mathbb{0} \to \mathbb{1}$, which is the type of all functions.[1] This means that typecases can distinguish functions from non-functions but cannot distinguish, for instance, the functions that have type $\mathsf{Int} \to \mathsf{Int}$ from those that do not. In previous work on semantic subtyping, there is no such restriction. However, if we allowed tests on function types, in a practical implementation the semantics would depend on the behaviour of the type checking or type inference algorithms. Thanks to this restriction, instead, the semantics does not depend on the type system: it could be implemented without keeping track of compile-time types at runtime. Moreover, the interest of the typecase construct in this work is mostly to encode a pattern matching construct. Standard pattern matching cannot check function types, so the restriction is not a problem for this. Typecases of this form also have the same expressiveness as the type-testing primitives of dynamic languages like JavaScript and Racket.

3.1.2 Semantics

We define the operational semantics of the language in small-step style. The evaluation is call-by-value. First, we define the values of the language.

3.2 DEFINITION: A *value* v is an expression generated by the grammar

$$v ::= c \mid \lambda x.\, e \mid (v_1, v_2)$$

and that is *closed*, that is, that does not contain any free variable.

We write Values for the set of all values. $\square$

1 Actually, we could remove even $\mathbb{0} \to \mathbb{1}$ from the grammar: it can be expressed without using arrow types, because it is equivalent to $\neg(\mathbb{1}_{\mathsf{B}} \vee (\mathbb{1} \times \mathbb{1}))$, the type of all values that are neither constants nor pairs. We leave it in the grammar for clarity.

$$
\begin{array}{rrcll}
[\mathrm{R}_{\mathrm{app}}] & v_1\, v_2 & \rightsquigarrow & e[v_2/x] & \text{if } v_1 = \lambda x.\, e \\
[\mathrm{R}_{\mathrm{proj}}] & \pi_i\, (v_1, v_2) & \rightsquigarrow & v_i & \\
[\mathrm{R}_{\mathsf{let}}] & \mathsf{let}\ x = v\ \mathsf{in}\ e & \rightsquigarrow & e[v/x] & \\
[\mathrm{R}^1_{\mathrm{case}}] & v \in \mathbf{t}\ ?\ e_1 : e_2 & \rightsquigarrow & e_1 & \text{if } \mathsf{typeof}(v) \leq \mathbf{t} \\
[\mathrm{R}^2_{\mathrm{case}}] & v \in \mathbf{t}\ ?\ e_1 : e_2 & \rightsquigarrow & e_2 & \text{if } \mathsf{typeof}(v) \leq \neg\mathbf{t} \\
[\mathrm{R}_{\mathrm{ctx}}] & E[e] & \rightsquigarrow & E[e'] & \text{if } e \rightsquigarrow e'
\end{array}
$$

FIGURE 3.1 Reduction rules

The semantics uses evaluation contexts to direct the order of evaluation. These are standard contexts for call-by-value, left-to-right reduction.

3.3 DEFINITION: A *context* is obtained from an expression by replacing one of the subterms with a hole, written [ ]. When C is a context, we write $C[e]$ for the expression obtained from C by replacing the hole with e.

An *evaluation context* E is a context generated by the following grammar:

$$E ::= [\,] \mid E\, e \mid v\, E \mid (E, e) \mid (v, E) \mid \pi_i\, E \mid E \in \mathbf{t}\ ?\ e : e \mid \mathsf{let}\ x = E\ \mathsf{in}\ e\,. \qquad \square$$

To define the semantics, we also use a standard definition of substitution mapping an expression variable to a value. The notation $e[v/x]$ indicates the expression obtained by replacing all free occurrences of x in e by v.

3.4 DEFINITION: The reduction relation $e \rightsquigarrow e'$ between expressions is defined by the rules in Figure 3.1. The rules use the typeof function, defined as

$$\mathsf{typeof}(v) \stackrel{\text{def}}{=} \begin{cases} b_c & \text{if } v = c \\ \mathbb{0} \to \mathbb{1} & \text{if } v = \lambda x.\, e \\ \mathsf{typeof}(v_1) \times \mathsf{typeof}(v_2) & \text{if } v = (v_1, v_2) \end{cases}$$

to map values to types for the evaluation of typecases. $\square$

The rules $[\mathrm{R}_{\mathrm{app}}]$, $[\mathrm{R}_{\mathrm{proj}}]$, and $[\mathrm{R}_{\mathsf{let}}]$ are entirely standard, as is the context closure rule $[\mathrm{R}_{\mathrm{ctx}}]$. Evaluation of typecases uses two rules, $[\mathrm{R}^1_{\mathrm{case}}]$ and $[\mathrm{R}^2_{\mathrm{case}}]$, which reduce the expression to either of its branches depending on whether the tested value has the type $\mathbf{t}$ or the type $\neg\mathbf{t}$ (Lemma 3.29 will show that exactly one of the two must hold). This test relies on the function typeof to map values to types. Note that typeof maps every λ-abstraction to $\mathbb{0} \to \mathbb{1}$, so it does not depend on static types. This approximation is allowed by the restriction on arrow types in typecases.

3.2 Type system

We now equip the language with a type system. We give a declarative definition of the type system: by *declarative* we mean that we rely on structural, non-syntax-directed rules for subtyping and for the introduction of intersection

types.[2] The next two chapters will focus instead on the study of algorithmic type inference.

The type system described here is very similar to a standard Hindley-Milner system: the differences are just the addition of subtyping and intersection introduction, as well as a rule for typecases. However, we will see in the next section that, to prove type soundness, we need to augment the system with a less standard rule.

As in Hindley-Milner type systems, we introduce a notion of type scheme separate from that of types.

3.5 DEFINITION: A *type scheme* is a term of the form $\forall\vec{\alpha}.\, t$. We view types as a subset of type schemes by identifying $\forall\vec{\alpha}.\, t$ with t itself if $\vec{\alpha}$ is empty. □

Type schemes are treated up to α-renaming of their bound variables. We extend $\mathsf{var}(\cdot)$ to type schemes by defining $\mathsf{var}(\forall\vec{\alpha}.\, t) = \mathsf{var}(t) \setminus \vec{\alpha}$. We extend the application of type substitutions to type schemes: $(\forall\vec{\alpha}.\, t)\sigma = \forall\vec{\alpha}.\, (t\sigma)$ when $\vec{\alpha} \sharp \sigma$ (i.e., when the variables in $\vec{\alpha}$ do not appear in $\mathsf{dom}(\sigma)$ and $\mathsf{var}(\sigma)$); this condition can always be ensured by α-renaming.

We give a standard definition for type environments too.

3.6 DEFINITION: *Type environments* Γ are finite mappings from expression variables to type schemes. We write $\varnothing$ for the empty type environment. □

We write $\mathsf{dom}(\Gamma)$ for the domain of the type environment. We write $\mathsf{var}(\Gamma)$ for the set of type variables that appear in Γ: that is, $\mathsf{var}(\Gamma) = \bigcup_{x \in \mathsf{dom}(\Gamma)} \mathsf{var}(\Gamma(x))$. We write type environments as finite sets of bindings with the standard notation $x_1 : \forall\vec{\alpha}_1.\, t_1, \ldots, x_n : \forall\vec{\alpha}_n.\, t_n$, where we assume that each x_i is distinct. We write $\Gamma, x : \forall\vec{\alpha}.\, t$ to denote the type environment obtained by extending Γ with the new binding $x : \forall\vec{\alpha}.\, t$, assuming that x does not already occur in Γ (which in practice is typically ensured by α-renaming). We extend the application of type substitutions to type environments as the pointwise application of the substitution to all type schemes in the environment.

We can now start to define the type system. Figure 3.2 presents ten of the typing rules defining the typing relation $\Gamma \vdash e : t$. The relation that we define formally (Definition 3.11) includes one more rule, which we need to prove soundness: a rule to type functions with negations of arrow types. That rule is less standard, and its inclusion demands more motivation. For now, we describe the ten rules of Figure 3.2. In the study of type inference, we will only consider the system with these ten rules. We refer to the inference system and associated typing relation defined by these rules as $\mathcal{T}$.

The first six rules – for variables, constants, λ-abstractions, applications, pairs, and projections – are entirely standard. So is the $[\mathrm{T}_{\mathsf{let}}]$ rule for let: it

2 In contrast, an *algorithmic* presentation does not use structural rules and embeds subtyping and intersection introduction into the other rules so as to be syntax-directed and closer to an actual typechecking algorithm. We use this terminology following, among others, Pierce (2002).

$$[\mathrm{T}_x]\ \frac{}{\Gamma \vdash x : t[\vec{t}/\vec{\alpha}]}\ \Gamma(x) = \forall\vec{\alpha}.\, t \qquad [\mathrm{T}_c]\ \frac{}{\Gamma \vdash c : b_c}$$

$$[\mathrm{T}_\lambda]\ \frac{\Gamma, x : t' \vdash e : t}{\Gamma \vdash \lambda x.\, e : t' \to t} \qquad [\mathrm{T}_{\mathrm{app}}]\ \frac{\Gamma \vdash e_1 : t' \to t \qquad \Gamma \vdash e_2 : t'}{\Gamma \vdash e_1\ e_2 : t}$$

$$[\mathrm{T}_{\mathrm{pair}}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad [\mathrm{T}_{\mathrm{proj}}]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_i\, e : t_i}$$

$$[\mathrm{T}_{\mathrm{case}}]\ \frac{\Gamma \vdash e_0 : t_0 \qquad \text{either } t_0 \leq \neg\mathbf{t} \text{ or } \Gamma \vdash e_1 : t \qquad \text{either } t_0 \leq \mathbf{t} \text{ or } \Gamma \vdash e_2 : t}{\Gamma \vdash (e_0 \in \mathbf{t}\ ?\ e_1 : e_2) : t}$$

$$[\mathrm{T}_{\mathsf{let}}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, x : \forall\vec{\alpha}.\, t_1 \vdash e_2 : t}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : t}\ \vec{\alpha} \sharp \Gamma$$

$$[\mathrm{T}_{\leq}]\ \frac{\Gamma \vdash e : t'}{\Gamma \vdash e : t}\ t' \leq t \qquad [\mathrm{T}_{\wedge}]\ \frac{\Gamma \vdash e : t_1 \qquad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2}$$

FIGURE 3.2 $\mathcal{T}$: Typing rules

allows generalization of the variables $\vec{\alpha}$, which must not occur in Γ; this is expressed by the notation $\vec{\alpha} \sharp \Gamma$, which, following our conventions (Section 1.5), means $\vec{\alpha} \cap \mathsf{var}(\Gamma) = \varnothing$. The $[\mathrm{T}_{\leq}]$ rule is a subsumption rule, using the subtyping relation of Definition 2.5. The $[\mathrm{T}_{\wedge}]$ is a standard intersection-introduction rule.

The most complex rule is $[\mathrm{T}_{\mathrm{case}}]$, to type typecase expressions. It corresponds to four distinct rules with different side conditions, which are written in a compact form here using the "either … or …" shorthand. To type a typecase $e_0 \in \mathbf{t}\ ?\ e_1 : e_2$, we first type e_0 with some type t_0. Then, we can type both branches e_1 and e_2 with the type t. However, if $t_0 \leq \neg\mathbf{t}$, we do not need to type the first branch; if $t_0 \leq \mathbf{t}$, we do not need to type the second. This is because, if either of the two condition holds, we can predict statically that the corresponding branch can never be selected at run time. The typecase reduces to its first branch if e_0 reduces to a value of type $\mathbf{t}$, but this cannot happen if e_0 has type $\neg\mathbf{t}$; likewise for the second branch. If $t_0 \leq \mathbf{t}$ and $t_0 \leq \neg\mathbf{t}$ both hold, then e_0 will diverge (because then $t_0 \leq \mathbb{0}$, and there are no values of type $\mathbb{0}$), so we type neither branch.

REMARK (Typing of typecases): It is very important for the type system that we do not always need to type both branches of a typecase. Changing the rule so that both premises must always be typed makes intersection types less useful to type functions defined with typecases.

For example, the negation function $\lambda x.\, x \in b_{\mathsf{true}}\ ?\ \mathsf{false} : \mathsf{true}$ can be given

the type $(b_{\mathsf{true}} \to b_{\mathsf{false}}) \wedge (b_{\mathsf{false}} \to b_{\mathsf{true}})$. This relies on the fact that, in the derivation for the type $b_{\mathsf{true}} \to b_{\mathsf{false}}$, we know that the second branch will never be taken (and vice versa for the other arrow type). Otherwise, we could only derive the less precise type $\mathsf{Bool} \to \mathsf{Bool}$. □

REMARK (Recursive functions): Since types can be recursive, we do not need to introduce recursive functions explicitly. We can represent the recursive function $\mu f.\, \lambda x.\, e$ as $\mathsf{fix}\ (\lambda f.\, \lambda x.\, e)$, where fix is the call-by-value fixpoint combinator $\mathsf{fix} \equiv \lambda f.\, (\lambda x.\, \lambda y.\, f\ (x\ x)\ y)\ (\lambda x.\, \lambda y.\, f\ (x\ x)\ y)$.

Assume that we want to type the recursive function with the type $t = \bigwedge_{i \in I} t'_i \to t_i$, so we assume to have $(f : t) \vdash \lambda x.\, e : t$. Hence, $\lambda f.\, \lambda x.\, e$ has type $t \to t$. We must therefore type fix as $(t \to t) \to t$. In particular, we type it as follows:

$$[\mathrm{T}_\lambda]\ \frac{[\mathrm{T}_{\mathrm{app}}]\ \dfrac{f : (t \to t) \vdash (\lambda x.\, \lambda y.\, f\ (x\ x)\ y) : \bar{t} \to t \qquad f : (t \to t) \vdash (\lambda x.\, \lambda y.\, f\ (x\ x)\ y) : \bar{t}}{f : (t \to t) \vdash (\lambda x.\, \lambda y.\, f\ (x\ x)\ y)\ (\lambda x.\, \lambda y.\, f\ (x\ x)\ y) : t}}{\varnothing \vdash \mathsf{fix} : (t \to t) \to t}$$

where $\bar{t}$ is the recursive type satisfying $\bar{t} = \bar{t} \to t$. The typing derivation for $(\lambda x.\, \lambda y.\, f\ (x\ x)\ y)$ is

$$[\mathrm{T}_\lambda]\ \frac{[\mathrm{T}_\wedge]^*\ \dfrac{\forall i \in I.\ \left\{ [\mathrm{T}_\lambda]\ \dfrac{\dfrac{\vdots}{f : (t \to t), x : \bar{t}, y : t'_i \vdash f\ (x\ x)\ y : t_i}}{f : (t \to t), x : \bar{t} \vdash \lambda y.\, f\ (x\ x)\ y : t'_i \to t_i} \right.}{f : (t \to t), x : \bar{t} \vdash \lambda y.\, f\ (x\ x)\ y : t}}{f : (t \to t) \vdash (\lambda x.\, \lambda y.\, f\ (x\ x)\ y) : \bar{t} \to t}$$

where $[\mathrm{T}_\wedge]^*$ denotes multiple applications of the rule $[\mathrm{T}_\wedge]$. To type $f\ (x\ x)\ y$, note that $x\ x$ has type t and therefore $f\ (x\ x)$ has type t too. Since $t \leq t'_i \to t_i$, the application $f\ (x\ x)\ y$ has type t_i. □

3.3 Type soundness

We want to show that the system $\mathcal{T}$ is type safe by establishing a *type soundness* result which states that "well-typed programs do not go wrong". More precisely, a program (i.e., a closed expression) that is well typed must either reduce to a value or diverge: it cannot get stuck.

Following the well-known syntactic approach of Wright and Felleisen (1994), we show type soundness as a corollary of the following two properties.

Progress: closed, well-typed expressions that cannot reduce are values.

Subject reduction or *type preservation:* reduction preserves types.

However, the system $\mathcal{T}$ defined by the rules in Figure 3.2 does not satisfy subject reduction. It can occur that $\Gamma \vdash e : t$ and $e \rightsquigarrow e'$, while $\Gamma \vdash e' : t$ does not hold. In the following, we show why this is the case. Then, we describe

how to augment $\mathcal{T}$ with one more rule to recover subject reduction, albeit partially – it will only hold for expressions typed with ground types (i.e., types without type variables), but this is enough for soundness to hold. Then, we develop all the lemmas we need and prove soundness. Proving soundness for the system extended with this rule (which we will denote by $\mathcal{T}^{\lambda\neg}$) also implies soundness for the system of Figure 3.2, since the latter allows fewer derivations.

3.3.1 Why subject reduction does not hold

The problem with subject reduction arises from the presence of negation types and from the set-theoretic definition of subtyping. In particular, these make it so that, for subject reduction to hold, the following property must be true.

> For every type t and every well-typed value v, (⋆)
> either $\varnothing \vdash v : t$ or $\varnothing \vdash v : \neg t$ holds.

We first illustrate why this is needed. Consider the expression $\lambda x.\,(x, x)$ and the following typing derivation (for some arbitrary type t).

$$[\mathrm{T}_{\leq}]\ \dfrac{[\mathrm{T}_{\wedge}]\ \dfrac{\dfrac{\vdots}{\varnothing \vdash \lambda x.\,(x,x) : t \to (t \times t)} \qquad \dfrac{\vdots}{\varnothing \vdash \lambda x.\,(x,x) : \neg t \to (\neg t \times \neg t)}}{\varnothing \vdash \lambda x.\,(x,x) : \big(t \to (t \times t)\big) \wedge \big(\neg t \to (\neg t \times \neg t)\big)}}{\varnothing \vdash \lambda x.\,(x,x) : \mathbb{1} \to \big((t \times t) \vee (\neg t \times \neg t)\big)}$$

The subsumption rule can be applied because

$$\big(t \to (t \times t)\big) \wedge \big(\neg t \to (\neg t \times \neg t)\big) \leq \mathbb{1} \to \big((t \times t) \vee (\neg t \times \neg t)\big) :$$

in general, it holds that $(t_1' \to t_1) \wedge (t_2' \to t_2) \leq (t_1' \vee t_2') \to (t_1 \vee t_2)$, and $t \vee \neg t \simeq \mathbb{1}$. Now consider an arbitrary type t and a well-typed value v. Since v has type $\mathbb{1}$ by subsumption, the application $(\lambda x.\,(x, x))\ v$ can be typed as $(t \times t) \vee (\neg t \times \neg t)$. This application reduces to (v, v) by the rule $[\mathrm{R}_{\mathrm{app}}]$. Therefore, either (v, v) has type $(t \times t) \vee (\neg t \times \neg t)$ or subject reduction does not hold. Since $t \times t$ and $\neg t \times \neg t$ are disjoint, to derive the union type for v we need v to have either type t or type $\neg t$. This illustrates the need for the property above.

Unfortunately, that property does not hold for the typing relation $\mathcal{T}$ defined by the rules of Figure 3.2. The problems concern type variables and arrow types: the following are two examples.

- Take $v = 3$ and $t = \alpha$. The most precise type we can derive for 3 is its singleton type b_3. By definition of subtyping, $b_3 \not\leq \alpha$ and $b_3 \not\leq \neg\alpha$. Therefore, we can derive neither $\varnothing \vdash 3 : \alpha$ nor $\varnothing \vdash 3 : \neg\alpha$.
- Take $v = \lambda x.\,x$ and $t = \mathsf{Int} \to \mathsf{Bool}$. Clearly, $\varnothing \vdash v : t$ is not, and should not, be derivable. We would need $\varnothing \vdash v : \neg t$, but does it hold?

  A λ-abstraction can be typed with an arrow type using $[\mathrm{T}_{\lambda}]$. The rules $[\mathrm{T}_{\wedge}]$ and $[\mathrm{T}_{\leq}]$ can be used to intersect multiple types and to derive

supertypes. To derive $\neg t$, we would need $(t'_1 \to t_1) \wedge \cdots \wedge (t'_n \to t_n) \leq \neg t$, where each arrow in the intersection can be derived by $[\mathrm{T}_\lambda]$.

Note that $(t'_1 \to t_1) \wedge \cdots \wedge (t'_n \to t_n) \leq \neg t$ holds if and only if $(t'_1 \to t_1) \wedge \cdots \wedge (t'_n \to t_n) \wedge t \leq \mathbb{0}$. But an intersection of arrows is never empty: all arrows are supertypes of $\mathbb{1} \to \mathbb{0}$, whose interpretation is non-empty (intuitively, it contains functions that always diverge).

The problem with type variables can be avoided by showing a restricted version of subject reduction where we only consider expressions typed with ground types. For a proof by induction to work, we also require that the environment be ground (as for types, Γ is ground if $\mathsf{var}(\Gamma) = \varnothing$). We obtain the following statement.

Let Γ be a ground type environment and t a ground type.
If $\Gamma \vdash e : t$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : t$.

This statement is sufficient to show soundness, since soundness only involves *programs*, that is, closed expressions typed in the empty environment.

Once we restrict to considering only ground types, the property above is sensible: if a goal of semantic subtyping is to be able to see (ground) types as sets of values, then we expect any value that is not in a given type to be in its complement. However, the problem with arrow types remains. To solve it, we need a rule to derive negations of arrow types.

3.3.2 Negation types for functions

The difficulty we have described is not unique to our system: it arises naturally from the combination of semantic subtyping, negation types, and intersection introduction. Frisch, Castagna, and Benzaken (2008) solve it, but in a different setting: their language has explicitly typed functions and no polymorphism. In their system, functions are typed using the following rule.

$$\frac{\forall i \in I.\ \ \Gamma, x : t'_i \vdash e : t_i}{\Gamma \vdash (\lambda^{\mathbb{I}} x.\, e) : \mathbb{I} \wedge t} \begin{cases} \mathbb{I} = \bigwedge_{i \in I} t'_i \to t_i \\ t = \bigwedge_{j \in J} \neg(t'_j \to t_j) \\ \mathbb{I} \wedge t \not\simeq \mathbb{0} \end{cases}$$

A function is explicitly annotated with a finite intersection $\mathbb{I}$ of arrow types, its *interface*. It is well-typed if its body satisfies all the arrow types in $\mathbb{I}$. Then, we can assign to it any type made by intersecting $\mathbb{I}$ with any number of negated arrows, with the only constraint that the type must be non-empty. (Both I and J must be finite because of the contractivity condition on types.)

This rule is arguably counter-intuitive. We can use it, for instance, to type $\lambda^{\mathsf{Int}\to\mathsf{Int}} x.\, x$ as $(\mathsf{Int} \to \mathsf{Int}) \wedge \neg(\mathsf{Bool} \to \mathsf{Bool})$ even though the identity function could very well be given the type $\mathsf{Bool} \to \mathsf{Bool}$. However, the language is explicitly typed: as such, this annotated version of the identity function cannot be given the type $\mathsf{Bool} \to \mathsf{Bool}$, so it makes sense to derive its negation.

The rule ensures that, for any type $t' \to t$, either $t' \to t$ can be obtained by subsumption from $\mathbb{I}$ or $\neg(t' \to t)$ can be added to the intersection. In turn,

this ensures that, for any function and any type t, either the function has type t or it has type $\neg t$.

Our setting is different because of the lack of function interfaces. There is no explicit syntactic information in the λ-abstraction specifying and limiting which arrow types can be derived for it. The obvious adaptation of the rule above would be

$$\frac{\forall i \in I.\ \ \Gamma, x\colon t'_i \vdash e\colon t_i}{\Gamma \vdash (\lambda x.\, e)\colon \mathbb{I} \wedge t} \begin{cases} \mathbb{I} = \bigwedge_{i\in I} t'_i \to t_i \\ t = \bigwedge_{j\in J} \neg(t'_j \to t_j) \\ \mathbb{I} \wedge t \not\simeq \mathbb{0} \end{cases}$$

but it cannot be used because, in conjunction with [$\mathrm{T}_\wedge$], it would allow us to assign empty types to functions, as follows. We could use it to derive $\Gamma \vdash \lambda x.\, x\colon (\mathsf{Int} \to \mathsf{Int}) \wedge \neg(\mathsf{Bool} \to \mathsf{Bool})$, but also $\Gamma \vdash \lambda x.\, x\colon \mathsf{Bool} \to \mathsf{Bool}$. Using [$\mathrm{T}_\wedge$] we would then get an empty type.

We need a rule by which an arrow type $\neg(t' \to t)$ can be derived for $\lambda x.\, e$ if and only if $t' \to t$ *cannot* be derived. Since we only need to derive negation types for values, we can assume that $\lambda x.\, e$ is closed (this simplifies the problem because then its typing does not depend on Γ). We can consider ground arrow types only because of the aforementioned restriction of subject reduction. We want the rule for negation to be something like

$$\frac{x\colon t'_1 \vdash e\colon t_1}{\Gamma \vdash \lambda x.\, e\colon \neg(t' \to t)} \begin{cases} \lambda x.\, e \text{ closed} \\ t' \to t \text{ closed} \\ \lambda x.\, e \text{ cannot have type } t' \to t \end{cases}$$

(the premise $x\colon t'_1 \vdash e\colon t_1$ is there just to ensure that the rule can only be applied to functions whose body is well typed) but it remains, of course, to define what "$\lambda x.\, e$ cannot have type $t' \to t$" means.

When can a type $t' \to t$ be derived for a function $\lambda x.\, e$? It must be a supertype of some intersection of arrow types which we can assign to the function, that is, it must hold that

$$\exists\{(t'_1, t_1), \ldots, (t'_n, t_n)\}.\\ \qquad \big(\forall i \in \{1, \ldots, n\}.\ \ x\colon t'_i \vdash e\colon t_i\big) \wedge \big(\textstyle\bigwedge_{i\in\{1,\ldots,n\}} t'_i \to t_i \leq t' \to t\big)\ .$$

Hence, we might define "$\lambda x.\, e$ cannot have type $t' \to t$" as the negation

$$\forall\{(t'_1, t_1), \ldots, (t'_n, t_n)\}.\\ \qquad \big(\forall i \in \{1, \ldots, n\}.\ \ x\colon t'_i \vdash e\colon t_i\big) \implies \big(\textstyle\bigwedge_{i\in\{1,\ldots,n\}} t'_i \to t_i \not\leq t' \to t\big)\ .$$

Of course, this definition cannot be used because it depends on the definition of $\Gamma \vdash e\colon t$ itself, which – when we introduce the rule for negation – is the very system we are defining. This rule cannot be written in an inference system and, indeed, it would correspond to an inference operator that is not monotone. This means that there is no guarantee that a fixed point exists, so we cannot use the operator to define a relation by induction.

The solution that we present below is based on recognizing that we do not need the side condition to refer to the typing relation itself. Note that the typing rule we want to add is used to type the expression $\lambda x.\, e$, whereas the

side condition considers derivations for e, which is a strictly smaller expression. We can therefore use in the side condition a restricted typing relation which can only type expressions that are strictly smaller than $\lambda x.\, e$. We explain below this notion of a stratified inference system and how we can use it to define a rule to derive negations of arrow types.[3]

3.3.3 Deriving negations of arrow types

We associate to each expression a size which we compute straightforwardly.

3.7 DEFINITION: The *size* $s(e)$ of an expression e is defined as follows.

$$\begin{aligned} s(x) &= 1 & s((e_1, e_2)) &= 1 + s(e_1) + s(e_2) \\ s(c) &= 1 & s(\pi_i\ e) &= 1 + s(e) \\ s(\lambda x.\, e) &= 1 + s(e) & s(e_0 \in \mathbf{t}\ ?\ e_1 : e_2) &= 1 + s(e_0) + s(e_1) + s(e_2) \\ s(e_1\ e_2) &= 1 + s(e_1) + s(e_2) & s(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) &= 1 + s(e_1) + s(e_2) \end{aligned}$$ □

For any natural number n, we define a typing relation that can only type expressions which are no larger than n. It uses the same rules we have presented before (except for the restriction on size) plus the rule $[\mathrm{T}^{\mathrm{n}}_{\lambda\neg}]$ for negations of arrows. This last rule can only be applied when n is positive, and it has a side condition referring to the typing relation for a strictly smaller size.

3.8 DEFINITION: For any natural number n,

- the *n-th size-indexed typing relation* $\Gamma \vdash_n e : t$ is defined by the rules in Figure 3.3;
- the relation $\not\in_n$ between closed λ-abstractions and closed arrow types is defined by

$$\begin{aligned} \lambda x.\, e \not\in_n t' \to t \stackrel{\text{def}}{\iff} \forall\{\,(t'_i, t_i) \mid i \in I\,\}.\ & \\ \big(\forall i \in I.\ \ x : t'_i \vdash_n e : t_i\big) \implies \textstyle\bigwedge_{i\in I} t'_i \to t_i \not\leq t' \to t & \end{aligned}$$

  (where I must be non-empty and finite). □

The definition is by induction on n. For $n < 2$, the rule $[\mathrm{T}^{\mathrm{n}}_{\lambda\neg}]$ is not applicable, because $s(\lambda x.\, e) \geq 2$. For $n \geq 2$, the rule is applicable and the relation $\not\in_{s(e)}$ is well defined, since it relies of the size-indexed typing relation for $s(e)$, which is strictly less than n.

Defining an infinite family of type systems could seem a cumbersome technique; however, the systems are all so similar that relating them is very simple. The following properties hold.

3.9 LEMMA: If $\Gamma \vdash_n e : t$, then $s(e) \leq n$. □

3 Chugh, Rondon, and Jhala (2012) use a similar stratification technique in a type system with refinement types where refinement predicates include typing judgments and negation (which introduces a circularity similar to ours).

$$[\mathrm{T}^{\mathrm{n}}_{x}]\ \frac{}{\Gamma \vdash_n x\colon t[\vec{t}/\vec{\alpha}]}\ \begin{cases}\Gamma(x) = \forall\vec{\alpha}.\, t\\ s(x) \le n\end{cases} \qquad [\mathrm{T}^{\mathrm{n}}_{c}]\ \frac{}{\Gamma \vdash_n c\colon b_c}\ s(c) \le n$$

$$[\mathrm{T}^{\mathrm{n}}_{\lambda}]\ \frac{\Gamma, x\colon t' \vdash_n e\colon t}{\Gamma \vdash_n \lambda x.\, e\colon t' \to t}\ s(\lambda x.\, e) \le n \qquad [\mathrm{T}^{\mathrm{n}}_{\mathrm{app}}]\ \frac{\Gamma \vdash_n e_1\colon t' \to t \qquad \Gamma \vdash_n e_2\colon t'}{\Gamma \vdash_n e_1\ e_2\colon t}\ s(e_1\ e_2) \le n$$

$$[\mathrm{T}^{\mathrm{n}}_{\mathrm{pair}}]\ \frac{\Gamma \vdash_n e_1\colon t_1 \qquad \Gamma \vdash_n e_2\colon t_2}{\Gamma \vdash_n (e_1, e_2)\colon t_1 \times t_2}\ s((e_1, e_2)) \le n \qquad [\mathrm{T}^{\mathrm{n}}_{\mathrm{proj}}]\ \frac{\Gamma \vdash_n e\colon t_1 \times t_2}{\Gamma \vdash_n \pi_i\ e\colon t_i}\ s(\pi_i\ e) \le n$$

$$[\mathrm{T}^{\mathrm{n}}_{\mathrm{case}}]\ \frac{\begin{array}{c}\Gamma \vdash_n e_0\colon t_0\\ \text{either } t_0 \le \neg\mathbf{t} \text{ or } \Gamma \vdash_n e_1\colon t\\ \text{either } t_0 \le \mathbf{t} \text{ or } \Gamma \vdash_n e_2\colon t\end{array}}{\Gamma \vdash_n (e_0 \in \mathbf{t}\ ?\ e_1 : e_2)\colon t}\ s(e_0 \in \mathbf{t}\ ?\ e_1 : e_2) \le n$$

$$[\mathrm{T}^{\mathrm{n}}_{\mathsf{let}}]\ \frac{\Gamma \vdash_n e_1\colon t_1 \qquad \Gamma, x\colon \forall\vec{\alpha}.\, t_1 \vdash_n e_2\colon t}{\Gamma \vdash_n \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\colon t}\ \begin{cases}\vec{\alpha} \sharp \Gamma\\ s(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) \le n\end{cases}$$

$$[\mathrm{T}^{\mathrm{n}}_{\le}]\ \frac{\Gamma \vdash_n e\colon t'}{\Gamma \vdash_n e\colon t}\ \begin{cases}t' \le t\\ s(e) \le n\end{cases} \qquad [\mathrm{T}^{\mathrm{n}}_{\wedge}]\ \frac{\Gamma \vdash_n e\colon t_1 \qquad \Gamma \vdash_n e\colon t_2}{\Gamma \vdash_n e\colon t_1 \wedge t_2}\ s(e) \le n$$

$$[\mathrm{T}^{\mathrm{n}}_{\lambda\neg}]\ \frac{x\colon t'_1 \vdash_n e\colon t_1}{\Gamma \vdash_n \lambda x.\, e\colon \neg(t' \to t)}\ \begin{cases}\lambda x.\, e \text{ closed}\\ t' \to t \text{ closed}\\ s(\lambda x.\, e) \le n\\ \lambda x.\, e\ \not\in_{s(e)}\ t' \to t\end{cases}$$

FIGURE 3.3 $\mathcal{T}^{\mathrm{n}}$: Size-indexed typing rules

Proof: Immediate, because all the rules require $s(e) \leq n$. □

3.10 LEMMA: If $\Gamma \vdash_n e : t$, then $\Gamma \vdash_{n'} e : t$ holds for all $n' \geq s(e)$. □

Proof: By induction on the derivation of $\Gamma \vdash_n e : t$ and by case analysis on the last rule applied. All cases are straightforward. □

Finally, we define the typing relation we wanted to define previously, using the rules of Figure 3.2 plus a rule for negations of arrows that relies on the size-indexed systems.

3.11 DEFINITION: The typing relation $\Gamma \vdash e : t$ is defined by the rules of Figure 3.2 plus the following rule:

$$[\mathrm{T}_{\lambda\neg}]\ \frac{x : t_1' \vdash e : t_1}{\Gamma \vdash \lambda x.\, e : \neg(t' \to t)} \begin{cases} \lambda x.\, e \text{ closed} \\ t' \to t \text{ closed} \\ \lambda x.\, e \not\in_{s(e)} t' \to t \end{cases}$$

□

We write $\mathcal{T}^{\lambda\neg}$ to refer to this typing relation and the inference system it is defined from. (A list of the inference systems used throughout the thesis can be found on page 21.)

This is the relation for which we will prove soundness by progress and (a restricted form of) subject reduction. However, soundness will also hold for $\mathcal{T}$ (the system without $[\mathrm{T}_{\lambda\neg}]$), since $\mathcal{T}$ allows fewer derivations than $\mathcal{T}^{\lambda\neg}$.

We can relate this typing relation to the stratified systems as follows.

3.12 LEMMA: $\Gamma \vdash e : t \iff \big(\exists n.\ \Gamma \vdash_n e : t\big)$. □

Proof: Both directions are shown easily by induction on the typing derivations and by case analysis on the last rule applied. □

We now proceed to develop the needed lemmas in order to show progress (Lemma 3.31), subject reduction (Lemma 3.32), and finally soundness as a corollary of the two (Corollary 3.33). Throughout the rest of this chapter, we always consider the typing relation $\mathcal{T}^{\lambda\neg}$.

3.3.4 Substitution and weakening properties

We begin the proof of soundness by showing some standard properties of the typing relation.

3.13 LEMMA (Stability under type substitutions): If $\Gamma \vdash e : t$, then $\Gamma\sigma \vdash e : t\sigma$ for any type substitution σ. □

Proof: By induction on the derivation of $\Gamma \vdash e : t$ and by case analysis on the last rule applied.

Case: $[\mathrm{T}_x]$
We have $\Gamma(x) = \forall \vec{\alpha}.\, t_x$ and $t = t_x[\vec{t}/\vec{\alpha}]$.
By α-renaming, we can assume $\vec{\alpha} \sharp \sigma$. Then, $(\Gamma\sigma)(x) = \forall \vec{\alpha}.\, t_x\sigma$.
By $[\mathrm{T}_x]$, we derive $\Gamma\sigma \vdash x : t_x\sigma[\vec{t}\sigma/\vec{\alpha}]$.
We have $t_x\sigma[\vec{t}\sigma/\vec{\alpha}] = t_x[\vec{t}/\vec{\alpha}]\sigma$.

Case: $[\mathrm{T}_c]$ Straightforward.

Case: $[\mathrm{T}_\lambda]$, $[\mathrm{T}_{\mathrm{app}}]$, $[\mathrm{T}_{\mathrm{pair}}]$, $[\mathrm{T}_{\mathrm{proj}}]$, $[\mathrm{T}_\wedge]$ Direct application of the IH.

Case: $[\mathrm{T}_{\mathrm{case}}]$
We have:

$$\Gamma \vdash (e_0 \in \mathbf{t}\ ?\ e_1 : e_2) : t \qquad \Gamma \vdash e_0 : t_0$$
$$\text{if } t_0 \not\leq \neg\mathbf{t} \text{ then } \Gamma \vdash e_1 : t \qquad \text{if } t_0 \not\leq \mathbf{t} \text{ then } \Gamma \vdash e_2 : t\ .$$

By IH, we have $\Gamma\sigma \vdash e_0 : t_0\sigma$.
Note that $\mathbf{t}$ is ground, therefore $\mathbf{t}\sigma = \mathbf{t}$.
If $t_0\sigma \not\leq \neg\mathbf{t}$, then we have $t_0 \not\leq \neg\mathbf{t}$ by the contrapositive of Proposition 2.11.
Then, by IH, we have $\Gamma\sigma \vdash e_1 : t\sigma$. Similarly, if $t_0\sigma \not\leq \mathbf{t}$, we have $\Gamma\sigma \vdash e_2 : t$.
Therefore, by $[\mathrm{T}_{\mathrm{case}}]$, we have $\Gamma\sigma \vdash (e_0 \in \mathbf{t}\ ?\ e_1 : e_2) : t\sigma$.

Case: $[\mathrm{T}_{\mathsf{let}}]$
We have:

$$\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : t$$
$$\text{Ⓐ}\ \ \Gamma \vdash e_1 : t_1 \qquad \text{Ⓑ}\ \ \Gamma, x : \forall \vec{\alpha}.\, \vec{t} \vdash e_2 : t \qquad \vec{\alpha} \sharp \Gamma\ .$$

We choose $\vec{\beta}$ such that $\vec{\beta} \sharp \Gamma, \sigma$, and we define $\rho = [\vec{\beta}/\vec{\alpha}]$.
By IH (applying $\sigma \circ \rho$ to Ⓐ and σ to Ⓑ), we have

$$\Gamma\rho\sigma \vdash e_1 : t_1\rho\sigma \qquad \Gamma\sigma, x : (\forall \vec{\alpha}.\, t_1)\sigma \vdash e_2 : t\sigma\ .$$

We have $\Gamma\rho\sigma = \Gamma\sigma$, since $\vec{\alpha} \sharp \Gamma$.
We have $(\forall \vec{\alpha}.\, t_1)\sigma = (\forall \vec{\beta}.\, t_1\rho\sigma)$ by α-renaming, since $\vec{\beta} \sharp \sigma$. Therefore:

$$\Gamma\sigma \vdash e_1 : t_1\rho\sigma \qquad \Gamma\sigma, x : \forall \vec{\beta}.\, t_1\rho\sigma \vdash e_2 : t\sigma \qquad \vec{\beta} \sharp \Gamma\sigma\ .$$

We conclude by $[\mathrm{T}_{\mathsf{let}}]$.

Case: $[\mathrm{T}_\leq]$
We have $\Gamma \vdash e : t'$ with $t' \leq t$.
By IH, we have $\Gamma\sigma \vdash e : t'\sigma$. By Proposition 2.11, $t'\sigma \leq t\sigma$. We conclude by $[\mathrm{T}_\leq]$.

Case: $[\mathrm{T}_{\lambda\neg}]$
We have $\Gamma \vdash \lambda x.\, e_1 : \neg(t_1' \to t_1)$.
We derive $\Gamma\sigma \vdash \lambda x.\, e_1 : \neg(t_1' \to t_1)\sigma$ by $[\mathrm{T}_{\lambda\neg}]$: the side conditions of $[\mathrm{T}_{\lambda\neg}]$ do not mention Γ, and $\neg(t_1' \to t_1)\sigma = \neg(t_1' \to t_1)$ because the type is closed. □

We define an order of generality on type schemes according to instantiation and subtyping.

3.14 DEFINITION: A type scheme $\forall\vec{\alpha}_1.\, t_1$ is *more general than* a type scheme $\forall\vec{\alpha}_2.\, t_2$ – written $(\forall\vec{\alpha}_1.\, t_1) \leq^{\forall} (\forall\vec{\alpha}_2.\, t_2)$ – if for every type substitution $[\vec{t}_2/\vec{\alpha}_2]$ there exists a type substitution $[\vec{t}_1/\vec{\alpha}_1]$ such that $t_1[\vec{t}_1/\vec{\alpha}_1] \leq t_2[\vec{t}_2/\vec{\alpha}_2]$.

A type environment Γ_1 is *more general than* a type environment Γ_2 – written $\Gamma_1 \leq^{\forall} \Gamma_2$ – if, for all $x \in \mathsf{dom}(\Gamma_2)$, we have $x \in \mathsf{dom}(\Gamma_1)$ and $\Gamma_1(x) \leq^{\forall} \Gamma_2(x)$. □

The following lemma gives an alternative characterization of the relation.

3.15 LEMMA: Let $\forall\vec{\alpha}_1.\, t_1$ and $\forall\vec{\alpha}_2.\, t_2$ be two type schemes such that $\vec{\alpha}_2 \sharp t_1$. Then, $(\forall\vec{\alpha}_1.\, t_1) \leq^{\forall} (\forall\vec{\alpha}_2.\, t_2)$ holds if and only if there exists a type substitution $[\vec{t}/\vec{\alpha}_1]$ such that $t_1[\vec{t}/\vec{\alpha}_1] \leq t_2$. □

Proof: If $(\forall\vec{\alpha}_1.\, t_1) \leq^{\forall} (\forall\vec{\alpha}_2.\, t_2)$, then we have $\exists[\vec{t}_1/\vec{\alpha}_1].t_1[\vec{t}_1/\vec{\alpha}_1] \leq t_2$ by applying the definition of $\leq^{\forall}$ for the identity substitution $[\vec{\alpha}_2/\vec{\alpha}_2]$.

For the other direction, if there exists $[\vec{t}/\vec{\alpha}_1]$ such that $t_1[\vec{t}/\vec{\alpha}_1] \leq t_2$, then given $[\vec{t}_2/\vec{\alpha}_2]$ we take the substitution $[(\vec{t}[\vec{t}_2/\vec{\alpha}_2])/\vec{\alpha}_1]$. Since $\vec{\alpha}_2 \sharp t_1$, we have $t_1[(\vec{t}[\vec{t}_2/\vec{\alpha}_2])/\vec{\alpha}_1] = t_1[\vec{t}/\vec{\alpha}_1][\vec{t}_2/\vec{\alpha}_2]$. Moreover, $t_1[\vec{t}/\vec{\alpha}_1][\vec{t}_2/\vec{\alpha}_2] \leq t_2[\vec{t}_2/\vec{\alpha}_2]$. □

3.16 LEMMA (Weakening): If $\Gamma_2 \vdash e : t$ and $\Gamma_1 \leq^{\forall} \Gamma_2$, then $\Gamma_1 \vdash e : t$. □

Proof: By induction on the derivation of $\Gamma_2 \vdash e : t$ and by case analysis on the last rule applied.

Case: $[\mathrm{T}_x]$
Since t is an instance of $\Gamma_2(x)$, by definition of $\Gamma_1 \leq^{\forall} \Gamma_2$ there is an instance t' of $\Gamma_1(x)$ such that $t' \leq t$.
We apply $[\mathrm{T}_x]$ and $[\mathrm{T}_{\leq}]$ to derive $\Gamma_1 \vdash x : t$.

Case: $[\mathrm{T}_c]$, $[\mathrm{T}_{\lambda\neg}]$
Immediate, because the environment is not used in the rules.

Case: $[\mathrm{T}_{\lambda}]$, $[\mathrm{T}_{\mathrm{app}}]$, $[\mathrm{T}_{\mathrm{pair}}]$, $[\mathrm{T}_{\mathrm{proj}}]$, $[\mathrm{T}_{\mathrm{case}}]$, $[\mathrm{T}_{\leq}]$, $[\mathrm{T}_{\wedge}]$
Straightforward by IH.

Case: $[\mathrm{T}_{\mathsf{let}}]$
We have:

$$\text{Ⓐ}\ \Gamma_2 \vdash e_1 : t_1 \qquad \text{Ⓑ}\ \Gamma_2, x : \forall\vec{\alpha}.\, t_1 \vdash e_2 : t \qquad \text{Ⓒ}\ \vec{\alpha} \sharp \Gamma_2$$

We choose $\vec{\beta}$ such that $\vec{\beta} \sharp \Gamma_1$ and let $\rho = [\vec{\beta}/\vec{\alpha}]$. The type schemes $\forall\vec{\alpha}.\, t_1$

and $\forall\vec{\beta}.\, t_1\rho$ are equivalent by α-renaming.
From Ⓐ by Lemma 3.13 we have Ⓓ $\Gamma_2\rho \vdash e_1 : t_1\rho$.
We have $\Gamma_2\rho = \Gamma_2$ by Ⓒ.
By IH from Ⓓ and Ⓑ (using $\Gamma_1, x : \forall\vec{\beta}.\, t_1\rho \leq^{\forall} \Gamma_2, x : \forall\vec{\alpha}.\, t_1$) we have

$$\Gamma_1 \vdash e_1 : t_1\rho \qquad \Gamma_1, x : \forall\vec{\beta}.\, t_1\rho \vdash e_2 : t$$

and we conclude by $[\mathrm{T}_{\mathsf{let}}]$. □

We prove two further lemmas concerning the type environment. The first is that we can remove useless bindings from an environment while preserving typing. We denote as $\Gamma\setminus\vec{x}$ the restriction of Γ to the variables not in $\vec{x}$.

3.17 LEMMA: If $\Gamma \vdash e : t$ and if no variable in $\vec{x}$ occurs free in e, then we have $\Gamma\setminus\vec{x} \vdash e : t$. □

Proof: By induction on the derivation of $\Gamma \vdash e : t$ and by case analysis on the last rule applied. All cases are straightforward. □

Finally, we have a standard property allowing substitution of values for variables.

3.18 LEMMA: If $\Gamma, x : \forall\vec{\alpha}.\, t' \vdash e : t$ and $\Gamma \vdash v : t'$, then $\Gamma \vdash e[v/x] : t$. □

Proof: By induction on the derivation of $\Gamma, x : \forall\vec{\alpha}.\, t' \vdash e : t$ and by case analysis on the last rule applied.

Case: $[\mathrm{T}_x]$
We have either $e = x$ or $e = y$ with $y \neq x$.
In the former case, we have $\Gamma, x : \forall\vec{\alpha}.\, t' \vdash x : t$ with $t = t'[\vec{t}/\vec{\alpha}]$. Since $x[v/x] = v$, we must derive $\Gamma \vdash v : t'[\vec{t}/\vec{\alpha}]$ to conclude.
We have $\Gamma \vdash v : t'$ by hypothesis.
By Lemma 3.17, $\varnothing \vdash v : t'$ (note that values have no free variables).
By Lemma 3.13, $\varnothing \vdash v : t'[\vec{t}/\vec{\alpha}]$. By Lemma 3.16, $\Gamma \vdash v : t'[\vec{t}/\vec{\alpha}]$.
In the latter case, we have $\Gamma, x : \forall\vec{\alpha}.\, t' \vdash y : t$ and we must derive $\Gamma \vdash y : t$, which holds because $(\Gamma, x : \forall\vec{\alpha}.\, t')(y) = \Gamma(y)$.

Case: $[\mathrm{T}_c]$, $[\mathrm{T}_{\lambda\neg}]$ Straightforward.

Case: $[\mathrm{T}_\lambda]$, $[\mathrm{T}_{\mathsf{let}}]$
The three cases are analogous: we consider the first. We have

$$\Gamma, x : \forall\vec{\alpha}.\, t' \vdash \lambda y.\, e : t_1 \to t_2 \qquad \text{Ⓐ}\ \Gamma, x : \forall\vec{\alpha}.\, t', y : t_1 \vdash e : t_2$$

and we must derive $\Gamma \vdash (\lambda y.\, e)[v/x] : t_1 \to t_2$.
We can assume $y \neq x$ by α-renaming. Then, $(\lambda y.\, e)[v/x] = \lambda y.\, (e[v/x])$ and $(\Gamma, x : \forall\vec{\alpha}.\, t', y : t_1) = (\Gamma, y : t_1, x : \forall\vec{\alpha}.\, t')$. By IH from Ⓐ we have $\Gamma, y : t_1 \vdash e[v/x] : t_2$. We obtain the result by $[\mathrm{T}_\lambda]$.

Case: $[\mathrm{T}_{\mathrm{app}}]$, $[\mathrm{T}_{\mathrm{pair}}]$, $[\mathrm{T}_{\mathrm{proj}}]$, $[\mathrm{T}_{\mathrm{case}}]$, $[\mathrm{T}_{\leq}]$, $[\mathrm{T}_{\wedge}]$ Straightforward by IH.

Case: $[\mathrm{T}_{\lambda\neg}]$

Straightforward because the environment is not used in the side conditions and because the expression is closed (and hence unaffected by $[v/x]$). □

3.3.5 Inversion of the typing relation

We now develop results on the inversion of the typing relation: that is, we show how, given a judgment $\Gamma \vdash e : t$, we can derive judgments for the sub-terms of e. We will use these results in Section 3.3.6 to study which values belong to a ground type and prove the property (⋆) that we have identified as necessary at the beginning of Section 3.3.1.

To study inversion, we give a different characterization of the type system which is syntax-directed, with one rule for each shape of expression.

3.19 DEFINITION (Syntax-directed typing rules): The relation $\Gamma \vdash_{\mathsf{sd}} e : t$ is defined inductively by the following rules.

$$\frac{}{\Gamma \vdash_{\mathsf{sd}} x : t} \begin{cases} \Gamma(x) = \forall\vec{\alpha}.\, t' \\ \bigwedge_{i\in I} t'[\vec{t}_i/\vec{\alpha}] \leq t \end{cases} \qquad \frac{}{\Gamma \vdash_{\mathsf{sd}} c : t}\ b_c \leq t$$

$$\frac{\forall i \in I.\ \ \Gamma, x : t'_i \vdash_{\mathsf{sd}} e : t_i}{\Gamma \vdash_{\mathsf{sd}} \lambda x.\, e : t} \begin{cases} \bigwedge_{i\in I} t'_i \to t_i \wedge \bigwedge_{j\in J} \neg(t'_j \to t_j) \leq t \\ \forall j \in J.\ \ \lambda x.\, e \not\simeq_{s(e)} t'_j \to t_j \\ I \neq \emptyset \\ J = \emptyset \text{ or } \lambda x.\, e \text{ closed} \\ \forall j \in J.\ \ t'_j \to t_j \text{ ground} \end{cases}$$

$$\frac{\Gamma \vdash_{\mathsf{sd}} e_1 : t' \to t \qquad \Gamma \vdash_{\mathsf{sd}} e_2 : t'}{\Gamma \vdash_{\mathsf{sd}} e_1\ e_2 : t}$$

$$\frac{\Gamma \vdash_{\mathsf{sd}} e_1 : t_1 \qquad \Gamma \vdash_{\mathsf{sd}} e_2 : t_2}{\Gamma \vdash_{\mathsf{sd}} (e_1, e_2) : t}\ t_1 \times t_2 \leq t \qquad \frac{\Gamma \vdash_{\mathsf{sd}} e : t_1 \times t_2}{\Gamma \vdash_{\mathsf{sd}} \pi_i\ e : t_i}$$

$$\frac{\Gamma \vdash_{\mathsf{sd}} e_0 : t_0 \qquad t_0 \leq \neg\mathbf{t} \text{ or } \Gamma \vdash_{\mathsf{sd}} e_1 : t \qquad t_0 \leq \mathbf{t} \text{ or } \Gamma \vdash_{\mathsf{sd}} e_2 : t}{\Gamma \vdash_{\mathsf{sd}} (e_0 \in \mathbf{t}\ ?\ e_1 : e_2) : t}$$

$$\frac{\Gamma \vdash_{\mathsf{sd}} e_1 : t_1 \qquad \Gamma, x : \forall\vec{\alpha}.\, t_1 \vdash_{\mathsf{sd}} e_2 : t}{\Gamma \vdash_{\mathsf{sd}} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : t}\ \vec{\alpha} \sharp \Gamma$$

(In the first and third rules, the variables that appear only in the side conditions are implicitly existentially quantified.) □

The system is obtained by embedding the uses of $[\mathrm{T}_{\leq}]$ in the rules for variables, constants, functions, and pairs and the uses of $[\mathrm{T}_{\wedge}]$ in the rules for variables and functions. We prove that the two rules $[\mathrm{T}_{\leq}]$ and $[\mathrm{T}_{\wedge}]$ are admissible in this system.

3.20 LEMMA: If $\Gamma \vdash_{\mathsf{sd}} e : t'$ and $t' \leq t$, then $\Gamma \vdash_{\mathsf{sd}} e : t$. □

Proof: By induction on the derivation of $\Gamma \vdash_{\mathsf{sd}} e : t'$ and by case analysis on the last rule applied. All cases are straightforward. □

3.21 LEMMA: If $\Gamma \vdash_{\mathsf{sd}} e : t_1$ and $\Gamma \vdash_{\mathsf{sd}} e : t_2$, then $\Gamma \vdash_{\mathsf{sd}} e : t_1 \wedge t_2$. □

Proof: By structural induction on e and by case analysis on the shape of e. Since all the rules are syntax-directed, if we know the shape of e, we also know the last rule applied in both derivations.

Case: $e = x$
Immediate: we just apply the rule to type x with an intersection containing all instantiations used in both derivations.

Case: $e = c$
Immediate: if $b_c \leq t_1$ and $b_c \leq t_2$, then $b_c \leq t_1 \wedge t_2$.

Case: $e = \lambda x.\, e'$
We have:

$$\bigwedge_{i \in I_1} (t'_i \to t_i) \wedge \bigwedge_{j \in J_1} \neg(t'_j \to t_j) \leq t_1$$
$$\forall i \in I_1.\ \Gamma, x : t'_i \vdash_{\mathsf{sd}} e' : t_i \qquad \forall j \in J_1.\ \lambda x.\, e' \not\in_{s(e')} t'_j \to t_j$$
$$I_1 \neq \varnothing \qquad J_1 = \varnothing \text{ or } \lambda x.\, e' \text{ closed} \qquad \forall j \in J_1.\ t'_j \to t_j \text{ ground}$$
$$\bigwedge_{i \in I_2} (t'_i \to t_i) \wedge \bigwedge_{j \in J_2} \neg(t'_j \to t_j) \leq t_2$$
$$\forall i \in I_2.\ \Gamma, x : t'_i \vdash_{\mathsf{sd}} e' : t_i \qquad \forall j \in J_2.\ \lambda x.\, e' \not\in_{s(e')} t'_j \to t_j$$
$$I_2 \neq \varnothing \qquad J_2 = \varnothing \text{ or } \lambda x.\, e' \text{ closed} \qquad \forall j \in J_2.\ t'_j \to t_j \text{ ground}$$

Therefore we have

$$\bigwedge_{i \in I_1 \cup I_2} (t'_i \to t_i) \wedge \bigwedge_{j \in J_1 \cup J_2} \neg(t'_j \to t_j) \leq t_1 \wedge t_2$$
$$\forall i \in I_1 \cup I_2.\ \Gamma, x : t'_i \vdash_{\mathsf{sd}} e' : t_i \qquad \forall j \in J_1 \cup J_2.\ \lambda x.\, e' \not\in_{s(e')} t'_j \to t_j$$
$$I_1 \cup I_2 \neq \varnothing \quad J_1 \cup J_2 = \varnothing \text{ or } \lambda x.\, e' \text{ closed} \quad \forall j \in J_1 \cup J_2.\ t'_j \to t_j \text{ ground}$$

and we conclude using the rule for λ-abstraction.

Case: $e = e_1\, e_2$
We have:

$$\Gamma \vdash_{\mathsf{sd}} e_1 : t'_1 \to t_1 \qquad \Gamma \vdash_{\mathsf{sd}} e_2 : t'_1 \qquad \Gamma \vdash_{\mathsf{sd}} e_1 : t'_2 \to t_2 \qquad \Gamma \vdash_{\mathsf{sd}} e_2 : t'_2$$

By IH, we have

$$\Gamma \vdash_{\mathsf{sd}} e_1 : (t'_1 \to t_1) \wedge (t'_2 \to t_2) \qquad \Gamma \vdash_{\mathsf{sd}} e_2 : t'_1 \wedge t'_2$$

and, since $(t'_1 \to t_1) \wedge (t'_2 \to t_2) \leq (t'_1 \wedge t'_2) \to (t_1 \wedge t_2)$, by Lemma 3.20 we have $\Gamma \vdash_{\mathsf{sd}} e_1 : (t'_1 \wedge t'_2) \to (t_1 \wedge t_2)$. We conclude using the rule for applications.

Case: $e = (e_1, e_2)$, $e = \pi_i\, e'$, or $e = (e_0 \in \mathbf{t}\ ?\ e_1 : e_2)$
Similar to the previous cases.

Case: $e = (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)$
We have:

$$\Gamma \vdash_{\mathsf{sd}} e_1 \colon t_1' \qquad \Gamma, x \colon \forall \vec{\alpha}_1.\, t_1' \vdash_{\mathsf{sd}} e_2 \colon t_1 \qquad \vec{\alpha}_1 \sharp \Gamma$$
$$\Gamma \vdash_{\mathsf{sd}} e_1 \colon t_2' \qquad \Gamma, x \colon \forall \vec{\alpha}_2.\, t_2' \vdash_{\mathsf{sd}} e_2 \colon t_2 \qquad \vec{\alpha}_2 \sharp \Gamma$$

By IH we have $\Gamma \vdash_{\mathsf{sd}} e_1 \colon t_1' \wedge t_2'$.
We have $(\forall \vec{\alpha}_1, \vec{\alpha}_2.\, t_1' \wedge t_2') \leq^{\forall} (\forall \vec{\alpha}_i.\, t_i')$ for both i.
Hence, by Lemma 3.16, we have $\Gamma, x \colon \forall \vec{\alpha}_1, \vec{\alpha}_2.\, t_1' \wedge t_2' \vdash_{\mathsf{sd}} e_2 \colon t_i$ for both i.
By IH we obtain $\Gamma, x \colon \forall \vec{\alpha}_1, \vec{\alpha}_2.\, t_1' \wedge t_2' \vdash_{\mathsf{sd}} e_2 \colon t_1 \wedge t_2$, and we conclude using the rule for let. □

Now, we prove that the syntax-directed typing relation is equivalent to the relation of Definition 3.11. We will use this result to invert typing judgments $\Gamma \vdash e \colon t$ and derive judgments on the subterms of e.

3.22 LEMMA: $\Gamma \vdash_{\mathsf{sd}} e \colon t$ holds if and only if $\Gamma \vdash e \colon t$. □

Proof: Both implications are proved easily by induction on the derivation and by case analysis on the last rule applied.

To prove that $\Gamma \vdash_{\mathsf{sd}} e \colon t$ implies $\Gamma \vdash e \colon t$, in all cases we obtain $\Gamma \vdash e \colon t$ from the judgments obtained by IH from the premises of $\Gamma \vdash_{\mathsf{sd}} e \colon t$, applying the rules specific to the shape of e (both $[\mathrm{T}_\lambda]$ and $[\mathrm{T}_{\lambda\neg}]$ for λ-abstractions) plus $[\mathrm{T}_\leq]$ and $[\mathrm{T}_\wedge]$.

To prove that $\Gamma \vdash e \colon t$ implies $\Gamma \vdash_{\mathsf{sd}} e \colon t$, if the last rule applied is $[\mathrm{T}_\leq]$, we apply the IH and Lemma 3.20; if it is $[\mathrm{T}_\wedge]$, we apply the IH and Lemma 3.21; in all other cases, we apply the IH and then the rule corresponding to the shape of e. □

3.3.6 Relating ground types and sets of values

Now we establish some results relating sets of values in different ground types. These results show that (as far as ground types are concerned) union, intersection, and negation types correspond to the set-theoretic notions: for instance, Lemma 3.26 proves that the values in a union of ground types $t_1 \vee t_2$ are exactly those in t_1 and those in t_2. We map types to sets of values using the function $\mathcal{V}(t) \stackrel{\text{def}}{=} \{\, v \mid \varnothing \vdash v \colon t \,\}$. First, we check that the empty type $\mathbb{0}$ is actually uninhabited.

3.23 LEMMA: $\mathcal{V}(\mathbb{0}) = \varnothing$. □

Proof: We show that $\varnothing \vdash v \colon t$ implies $t \not\leq \mathbb{0}$, by induction on v and using Lemma 3.22.

Case: $v = c$ We have $b_c \leq t$ and b_c is not empty: therefore $t \not\leq \mathbb{0}$.

Case: $v = \lambda x.\, e$

We have $\bigwedge_{i\in I} t'_i \to t_i \wedge \bigwedge_{j\in J} \neg(t'_j \to t_j) \leq t$.
We show that, for all $j \in J$, $\bigwedge_{i\in I} t'_i \to t_i \not\leq t'_j \to t_j$.
For all $i \in I$, we have $x\colon t'_i \vdash e\colon t_i$.
By Lemmas 3.10 and 3.12, we have $x\colon t'_i \vdash_{s(e)} e\colon t_i$.
For all $j \in J$, we have $\lambda x.\, e \not\in_{s(e)} t'_j \to t_j$. By definition, this implies that $\bigwedge_{i\in I} t'_i \to t_i \not\leq t'_j \to t_j$.
By the contrapositive of Corollary 2.17, we have $\bigwedge_{i\in I} t'_i \to t_i \not\leq \bigvee_{j\in J} t'_j \to t_j$, which is $\bigwedge_{i\in I} t'_i \to t_i \wedge \bigwedge_{j\in J} \neg(t'_j \to t_j) \not\leq \mathbb{0}$.

Case: $v = (v_1, v_2)$
We have $\varnothing \vdash v_1\colon t_1$, $\varnothing \vdash v_2\colon t_2$, and $t_1 \times t_2 \leq t$.
By IH, $t_1 \not\leq \mathbb{0}$ and $t_2 \not\leq \mathbb{0}$: therefore, $t_1 \times t_2 \not\leq \mathbb{0}$. □

We show that the values in a ground intersection type $t_1 \wedge t_2$ are exactly those in both t_1 and t_2.

3.24 LEMMA: Let t_1 and t_2 be ground types. Then, $\mathcal{V}(t_1 \wedge t_2) = \mathcal{V}(t_1) \cap \mathcal{V}(t_2)$. □

Proof: If $\varnothing \vdash v\colon t_1 \wedge t_2$, then by $[\mathrm{T}_{\leq}]$ we have $\varnothing \vdash v\colon t_1$ and $\varnothing \vdash v\colon t_2$
Conversely, if $\varnothing \vdash v\colon t_1$ and $\varnothing \vdash v\colon t_2$, then $\varnothing \vdash v\colon t_1 \wedge t_2$ holds by $[\mathrm{T}_{\wedge}]$. □

Now, we prove that all well-typed values – that is, values in $\mathcal{V}(\mathbb{1})$ – are either in $\mathcal{V}(t)$ or $\mathcal{V}(\neg t)$, for any ground type t. This is the result ($\star$) that we stated in Section 3.3.1.

3.25 LEMMA: Let t be a ground type. Then, $\mathcal{V}(\neg t) = \mathcal{V}(\mathbb{1}) \setminus \mathcal{V}(t)$. □

Proof: Using the previous two results, we have that, if $v \in \mathcal{V}(t) \cap \mathcal{V}(\neg t)$, then $v \in \mathcal{V}(t \wedge \neg t)$; but then $v \in \mathcal{V}(\mathbb{0})$, which is impossible. Therefore, $\mathcal{V}(t)$ and $\mathcal{V}(\neg t)$ are disjoint.

We show that $\mathcal{V}(t) \cup \mathcal{V}(\neg t) = \mathcal{V}(\mathbb{1})$, which yields the result we need. We show this by proving, for every well-typed value v and every ground type t, that either $\varnothing \vdash v\colon t$ or $\varnothing \vdash v\colon \neg t$ holds. The proof is by induction on (v, t).

Case: $t = b$
If a well-typed value v is not a constant, it always has type $\neg b$.
If v is a constant c, it has type b_c. Since $[\![b_c]\!]$ is a singleton, it is a subset of either $[\![b]\!]$ or $[\![\neg b]\!]$: therefore, v has either type b or type $\neg b$ by $[\mathrm{T}_{\leq}]$.

Case: $t = t_1 \times t_2$
If a well-typed value v is not a pair, it always has type $\neg(t_1 \times t_2)$.
If $v = (v_1, v_2)$, then, by IH, v_1 has either type t_1 or $\neg t_1$, and v_2 has either type t_2 or $\neg t_2$. Then, v has one of these four types: $(t_1 \times t_2)$, $(\neg t_1 \times t_2)$, $(t_1 \times \neg t_2)$, or $(\neg t_1 \times \neg t_2)$. In the last three cases, by $[\mathrm{T}_{\leq}]$ it has type $\neg(t_1 \times t_2)$.

Case: $t = t_1 \to t_2$
If a well-typed value v is not a λ-abstraction, it always has type $\neg(t_1 \to t_2)$.

Otherwise, we have $v = \lambda x.\, e$. Either we can derive $\varnothing \vdash \lambda x.\, e \colon \neg(t_1 \to t_2)$ using $[\mathrm{T}_{\lambda\neg}]$ or not. In the latter case, we show $\varnothing \vdash \lambda x.\, e \colon t_1 \to t_2$.
If we cannot apply $[\mathrm{T}_{\lambda\neg}]$, then it must be either because no premise cannot be found or because one of the side conditions does not hold. The first possibility cannot actually occur because $\lambda x.\, e$ is well typed: therefore its body must be well typed under some assumption for x. Therefore, it must be that $\lambda x.\, e \not\in_{s(e)} t_1 \to t_2$ does not hold.
As a consequence, we have

$$\exists\{\, (t'_i, t_i) \mid i \in I \,\}.\ \big(\forall i \in I.\ \ x \colon t'_i \vdash_{s(e)} e \colon t_i\big) \wedge \big(\textstyle\bigwedge_{i\in I} t'_i \to t_i \leq t_1 \to t_2\big)$$

(where I is finite and non-empty). By Lemma 3.12 and by $[\mathrm{T}_\lambda]$, $[\mathrm{T}_\wedge]$, and $[\mathrm{T}_\leq]$, we obtain $\varnothing \vdash \lambda x.\, e \colon t_1 \to t_2$.

Case: $t = t_1 \vee t_2$
By IH, v has either type t_1 or $\neg t_1$, and either type t_2 or $\neg t_2$.
Therefore, either it has type $t_1 \vee t_2$ by $[\mathrm{T}_\leq]$ or it has both types $\neg t_1$ and $\neg t_2$, in which case it has type $\neg(t_1 \vee t_2)$ by $[\mathrm{T}_\wedge]$ and $[\mathrm{T}_\leq]$.

Case: $t = \neg t'$ Straightforward by IH.

Case: $t = \mathbb{0}$ Since v is well typed, it has type $\neg\mathbb{0}$ by $[\mathrm{T}_\leq]$. □

As a consequence of these results, the values in a ground union type are exactly those in at least one of the types in the union.

3.26 LEMMA: Let t_1 and t_2 be ground types. Then, $\mathcal{V}(t_1 \vee t_2) = \mathcal{V}(t_1) \cup \mathcal{V}(t_2)$. □

Proof: If $v \in \mathcal{V}(t_1)$, then $\varnothing \vdash v \colon t_1$. Then, by $[\mathrm{T}_\leq]$, $\varnothing \vdash v \colon t_1 \vee t_2$. Hence, $v \in \mathcal{V}(t_1 \vee t_2)$. Likewise if $v \in \mathcal{V}(t_2)$.
If $v \in \mathcal{V}(t_1 \vee t_2)$, then $\varnothing \vdash v \colon t_1 \vee t_2$. Since v is well typed, we have $v \in \mathcal{V}(\mathbb{1})$. By Lemma 3.25, either $\varnothing \vdash v \colon t_1$ or $\varnothing \vdash v \colon \neg t_1$ must hold. In the former case, we have $v \in \mathcal{V}(t_1)$. In the latter, since $(t_1 \vee t_2) \wedge \neg t_1 \simeq t_2$, we have $\varnothing \vdash v \colon t_2$ by $[\mathrm{T}_\wedge]$ and $[\mathrm{T}_\leq]$; hence, $v \in \mathcal{V}(t_2)$. □

3.27 COROLLARY: If $\Gamma \vdash v \colon \bigvee_{i\in I} t_i$ and if $\bigvee_{i\in I} t_i$ is ground, then there exists an $i_0 \in I$ such that $\Gamma \vdash v \colon t_{i_0}$. □

Proof: Consequence of Lemma 3.26, shown by induction on $|I|$ (note that I is necessarily finite). □

3.3.7 Progress, subject reduction, and soundness

We prove three auxiliary lemmas and then the main results of progress and subject reduction. The first lemma is a result of inversion of typing for values.

3.28 LEMMA: The following hold:

- if $\Gamma \vdash v\colon t' \to t$, then $v = \lambda x.\, e$ and there exists a non-empty intersection $\bigwedge_{i\in I} t'_i \to t_i$ such that $\bigwedge_{i\in I} t'_i \to t_i \leq t' \to t$ and that, for all $i \in I$, we have $\Gamma, x : t'_i \vdash e\colon t_i$;
- if $\Gamma \vdash v\colon t_1 \times t_2$, then $v = (v_1, v_2)$ and $\Gamma \vdash v_1\colon t_1$ and $\Gamma \vdash v_2\colon t_2$. □

Proof: Both points are consequences of Lemma 3.22.

In particular, when $\Gamma \vdash v\colon t' \to t$, by Lemma 3.22 we know that v must be of the form $\lambda x.\, e$. Then, we have $\bigwedge_{i\in I} t'_i \to t_i \wedge \bigwedge_{j\in J} \neg(t'_j \to t_j) \leq t' \to t$. However, since $\bigwedge_{i\in I} t'_i \to t_i \wedge \bigwedge_{j\in J} \neg(t'_j \to t_j)$ is not empty (by Lemma 3.23), we also have $\bigwedge_{i\in I} t'_i \to t_i \leq t' \to t$ by Corollary 2.17. □

The following lemma ensures that the evaluation of a well-typed typecase cannot get stuck.

3.29 LEMMA: For every v and $\mathbf{t}$, either $\mathsf{typeof}(v) \leq \mathbf{t}$ or $\mathsf{typeof}(v) \leq \neg\mathbf{t}$. □

Proof: By induction on the pair $(v, \mathbf{t})$ and by case analysis on the shape of $\mathbf{t}$.

Case: $\mathbf{t} = b$

If v is a function or a pair, $\mathsf{typeof}(v) \leq \neg\mathbf{t}$.

If v is a constant c, then $\mathsf{typeof}(c) = b_c$. The type b_c is a singleton type, that is, $\mathbb{B}(b_c) = \{c\}$. As a result, we have either $\mathbb{B}(b_c) \subseteq \mathbb{B}(b)$ or $\mathbb{B}(b_c) \subseteq \mathsf{Const} \setminus \mathbb{B}(b)$. This implies that either $b_c \leq b$ or $b_c \leq \neg b$.

Case: $\mathbf{t} = \mathbf{t}_1 \times \mathbf{t}_2$

If v is a constant or a function, then $\mathsf{typeof}(v) \leq \neg\mathbf{t}$.

If v is a pair (v_1, v_2), then $\mathsf{typeof}(v) = \mathsf{typeof}(v_1) \times \mathsf{typeof}(v_2)$.

By IH, we have

$$\text{either } \mathsf{typeof}(v_1) \leq \mathbf{t}_1 \text{ or } \mathsf{typeof}(v_1) \leq \neg\mathbf{t}_1$$
$$\text{either } \mathsf{typeof}(v_2) \leq \mathbf{t}_2 \text{ or } \mathsf{typeof}(v_2) \leq \neg\mathbf{t}_2\ .$$

If $\mathsf{typeof}(v_1) \leq \mathbf{t}_1$ and $\mathsf{typeof}(v_2) \leq \mathbf{t}_2$, then $\mathsf{typeof}(v) \leq \mathbf{t}$. In all other cases, $\mathsf{typeof}(v) \leq \neg\mathbf{t}$.

Case: $\mathbf{t} = \mathbb{0} \to \mathbb{1}$

If v is a constant or a pair, then $\mathsf{typeof}(v) \leq \neg\mathbf{t}$.

If v is a function, then $\mathsf{typeof}(v) = \mathbf{t}$.

Case: $\mathbf{t} = \mathbf{t}_1 \vee \mathbf{t}_2$

By IH, we have

$$\text{either } \mathsf{typeof}(v) \leq \mathbf{t}_1 \text{ or } \mathsf{typeof}(v) \leq \neg\mathbf{t}_1$$
$$\text{either } \mathsf{typeof}(v) \leq \mathbf{t}_2 \text{ or } \mathsf{typeof}(v) \leq \neg\mathbf{t}_2\ .$$

If $\mathsf{typeof}(v) \leq \mathbf{t}_1$ or $\mathsf{typeof}(v) \leq \mathbf{t}_2$, then $\mathsf{typeof}(v) \leq \mathbf{t}$.

Otherwise, we have $\mathsf{typeof}(v) \leq \neg\mathbf{t}_1$ and $\mathsf{typeof}(v) \leq \neg\mathbf{t}_2$. Then, we have $\mathsf{typeof}(v) \leq \neg\mathbf{t}_1 \wedge \neg\mathbf{t}_2$, and $\neg\mathbf{t}_1 \wedge \neg\mathbf{t}_2 \simeq \neg\mathbf{t}$.

Case: $\mathbf{t} = \neg\mathbf{t}'$ By IH.

Case: $\mathbf{t} = \mathbb{0}$ We have $\mathsf{typeof}(v) \leq \neg\mathbf{t} \simeq \mathbb{1}$. □

The next lemma proves that, for every well-typed v, $\mathsf{typeof}(v)$ is indeed a derivable type for v.

3.30 LEMMA: If $\Gamma \vdash v : t$, then $\Gamma \vdash v : \mathsf{typeof}(v)$. □

Proof: By induction on v and by case analysis on the shape of v.

Case: $v = c$ We have $\Gamma \vdash c : b_c$ by $[\mathrm{T}_c]$.

Case: $v = \lambda x.\, e$
By Lemma 3.22, we have $\Gamma, x : t' \vdash e : t''$ for some t' and t''.
Hence, by $[\mathrm{T}_\lambda]$, $\Gamma \vdash \lambda x.\, e : t' \to t''$ and, by $[\mathrm{T}_\leq]$, $\Gamma \vdash \lambda x.\, e : \mathbb{0} \to \mathbb{1}$.

Case: $v = (v_1, v_2)$
By Lemma 3.22, v_1 and v_2 are well typed. Then, by IH, we have both $\Gamma \vdash v_1 : \mathsf{typeof}(v_1)$ and $\Gamma \vdash v_2 : \mathsf{typeof}(v_2)$. We conclude by $[\mathrm{T}_{\mathrm{pair}}]$. □

Finally, we can prove progress and subject reduction.

3.31 LEMMA (Progress): Let e be a closed expression. If $\varnothing \vdash e : t$, then either e is a value or there exists an expression e' such that $e \rightsquigarrow e'$. □

Proof: By induction on the derivation of $\varnothing \vdash e : t$ and by case analysis on the last rule applied.

Case: $[\mathrm{T}_x]$ Impossible, because a variable is not closed.

Case: $[\mathrm{T}_c]$, $[\mathrm{T}_\lambda]$, $[\mathrm{T}_{\lambda\neg}]$ The expression is a value.

Case: $[\mathrm{T}_{\mathrm{app}}]$
We have $e = e_1\, e_2$, and both e_1 and e_2 are closed and well typed.
We apply the IH to both sub-expressions. If e_1 reduces, or if e_1 is a value and e_2 reduces, then e reduces by $[\mathrm{R}_{\mathrm{ctx}}]$.
Otherwise, e_1 and e_2 are both values. Then, by Lemma 3.28, since $\varnothing \vdash e_1 : t' \to t$, we have $e_1 = \lambda x.\, e'$, and e reduces by $[\mathrm{R}_{\mathrm{app}}]$.

Case: $[\mathrm{T}_{\mathrm{pair}}]$
We have $e = (e_1, e_2)$, and both e_1 and e_2 are closed and well typed.
By IH, either e_1 is a value or it reduces; in the latter case, e reduces by $[\mathrm{R}_{\mathrm{ctx}}]$.
In the former case, by IH either e_2 is a value or it reduces. If it is a value, then e is a value as well. Otherwise, it reduces by $[\mathrm{R}_{\mathrm{ctx}}]$.

Case: $[\mathrm{T}_{\mathrm{proj}}]$
We have $e = \pi_i\, e'$, and e' is closed and well typed.

Therefore, by IH, either e' is a value or it reduces.
In the latter case, e reduces by $[\mathrm{R}_{\mathrm{ctx}}]$.
In the former case, by Lemma 3.28, since $\varnothing \vdash e' \colon t_1 \times t_2$, we have $e' = (v_1, v_2)$. Then, e reduces by $[\mathrm{R}_{\mathrm{proj}}]$.

Case: $[\mathrm{T}_{\mathrm{case}}]$
We have $e = (e_0 \in \mathbf{t}\ ?\ e_1 : e_2)$, and e_0 is closed and well typed.
Therefore, by IH, either e_0 is a value or it reduces.
In the latter case, e reduces by $[\mathrm{R}_{\mathrm{ctx}}]$.
In the former case, e reduces either by $[\mathrm{R}^1_{\mathrm{case}}]$ or by $[\mathrm{R}^2_{\mathrm{case}}]$ according to whether $\mathsf{typeof}(e_0) \leq \mathbf{t}$ or $\mathsf{typeof}(e_0) \leq \neg\mathbf{t}$ holds. By Lemma 3.29, either must hold.

Case: $[\mathrm{T}_{\mathsf{let}}]$
We have $e = (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)$, and e_1 is closed and well typed.
Therefore, by IH, either e_1 is a value or it reduces.
Hence, e reduces by either $[\mathrm{R}_{\mathsf{let}}]$ or $[\mathrm{R}_{\mathrm{ctx}}]$.

Case: $[\mathrm{T}_{\leq}]$, $[\mathrm{T}_{\wedge}]$ Immediate by application of IH. □

3.32 LEMMA (Subject reduction): Let e be an expression. Let Γ be a ground type environment and t a ground type. If $\Gamma \vdash e \colon t$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' \colon t$. □

Proof: By induction on the derivation of $\Gamma \vdash e \colon t$ and by case analysis on the last typing rule applied and on the reduction rule.

Case: $[\mathrm{T}_x]$, $[\mathrm{T}_c]$, $[\mathrm{T}_\lambda]$, $[\mathrm{T}_{\lambda\neg}]$
Impossible, because such expressions cannot reduce.

Case: $[\mathrm{T}_{\leq}]$, $[\mathrm{T}_{\wedge}]$
The conclusion follows directly by IH.

Case: $[\mathrm{T}_{\mathrm{app}}]$, $[\mathrm{T}_{\mathrm{pair}}]$, $[\mathrm{T}_{\mathrm{proj}}]$, $[\mathrm{T}_{\mathrm{case}}]$, $[\mathrm{T}_{\mathsf{let}}]$ when $e \rightsquigarrow e'$ occurs by $[\mathrm{R}_{\mathrm{ctx}}]$
Straightforward by IH.

Case: $[\mathrm{T}_{\mathrm{app}}]$ when $e \rightsquigarrow e'$ occurs by $[\mathrm{R}_{\mathrm{app}}]$
We have $\Gamma \vdash v_1\, v_2 \colon t$ derived from

$$\text{Ⓐ}\ \ \Gamma \vdash v_1 \colon t' \to t \qquad \text{Ⓑ}\ \ \Gamma \vdash v_2 \colon t' \qquad \text{Ⓒ}\ \ v_1 = \lambda x.\, e_1$$

and we must show $\Gamma \vdash e_1[v_2/x] \colon t$.
From Ⓐ and Ⓒ, by Lemma 3.28, we find $\bigwedge_{i\in I} t'_i \to t_i$ such that

$$\text{Ⓓ}\ \ \bigwedge_{i\in I} t'_i \to t_i \leq t' \to t \qquad \text{Ⓔ}\ \ \forall i \in I.\ \ \Gamma, x : t'_i \vdash e_1 \colon t_i\ .$$

Let $\vec{\alpha} = \mathsf{var}(\bigwedge_{i\in I} t'_i \to t_i) \cup \mathsf{var}(t')$ and let $\sigma = [\mathbb{0}/\vec{\alpha}]$. (The choice of $\mathbb{0}$ is arbitrary: any ground type can replace it.)
By Proposition 2.11 from Ⓓ and by Lemma 3.13 from Ⓔ we have

$$\text{Ⓕ}\ \ \bigwedge_{i\in I} t'_i\sigma \to t_i\sigma \leq t'\sigma \to t \qquad \text{Ⓖ}\ \ \forall i \in I.\ \ \Gamma, x : t'_i\sigma \vdash e_1 \colon t_i\sigma$$

(note that Γ and t are ground and hence unaffected by σ).
From Ⓕ, by Lemma 2.16, we have $t'\sigma \leq \bigvee_{i\in I} t'_i\sigma$.
From Ⓑ, by Lemma 3.13, we have $\Gamma \vdash v_2 : t'\sigma$.
By $[\mathrm{T}_{\leq}]$, we have $\Gamma \vdash v_2 : \bigvee_{i\in I} t'_i\sigma$.
By Corollary 3.27, since $\bigvee_{i\in I} t'_i\sigma$ is ground, then there exists an $i_0 \in I$ such that Ⓗ $\Gamma \vdash v_2 : t'_{i_0}\sigma$.
From Ⓖ, we have Ⓘ $\Gamma, x : t'_{i_0}\sigma \vdash e_1 : t_{i_0}\sigma$.
From Ⓗ and Ⓘ, by Lemma 3.18, we have $\Gamma \vdash e_1[v_2/x] : t$.

Case: $[\mathrm{T}_{\mathrm{proj}}]$ when $e \rightsquigarrow e'$ occurs by $[\mathrm{R}_{\mathrm{proj}}]$
We have

$$\Gamma \vdash \pi_i\,(v_1, v_2) : t_i \qquad \text{Ⓐ}\ \ \Gamma \vdash (v_1, v_2) : t_1 \times t_2$$

and we must show $\Gamma \vdash v_i : t_i$.
From Ⓐ, by Lemma 3.28, we obtain $\Gamma \vdash v_1 : t_1$ and $\Gamma \vdash v_2 : t_2$, which yields the result we need.

Case: $[\mathrm{T}_{\mathrm{case}}]$ when $e \rightsquigarrow e'$ occurs by $[\mathrm{R}^1_{\mathrm{case}}]$
We have $\Gamma \vdash (v \in \mathbf{t}\ ?\ e_1 : e_2) : t$ derived from

$$\text{Ⓐ}\ \ \Gamma \vdash v : t_0 \qquad \text{Ⓑ}\ \ t_0 \leq \neg\mathbf{t} \text{ or } \Gamma \vdash e_1 : t \qquad \text{Ⓒ}\ \ t_0 \leq \mathbf{t} \text{ or } \Gamma \vdash e_2 : t$$

and we have Ⓓ $\mathsf{typeof}(v) \leq \mathbf{t}$. We must show $\Gamma \vdash e_1 : t$.
First, we derive Ⓔ $\Gamma \vdash v : t_0 \wedge \mathbf{t}$.
From Ⓐ, by Lemma 3.30, we have $\Gamma \vdash v : \mathsf{typeof}(v)$.
Applying $[\mathrm{T}_{\leq}]$, using Ⓓ, we have $\Gamma \vdash v : \mathbf{t}$.
Then, from Ⓐ and applying $[\mathrm{T}_{\wedge}]$, we have $\Gamma \vdash v : t_0 \wedge \mathbf{t}$.
We prove by contradiction that $t_0 \leq \neg\mathbf{t}$ does not hold.
Assume that $t_0 \leq \neg\mathbf{t}$ holds. Then, $t_0 \wedge \mathbf{t} \leq \mathbb{0}$.
By $[\mathrm{T}_{\leq}]$ from Ⓔ, $\Gamma \vdash v : \mathbb{0}$.
By Lemma 3.17, since v has no free variables, we have $\varnothing \vdash v : \mathbb{0}$.
Hence, $v \in \mathcal{V}(\mathbb{0})$, which is impossible by Lemma 3.23.
Since $t_0 \leq \neg\mathbf{t}$, from Ⓑ er have $\Gamma \vdash e_1 : t$.

Case: $[\mathrm{T}_{\mathrm{case}}]$ when $e \rightsquigarrow e'$ occurs by $[\mathrm{R}^2_{\mathrm{case}}]$
Analogous to the previous case.
Instead of Ⓓ, we have $\mathsf{typeof}(v) \leq \neg\mathbf{t}$. We use it to show that $t_0 \leq \mathbf{t}$ is impossible, so from Ⓒ we obtain $\Gamma \vdash e_2 : t$.

Case: $[\mathrm{T}_{\mathsf{let}}]$ when $e \rightsquigarrow e'$ occurs by $[\mathrm{R}_{\mathsf{let}}]$
We have $\Gamma \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e_2 : t$ derived from

$$\text{Ⓐ}\ \ \Gamma \vdash v : t_1 \qquad \text{Ⓑ}\ \ \Gamma, x : \forall\vec{\alpha}.\, t_1 \vdash e_2 : t \qquad \vec{\alpha} \sharp \Gamma$$

and we must show $\Gamma \vdash e_2[v/x] : t$.
We obtain it by Lemma 3.18 from Ⓐ and Ⓑ. □

3.33 COROLLARY (Type soundness): Let e be a closed expression. If $\varnothing \vdash e : t$, then either e diverges or there exists a value v such that $e \rightsquigarrow^* v$. □

Proof: Let $\sigma = [\vec{t}/\vec{\alpha}]$ where $\vec{\alpha} = \mathsf{var}(t)$ and where all types in $\vec{t}$ are ground. Then, $t\sigma$ is ground. By Lemma 3.13, we have $\varnothing \vdash e : t\sigma$.

If e does not diverge, then there exists a reduction sequence $e_0 \rightsquigarrow \cdots \rightsquigarrow e_n$ such that $e = e_0$ and that e_n does not reduce. By Lemma 3.32, we have $\varnothing \vdash e_n : t\sigma$. Then, by Lemma 3.31, e_n is a value. □

4 Type inference

This chapter studies the problem of *type inference* or *type reconstruction*[1] for the type system of the previous chapter. We consider the typing relation $\mathcal{T}$ given by the rules of Figure 3.2 and not the full typing relation $\mathcal{T}^{\lambda\neg}$ of Definition 3.11. That is, we do not attempt to infer the negation types that can be derived using $[\mathrm{T}_{\lambda\neg}]$ (we added that rule only to be able to prove type soundness).

The system $\mathcal{T}$ includes the intersection-introduction rule $[\mathrm{T}_{\wedge}]$. It is well known that intersection types can be used to define type systems that can type all and only those λ-terms that are strongly normalizable (Coppo and Dezani-Ciancaglini, 1980); as a consequence, type inference is undecidable for such systems. That result does not hold directly in our case – though $\mathcal{T}$ is not known to be decidable – since we can also type diverging terms (using recursive types). In any case, in this chapter, we will not attempt to infer intersection types: that would complicate type inference because we cannot easily know how many types we should infer and intersect for a given expression, notably for a function. Therefore, we will prove that type inference is sound with respect to $\mathcal{T}$ and that it is complete with respect to the restriction of $\mathcal{T}$ without the rule $[\mathrm{T}_{\wedge}]$.

CHAPTER OUTLINE:

Section 4.1 We describe a new declarative type system – closely based on the "reformulated rules" of Dolan and Mycroft (2017) – which is better suited to being compared to an inference algorithm. We prove a result of equivalence between $\mathcal{T}$ and the new system.

Section 4.2 We start to describe type inference, which consists of constraint generation and solving. We define the notions of constraints and constraint satisfaction. We show how to generate constraints from expressions. Then, we relate typing with constraint satisfaction, proving results of soundness and completeness.

Section 4.3 We describe how to solve constraints algorithmically, reusing the *tallying* algorithm of Castagna et al. (2015b). We prove soundness and completeness of the algorithm with respect to the declarative notion of constraint satisfaction.

Section 4.4 We summarize and discuss the results of the whole chapter. We also outline two possible modifications of the inference algorithm.

1 Throughout this thesis, we refer to the process of reconstructing type information for programs as *type inference*. The term is widely used in this sense, but the process is also called *type reconstruction*, notably by Pierce (2002).

$$[\mathrm{T}_{\hat{x}}]\ \frac{}{\Gamma \vdash \hat{x} : t[\vec{t}/\vec{\alpha}]}\ \Gamma(\hat{x}) = \forall\vec{\alpha}.\, t \qquad [\mathrm{T}_{x}]\ \frac{}{\Gamma \vdash x : t}\ \Gamma(x) = t \qquad [\mathrm{T}_{c}]\ \frac{}{\Gamma \vdash c : b_c}$$

$$[\mathrm{T}_{\lambda}]\ \frac{\Gamma, x : t' \vdash e : t}{\Gamma \vdash \lambda x.\, e : t' \to t} \qquad [\mathrm{T}_{\mathrm{app}}]\ \frac{\Gamma \vdash e_1 : t' \to t \qquad \Gamma \vdash e_2 : t'}{\Gamma \vdash e_1\ e_2 : t}$$

$$[\mathrm{T}_{\mathrm{pair}}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad [\mathrm{T}_{\mathrm{proj}}]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_i\ e : t_i}$$

$$[\mathrm{T}_{\mathrm{case}}]\ \frac{\Gamma \vdash e_0 : t_0 \qquad \text{either } t_0 \leq \neg\mathbf{t} \text{ or } \Gamma \vdash e_1 : t \qquad \text{either } t_0 \leq \mathbf{t} \text{ or } \Gamma \vdash e_2 : t}{\Gamma \vdash (e_0 \in \mathbf{t}\ ?\ e_1 : e_2) : t}$$

$$[\mathrm{T}_{\mathsf{let}}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, \hat{x} : \forall\vec{\alpha}.\, t_1 \vdash e_2 : t}{\Gamma \vdash \mathsf{let}\ \hat{x} = e_1\ \mathsf{in}\ e_2 : t}\ \vec{\alpha} \sharp \Gamma$$

$$[\mathrm{T}_{\leq}]\ \frac{\Gamma \vdash e : t'}{\Gamma \vdash e : t}\ t' \leq t \qquad [\mathrm{T}_{\wedge}]\ \frac{\Gamma \vdash e : t_1 \qquad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2}$$

FIGURE 4.1 $\mathcal{T}^{\mathrm{i}}$: Typing rules

NOTATION AND CONVENTIONS: Throughout this chapter and the next, we distinguish syntactically the variables bound by `let` bindings from those bound by λ-abstractions. We have not done so in the previous chapter because they could be treated uniformly. Now, it is more convenient to distinguish them: we will use $\hat{x}$ for variables bound by `let` and keep x for those bound by λ-abstractions. We therefore use the following syntax for expressions

$$e ::= x \mid \hat{x} \mid c \mid \lambda x.\, e \mid e\ e \mid (e, e) \mid \pi_i\ e \mid e \in \mathbf{t}\ ?\ e : e \mid \mathsf{let}\ \hat{x} = e\ \mathsf{in}\ e$$

and we also distinguish the variables in the domain of type environments: only $\hat{x}$ variables can be bound to type schemes with quantified variables. We denote the set of all $\hat{x}$ variables as $\mathsf{EVar}_{\mathsf{let}}$ and that of all x variables as EVar_{λ}; the set EVar of Section 3.1.1 is their union.

The typing rules that we will consider are those of Figure 4.1: we refer to them as $\mathcal{T}^{\mathrm{i}}$ (the "i" marks it as the system we study for type *inference*). They are the rules of $\mathcal{T}$ (Figure 3.2), except that we use two rules for the two kinds of variables and change the rule for `let` to use $\hat{x}$ instead of x. As mentioned earlier, we do not include the rule $[\mathrm{T}_{\lambda\neg}]$.

We will often consider the restriction of $\mathcal{T}^{\mathrm{i}}$ without the rule $[\mathrm{T}_{\wedge}]$: we refer to this restricted system as $\mathcal{T}^{\mathrm{i}\setminus\wedge}$. (A list of the different inference systems we use with pointers to their definition can be found on page 21.)

4.1 The reformulated type system

As a first step in our study of type inference, we define a new type system which is easier to relate to the inference algorithm that we will define next. The two systems differ in the treatment of type environments and generalization.

This *reformulated type system* is based on the *lambda-lifted* presentation of type systems from previous work on type inference with subtyping. The *reformulated typing rules* of Dolan and Mycroft (2017) – described in more detail in Dolan's PhD thesis (Dolan, 2016) – are the closest model. Earlier work include that of Trifonov and Smith (1996) and Pottier (1998).

We begin by describing why this alternative type system is useful: handling generalization during type inference is problematic in our system. Then, we describe the system itself. Finally, we study its relation with the type system $\mathcal{T}^{\mathrm{i}}$ and prove that, for each closed expression, the two systems derive exactly the same types.

4.1.1 The problem with generalization

A subtlety of the Hindley-Milner type system is in generalization: to type e_2 in $\mathsf{let}\ \hat{x} = e_1\ \mathsf{in}\ e_2$, we can assign to $\hat{x}$ the type scheme obtained from the type of e_1 by quantifying over all type variables *except those that are free in the type environment*. This restriction is needed to ensure soundness.

Therefore, whether the binding for a variable $\hat{x}$ is polymorphic or not (and if it is, which type variables we can instantiate) depends on a comparison of the type variables that appear syntactically in the type of the bound expression and in the type environment.

This is problematic with semantic subtyping: we want to see types up to the equivalence relation $\simeq$ (that is, to identify types with the same set-theoretic interpretation), but two types can be equivalent while having different type variables in them. For instance, $\alpha \wedge \mathbb{0}$ and $\alpha \setminus \alpha$ are both equivalent to $\mathbb{0}$, but α occurs in them and not in $\mathbb{0}$.

This mismatch is not a problem in the type system, but it complicates the definition of type inference. Let us examine how type inference for $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ in a type environment Γ could proceed.

1. We assign a type variable α to stand for the type of e_1.
2. We attempt to infer the type of e_1. Assuming we obtain a solution, this solution is a type substitution σ, and the inferred type of e_1 is $\alpha\sigma$. Note that σ can also instantiate type variables that appear in Γ.
3. We add $(x : \forall\vec{\alpha}.\, \alpha\sigma)$, where $\vec{\alpha} = \mathsf{var}(\alpha\sigma) \setminus \mathsf{var}(\Gamma\sigma)$, to the environment.
4. We attempt to infer the type of e_2 in the expanded environment.

The third step compares the variables that occur in $\alpha\sigma$ and $\Gamma\sigma$ to compute $\vec{\alpha}$. This implies that replacing σ with a σ' such that $\forall\alpha.\, \alpha\sigma \simeq \alpha\sigma'$ can change $\vec{\alpha}$: type substitutions cannot be seen up to equivalence in this step. This is undesirable, because it means that type inference must consider types syntactically

(taking care to introduce as few variables as possible) and not up to equivalence. For instance, in our work we want to reuse the *tallying* algorithm (Castagna et al., 2015b) to compute solutions (just like unification can be used as a step in Hindley-Milner type inference). Tallying has a completeness property that is stated up to equivalence: any solution σ of a set of subtyping constraints[2] $\{(t_1^1 \mathrel{\dot{\leq}} t_1^2), \ldots, (t_n^1 \mathrel{\dot{\leq}} t_n^2)\}$ is equivalent to some instantiation of a solution σ' found by tallying. However, the substitution found by tallying could introduce more type variables than needed (e.g., by mapping some variable to $\alpha \setminus \alpha$ instead of $\mathbb{0}$, but more complex cases exist, of course). Therefore, we cannot reuse tallying for type inference unless we describe its behaviour in more syntactic detail, which is inconvenient and runs counter to the principles of semantic subtyping.

In previous work (Castagna, Petrucciani, and Nguyễn, 2016), we have tried to overcome this difficulty by introducing a notion of *meaningful type variables* of a type. These are given by $\mathsf{mvar}(t) = \min_{\subseteq}\{\, \mathsf{var}(t') \mid t' \simeq t \,\}$: the meaningful type variables of t are those that occur in every type t' equivalent to t. In the cited work, they are defined as $\mathsf{mvar}(t) = \{\, \alpha \in \mathsf{var}(t) \mid t[\mathbb{0}/\alpha] \not\simeq t \,\}$; the two definitions are equivalent. This notion is interesting because equivalent types have the same meaningful variables. We have used mvar instead of var for generalization in a type inference algorithm. We previously believed that we had proven the algorithm sound and complete; however, we have later found a mistake in the proof of completeness, and we have realized that the approach was not wholly correct. Indeed, mvar is not as convenient to use as var, because it is difficult to determine the type variables that occur in $\mathsf{mvar}(t\sigma)$ knowing t and σ; in contrast, for var, we have the equality $\mathsf{var}(t\sigma) = \bigcup_{\alpha \in \mathsf{var}(t)} \mathsf{var}(\alpha\sigma)$. A step of the proof implicitly, and wrongly, assumed this equality also for mvar. To correct the proof, we would need to consider the behaviour of constraint solving in greater detail than we did, to prove that it does not introduce too many type variables. We conjecture that it is possible, but it seems to tie up too closely the general process of inference to the specifics of constraint solving.

Here, we follow a different approach: we introduce the reformulated type system, where type schemes and generalization are replaced by *typing schemes* that record dependence on the environment explicitly.

To illustrate the difference between the two type systems, consider the expression $\lambda x.\, (\mathsf{let}\ \hat{x} = \lambda y.\, (x, y)\ \mathsf{in}\ e)$, for some e. In the type system of the previous chapter, we can choose α as the type of x and type $\lambda y.\, (x, y)$ as $\beta \to \alpha \times \beta$. Then, to type e, we can assign to $\hat{x}$ the type scheme $\forall \beta.\, \beta \to \alpha \times \beta$. While β can be quantified, α cannot since it appears free in the environment: the let construct is typed assuming $(x \colon \alpha)$.

In the reformulated system, in contrast, $\hat{x}$ could be assigned the *typing scheme* $\langle x \colon \alpha \rangle (\beta \to \alpha \times \beta)$ (typing schemes are defined formally below). In this typing scheme, we treat all type variables as implicitly quantified (the typing rules allow us to instantiate any variable). Instead of distinguishing

2 We write $(t^1 \mathrel{\dot{\leq}} t^2)$ to denote a constraint that requires the solution to satisfy subtyping between the substitution instances of t^1 and t^2: this is defined formally in Definition 4.17.

between quantified and non-quantified variables, the typing scheme records explicitly the assumptions made on the type of free expression variables: in this case, $\langle x\colon \alpha\rangle$. We could equivalently choose for $\hat{x}$ the typing scheme $\langle x\colon \gamma\rangle(\delta \to \gamma \times \delta)$: we do not care which type variables we use, but only that the dependency is recorded correctly.

Using this system, the difficulties with generalization do not arise because we do not rely on comparing the type variables that occur in a type and in the environment. We will show how to build a type inference algorithm for this system. However, we actually want type inference for the previous, more standard system, $\mathcal{T}^{\mathrm{i}}$. Therefore, we also need to study the relation between the standard and the reformulated system.

4.1.2 Definition of the reformulated type system

Instead of using a single type environment Γ for both λ- and let-bound identifiers, the reformulated type system uses two separate ones: a *let-environment* P for let-bound, polymorphic binders, and a λ*-environment* M for monomorphic ones. More importantly, let-environments do not use type schemes: rather, they use *typing schemes* which record explicitly (using a λ-environment) the assumptions on the types of λ-bound variables.

4.1 DEFINITION: A λ*-environment* M is a finite mapping of variables in EVar_λ to types. A *typing scheme* is a pair of a λ-environment and a type, written $\langle M\rangle t$. A *let-environment* P is a finite mapping of variables in $\mathsf{EVar}_{\mathsf{let}}$ to typing schemes. □

We adopt the same notation to write these environments as for normal type environments. On λ-environments, we define some additional notions. We write $M_1 \leq M_2$ when, for every binding $(x\colon t_2)$ in M_2, there is a binding $(x\colon t_1)$ in M_1 such that $t_1 \leq t_2$. We write $M_1 \wedge M_2$ for the λ-environment whose domain is the union of the two domains and such that

$$(M_1 \wedge M_2)(x) = \begin{cases} M_1(x) & \text{if } x \in \mathsf{dom}(M_1) \setminus \mathsf{dom}(M_2) \\ M_2(x) & \text{if } x \in \mathsf{dom}(M_2) \setminus \mathsf{dom}(M_1) \\ M_1(x) \wedge M_2(x) & \text{if } x \in \mathsf{dom}(M_2) \cap \mathsf{dom}(M_1) \end{cases}$$

We write $M \setminus x$ for M with the binding for x removed.

The reformulated type system is then defined by typing rules very similar to those of the standard system. Following Dolan and Mycroft (2017), we use the symbol ⊩ in the judgments instead of ⊢.

4.2 DEFINITION: The reformulated typing relation $P; M \Vdash e\colon t$ is defined by the rules of Figure 4.2. □

We write $\mathcal{T}^{\mathrm{r}}$ to refer to this system and $\mathcal{T}^{\mathrm{r}\setminus\wedge}$ to refer to its restriction without the rule $[\mathrm{T}^{\mathrm{r}}_{\wedge}]$.

$$[\mathrm{T}^{\mathrm{r}}_{\hat{x}}]\ \frac{}{P; M\sigma \Vdash \hat{x} : t\sigma}\ P(\hat{x}) = \langle M \rangle t \qquad [\mathrm{T}^{\mathrm{r}}_{x}]\ \frac{}{P; M \Vdash x : t}\ M(x) = t \qquad [\mathrm{T}^{\mathrm{r}}_{c}]\ \frac{}{P; M \Vdash c : b_c}$$

$$[\mathrm{T}^{\mathrm{r}}_{\lambda}]\ \frac{P; (M, x : t') \Vdash e : t}{P; M \Vdash \lambda x.\, e : t' \to t} \qquad [\mathrm{T}^{\mathrm{r}}_{\mathrm{app}}]\ \frac{P; M \Vdash e_1 : t' \to t \qquad P; M \Vdash e_2 : t'}{P; M \Vdash e_1\ e_2 : t}$$

$$[\mathrm{T}^{\mathrm{r}}_{\mathrm{pair}}]\ \frac{P; M \Vdash e_1 : t_1 \qquad P; M \Vdash e_2 : t_2}{P; M \Vdash (e_1, e_2) : t_1 \times t_2} \qquad [\mathrm{T}^{\mathrm{r}}_{\mathrm{proj}}]\ \frac{P; M \Vdash e : t_1 \times t_2}{P; M \Vdash \pi_i\ e : t_i}$$

$$[\mathrm{T}^{\mathrm{r}}_{\mathrm{case}}]\ \frac{\begin{array}{c} P; M \Vdash e_0 : t_0 \\ \text{either } t_0 \leq \neg\mathbf{t} \text{ or } P; M \Vdash e_1 : t \qquad \text{either } t_0 \leq \mathbf{t} \text{ or } P; M \Vdash e_2 : t \end{array}}{P; M \Vdash (e_0 \in \mathbf{t}\ ?\ e_1 : e_2) : t}$$

$$[\mathrm{T}^{\mathrm{r}}_{\mathsf{let}}]\ \frac{P; M_1 \Vdash e_1 : t_1 \qquad (P, \hat{x} : \langle M_1 \rangle t_1); M \Vdash e_2 : t}{P; M \Vdash \mathsf{let}\ \hat{x} = e_1\ \mathsf{in}\ e_2 : t}\ \exists\sigma.\, M \leq M_1\sigma$$

$$[\mathrm{T}^{\mathrm{r}}_{\leq}]\ \frac{P; M' \Vdash e : t'}{P; M \Vdash e : t}\ \begin{cases} t' \leq t \\ M \leq M' \end{cases} \qquad [\mathrm{T}^{\mathrm{r}}_{\wedge}]\ \frac{P; M \Vdash e : t_1 \qquad P; M \Vdash e : t_2}{P; M \Vdash e : t_1 \wedge t_2}$$

FIGURE 4.2 $\mathcal{T}^{\mathrm{r}}$: Reformulated typing rules

Compared to the rules of Figure 4.1, the interesting differences are for $[\mathrm{T}^{\mathrm{r}}_{\hat{x}}]$, $[\mathrm{T}^{\mathrm{r}}_{\mathsf{let}}]$, and $[\mathrm{T}^{\mathrm{r}}_{\leq}]$. For $[\mathrm{T}^{\mathrm{r}}_{\hat{x}}]$, we can instantiate all type variables in the typing scheme $\langle M \rangle t$ of $\hat{x}$: there is no restriction on the domain of σ. In this sense we say that typing schemes behave with respect to typing as if all type variables in them were implicitly quantified. However, note that the λ-environment must correspond to the substitution. In $[\mathrm{T}^{\mathrm{r}}_{\mathsf{let}}]$, to type e_1 we can use a different λ-environment than the one in the main derivation. However, we must make sure that the assumptions used to type e_1 are reflected in M. To do so, we could ask $M \leq M_1$. We require instead the weaker condition $\exists\sigma.\, M \leq M_1\sigma$, which simplifies the proofs that relate this system with inference. The subsumption rule $[\mathrm{T}^{\mathrm{r}}_{\leq}]$ acts on both the type and the λ-environment.

COMPARISON TO THE RULES OF DOLAN AND MYCROFT: Our reformulated typing rules are very similar to those of Dolan (2016) and Dolan and Mycroft (2017). The main difference is that they put M to the right of the turnstile, so that the rules derive a typing scheme and not a type: $P \Vdash e : \langle M \rangle t$ (or $\Pi \Vdash e : [\Delta]\tau$ using their metavariables and notation). They allow instantiation in the rule $[\mathrm{T}^{\mathrm{r}}_{\leq}]$, while we allow it in $[\mathrm{T}^{\mathrm{r}}_{\hat{x}}]$. We choose our presentation for ease of comparison with the standard rules and with type inference.

4.1.3 Relating the systems $\mathcal{T}^{\mathrm{i}}$ and $\mathcal{T}^{\mathrm{r}}$

We want to relate the standard type system $\mathcal{T}^{\mathrm{i}}$ and the reformulated system $\mathcal{T}^{\mathrm{r}}$ so that the results we develop next on type inference, which consider the latter, can be transferred also to the former.

In the work of Trifonov and Smith (1996) and Pottier (1998), the lambda-lifted style was used to define the type system for which type soundness was proven. However, it has the disadvantage of being a less standard way to describe a type system. A claim of Dolan and Mycroft (2017) is that they can relate the standard and the reformulated type systems, proving that (with our notation) for every e and t, $\varnothing \vdash e : t$ holds if and only if $\varnothing; \varnothing \Vdash e : t$. This is the result we want too.

The proof of Dolan and Mycroft is described in the first author's PhD thesis (Dolan, 2016). It relies on two lemmas (Lemmas 33 and 34) to prove the two implications. For induction to work, the lemmas also consider non-empty environments and show how to convert Γ into P and M, and vice versa. Unfortunately, Lemma 34 – which converts derivations in the reformulated system to derivations in the standard one – does not actually hold.[3]

We develop a different proof to show the same result. Our proof relies on the presence of the rule $[\mathrm{T}^{\mathrm{r}}_{\wedge}]$, which Dolan and Mycroft do not have. In the rest of the section, we prove this equivalence result:

$$\forall e, t.\ \ \varnothing \vdash e : t \iff \varnothing; \varnothing \Vdash e : t\ .$$

Additionally, we prove that the implication $\Longrightarrow$ holds also in the restricted systems $\mathcal{T}^{\mathrm{i}\setminus\wedge}$ and $\mathcal{T}^{\mathrm{r}\setminus\wedge}$ (those without the rules $[\mathrm{T}_{\wedge}]$ and $[\mathrm{T}^{\mathrm{r}}_{\wedge}]$, respectively). We cannot prove the reverse implication for the restricted systems, because the proof relies on using $[\mathrm{T}^{\mathrm{r}}_{\wedge}]$. However, we conjecture that it holds too.

Converting derivations in $\mathcal{T}^{\mathrm{i}}$ to derivations in $\mathcal{T}^{\mathrm{r}}$ is fairly simple. We give the following definition to express when a pair of a P and an M can be used to represent a Γ.

4.3 DEFINITION: A pair of a let-environment P and a λ-environment M is *adequate to represent* a type environment Γ, written $P; M \vDash \Gamma$, if:

- for every binding $(x : t)$ in Γ, there is a binding $(x : t')$ in M and $t' \leq t$;
- for every binding $(\hat{x} : \forall \vec{\alpha}.\, t)$ in Γ, there is a binding $(\hat{x} : \langle M' \rangle t)$ in P with $M \leq M'$ and $\vec{\alpha} \sharp M'$;
- $\mathsf{var}(M) \subseteq \mathsf{var}(\Gamma)$. □

3 Confirmed by Dolan in personal communication with the author.

Lemma 34 states that "if $\Pi \Vdash e : [\Delta]\tau$, then $r(\Pi) \sqcap \Delta \vdash e : \tau$". However, if we take $\Pi = (\hat{x} : [x : \alpha]\alpha)$, then using (VAR-$\Pi$) and (SUB) with the substitution $[\mathsf{Int}/\alpha]$ we have $\Pi \Vdash \hat{x} : [x : \mathsf{Int}]\mathsf{Int}$. If the lemma held, we should be able to derive $r(\Pi) \sqcap (x : \mathsf{Int}) \vdash \hat{x} : \mathsf{Int}$. However, $r(\Pi) \sqcap (x : \mathsf{Int})$ is $(x : \alpha, \hat{x} : \alpha) \sqcap (x : \mathsf{Int}) = (x : \alpha \sqcap \mathsf{Int}, \hat{x} : \alpha)$, which does not allow this derivation.

Dolan proposes an alternative proof which relies on encoding expressions so that, in each $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, e_1 has no free λ-bound variables. While appealing, this proof is not fully developed yet.

Next we prove the following result to convert a typing derivation $\Gamma \vdash e : t$ in $\mathcal{T}^{\mathrm{i}}$ to a derivation $P; M \Vdash e : t$ in $\mathcal{T}^{\mathrm{r}}$.

4.4 LEMMA: If $\Gamma \vdash e : t$ and $P; M \vDash \Gamma$, then $P; M \Vdash e : t$.

Moreover, if $\Gamma \vdash e : t$ can be derived in $\mathcal{T}^{\mathrm{i}\setminus\wedge}$, then $P; M \Vdash e : t$ can be derived in $\mathcal{T}^{\mathrm{r}\setminus\wedge}$. □

Proof: By induction on the derivation of $\Gamma \vdash e : t$ and by case analysis on the last rule applied. Note that we only apply $[\mathrm{T}^{\mathrm{r}}_{\wedge}]$ in the case for $[\mathrm{T}_{\wedge}]$.

Case: $[\mathrm{T}_{\hat{x}}]$
We have $\Gamma(\hat{x}) = \forall\vec{\alpha}.\, t'$ and $t = t'[\vec{t}/\vec{\alpha}]$. Then, $P(\hat{x}) = \langle M'\rangle t'$, and $M \leq M'$ and $\vec{\alpha} \sharp M'$.
We derive $P; M'[\vec{t}/\vec{\alpha}] \Vdash \hat{x} : t'[\vec{t}/\vec{\alpha}]$ by $[\mathrm{T}^{\mathrm{r}}_{\hat{x}}]$. Since $\vec{\alpha} \sharp M'$, we have $M'[\vec{t}/\vec{\alpha}] = M'$. We obtain $P; M \Vdash \hat{x} : t'[\vec{t}/\vec{\alpha}]$ by $[\mathrm{T}^{\mathrm{r}}_{\leq}]$.

Case: $[\mathrm{T}_{x}]$
We have $\Gamma(x) = t$, therefore $M(x) \leq t$. We derive the conclusion by $[\mathrm{T}^{\mathrm{r}}_{x}]$ and $[\mathrm{T}^{\mathrm{r}}_{\leq}]$.

Case: $[\mathrm{T}_{c}]$ Immediate.

Case: $[\mathrm{T}_{\lambda}]$
We have $\Gamma \vdash \lambda x.\, e' : t_1 \to t_2$ and $\Gamma, x : t_1 \vdash e' : t_2$.
We have $P; (M, x : t_1) \vDash (\Gamma, x : t_1)$. (In particular, note that $M, x : t_1 \leq M$ and therefore $M \leq M'$ implies $M, x : t_1 \leq M'$.)
By IH, we have $P; (M, x : t_1) \Vdash e' : t_2$. We conclude by $[\mathrm{T}^{\mathrm{r}}_{\lambda}]$.

Case: $[\mathrm{T}_{\mathrm{app}}]$, $[\mathrm{T}_{\mathrm{pair}}]$, $[\mathrm{T}_{\mathrm{proj}}]$, $[\mathrm{T}_{\mathrm{case}}]$, $[\mathrm{T}_{\leq}]$, $[\mathrm{T}_{\wedge}]$
Straightforward by IH. Note that the case for $[\mathrm{T}_{\wedge}]$ is the only one for which we must use the rule $[\mathrm{T}^{\mathrm{r}}_{\wedge}]$.

Case: $[\mathrm{T}_{\mathsf{let}}]$
We have $\Gamma \vdash \mathsf{let}\ \hat{x} = e_1\ \mathsf{in}\ e_2 : t$, derived from

$$\Gamma \vdash e_1 : t_1 \qquad \Gamma, \hat{x} : \forall\vec{\alpha}.\, t_1 \vdash e_2 : t \qquad \vec{\alpha} \sharp \Gamma\,.$$

We have $(P, \hat{x} : \langle M\rangle t_1); M \vDash (\Gamma, \hat{x} : \forall\vec{\alpha}.\, t_1)$. (In particular, note that $P; M \vDash \Gamma$ implies $\mathsf{var}(M) \subseteq \mathsf{var}(\Gamma)$, therefore $\vec{\alpha} \sharp M$.)
By IH we obtain

$$P; M \Vdash e_1 : t_1 \qquad (P, \hat{x} : \langle M\rangle t_1); M \Vdash e_2 : t$$

and we conclude by $[\mathrm{T}^{\mathrm{r}}_{\mathsf{let}}]$. □

The other direction is more challenging. The proof requires us to convert a pair of a let-environment and a λ-environment to a type environment. We do so by exploiting a property of type systems with intersection types: instead of using a type scheme containing quantified variables, we can use the type formed by taking the intersection of all the instantiations of the type scheme

that we actually use in the derivation. There are always finitely many different instantiations (at most one for each use of the bound variable).

We give a description of the reformulated type system where the derivation keeps track of the instantiations used for each typing scheme. We do so by adding a new part to the typing judgment: a function that maps each variable in $\mathsf{EVar}_{\mathsf{let}}$ to the set of type substitutions used to instantiate that variable in the derivation.

4.5 DEFINITION: An *instantiation map* $\mathcal{I}$ is a total function from $\mathsf{EVar}_{\mathsf{let}}$ to finite sets of type substitutions.

We write ϵ for the instantiation map such that $\epsilon(\hat{x}) = \emptyset$ for every variable $\hat{x}$. We write $(\hat{x} \mapsto \{\sigma\})$ for the instantiation map such that $(\hat{x} \mapsto \{\sigma\})(\hat{x}) = \{\sigma\}$ and $(\hat{x} \mapsto \{\sigma\})(\hat{y}) = \emptyset$ for every variable $\hat{y} \neq \hat{x}$. □

Given two instantiation maps $\mathcal{I}_1$ and $\mathcal{I}_2$, we can define their pointwise union $\mathcal{I}_1 \sqcup \mathcal{I}_2$, such that $(\mathcal{I}_1 \sqcup \mathcal{I}_2)(\hat{x}) = \mathcal{I}_1(\hat{x}) \cup \mathcal{I}_2(\hat{x})$ for every $\hat{x}$.

Given an instantiation map $\mathcal{I}$ and a finite set $\{\, \sigma_i \mid i \in I \,\}$ of type substitutions, we write $\mathcal{I}\{\, \sigma_i \mid i \in I \,\}$ for the instantiation map such that

$$(\mathcal{I}\{\, \sigma_i \mid i \in I \,\})(\hat{x}) = \{\, \sigma_i \circ \sigma \mid \sigma \in \mathcal{I}(\hat{x}), i \in I \,\} .$$

We write $\mathcal{I}\sigma$ for $\mathcal{I}\{\sigma\}$.

We write $\mathcal{I}_1 \sqsubseteq \mathcal{I}_2$ when $\mathcal{I}_1(\hat{x}) \subseteq \mathcal{I}_2(\hat{x})$ for every $\hat{x}$.

We write $\mathcal{I} \setminus \hat{x}$ for the instantiation map such that $(\mathcal{I} \setminus \hat{x})(\hat{x}) = \emptyset$ and that $(\mathcal{I} \setminus \hat{y})(\hat{y}) = \mathcal{I}(\hat{y})$ for every $\hat{y} \neq \hat{x}$.

4.6 LEMMA: For any two instantiation maps $\mathcal{I}_1$ and $\mathcal{I}_2$, we have $\mathcal{I}_1 \sqsubseteq \mathcal{I}_1 \sqcup \mathcal{I}_2$ and $\mathcal{I}_2 \sqsubseteq \mathcal{I}_1 \sqcup \mathcal{I}_2$. □

Proof: Straightforward. □

4.7 DEFINITION: The reformulated typing relation with explicit instantiations $P; M \Vdash e : t \mid \mathcal{I}$ is defined by the rules of Figure 4.3. □

We refer to this modified system as $\mathcal{T}^{\mathrm{ri}}$. Adding instantiations to the rules is mostly straightforward. The most complex case is that of $[\mathrm{T}^{\mathrm{ri}}_{\mathsf{let}}]$: we compose the instantiations needed to type e_1 with the instantiations of $\hat{x}$ used to type e_2 (plus the substitution σ that we already had in the side condition of $[\mathrm{T}^{\mathrm{r}}_{\mathsf{let}}]$).

The following are a few lemmas that relate the rules with explicit instantiations with the previous ones and the derived type and set of instantiations with the environments.

4.8 LEMMA: $P; M \Vdash e : t$ holds in $\mathcal{T}^{\mathrm{r}}$ if and only if there exists an instantiation map $\mathcal{I}$ such that $P; M \Vdash e : t \mid \mathcal{I}$ holds in $\mathcal{T}^{\mathrm{ri}}$. □

$$[\mathrm{T}^{\mathrm{ri}}_{\hat{x}}]\ \frac{}{P; M\sigma \Vdash \hat{x} \colon t\sigma \mid (\hat{x} \mapsto \{\sigma\})}\ P(\hat{x}) = \langle M \rangle t \qquad [\mathrm{T}^{\mathrm{ri}}_{x}]\ \frac{}{P; M \Vdash x \colon t \mid \epsilon}\ M(x) = t$$

$$[\mathrm{T}^{\mathrm{ri}}_{c}]\ \frac{}{P; M \Vdash c \colon b_c \mid \epsilon}$$

$$[\mathrm{T}^{\mathrm{ri}}_{\lambda}]\ \frac{P; (M, x \colon t') \Vdash e \colon t \mid \mathcal{I}}{P; M \Vdash \lambda x.\, e \colon t' \to t \mid \mathcal{I}} \qquad [\mathrm{T}^{\mathrm{ri}}_{\mathrm{app}}]\ \frac{P; M \Vdash e_1 \colon t' \to t \mid \mathcal{I}_1 \qquad P; M \Vdash e_2 \colon t' \mid \mathcal{I}_2}{P; M \Vdash e_1\ e_2 \colon t \mid \mathcal{I}_1 \sqcup \mathcal{I}_2}$$

$$[\mathrm{T}^{\mathrm{ri}}_{\mathrm{pair}}]\ \frac{P; M \Vdash e_1 \colon t_1 \mid \mathcal{I}_1 \qquad P; M \Vdash e_2 \colon t_2 \mid \mathcal{I}_2}{P; M \Vdash (e_1, e_2) \colon t_1 \times t_2 \mid \mathcal{I}_1 \sqcup \mathcal{I}_2} \qquad [\mathrm{T}^{\mathrm{ri}}_{\mathrm{proj}}]\ \frac{P; M \Vdash e \colon t_1 \times t_2 \mid \mathcal{I}}{P; M \Vdash \pi_i\, e \colon t_i \mid \mathcal{I}}$$

$$[\mathrm{T}^{\mathrm{ri}}_{\mathrm{case}}]\ \frac{\begin{array}{c} P; M \Vdash e_0 \colon t_0 \mid \mathcal{I}_0 \\ \text{either } t_0 \leq \neg\mathbf{t} \text{ and } \mathcal{I}_1 = \epsilon \text{ or } P; M \Vdash e_1 \colon t \mid \mathcal{I}_1 \\ \text{either } t_0 \leq \mathbf{t} \text{ and } \mathcal{I}_2 = \epsilon \text{ or } P; M \Vdash e_2 \colon t \mid \mathcal{I}_2 \end{array}}{P; M \Vdash (e_0 \in \mathbf{t}\ ?\ e_1 : e_2) \colon t \mid \mathcal{I}_0 \sqcup \mathcal{I}_1 \sqcup \mathcal{I}_2}$$

$$[\mathrm{T}^{\mathrm{ri}}_{\mathsf{let}}]\ \frac{\begin{array}{c} P; M_1 \Vdash e_1 \colon t_1 \mid \mathcal{I}_1 \\ (P, \hat{x} \colon \langle M_1 \rangle t_1); M \Vdash e_2 \colon t \mid \mathcal{I}_2 \end{array}}{P; M \Vdash \mathsf{let}\ \hat{x} = e_1\ \mathsf{in}\ e_2 \colon t \mid \mathcal{I}}\ \exists\sigma. \begin{cases} M \leq M_1\sigma \\ \mathcal{I} = \big(\mathcal{I}_1\sigma\big) \sqcup \big(\mathcal{I}_1\,(\mathcal{I}_2(\hat{x}))\big) \sqcup \big(\mathcal{I}_2 \setminus \hat{x}\big) \end{cases}$$

$$[\mathrm{T}^{\mathrm{ri}}_{\leq}]\ \frac{P; M' \Vdash e \colon t' \mid \mathcal{I}}{P; M \Vdash e \colon t \mid \mathcal{I}} \begin{cases} t' \leq t \\ M \leq M' \end{cases} \qquad [\mathrm{T}^{\mathrm{ri}}_{\wedge}]\ \frac{P; M \Vdash e \colon t_1 \mid \mathcal{I}_1 \qquad P; M \Vdash e \colon t_2 \mid \mathcal{I}_2}{P; M \Vdash e \colon t_1 \wedge t_2 \mid \mathcal{I}_1 \sqcup \mathcal{I}_2}$$

FIGURE 4.3 $\mathcal{T}^{\mathrm{ri}}$: Reformulated typing rules with explicit instantiations

Proof: Straightforward proofs by induction on the typing derivations. □

4.9 LEMMA: If $P; M \Vdash e : t \mid \mathcal{I}$, then for every σ there exists an $\mathcal{I}'$ such that $P; M\sigma \Vdash e : t\sigma \mid \mathcal{I}'$ and $\mathcal{I}' \sqsubseteq \mathcal{I}\sigma$. □

Proof: By induction on the derivation of $P; M \Vdash e : t \mid \mathcal{I}$ and by case analysis on the last rule applied.

Case: $[\mathrm{T}^{\mathrm{ri}}_x]$
By hypothesis we have $P; M\sigma' \Vdash \hat{x} : t\sigma' \mid (\hat{x} \mapsto \{\sigma'\})$.
We can derive $P; M\sigma'\sigma \Vdash \hat{x} : t\sigma'\sigma \mid (\hat{x} \mapsto \{\sigma \circ \sigma'\})$.
We have $(\hat{x} \mapsto \{\sigma \circ \sigma'\}) = (\hat{x} \mapsto \{\sigma'\})\sigma$.

Case: $[\mathrm{T}^{\mathrm{ri}}_x]$, $[\mathrm{T}^{\mathrm{ri}}_c]$ Straightforward.

Case: $[\mathrm{T}^{\mathrm{ri}}_\lambda]$, $[\mathrm{T}^{\mathrm{ri}}_{\mathrm{app}}]$, $[\mathrm{T}^{\mathrm{ri}}_{\mathrm{pair}}]$, $[\mathrm{T}^{\mathrm{ri}}_{\mathrm{proj}}]$, $[\mathrm{T}^{\mathrm{ri}}_{\mathrm{case}}]$, $[\mathrm{T}^{\mathrm{ri}}_\leq]$, $[\mathrm{T}^{\mathrm{ri}}_\wedge]$
Straightforward by IH.
Note that $\mathcal{I}_1' \sqsubseteq \mathcal{I}_1\sigma$ and $\mathcal{I}_2' \sqsubseteq \mathcal{I}_2\sigma$ imply $\mathcal{I}_1' \sqcup \mathcal{I}_2' \sqsubseteq (\mathcal{I}_1 \sqcup \mathcal{I}_2)\sigma$.
The case for $[\mathrm{T}^{\mathrm{ri}}_{\mathrm{case}}]$ is the only one in which we do not have $\mathcal{I}' = \mathcal{I}\sigma$, because $t_0\sigma \leq \neg\mathbf{t}$ and $t_0\sigma \leq \mathbf{t}$ could hold when $t_0 \leq \neg\mathbf{t}$ and $t_0 \leq \mathbf{t}$ do not.

Case: $[\mathrm{T}^{\mathrm{ri}}_{\mathsf{let}}]$
We have

$$P; M \Vdash \mathsf{let}\ \hat{y} = e_1\ \mathsf{in}\ e_2 : t \mid \mathcal{I}$$
$$\text{Ⓐ}\ P; M_1 \Vdash e_1 : t_1 \mid \mathcal{I}_1 \qquad \text{Ⓑ}\ (P, \hat{y} : \langle M_1 \rangle t_1); M \Vdash e_2 : t \mid \mathcal{I}_2$$
$$\text{Ⓒ}\ M \leq M_1\sigma' \qquad \mathcal{I} = \big(\mathcal{I}_1\sigma'\big) \sqcup \big(\mathcal{I}_1\,(\mathcal{I}_2(\hat{y}))\big) \sqcup \big(\mathcal{I}_2 \setminus \hat{y}\big)$$

By IH from Ⓑ we have $(P, \hat{y} : \langle M_1 \rangle t_1); M\sigma \Vdash e_2 : t\sigma \mid \mathcal{I}_2'$ with $\mathcal{I}_2' \sqsubseteq \mathcal{I}_2\sigma$.
From Ⓒ we obtain $M\sigma \leq M_1\sigma'\sigma$.
Applying $[\mathrm{T}^{\mathrm{ri}}_{\mathsf{let}}]$ to Ⓐ and Ⓑ, we have:

$$P; M\sigma \Vdash \mathsf{let}\ \hat{y} = e_1\ \mathsf{in}\ e_2 : t\sigma \mid \mathcal{I}'$$
$$\mathcal{I}' = \big(\mathcal{I}_1\sigma'\sigma\big) \sqcup \big(\mathcal{I}_1\,(\mathcal{I}_2'(\hat{y}))\big) \sqcup \big(\mathcal{I}_2' \setminus \hat{y}\big)$$

and we conclude by observing that $\mathcal{I}' \sqsubseteq \mathcal{I}\sigma$. □

4.10 LEMMA: If $P; M \Vdash e : t \mid \mathcal{I}$ then, for every $(\hat{x} : \langle M' \rangle t')$ in P and $\sigma \in \mathcal{I}(\hat{x})$, we have $M \leq M'\sigma$. □

Proof: By induction on the derivation of $P; M \Vdash e : t \mid \mathcal{I}$ and by case analysis on the last rule applied.

Case: $[\mathrm{T}^{\mathrm{ri}}_{\hat{x}}]$, $[\mathrm{T}^{\mathrm{ri}}_x]$, $[\mathrm{T}^{\mathrm{ri}}_c]$ Straightforward.

Case: $[\mathrm{T}^{\mathrm{ri}}_\lambda]$
We have $e = \lambda x.\, e'$. We assume by α-renaming that x does not occur in the typing schemes in P.
By IH, $(M, x : t') \leq M'\sigma$. Since x is not in P, $M \leq M'\sigma$.

Case: $[\mathrm{T}^{\mathrm{ri}}_{\mathrm{app}}]$, $[\mathrm{T}^{\mathrm{ri}}_{\mathrm{pair}}]$, $[\mathrm{T}^{\mathrm{ri}}_{\mathrm{proj}}]$, $[\mathrm{T}^{\mathrm{ri}}_{\mathrm{case}}]$, $[\mathrm{T}^{\mathrm{ri}}_{\leq}]$, $[\mathrm{T}^{\mathrm{ri}}_{\wedge}]$
Straightforward by IH.

Case: $[\mathrm{T}^{\mathrm{ri}}_{\mathsf{let}}]$
We have

$$P; M \Vdash \mathsf{let}\ \hat{y} = e_1\ \mathsf{in}\ e_2 : t \mid \mathcal{I} \qquad P(\hat{x}) = \langle M' \rangle t' \qquad \sigma \in \mathcal{I}(\hat{x})$$
$$\text{Ⓐ}\ \ P; M_1 \Vdash e_1 : t_1 \mid \mathcal{I}_1 \qquad \text{Ⓑ}\ \ (P, \hat{y} : \langle M_1 \rangle t_1); M \Vdash e_2 : t \mid \mathcal{I}_2$$
$$\text{Ⓒ}\ \ M \leq M_1\sigma' \qquad \mathcal{I} = \big(\mathcal{I}_1\sigma'\big) \sqcup \big(\mathcal{I}_1\,(\mathcal{I}_2(\hat{y}))\big) \sqcup \big(\mathcal{I}_2 \setminus \hat{y}\big)$$

and we must show $M \leq M'\sigma$.
We can assume by α-renaming that $\hat{x} \neq \hat{y}$.
There are three cases.

Subcase: $\sigma \in \big(\mathcal{I}_1\sigma'\big)(\hat{x})$
Then $\sigma = \sigma' \circ \sigma_1$ with $\sigma_1 \in \mathcal{I}_1(\hat{x})$.
By IH from Ⓐ we have $M_1 \leq M'\sigma_1$. We obtain $M \leq M'\sigma$ from Ⓒ.

Subcase: $\sigma \in \big(\mathcal{I}_1\,(\mathcal{I}_2(\hat{y}))\big)(\hat{x})$
Then $\sigma = \sigma_2 \circ \sigma_1$ with $\sigma_1 \in \mathcal{I}_1(\hat{x})$ and $\sigma_2 \in \mathcal{I}_2(\hat{y})$.
By IH from Ⓐ we have $M_1 \leq M'\sigma_1$ and from Ⓑ we have $M \leq M_1\sigma_2$.
We have $M_1\sigma_2 \leq M'\sigma_1\sigma_2$ and therefore $M \leq M'\sigma$.

Subcase: $\sigma \in \big(\mathcal{I}_2 \setminus \hat{y}\big)(\hat{x})$
Then $\sigma \in \mathcal{I}_2(\hat{x})$, and we obtain the result by IH from Ⓑ. □

We define when a type environment Γ can represent a triple of P, M, and $\mathcal{I}$.

4.11 DEFINITION: A type environment Γ is *adequate to represent* a triple of a let-environment P, a λ-environment M, and an instantiation map $\mathcal{I}$, written $\Gamma \vDash P; M; \mathcal{I}$, if:

- for every binding $(x : t)$ in M, there is a binding $(x : t')$ in Γ and $t' \leq t$;
- for every binding $(\hat{x} : \langle M' \rangle t)$ in P and every $\sigma \in \mathcal{I}(\hat{x})$, there is a binding $(\hat{x} : \forall \vec{\alpha}.\, t')$ in Γ and a vector $\vec{t}$ such that $t'[\vec{t}/\vec{\alpha}] \leq t\sigma$. □

4.12 LEMMA: If $\Gamma \vDash P; M; \mathcal{I}$ and $\mathcal{I}' \sqsubseteq \mathcal{I}$, then $\Gamma \vDash P; M; \mathcal{I}'$. □

Proof: Straightforward. □

We now show how to convert derivations in $\mathcal{T}^{\mathrm{r}}$ to those in $\mathcal{T}^{\mathrm{i}}$.

4.13 LEMMA: If $P; M \Vdash e : t \mid \mathcal{I}$ and $\Gamma \vDash P; M; \mathcal{I}$, then $\Gamma \vdash e : t$. □

Proof: By induction on the derivation of $P; M \Vdash e : t \mid \mathcal{I}$ and by case analysis on the last rule applied.

Case: $[\mathrm{T}^{\mathrm{ri}}_{\hat{x}}]$
We have $P; M\sigma \Vdash \hat{x} : t\sigma \mid (\hat{x} \mapsto \{\sigma\})$ and $\Gamma \vDash P; M\sigma; (\hat{x} \mapsto \{\sigma\})$.

Therefore, we have $\Gamma(\hat{x}) = \forall\vec{\alpha}.\, t'$ and $t'[\vec{t}/\vec{\alpha}] \leq t\sigma$ for some $\forall\vec{\alpha}.\, t'$ and $\vec{t}$.
We derive $\Gamma \vdash \hat{x} \colon t\sigma$ by $[\mathrm{T}_{\hat{x}}]$ and $[\mathrm{T}_{\leq}]$.

Case: $[\mathrm{T}^{\mathrm{ri}}_{x}]$
We obtain $\Gamma \vdash x \colon t$ by $[\mathrm{T}_{x}]$ and $[\mathrm{T}_{\leq}]$ since $M(x) = t$ and $\Gamma(x) \leq M(x)$.

Case: $[\mathrm{T}^{\mathrm{ri}}_{c}]$ Immediate.

Case: $[\mathrm{T}^{\mathrm{ri}}_{\lambda}]$
Since $\Gamma \vDash P; M; \mathcal{I}$, we have $(\Gamma, x \colon t') \vDash P; (M, x \colon t'); \mathcal{I}$.
We apply the IH and conclude using $[\mathrm{T}_{\lambda}]$.

Case: $[\mathrm{T}^{\mathrm{ri}}_{\mathrm{app}}]$, $[\mathrm{T}^{\mathrm{ri}}_{\mathrm{pair}}]$, $[\mathrm{T}^{\mathrm{ri}}_{\mathrm{proj}}]$, $[\mathrm{T}^{\mathrm{ri}}_{\mathrm{case}}]$, $[\mathrm{T}^{\mathrm{ri}}_{\wedge}]$
Straightforward by IH using Lemmas 4.6 and 4.13.

Case: $[\mathrm{T}^{\mathrm{ri}}_{\mathsf{let}}]$
We have

$$P; M \Vdash \mathsf{let}\ \hat{x} = e_1\ \mathsf{in}\ e_2 \colon t \mid \mathcal{I} \qquad \text{Ⓐ}\ \ \Gamma \vDash P; M; \mathcal{I}$$
$$\text{Ⓑ}\ \ P; M_1 \Vdash e_1 \colon t_1 \mid \mathcal{I}_1 \qquad \text{Ⓒ}\ \ (P, \hat{x} \colon \langle M_1 \rangle t_1); M \Vdash e_2 \colon t \mid \mathcal{I}_2$$
$$\text{Ⓓ}\ \ M \leq M_1\sigma \qquad \mathcal{I} = \big(\mathcal{I}_1\sigma\big) \sqcup \big(\mathcal{I}_1\,(\mathcal{I}_2(\hat{x}))\big) \sqcup \big(\mathcal{I}_2 \setminus \hat{x}\big)$$

and we must derive $\Gamma \vdash \mathsf{let}\ \hat{x} = e_1\ \mathsf{in}\ e_2 \colon t$.
Let $\{\, \sigma_k \mid k \in K \,\} = \mathcal{I}_2(\hat{x})$.
From Ⓑ by Lemma 4.9 we have

$$P; M_1\sigma \Vdash e_1 \colon t_1\sigma \mid \mathcal{I}_1' \qquad \mathcal{I}_1' \sqsubseteq \mathcal{I}_1\sigma$$
$$P; M_1\sigma_k \Vdash e_1 \colon t_1\sigma_k \mid \mathcal{I}_1^k \qquad \mathcal{I}_1^k \sqsubseteq \mathcal{I}_1\sigma_k.$$

From Ⓒ by Lemma 4.10 we have $M \leq M_1\sigma_k$ for every $k \in K$.
Therefore, by $[\mathrm{T}^{\mathrm{ri}}_{\leq}]$ we have

$$P; M \Vdash e_1 \colon t_1\sigma \mid \mathcal{I}_1' \qquad P; M \Vdash e_1 \colon t_1\sigma_k \mid \mathcal{I}_1^k$$

and, by $[\mathrm{T}^{\mathrm{ri}}_{\wedge}]$, Ⓔ $P; M \Vdash e_1 \colon t_1' \mid \mathcal{I}_1''$ where

$$t_1' = t_1\sigma \wedge \textstyle\bigwedge_{k \in K} \sigma_k \qquad \mathcal{I}_1'' = \mathcal{I}_1' \sqcup \bigsqcup_{k \in K} \mathcal{I}_1^k\ .$$

We have $\Gamma \vDash P; M; \mathcal{I}_1''$ by Lemma 4.12 since $\mathcal{I}_1'' \sqsubseteq \mathcal{I}$.
Therefore, by IH from Ⓔ, we have Ⓕ $\Gamma \vdash e_1 \colon t_1'$.
We show $(\Gamma, \hat{x} \colon t_1') \vDash (P, \hat{x} \colon \langle M_1 \rangle t_1); M; \mathcal{I}_2$.
(It suffices to observe that $t_1' \leq t_1\sigma_k$ for all $k \in K$.)
Therefore, by IH from Ⓒ, we have Ⓖ $\Gamma, \hat{x} \colon t_1' \vdash e_2 \colon t$.
We conclude by $[\mathrm{T}_{\mathsf{let}}]$ from Ⓕ and Ⓖ.

Case: $[\mathrm{T}^{\mathrm{ri}}_{\leq}]$
Since $\Gamma \vDash P; M; \mathcal{I}$ and $M \leq M'$, we have $\Gamma \vDash P; M'; \mathcal{I}$.
We apply the IH and conclude using $[\mathrm{T}_{\leq}]$. □

Finally, we obtain that the two systems assign the same types to every expression in empty environments.

4.14 THEOREM (Equivalence of $\mathcal{T}^{\mathrm{i}}$ and $\mathcal{T}^{\mathrm{r}}$): For any e and t, $\varnothing \vdash e : t$ holds if and only if $\varnothing; \varnothing \Vdash e : t$.

Moreover, if $\varnothing \vdash e : t$ can be derived in $\mathcal{T}^{\mathrm{i}\setminus\wedge}$, then $\varnothing; \varnothing \Vdash e : t$ can be derived in $\mathcal{T}^{\mathrm{r}\setminus\wedge}$. □

Proof: If $\varnothing \vdash e : t$, we can obtain $\varnothing; \varnothing \Vdash e : t$ by Lemma 4.4 because $\varnothing; \varnothing \vDash \varnothing$ holds by Definition 4.3.

If $\varnothing; \varnothing \Vdash e : t$, by Lemma 4.8 we have $\varnothing; \varnothing \Vdash e : t \mid \mathcal{I}$ for some instantiation map $\mathcal{I}$ (in particular, $\mathcal{I}$ will be ϵ because the let-environment is empty). By Definition 4.11, we have $\varnothing \vDash \varnothing; \varnothing; \epsilon$. We obtain $\varnothing \vdash e : t$ by Lemma 4.13. □

Theorem 4.14 is the result we need to relate the two systems. It is inconvenient, however, that we have proven the equality for the full system, but only one implication for the restricted systems without $[\mathrm{T}_\wedge]$. This is unavoidable with this proof technique, but we conjecture that the equivalence also holds for the restricted systems. In particular, as suggested by Dolan and Mycroft (see footnote 3 on p. 93), typing an expression in the reformulated rules seems to correspond to typing a "lifted" expression in the standard rules, where this lifting ensures that let-bound expressions have no free λ-bound variables, for example by transforming $\lambda x.\, \mathsf{let}\ \hat{x} = (x, 3)\ \mathsf{in}\ \hat{x}\ \mathsf{to}\ \lambda x.\, \mathsf{let}\ \hat{x} = \lambda y.\, (y, 3)\ \mathsf{in}\ \hat{x}\ x$. If we proved that $\varnothing; \varnothing \Vdash e : t$ implies $\varnothing \vdash \mathsf{lift}(e) : t$, then it would only remain to prove that the latter implies $\varnothing \vdash e : t$, which seems intuitively correct. However, we have not attempted to develop this proof in detail yet.

4.1.4 Inversion for the type system $\mathcal{T}^{\mathrm{r}\setminus\wedge}$

We show here a result on the inversion of the typing rules $\mathcal{T}^{\mathrm{r}\setminus\wedge}$, that is, the reformulated typing rules without $[\mathrm{T}^{\mathrm{r}}_\wedge]$. We will use it later to relate this system to constraint satisfaction. Similarly to what we did in Section 3.3.5, we give a syntax-directed characterization of the system.

4.15 DEFINITION (Syntax-directed reformulated typing rules): The relation $P; M \Vdash_{\mathsf{sd}} e : t$ is defined inductively by the following rules.

$$\frac{}{P; M \Vdash_{\mathsf{sd}} \hat{x} : t} \begin{cases} P(\hat{x}) = \langle M' \rangle t' \\ \exists \sigma. \begin{cases} t'\sigma \le t \\ M \le M'\sigma \end{cases} \end{cases} \qquad \frac{}{P; M \Vdash_{\mathsf{sd}} x : t}\ M(x) \le t$$

$$\frac{}{P; M \Vdash_{\mathsf{sd}} c : t}\ b_c \le t$$

$$\frac{P; (M, x : t_1) \Vdash_{\mathsf{sd}} e : t_2}{P; M \Vdash_{\mathsf{sd}} \lambda x.\, e : t}\ t_1 \to t_2 \le t \qquad \frac{P; M \Vdash_{\mathsf{sd}} e_1 : t' \to t \qquad P; M \Vdash_{\mathsf{sd}} e_2 : t'}{P; M \Vdash_{\mathsf{sd}} e_1\ e_2 : t}$$

$$\frac{P; M \Vdash_{\mathsf{sd}} e_1 : t_1 \qquad P; M \Vdash_{\mathsf{sd}} e_2 : t_2}{P; M \Vdash_{\mathsf{sd}} (e_1, e_2) : t}\ t_1 \times t_2 \le t \qquad \frac{P; M \Vdash_{\mathsf{sd}} e : t_1 \times t_2}{P; M \Vdash_{\mathsf{sd}} \pi_i\ e : t_i}$$

$$\frac{P; M \Vdash_{\mathsf{sd}} e_0 : t_0 \qquad t_0 \leq \neg\mathbf{t} \text{ or } P; M \Vdash_{\mathsf{sd}} e_1 : t \qquad t_0 \leq \mathbf{t} \text{ or } P; M \Vdash_{\mathsf{sd}} e_2 : t}{P; M \Vdash_{\mathsf{sd}} (e_0 \in \mathbf{t}\ ?\ e_1 : e_2) : t}$$

$$\frac{P; M_1 \Vdash_{\mathsf{sd}} e_1 : t_1 \qquad (P, \hat{x} : \langle M_1 \rangle t_1); M \Vdash_{\mathsf{sd}} e_2 : t}{P; M \Vdash_{\mathsf{sd}} \mathsf{let}\ \hat{x} = e_1\ \mathsf{in}\ e_2 : t}\ \exists\sigma.\, M \leq M_1\sigma$$

□

Compared to the rules of $\mathcal{T}^{\mathrm{r}\setminus\wedge}$, the difference is that $[\mathrm{T}^{\mathrm{r}}_{\leq}]$ has been merged with the rules for variables, constants, functions, and pairs.

4.16 LEMMA: $P; M \Vdash_{\mathsf{sd}} e : t$ holds if and only if $P; M \Vdash e : t$ can be derived in $\mathcal{T}^{\mathrm{r}\setminus\wedge}$. □

Proof: First, we can prove by induction that

$$\left.\begin{array}{l} P; M' \Vdash_{\mathsf{sd}} e : t' \\ t' \leq t \\ M \leq M' \end{array}\right\} \implies P; M \Vdash_{\mathsf{sd}} e : t$$

(the proof is straightforward).

Using this fact, both implications are shown easily by induction. □

4.2 Constraints and constraint generation

In this section, we begin to describe type inference itself. Inference consists in constraint generation and constraint solving. Here, we introduce constraints and a notion of constraint satisfaction. We show how to generate constraints from expressions to describe the conditions under which an expression has a given type. Finally, we relate the type system $\mathcal{T}^{\mathrm{r}\setminus\wedge}$ (the reformulated system without $[\mathrm{T}^{\mathrm{r}}_{\wedge}]$) with constraint satisfaction, proving results of soundness and completeness.

4.2.1 Constraints and constraint satisfaction

We introduce two notions of constraint. The first, *type constraints* ($t_1 \mathrel{\dot{\leq}} t_2$), constrain a solution (a type substitution σ) to satisfy subtyping between two types (that is, to satisfy $t_1\sigma \leq t_2\sigma$).

4.17 DEFINITION (Type constraints and satisfaction): A *type constraint* is a term of the form ($t_1 \mathrel{\dot{\leq}} t_2$). A *type-constraint set* is a finite set of type constraints. We use the metavariable D to range over type-constraint sets.

A type substitution σ *satisfies* a type constraint ($t_1 \mathrel{\dot{\leq}} t_2$) if $t_1\sigma \leq t_2\sigma$; it satisfies a type-constraint set if it satisfies every type constraint in it. We write respectively $\sigma \Vdash (t_1 \mathrel{\dot{\leq}} t_2)$ and $\sigma \Vdash D$ to denote this relation.

When Δ is a finite set of type variables, we write $\sigma \Vdash_{\Delta} D$ to mean that $\sigma \Vdash D$ and that $\mathsf{dom}(\sigma) \sharp \Delta$. □

In the absence of let-polymorphism, the type inference problem can be reduced to solving such type constraints, as done by Wand (1987) for unification. In our setting, as for type inference for ML, it would force us to mix constraint generation with constraint solving. Therefore, we introduce *structured constraints*, which allow us to keep the two phases of constraint generation and constraint solving separate. These constraints can mention expression variables and include binders to introduce new variables. Constraints are closely related to those in the work of Pottier and Rémy (2005) on type inference for ML.

4.18 DEFINITION (Structured constraints): A *structured constraint* is a term C generated inductively by the following grammar:

$$C ::= (t \mathrel{\dot{\leq}} t) \mid (x \mathrel{\dot{\leq}} t) \mid (\hat{x} \mathrel{\dot{\leq}} t) \mid C \wedge C \mid C \vee C \mid \exists \vec{\alpha}.\, C$$
$$\mid \mathsf{def}\ x\colon t\ \mathsf{in}\ C \mid \mathsf{let}\ \hat{x}\colon \forall \alpha[C].\, \alpha\ \mathsf{in}\ C \qquad \square$$

Structured constraints are treated up to α-renaming of bound variables. In $\exists \vec{\alpha}.\, C$, the $\vec{\alpha}$ variables are bound in C. In $\mathsf{def}\ x\colon t\ \mathsf{in}\ C$, x is bound in C. In $\mathsf{let}\ \hat{x}\colon \forall \alpha[C_1].\, \alpha\ \mathsf{in}\ C_2$, α is bound in C_1 and $\hat{x}$ is bound in C_2.

Structured constraints include type constraints but also several other forms. The two forms $(x \mathrel{\dot{\leq}} t)$ and $(\hat{x} \mathrel{\dot{\leq}} t)$ constrain the type or typing scheme of the variable. Constraints include conjunction and disjunction. The existential constraint $\exists \vec{\alpha}.\, C$ introduces new type variables: this is useful to simplify freshness conditions. Finally, the def and let constraints introduce the two forms of expression variables and are used to describe the constraints for λ-abstractions and let constructs.

We describe the meaning of these constraints by defining a *constraint satisfaction* relation: it describes when two environments P and M and a type substitution σ satisfy a structured constraint C.

4.19 DEFINITION (Structured-constraint satisfaction): The *structured-constraint satisfaction* relation $P; M; \sigma \Vdash C$ is defined by the rules of Figure 4.4. $\square$

We refer to the rules of Figure 4.4 and the resulting relation as C^{sat}.

The rule $[\mathrm{C}^{\mathrm{sat}}_{\dot{\leq}}]$ corresponds to type-constraint satisfaction. The rule $[\mathrm{C}^{\mathrm{sat}}_{x}]$ can be understood as the combination of $[\mathrm{T}^{\mathrm{r}}_{x}]$ and $[\mathrm{T}^{\mathrm{r}}_{\leq}]$ of the reformulated system; likewise for $[\mathrm{C}^{\mathrm{sat}}_{\hat{x}}]$, which corresponds to $[\mathrm{T}^{\mathrm{r}}_{\hat{x}}]$ and $[\mathrm{T}^{\mathrm{r}}_{\leq}]$. The rules $[\mathrm{C}^{\mathrm{sat}}_{\wedge}]$, $[\mathrm{C}^{\mathrm{sat}}_{\vee}]$, and $[\mathrm{C}^{\mathrm{sat}}_{\exists}]$ are unsurprising. The rule $[\mathrm{C}^{\mathrm{sat}}_{\mathsf{def}}]$ expands the λ-environment, applying σ to t (note that, when the λ-environment is used in $[\mathrm{C}^{\mathrm{sat}}_{x}]$, σ is not applied to it because it has already been applied here). Finally, $[\mathrm{C}^{\mathrm{sat}}_{\mathsf{let}}]$ corresponds closely to $[\mathrm{T}^{\mathrm{r}}_{\mathsf{let}}]$.

4.2.2 Constraint generation

We now define a function $\langle\!\langle (\cdot)\colon (\cdot) \rangle\!\rangle$ that, given an expression e and a type t, yields a structured constraint $\langle\!\langle e\colon t \rangle\!\rangle$. This constraint expresses the conditions under which e has type $t\sigma$ for some type substitution σ.

$$[\mathrm{C}^{\mathrm{sat}}_{\dot\leq}]\ \frac{}{P; M; \sigma \Vdash (t_1 \mathrel{\dot\leq} t_2)}\ t_1\sigma \leq t_2\sigma$$

$$[\mathrm{C}^{\mathrm{sat}}_{x}]\ \frac{}{P; M; \sigma \Vdash (x \mathrel{\dot\leq} t)}\ M(x) \leq t\sigma \qquad [\mathrm{C}^{\mathrm{sat}}_{\hat{x}}]\ \frac{}{P; M; \sigma \Vdash (\hat{x} \mathrel{\dot\leq} t)} \begin{cases} P(\hat{x}) = \langle M_1 \rangle t_1 \\ \exists \sigma_1. \begin{cases} t_1\sigma_1 \leq t\sigma \\ M \leq M_1\sigma_1 \end{cases} \end{cases}$$

$$[\mathrm{C}^{\mathrm{sat}}_{\wedge}]\ \frac{P; M; \sigma \Vdash C_1 \qquad P; M; \sigma \Vdash C_2}{P; M; \sigma \Vdash C_1 \wedge C_2} \qquad [\mathrm{C}^{\mathrm{sat}}_{\vee}]\ \frac{P; M; \sigma \Vdash C_i}{P; M; \sigma \Vdash C_1 \vee C_2}$$

$$[\mathrm{C}^{\mathrm{sat}}_{\exists}]\ \frac{P; M; \sigma \cup [\vec{t}/\vec{\alpha}] \Vdash C}{P; M; \sigma \Vdash \exists \vec{\alpha}.\, C} \qquad [\mathrm{C}^{\mathrm{sat}}_{\mathsf{def}}]\ \frac{P; (M, x\colon t\sigma); \sigma \Vdash C}{P; M; \sigma \Vdash \mathsf{def}\ x\colon t\ \mathsf{in}\ C}$$

$$[\mathrm{C}^{\mathrm{sat}}_{\mathsf{let}}]\ \frac{P; M_1; \sigma_1 \Vdash C_1 \qquad (P, \hat{x}\colon \langle M_1 \rangle \alpha\sigma_1); M; \sigma \Vdash C_2}{P; M; \sigma \Vdash \mathsf{let}\ \hat{x}\colon \forall \alpha[C_1].\, \alpha\ \mathsf{in}\ C_2}\ \exists \sigma_1'.\, M \leq M_1\sigma_1'$$

FIGURE 4.4 C^{sat}: Constraint satisfaction rules

$$\begin{aligned}
\langle\!\langle \hat{x}\colon t \rangle\!\rangle &= (\hat{x} \mathrel{\dot\leq} t) \\
\langle\!\langle x\colon t \rangle\!\rangle &= (x \mathrel{\dot\leq} t) \\
\langle\!\langle c\colon t \rangle\!\rangle &= (b_c \mathrel{\dot\leq} t) \\
\langle\!\langle (\lambda x.\, e)\colon t \rangle\!\rangle &= \exists \alpha_1, \alpha_2.\, (\mathsf{def}\ x\colon \alpha_1\ \mathsf{in}\ \langle\!\langle e\colon \alpha_2 \rangle\!\rangle) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot\leq} t) && \alpha_1, \alpha_2 \sharp t \\
\langle\!\langle e_1\ e_2\colon t \rangle\!\rangle &= \exists \alpha.\, \langle\!\langle e_1\colon \alpha \to t \rangle\!\rangle \wedge \langle\!\langle e_2\colon t \rangle\!\rangle && \alpha \sharp t \\
\langle\!\langle (e_1, e_2)\colon t \rangle\!\rangle &= \exists \alpha_1, \alpha_2.\, \langle\!\langle e_1\colon \alpha_1 \rangle\!\rangle \wedge \langle\!\langle e_2\colon \alpha_2 \rangle\!\rangle \wedge (\alpha_1 \times \alpha_2 \mathrel{\dot\leq} t) && \alpha_1, \alpha_2 \sharp t \\
\langle\!\langle \pi_i\ e\colon t \rangle\!\rangle &= \exists \alpha_1, \alpha_2.\, \langle\!\langle e\colon \alpha_1 \times \alpha_2 \rangle\!\rangle \wedge (\alpha_i \mathrel{\dot\leq} t) && \alpha_1, \alpha_2 \sharp t \\
\langle\!\langle (e_0 \in \mathbf{t}\ ?\ e_1 : e_2)\colon t \rangle\!\rangle &= \exists \alpha.\, \langle\!\langle e_0\colon \alpha \rangle\!\rangle \wedge \big((\alpha \mathrel{\dot\leq} \neg\mathbf{t}) \vee \langle\!\langle e_1\colon t \rangle\!\rangle\big) \wedge \big((\alpha \mathrel{\dot\leq} \mathbf{t}) \vee \langle\!\langle e_2\colon t \rangle\!\rangle\big) && \alpha \sharp t \\
\langle\!\langle (\mathsf{let}\ \hat{x} = e_1\ \mathsf{in}\ e_2)\colon t \rangle\!\rangle &= \mathsf{let}\ \hat{x}\colon \forall \alpha[\langle\!\langle e_1\colon \alpha \rangle\!\rangle].\, \alpha\ \mathsf{in}\ \langle\!\langle e_2\colon t \rangle\!\rangle
\end{aligned}$$

FIGURE 4.5 Constraint generation

4.20 DEFINITION: The *constraint generation* function $\langle\!\langle (\cdot) \colon (\cdot) \rangle\!\rangle$ is defined by the equations in Figure 4.5. □

This definition is closely based on that of Pottier and Rémy (2005). We use def constraints to introduce function parameters. This, together with let constraints for let expressions, allows constraint generation to be described independently from the environment; thanks to this, we can keep constraint generation separate from constraint solving. For typecases, we use disjunctive constraints $\vee$ to translate the conditions "either … or …" in $[\mathrm{T}^{\mathrm{r}}_{\mathrm{case}}]$.

Note that the constraint for a function associates to it a single arrow type $\alpha_1 \to \alpha_2$: as anticipated, we do not attempt to infer intersection types.

We have mentioned that existential constraints simplify freshness conditions: indeed, many of the cases mention that the bound variables should be distinct from those that occur in t, but we do not need global conditions. It is easy to check that the free type variables in $\langle\!\langle e \colon t \rangle\!\rangle$ are exactly those in t.

4.2.3 Relating typing with constraint satisfaction

In this section we connect the reformulated type system with constraint generation and constraint satisfaction, by showing (for all P, M, e, t, and σ):

$$P; M \Vdash e \colon t\sigma \text{ is derivable in } \mathcal{T}^{\mathrm{r}\setminus\wedge} \iff P; M; \sigma \Vdash \langle\!\langle e \colon t \rangle\!\rangle \,.$$

The two implications are proven next as Lemma 4.21 (soundness of constraints w.r.t. typing) and Lemma 4.22 (completeness).

4.21 LEMMA: If $P; M; \sigma \Vdash \langle\!\langle e \colon t \rangle\!\rangle$, then $P; M \Vdash e \colon t\sigma$ is derivable in $\mathcal{T}^{\mathrm{r}\setminus\wedge}$. □

Proof: By induction on e and by case analysis on the shape of e.

Case: $e = \hat{x}$
We have $P; M; \sigma \Vdash (\hat{x} \mathrel{\dot{\leq}} t)$, therefore:

$$P(\hat{x}) = \langle M_1 \rangle t_1 \qquad t_1\sigma_1 \leq t\sigma \qquad M \leq M_1\sigma_1 \,.$$

We derive $P; M \Vdash \hat{x} \colon t\sigma$ by $[\mathrm{T}^{\mathrm{r}}_{\hat{x}}]$ and $[\mathrm{T}^{\mathrm{r}}_{\leq}]$.

Case: $e = x$
We have $P; M; \sigma \Vdash (x \mathrel{\dot{\leq}} t)$, therefore $M(x) \leq t\sigma$.
We derive $P; M \Vdash x \colon t\sigma$ by $[\mathrm{T}^{\mathrm{r}}_{x}]$ and $[\mathrm{T}^{\mathrm{r}}_{\leq}]$.

Case: $e = c$
We have $P; M; \sigma \Vdash (b_c \mathrel{\dot{\leq}} t)$, therefore $b_c\sigma \leq t\sigma$.
We derive $P; M \Vdash c \colon t\sigma$ by $[\mathrm{T}^{\mathrm{r}}_{c}]$ and $[\mathrm{T}^{\mathrm{r}}_{\leq}]$.

Case: $e = \lambda x.\, e'$
We have:

$$P; M; \sigma \Vdash \exists \alpha_1, \alpha_2.\, (\mathsf{def}\ x \colon \alpha_1\ \mathsf{in}\ \langle\!\langle e' \colon \alpha_2 \rangle\!\rangle) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t) \quad \alpha_1, \alpha_2 \sharp t \,.$$

Therefore there exist t_1 and t_2 such that

$$P; (M, x\colon t_1); (\sigma \cup [t_1/\alpha_1, t_2/\alpha_2]) \Vdash \langle\!\langle e'\colon \alpha_2 \rangle\!\rangle \qquad t_1 \to t_2 \leq t\sigma\ .$$

We apply the IH and conclude by $[\mathrm{T}^{\mathrm{r}}_{\lambda}]$ and $[\mathrm{T}^{\mathrm{r}}_{\leq}]$.

Case: $e = e_1\ e_2$
We have:

$$P; M; \sigma \Vdash \exists\alpha.\ \langle\!\langle e_1\colon \alpha \to t \rangle\!\rangle \wedge \langle\!\langle e_2\colon \alpha \rangle\!\rangle \qquad \alpha \sharp t\ .$$

Therefore there exists a t' such that

$$P; M; (\sigma \cup [t'/\alpha]) \Vdash \langle\!\langle e_1\colon \alpha \to t \rangle\!\rangle \qquad P; M; (\sigma \cup [t'/\alpha]) \Vdash \langle\!\langle e_2\colon \alpha \rangle\!\rangle\ .$$

We apply the IH and conclude by $[\mathrm{T}^{\mathrm{r}}_{\mathrm{app}}]$.

Case: $e = (e_1, e_2)$ or $e = \pi_i\ e'$
Similar to the previous cases.

Case: $e = (e_0 \in \mathbf{t}\ ?\ e_1 : e_2)$
We have

$$P; M; \sigma \Vdash \exists\alpha.\ \langle\!\langle e_0\colon \alpha \rangle\!\rangle \wedge \big((\alpha \mathrel{\dot{\leq}} \neg\mathbf{t}) \vee \langle\!\langle e_1\colon t \rangle\!\rangle\big) \wedge \big((\alpha \mathrel{\dot{\leq}} \mathbf{t}) \vee \langle\!\langle e_2\colon t \rangle\!\rangle\big)$$

(with $\alpha \sharp t$), therefore for some t' we have:

$$P; M; \sigma \cup [t'/\alpha] \Vdash \langle\!\langle e_0\colon \alpha \rangle\!\rangle$$
$$t' \leq \neg\mathbf{t} \text{ or } P; M; \sigma \cup [t'/\alpha] \Vdash \langle\!\langle e_1\colon t \rangle\!\rangle$$
$$t' \leq \mathbf{t} \text{ or } P; M; \sigma \cup [t'/\alpha] \Vdash \langle\!\langle e_2\colon t \rangle\!\rangle$$

By IH we obtain

$$P; M \Vdash e_0\colon t' \qquad t' \leq \neg\mathbf{t} \text{ or } P; M \Vdash e_1\colon t\sigma \qquad t' \leq \mathbf{t} \text{ or } P; M \Vdash e_2\colon t\sigma$$

and we conclude by $[\mathrm{T}^{\mathrm{r}}_{\mathrm{case}}]$.

Case: $e = (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)$
We have $P; M; \sigma \Vdash \mathsf{let}\ \hat{x}\colon \forall\alpha[\langle\!\langle e_1\colon \alpha \rangle\!\rangle].\ \alpha\ \mathsf{in}\ \langle\!\langle e_2\colon t \rangle\!\rangle$. Therefore

$$P; M_1; \sigma_1 \Vdash \langle\!\langle e_1\colon \alpha \rangle\!\rangle \quad (P, \hat{x}\colon \langle M_1\rangle\alpha\sigma_1); M; \sigma \Vdash \langle\!\langle e_2\colon t \rangle\!\rangle \quad M \leq M_1\sigma_1'\ .$$

By IH we have

$$P; M_1 \Vdash e_1\colon \alpha\sigma_1 \qquad (P, \hat{x}\colon \langle M_1\rangle\alpha\sigma_1); M \Vdash e_2\colon t\sigma$$

and we conclude by $[\mathrm{T}^{\mathrm{r}}_{\mathsf{let}}]$. □

To prove completeness of constraint generation and satisfaction with respect to $\mathcal{T}^{\mathrm{r}\setminus\wedge}$, we use Lemma 4.16 to invert the typing derivation for e.

4.22 LEMMA: If $P; M \Vdash e\colon t\sigma$ can be derived in $\mathcal{T}^{\mathrm{r}\setminus\wedge}$, then $P; M; \sigma \Vdash \langle\!\langle e\colon t \rangle\!\rangle$. □

Proof: By induction on e and by case analysis on the shape of e.
In each case, we use Lemma 4.16 to invert the judgment $P; M \Vdash e\colon t\sigma$.

Case: $e = \hat{x}$
We have
$$P(\hat{x}) = \langle M' \rangle t' \qquad t'\sigma' \leq t\sigma \qquad M \leq M'\sigma'$$
therefore $P; M; \sigma \Vdash (\hat{x} \mathrel{\dot{\leq}} t)$.

Case: $e = x$
We have $M(x) \leq t\sigma$, therefore $P; M; \sigma \Vdash (x \mathrel{\dot{\leq}} t)$.

Case: $e = c$
We have $b_c \leq t\sigma$, therefore (since b_c is ground) $P; M; \sigma \Vdash (b_c \mathrel{\dot{\leq}} t)$.

Case: $e = \lambda x.\, e'$
We have $P; (M, x\colon t_1) \Vdash e'\colon t_2$ and $t_1 \to t_2 \leq t\sigma$.
Let α_1 and α_2 be such that $\alpha_1, \alpha_2 \sharp t, \sigma$. Let $\hat{\sigma} = \sigma \cup [t_1/\alpha_1, t_2/\alpha_2]$.
Then, $\langle\!\langle e\colon t \rangle\!\rangle = \exists \alpha_1, \alpha_2.\, (\mathsf{def}\, x\colon \alpha_1\, \mathsf{in}\, \langle\!\langle e'\colon \alpha_2 \rangle\!\rangle) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)$, and we have $P; (M, x\colon \alpha_1\hat{\sigma}) \Vdash e'\colon \alpha_2\hat{\sigma}$.
Therefore, by IH, $P; (M, x\colon \alpha_1\hat{\sigma}); \hat{\sigma} \Vdash \langle\!\langle e'\colon \alpha_2 \rangle\!\rangle$.
Hence, we have $P; M; \sigma \Vdash \langle\!\langle e\colon t \rangle\!\rangle$.

Case: $e = e_1\, e_2$
We have $P; M \Vdash e_1\colon t' \to t\sigma$ and $P; M \Vdash e_2\colon t'$.
Let α be such that $\alpha \sharp t$. Let $\hat{\sigma} = \sigma \cup [t'/\alpha]$.
Then, $\langle\!\langle e\colon t \rangle\!\rangle = \exists \alpha.\, \langle\!\langle e_1\colon \alpha \to t \rangle\!\rangle \wedge \langle\!\langle e_2\colon \alpha \rangle\!\rangle$.
We have $P; M \Vdash e_1\colon (\alpha \to t)\hat{\sigma}$ and $P; M \Vdash e_2\colon \alpha\hat{\sigma}$.
Therefore, by IH,
$$P; M; \hat{\sigma} \Vdash \langle\!\langle e_1\colon \alpha \to t \rangle\!\rangle \qquad P; M; \hat{\sigma} \Vdash \langle\!\langle e_2\colon \alpha \rangle\!\rangle\,.$$
Hence, we have $P; M; \sigma \Vdash \langle\!\langle e\colon t \rangle\!\rangle$.

Case: $e = (e_1, e_2)$ or $e = \pi_i\, e'$
Analogous to the previous cases.

Case: $e = (e_0 \in \mathbf{t}\ ?\ e_1 : e_2)$
We have:
$$P; M \Vdash e_0\colon t_0 \qquad t_0 \leq \neg\mathbf{t} \text{ or } P; M \Vdash e_1\colon t\sigma \qquad t_0 \leq \mathbf{t} \text{ or } P; M \Vdash e_2\colon t\sigma$$
Let α be such that $\alpha \sharp t$. Let $\hat{\sigma} = \sigma \cup [t_0/\alpha]$.
Then, $\langle\!\langle e\colon t \rangle\!\rangle = \exists \alpha.\, \langle\!\langle e_0\colon \alpha \rangle\!\rangle \wedge \big((\alpha \mathrel{\dot{\leq}} \neg\mathbf{t}) \vee \langle\!\langle e_1\colon t \rangle\!\rangle\big) \wedge \big((\alpha \mathrel{\dot{\leq}} \mathbf{t}) \vee \langle\!\langle e_2\colon t \rangle\!\rangle\big)$.
By IH, we have
$$P; M; \hat{\sigma} \Vdash \langle\!\langle e_0\colon \alpha \rangle\!\rangle$$
$$\alpha\hat{\sigma} \leq \neg\mathbf{t} \text{ or } P; M; \hat{\sigma} \Vdash \langle\!\langle e_1\colon t \rangle\!\rangle \qquad \alpha\hat{\sigma} \leq \mathbf{t} \text{ or } P; M; \hat{\sigma} \Vdash \langle\!\langle e_2\colon t \rangle\!\rangle$$
and therefore $P; M; \sigma \Vdash \langle\!\langle e\colon t \rangle\!\rangle$.

Case: $e = (\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2)$
We have:
$$P; M_1 \Vdash e_1\colon t_1 \qquad (P, \hat{x}\colon \langle M_1 \rangle t_1); M \Vdash e_2\colon t\sigma \qquad M \leq M_1\sigma'$$

We choose a type variable α, and we have $P; M_1 \Vdash e_1 : \alpha[t_1/\alpha]$.
Therefore, by IH,

$$P; M_1; [t_1/\alpha] \Vdash \langle\!\langle e_1 : \alpha \rangle\!\rangle \qquad (P, \hat{x} : \langle M_1 \rangle t_1); M; \sigma \Vdash \langle\!\langle e : t \rangle\!\rangle$$

and we obtain $P; M; \sigma \Vdash \langle\!\langle e : t \rangle\!\rangle$. □

4.2.4 Properties of structured-constraint satisfaction

We prove two weakening properties of structured-constraint satisfaction that we use in the next section to relate it to algorithmic constraint solving.

4.23 LEMMA: If $P; M; \sigma \Vdash C$ and $M' \leq M$, then $P; M'; \sigma \Vdash C$. □

Proof: Straightforward proof by structural induction on C. □

We introduce an order of generality on typing schemes and let-environments analogous to that of Definition 3.14 and its alternative characterization in Lemma 3.15. We write $\langle M_1 \rangle t_1 \leq^{\forall} \langle M_2 \rangle t_2$ if there exists σ_1 such that $t_1\sigma_1 \leq t_2$ and $M_2 \leq M_1\sigma_1$. We extend this pointwise to let-environments.

4.24 LEMMA: If $P'; M; \sigma \Vdash C$ and $P \leq^{\forall} P'$, then $P; M; \sigma \Vdash C$. □

Proof: By structural induction on C and by case analysis on the shape of C. All cases are straightforward except the following two.

Case: $C = (\hat{x} \mathrel{\dot{\leq}} t)$
We have:

$$P'(\hat{x}) = \langle M_1' \rangle t_1' \qquad t_1'\sigma_1 \leq t\sigma \qquad M \leq M_1'\sigma_1 \,.$$

Since $P \leq^{\forall} P'$, we have $P'(\hat{x}) = \langle M_1 \rangle t_1$ and there exists a σ' such that $t_1\sigma' \leq t_1'$ and $M_1' \leq M_1\sigma'$.
Hence, $t_1(\sigma_1 \circ \sigma') \leq t\sigma$ and $M \leq M_1(\sigma_1 \circ \sigma')$. We conclude by $[\mathrm{C}^{\mathrm{sat}}_{\hat{x}}]$.

Case: $C = (\mathsf{let}\ \hat{x} : \forall\alpha[C_1].\ \alpha\ \mathsf{in}\ C_2)$
We have:

$$P'; M; \sigma \Vdash \mathsf{let}\ \hat{x} : \forall\alpha[C_1].\ \alpha\ \mathsf{in}\ C_2$$
$$P'; M_1; \sigma_1 \Vdash C_1 \qquad (P', \hat{x} : \langle M_1 \rangle \alpha\sigma_1); M; \sigma \Vdash C_2 \qquad M \leq M_1\sigma_1'$$

Note that $(P, \hat{x} : \langle M_1 \rangle \alpha\sigma_1) \leq^{\forall} (P', \hat{x} : \langle M_1 \rangle \alpha\sigma_1)$.
By IH we obtain:

$$P; M_1; \sigma_1 \Vdash C_1 \qquad (P, \hat{x} : \langle M_1 \rangle \alpha\sigma_1); M; \sigma \Vdash C_2$$

and we conclude by $[\mathrm{C}^{\mathrm{sat}}_{\mathsf{let}}]$. □

This concludes the study of constraint satisfaction from a declarative perspective: in the next section, we show how to look for solutions algorithmically.

4.3 Constraint solving

To solve type-constraint sets, we reuse the *tallying* algorithm of Castagna et al. (2015b). We do not describe the algorithm in detail here: we state some properties of it below and rely only on them in the rest of the development. Then, we show how to solve structured constraints by simplifying them to type-constraint sets that can be solved by tallying.

4.3.1 Type-constraint solving by tallying

The *tallying* problem, as defined by Castagna et al. (2015b), is the problem of finding solutions to type-constraint sets. It is the analogue of the unification problem for subtyping, instead of equality, constraints.

The authors of the cited work study the problem in order to do local type inference for an explicitly typed polymorphic language with set-theoretic types (specifically, to infer instantiations of polymorphic functions). They define a sound and complete algorithm to solve tallying. We refer to this algorithm as tally, and assume it has the following properties.

4.25 PROPERTY: There exists a function $\mathsf{tally}_{(\cdot)}(\cdot)$ such that, when D is a type-constraint set and Δ is a finite set of type variables, $\mathsf{tally}_\Delta(D)$ is a finite set of type substitutions. Moreover, the following properties hold.

- *Soundness:* if $\sigma \in \mathsf{tally}_\Delta(D)$, then $\sigma \Vdash_\Delta D$.
- *Completeness:* if $\sigma \Vdash_\Delta D$, then there exist $\sigma' \in \mathsf{tally}_\Delta(D)$ and σ'' such that $\sigma \simeq \sigma'' \circ \sigma'$.
- If $\sigma \in \mathsf{tally}_\Delta(D)$, then $\mathsf{dom}(\sigma) \subseteq \mathsf{var}(D) \setminus \Delta$.

We write $\mathsf{tally}(D)$ to abbreviate $\mathsf{tally}_\varnothing(D)$. □

These results are proven as Theorems C.45 and C.46 in Castagna et al. (2015b). Moreover, Theorem C.47 states that tally always terminates.

The soundness property is straightforward. In the statement of completeness, by $\sigma \simeq \sigma'' \circ \sigma'$ we mean that $\alpha\sigma \simeq \alpha\sigma'\sigma''$ for every α. Note that tallying does not yield a single type substitution, but a finite set of them. If $\mathsf{tally}_\Delta(D) = \varnothing$, then (by completeness) there exists no σ such that $\sigma \Vdash_\Delta (D)$. Otherwise, all the type substitutions in $\mathsf{tally}_\Delta(D)$ are solutions, and every other solution can be obtained from one of them (by composition with some other substitution and up to equivalence $\simeq$). In this sense, the set of type substitutions is a principal solution, though none of the substitutions is itself principal.

We give two examples of why the principal solution cannot be a single type substitution. The constraint $\alpha_1 \times \alpha_2 \mathrel{\dot{\leq}} (\mathsf{Int} \times \mathsf{Int}) \vee (\mathsf{Bool} \times \mathsf{Bool})$ has two incomparable solutions: $[\mathsf{Int}/\alpha_1, \mathsf{Int}/\alpha_2]$ and $[\mathsf{Bool}/\alpha_1, \mathsf{Bool}/\alpha_2]$; no solution is more general than both. Likewise, the constraint $\mathsf{Int} \to \mathsf{Bool} \mathrel{\dot{\leq}} \alpha \to \beta$ has a solution $[(\mathsf{Int} \wedge \alpha)/\alpha, (\mathsf{Bool} \vee \beta)/\beta]$ (which is valid because $\mathsf{Int} \to \mathsf{Bool} \leq (\mathsf{Int} \wedge \alpha) \to (\mathsf{Bool} \vee \beta)$), but also $[\mathbb{0}/\alpha]$ (which is valid because arrow types of the form $\mathbb{0} \to t$ are greater than any arrow type).

$$[\mathrm{C}^{\mathrm{sim}}_{\dot\leq}]\ \frac{}{P \vdash (t_1 \mathrel{\dot\leq} t_2) \rightsquigarrow \{t_1 \mathrel{\dot\leq} t_2\} \mid \varnothing \mid \varnothing} \qquad [\mathrm{C}^{\mathrm{sim}}_{x}]\ \frac{}{P \vdash (x \mathrel{\dot\leq} t) \rightsquigarrow \varnothing \mid (x\colon t) \mid \varnothing}$$

$$[\mathrm{C}^{\mathrm{sim}}_{\hat x}]\ \frac{}{P \vdash (\hat x \mathrel{\dot\leq} t) \rightsquigarrow \{t_1[\vec\beta/\vec\alpha] \mathrel{\dot\leq} t\} \mid M_1[\vec\beta/\vec\alpha] \mid \vec\beta}\ \begin{cases} P(\hat x) = \langle M_1\rangle t_1 \\ \vec\alpha = \mathsf{var}(\langle M_1\rangle t_1) \\ \vec\beta \sharp t \end{cases}$$

$$[\mathrm{C}^{\mathrm{sim}}_{\wedge}]\ \frac{P \vdash C_1 \rightsquigarrow D_1 \mid M_1 \mid \vec\alpha_1 \qquad P \vdash C_2 \rightsquigarrow D_2 \mid M_2 \mid \vec\alpha_2}{P \vdash C_1 \wedge C_2 \rightsquigarrow D_1 \cup D_2 \mid M_1 \wedge M_2 \mid \vec\alpha_1 \cup \vec\alpha_2}\ \begin{cases} \vec\alpha_1 \sharp \vec\alpha_2, C_2 \\ \vec\alpha_2 \sharp C_1 \end{cases}$$

$$[\mathrm{C}^{\mathrm{sim}}_{\vee}]\ \frac{P \vdash C_i \rightsquigarrow D \mid M \mid \vec\alpha}{P \vdash C_1 \vee C_2 \rightsquigarrow D \mid M \mid \vec\alpha} \qquad [\mathrm{C}^{\mathrm{sim}}_{\exists}]\ \frac{P \vdash C \rightsquigarrow D \mid M \mid \vec\alpha'}{P \vdash \exists\vec\alpha.\, C \rightsquigarrow D \mid M \mid \vec\alpha' \cup \vec\alpha}\ \vec\alpha' \sharp \vec\alpha$$

$$[\mathrm{C}^{\mathrm{sim}}_{\mathsf{def}}]\ \frac{P \vdash C \rightsquigarrow D \mid M \mid \vec\alpha}{P \vdash \mathsf{def}\ x\colon t\ \mathsf{in}\ C \rightsquigarrow D \cup D' \mid M \setminus x \mid \vec\alpha}\ \begin{cases} D' = \begin{cases} \{t \mathrel{\dot\leq} M(x)\} & \text{if } x \in \mathsf{dom}(M) \\ \varnothing & \text{otherwise} \end{cases} \\ \vec\alpha \sharp t \end{cases}$$

$$[\mathrm{C}^{\mathrm{sim}}_{\mathsf{let}}]\ \frac{\begin{array}{c} P \vdash C_1 \rightsquigarrow D_1 \mid M_1 \mid \vec\alpha_1 \\ (P, \hat x\colon \langle M_1\sigma_1\rangle\alpha\sigma_1) \vdash C_2 \rightsquigarrow D_2 \mid M_2 \mid \vec\alpha_2 \end{array}}{P \vdash \mathsf{let}\ \hat x\colon \forall\alpha[C_1].\, \alpha\ \mathsf{in}\ C_2 \rightsquigarrow D_2 \mid M \mid \vec\alpha_2 \cup \vec\beta}\ \begin{cases} \sigma_1 \in \mathsf{tally}(D_1) \\ \vec\alpha = \mathsf{var}(M_1\sigma_1) \\ M = M_1\sigma_1[\vec\beta/\vec\alpha] \wedge M_2 \\ \vec\alpha_1 \sharp \alpha \\ \vec\beta \sharp C_1, \vec\alpha_2 \end{cases}$$

FIGURE 4.6 C^{sim}: Constraint simplification rules

REMARK (Introduction of fresh type variables in tally): The tallying algorithm described by Castagna et al. (2015b) introduces new type variables to convert subtyping constraints to equations: for example, $(\alpha \mathrel{\dot\leq} \mathsf{Int})$ becomes $\alpha = \mathsf{Int} \wedge \alpha'$. If $\vec\alpha$ are the type variables in the type-constraint set, then tallying introduces new variables $\vec\alpha'$, each corresponding to one in $\vec\alpha$.

In our description, we assume that tally returns type substitutions where we have already performed a renaming $[\vec\alpha/\vec\alpha']$ to map each new variable to the original one. For example, for the constraint above we assume that tally returns $[(\mathsf{Int} \wedge \alpha)/\alpha]$ instead of $[(\mathsf{Int} \wedge \alpha')/\alpha]$. As a result, the type substitutions in general are not idempotent, unlike in the specification of Castagna et al. (2015b).

This allows us to state completeness as we do. If $\hat\sigma \in \mathsf{tally}_\Delta(D)$ introduced new type variables, then we would have $(\sigma \cup \check\sigma) \simeq (\sigma \cup \check\sigma) \circ \hat\sigma$ instead of $\sigma \simeq \check\sigma \circ \hat\sigma$. This is because the new variables introduced by $\hat\sigma$, being fresh, would be different from those in the domain of σ, and $\check\sigma$ would need to instantiate them. □

4.3.2 Structured-constraint simplification

We solve structured constraints by simplifying them to type-constraint sets which can be solved by tallying. Because of let-polymorphism, the constraint simplification algorithm also uses tallying internally to simplify let constraints.

4.26 DEFINITION: The structured-constraint simplification relation $P \vdash C \rightsquigarrow D \mid M \mid \vec{\alpha}$ is defined by the rules in Figure 4.6. □

We refer to this system as C^{sim}. The rules are syntax-directed. We can read them as an algorithm that takes two inputs, a let-environment P and a structured constraint C, and produces three outputs: the type-constraint set D which we then solve by tallying, the λ-environment M which collects the $(x \mathrel{\dot{\leq}} t)$ constraints in C, and the vector $\vec{\alpha}$ of the type variables introduced during simplification (for example, to instantiate existential constraints).

The rules $[\mathrm{C}^{\text{sim}}_{\dot{\leq}}]$ and $[\mathrm{C}^{\text{sim}}_{x}]$ are straightforward. In $[\mathrm{C}^{\text{sim}}_{\hat{x}}]$, we take a fresh instance of the typing scheme $P(\hat{x})$, instantiating all its type variables with the new variables $\vec{\beta}$. The rules for conjunctive, disjunctive, and existential constraints are unsurprising. In $[\mathrm{C}^{\text{sim}}_{\text{def}}]$, we simplify the constraint C and then add one more constraint $(t \mathrel{\dot{\leq}} M(x))$, unless x is never used and thus does not occur in M, to remove the binding of x from M. Therefore, the domain of the λ-environment obtained from simplification is always the set of free λ-variables in C. In $[\mathrm{C}^{\text{sim}}_{\text{let}}]$, we first simplify the constraint C_1 and solve the resulting D_1 using tallying. We use a solution σ_1 to obtain the typing scheme for $\hat{x}$ and simplify C_2 in the expanded environment. The final λ-environment we return is the intersection of M_2 and a fresh renaming of $M_1\sigma_1$: this corresponds to the condition $M \leq M_1\sigma_1'$ in $[\mathrm{C}^{\text{sat}}_{\text{let}}]$. In most rules, the side conditions force the choice of fresh variables.

Constraint simplification is not deterministic: we can build different derivations from the same P and C. Apart from the choice of different variables for $\vec{\alpha}$ (which is immaterial as long as the disjointness conditions are satisfied) there are two sources of non-determinism: disjunctive constraints and the side-condition $\sigma_1 \in \mathsf{tally}(D_1)$ in $[\mathrm{C}^{\text{sim}}_{\text{let}}]$, since $\mathsf{tally}(D_1)$ can contain more than one type substitution. This means that a practical implementation will have to test multiple possible choices by backtracking, possibly compromising efficiency (we outline in Section 4.4.1 two approaches to mitigate this problem).

We want to connect structured-constraint satisfaction with simplification. First, we describe which type variables can occur in the D and M that we obtain by simplification.

We define $\mathsf{var}(\cdot)$ on type-constraint sets and on structured constraints. For type-constraint sets, we define $\mathsf{var}(D) = \bigcup_{(t_1 \mathrel{\dot{\leq}} t_2) \in D} \mathsf{var}(t_1) \cup \mathsf{var}(t_2)$. For struc-

tured constraints, we must consider binders, as follows.

$$\mathsf{var}(t_1 \mathrel{\dot\leq} t_2) = \mathsf{var}(t_1) \cup \mathsf{var}(t_2) \qquad \mathsf{var}(x \mathrel{\dot\leq} t) = \mathsf{var}(t) \qquad \mathsf{var}(\hat{x} \mathrel{\dot\leq} t) = \mathsf{var}(t)$$
$$\mathsf{var}(C_1 \wedge C_2) = \mathsf{var}(C_1) \cup \mathsf{var}(C_2) \qquad \mathsf{var}(C_1 \vee C_2) = \mathsf{var}(C_1) \cup \mathsf{var}(C_2)$$
$$\mathsf{var}(\exists \vec{\alpha}.\, C) = \mathsf{var}(C) \setminus \vec{\alpha} \qquad \mathsf{var}(\mathsf{def}\ x\colon t\ \mathsf{in}\ C) = \mathsf{var}(t) \cup \mathsf{var}(C)$$
$$\mathsf{var}(\mathsf{let}\ \hat{x}\colon \forall \alpha[C_1].\, \alpha\ \mathsf{in}\ C_2) = (\mathsf{var}(C_1) \setminus \{\alpha\}) \cup \mathsf{var}(C_2)$$

4.27 LEMMA: If $P \vdash C \rightsquigarrow D \mid M \mid \vec{\alpha}$, then $\mathsf{var}(D) \cup \mathsf{var}(M) \subseteq \mathsf{var}(C) \cup \vec{\alpha}$. □

Proof: Straightforward proof by structural induction on C. □

The following lemma proves that simplification is sound with respect to structured-constraint satisfaction.

4.28 LEMMA: If $P \vdash C \rightsquigarrow D \mid M \mid \vec{\alpha}$ and $\sigma \Vdash D$, then $P; M\sigma; \sigma|_{\setminus \vec{\alpha}} \Vdash C$. □

Proof: By structural induction on C.

Case: $C = (t_1 \mathrel{\dot\leq} t_2)$
Straightforward, because we have $t_1\sigma \leq t_2\sigma$ and $\sigma|_{\setminus \varnothing} = \sigma$.

Case: $C = (x \mathrel{\dot\leq} t)$
Straightforward: we must show $P; (x\colon t)\sigma; \sigma|_{\setminus \varnothing} \Vdash (x \mathrel{\dot\leq} t)$, which just requires $t\sigma \leq t\sigma|_{\setminus \varnothing}$.

Case: $C = (\hat{x} \mathrel{\dot\leq} t)$
We have

$$P \vdash C \rightsquigarrow \{t_1[\vec{\beta}/\vec{\alpha}] \mathrel{\dot\leq} t\} \mid M_1[\vec{\beta}/\vec{\alpha}] \mid \vec{\beta} \qquad t_1[\vec{\beta}/\vec{\alpha}]\sigma \leq t\sigma$$
$$P(\hat{x}) = \langle M_1 \rangle t_1 \qquad \vec{\alpha} = \mathsf{var}(\langle M_1 \rangle t_1) \qquad \vec{\beta} \sharp t$$

and we must show $P; M_1[\vec{\beta}/\vec{\alpha}]\sigma; \sigma|_{\setminus \vec{\beta}} \Vdash C$, which requires finding a σ_1 such that

$$t_1\sigma_1 \leq t\sigma|_{\setminus \vec{\beta}} \qquad M_1[\vec{\beta}/\vec{\alpha}]\sigma \leq M_1\sigma_1\ .$$

We choose $\sigma_1 = [\vec{\beta}/\vec{\alpha}]\sigma$. Note that $t\sigma = t\sigma|_{\setminus \vec{\beta}}$ since $\vec{\beta} \sharp t$.

Case: $C = (C_1 \wedge C_2)$
We have:

$$P \vdash C_1 \wedge C_2 \rightsquigarrow D_1 \cup D_2 \mid M_1 \wedge M_2 \mid \vec{\alpha}_1 \cup \vec{\alpha}_2 \qquad \sigma \Vdash D_1 \cup D_2$$
$$P \vdash C_1 \rightsquigarrow D_1 \mid M_1 \mid \vec{\alpha}_1 \qquad P \vdash C_2 \rightsquigarrow D_2 \mid M_2 \mid \vec{\alpha}_2$$
$$\vec{\alpha}_1 \sharp \vec{\alpha}_2, C_2 \qquad \vec{\alpha}_2 \sharp C_1$$

Since $\vec{\alpha}_1 \sharp \vec{\alpha}_1, C_2$, by Lemma 4.27 we have $\vec{\alpha}_1 \sharp D_2, M_2$.
Analogously, $\vec{\alpha}_2 \sharp D_1, M_1$.
Therefore, $\sigma|_{\setminus \vec{\alpha}_2} \Vdash D_1$ and $\sigma|_{\setminus \vec{\alpha}_1} \Vdash D_2$.

By IH, we obtain:

$$P; M_1\sigma|_{\setminus\vec{\alpha}_2}; \sigma|_{\setminus(\vec{\alpha}_1\cup\vec{\alpha}_2)} \Vdash C_1 \qquad P; M_2\sigma|_{\setminus\vec{\alpha}_1}; \sigma|_{\setminus(\vec{\alpha}_1\cup\vec{\alpha}_2)} \Vdash C_2$$

We conclude because $M_1\sigma|_{\setminus\vec{\alpha}_2} = M_1\sigma$ and $M_2\sigma|_{\setminus\vec{\alpha}_1} = M_2\sigma$.

Case: $C = (C_1 \vee C_2)$
We have:

$$P \vdash C_1 \vee C_2 \rightsquigarrow D \mid M \mid \vec{\alpha} \qquad \exists i.\ P \vdash C_i \rightsquigarrow D \mid M \mid \vec{\alpha}$$

By IH we obtain $P; M\sigma; \sigma|_{\setminus\vec{\alpha}} \Vdash C_i$. Therefore, $P; M\sigma; \sigma|_{\setminus\vec{\alpha}} \Vdash C_1 \vee C_2$.

Case: $C = (\exists\vec{\alpha}.\, C')$
We have:

$$P \vdash \exists\vec{\alpha}.\, C' \rightsquigarrow D \mid M \mid \vec{\alpha}' \cup \vec{\alpha} \qquad P \vdash C' \rightsquigarrow D \mid M \mid \vec{\alpha}'$$

By IH we obtain $P; M\sigma; \sigma|_{\setminus\vec{\alpha}'} \Vdash C'$.
Since $\sigma|_{\setminus\vec{\alpha}'} = \sigma|_{\setminus(\vec{\alpha}'\cup\vec{\alpha})} \cup [\vec{\alpha}\sigma/\vec{\alpha}]$, we have $P; M\sigma; \sigma|_{\setminus(\vec{\alpha}'\cup\vec{\alpha})} \cup [\vec{\alpha}\sigma/\vec{\alpha}] \Vdash C'$.
Therefore, $P; M\sigma; \sigma|_{\setminus(\vec{\alpha}'\cup\vec{\alpha})} \Vdash \exists\vec{\alpha}.\, C'$.

Case: $C = (\mathsf{def}\ x\colon t\ \mathsf{in}\ C')$
We have:

$$P \vdash \mathsf{def}\ x\colon t\ \mathsf{in}\ C' \rightsquigarrow D' \cup \{\, t \mathrel{\dot{\leq}} M'(x) \mid x \in \mathsf{dom}(M') \,\} \mid M' \setminus x \mid \vec{\alpha}$$
$$P \vdash C' \rightsquigarrow D' \mid M' \mid \vec{\alpha} \qquad \vec{\alpha} \sharp t$$

Since $\sigma \Vdash D$, we have $\sigma \Vdash D'$ and, if $x \in \mathsf{dom}(M')$, $t\sigma \leq M'(x)\sigma$.
By IH we obtain $P; M'\sigma; \sigma|_{\setminus\vec{\alpha}} \Vdash C'$.
We have $\big((M' \setminus x)\sigma, x\colon t(\sigma|_{\setminus\vec{\alpha}})\big) \leq M'\sigma$.
This amounts to showing that $t(\sigma|_{\setminus\vec{\alpha}}) \leq M'(x)\sigma$ if $x \in \mathsf{dom}(M')$.
It holds because, if $x \in \mathsf{dom}(M')$, $t\sigma \leq M'(x)\sigma$, and because $\vec{\alpha} \sharp t$.
By Lemma 4.23, we obtain $P; \big((M' \setminus x)\sigma, x\colon t(\sigma|_{\setminus\vec{\alpha}})\big); \sigma|_{\setminus\vec{\alpha}} \Vdash C'$.
Therefore, $P; (M' \setminus x)\sigma; \sigma|_{\setminus\vec{\alpha}} \Vdash C$.

Case: $C = (\mathsf{let}\ \hat{x}\colon \forall\alpha[C_1].\, \alpha\ \mathsf{in}\ C_2)$
We have:

$$P \vdash C \rightsquigarrow D_2 \mid M_1\sigma_1[\vec{\beta}/\vec{\alpha}] \wedge M_2 \mid \vec{\alpha}_2 \cup \vec{\beta} \qquad \sigma \Vdash D_2$$
$$P \vdash C_1 \rightsquigarrow D_1 \mid M_1 \mid \vec{\alpha}_1 \qquad (P, \hat{x}\colon \langle M_1\sigma_1\rangle\alpha\sigma_1) \vdash C_2 \rightsquigarrow D_2 \mid M_2 \mid \vec{\alpha}_2$$
$$\sigma_1 \in \mathsf{tally}(D_1) \qquad \vec{\alpha} = \mathsf{var}(M_1\sigma_1) \qquad \vec{\alpha}_1 \sharp \alpha \qquad \vec{\beta} \sharp C_1, \vec{\alpha}_2$$

By Property 4.25, we have $\sigma_1 \Vdash D_1$.
By Lemma 4.27, since $\vec{\beta} \sharp C_1, \vec{\alpha}_2$, then $\vec{\beta} \sharp D_2$. Therefore, $\sigma|_{\setminus\vec{\beta}} \Vdash D_2$.
By IH we obtain:

$$P; M_1\sigma_1; \sigma_1|_{\setminus\vec{\alpha}_1} \Vdash C_1 \qquad (P, \hat{x}\colon \langle M_1\sigma_1\rangle\alpha\sigma_1); M_2\sigma|_{\setminus\vec{\beta}}; \sigma_2|_{\setminus(\vec{\alpha}_2\cup\vec{\beta})} \Vdash C_2$$

We have $\alpha\sigma_1 = \alpha\sigma_1|_{\setminus\vec{\alpha}_1}$ because $\vec{\alpha}_1 \sharp \alpha$.
We have $M_2\sigma|_{\setminus\vec{\beta}} = M_2\sigma$ because $\vec{\beta} \sharp M_2$ (by Lemma 4.27).
Therefore, we have $(P, \hat{x}\colon \langle M_1\sigma_1\rangle\alpha\sigma_1|_{\setminus\vec{\alpha}_1}); M_2\sigma; \sigma_2|_{\setminus(\vec{\alpha}_2\cup\vec{\beta})} \Vdash C_2$.

We have $(M_1\sigma_1[\vec{\beta}/\vec{\alpha}] \wedge M_2)\sigma \leq M_2\sigma$.
Therefore, by Lemma 4.23,

$$(P, \hat{x}\colon \langle M_1\sigma_1\rangle\alpha\sigma_1|_{\setminus\vec{\alpha}_1}); (M_1\sigma_1[\vec{\beta}/\vec{\alpha}] \wedge M_2)\sigma; \sigma_2|_{\setminus(\vec{\alpha}_2\cup\vec{\beta})} \Vdash C_2 \ .$$

To conclude, we also need to find σ_1' such that

$$(M_1\sigma_1[\vec{\beta}/\vec{\alpha}] \wedge M_2)\sigma \leq M_1\sigma_1\sigma_1' \ :$$

we take $\sigma_1' = [\vec{\beta}/\vec{\alpha}]\sigma$. □

Completeness of structured-constraint simplification is proven by the following lemma.

4.29 LEMMA:

$$P; M; \sigma \Vdash C \implies \exists D, M', \vec{\alpha}, \sigma'. \begin{cases} P \vdash C \rightsquigarrow D \mid M' \mid \vec{\alpha} \\ \sigma \cup \sigma' \Vdash D \\ M \leq M'(\sigma \cup \sigma') \\ \mathsf{dom}(\sigma') \subseteq \vec{\alpha} \end{cases}$$

□

Proof: We use the metavariable $\mathfrak{U}$ to range over infinite subsets of TVar. We prove the following stronger claim (for all P, M, σ, C, and $\mathfrak{U}$).

$$\left.\begin{array}{l} P; M; \sigma \Vdash C \\ \mathfrak{U} \sharp C \end{array}\right\} \implies \exists D, M', \vec{\alpha}, \sigma'. \begin{cases} P \vdash C \rightsquigarrow D \mid M' \mid \vec{\alpha} \\ \sigma \cup \sigma' \Vdash D \\ M \leq M'(\sigma \cup \sigma') \\ \mathsf{dom}(\sigma') \subseteq \vec{\alpha} \subseteq \mathfrak{U} \end{cases}$$

This implies the statement: take $\mathfrak{U}$ to be $\mathsf{TVar} \setminus \mathsf{var}(C)$.

We prove the claim by structural induction on C.

Case: $C = (t_1 \dot{\leq} t_2)$
Straightforward: take $D = \{t_1 \dot{\leq} t_2\}$, $M' = \varnothing$, $\vec{\alpha}$ empty, and $\sigma' = [\,]$.

Case: $C = (x \dot{\leq} t)$
Take $D = \varnothing$, $M' = (x\colon t)$, $\vec{\alpha}$ empty, and $\sigma' = [\,]$.
We have $M \leq (x\colon t)(\sigma \cup \sigma')$ because $M(x) \leq t\sigma$.

Case: $C = (\hat{x} \dot{\leq} t)$
By hypothesis:

$$P(\hat{x}) = \langle M_1\rangle t_1 \qquad t_1\sigma_1 \leq t\sigma \qquad M \leq M_1\sigma_1 \ .$$

Let $\vec{\alpha}_1 = \mathsf{var}(\langle M_1\rangle t_1)$ and choose $\vec{\alpha}$ in $\mathfrak{U}$ (this ensures $\vec{\alpha} \sharp t$, since $\mathfrak{U} \sharp C$).
Then, we have $P \vdash (\hat{x} \dot{\leq} t) \rightsquigarrow \{t_1[\vec{\alpha}/\vec{\alpha}_1] \dot{\leq} t\} \mid M_1[\vec{\alpha}/\vec{\alpha}_1] \mid \vec{\alpha}$.
Take $\sigma' = [\vec{\alpha}_1\sigma_1/\vec{\alpha}]$.

We have:

$$t_1[\vec{\alpha}/\vec{\alpha}_1](\sigma \cup \sigma') = t_1\sigma_1 \leq t\sigma = t(\sigma \cup \sigma_1)$$
$$M \leq M_1\sigma_1 = M_1[\vec{\alpha}/\vec{\alpha}_1](\sigma \cup \sigma') \,.$$

Case: $C = (C_1 \wedge C_2)$
By hypothesis, we have $P; M; \sigma \Vdash C_1$ and $P; M; \sigma \Vdash C_2$.
We partition $\mathfrak{U}$ into two infinite sets $\mathfrak{U}_1$ and $\mathfrak{U}_2$.
By IH, we have:

$$\begin{aligned} P \vdash C_1 \rightsquigarrow D_1 \mid M_1' \mid \vec{\alpha}_1 \qquad & P \vdash C_2 \rightsquigarrow D_2 \mid M_2' \mid \vec{\alpha}_2 \\ \sigma \cup \sigma_1' \Vdash D_1 \qquad & \sigma \cup \sigma_2' \Vdash D_2 \\ M \leq M_1'(\sigma \cup \sigma_1') \qquad & M \leq M_2'(\sigma \cup \sigma_2') \\ \mathsf{dom}(\sigma_1') \subseteq \vec{\alpha}_1 \subseteq \mathfrak{U}_1 \qquad & \mathsf{dom}(\sigma_2') \subseteq \vec{\alpha}_2 \subseteq \mathfrak{U}_2 \end{aligned}$$

By Lemma 4.27 we obtain $\vec{\alpha}_1 \sharp D_2, M_2'$ and $\vec{\alpha}_2 \sharp D_1, M_1'$.
Therefore, we have:

$$\begin{aligned} P \vdash C \rightsquigarrow D_1 \cup D_2 \mid M_1' \wedge M_2' \mid \vec{\alpha}_1 \cup \vec{\alpha}_2 \qquad & \sigma \cup \sigma_1' \cup \sigma_2' \Vdash D_1 \cup D_2 \\ M \leq (M_1' \wedge M_2')(\sigma \cup \sigma_1' \cup \sigma_2') \qquad & \mathsf{dom}(\sigma_1' \cup \sigma_2') \subseteq \vec{\alpha}_1 \cup \vec{\alpha}_2 \subseteq \mathfrak{U} \end{aligned}$$

Case: $C = (C_1 \vee C_2)$
By hypothesis, there exists an i such that $P; M; \sigma \Vdash C_i$.
By IH, we obtain

$$\begin{aligned} P \vdash C_i \rightsquigarrow D \mid M' \mid \vec{\alpha} \qquad & \sigma \cup \sigma' \Vdash D \\ M \leq M'(\sigma \cup \sigma') \qquad & \mathsf{dom}(\sigma') \subseteq \vec{\alpha} \subseteq \mathfrak{U} \end{aligned}$$

We can conclude directly by applying $[\mathrm{C}_\vee^{\mathrm{sim}}]$.

Case: $C = (\exists \vec{\alpha}.\, C')$
Assume by α-renaming that $\vec{\alpha}$ is in $\mathfrak{U}$ and take $\mathfrak{U}' = \mathfrak{U} \setminus \vec{\alpha}$.
By hypothesis, for some $\vec{t}$ we have $P; M; \sigma \cup [\vec{t}/\vec{\alpha}] \Vdash C'$.
By IH, we obtain

$$\begin{aligned} P \vdash C' \rightsquigarrow D \mid M' \mid \vec{\alpha}' \qquad & \sigma \cup [\vec{t}/\vec{\alpha}] \cup \sigma_1' \Vdash D \\ M \leq M'(\sigma \cup [\vec{t}/\vec{\alpha}] \cup \sigma_1') \qquad & \mathsf{dom}(\sigma_1') \subseteq \vec{\alpha}' \subseteq \mathfrak{U}' \end{aligned}$$

We conclude by $[\mathrm{C}_\exists^{\mathrm{sim}}]$ and by taking $\sigma' = [\vec{t}/\vec{\alpha}] \cup \sigma_1'$.

Case: $C = (\mathsf{def}\ x \colon t\ \mathsf{in}\ C')$
By hypothesis, we have $P; (M, x \colon t\sigma); \sigma \Vdash C'$.
By IH, we obtain:

$$\begin{aligned} P \vdash C' \rightsquigarrow D \mid M' \mid \vec{\alpha} \qquad & \sigma \cup \sigma' \Vdash D \\ (M, x \colon t\sigma) \leq M'(\sigma \cup \sigma') \qquad & \mathsf{dom}(\sigma') \subseteq \vec{\alpha} \subseteq \mathfrak{U} \end{aligned}$$

Note that $\vec{\alpha} \sharp t$ because $\mathfrak{U} \sharp C$. Therefore, $t(\sigma \cup \sigma') = t\sigma$.

By $[\mathrm{C}^{\mathrm{sim}}_{\mathrm{def}}]$ we have:

$$P \vdash C \rightsquigarrow D \cup \{\, t \mathrel{\dot{\leq}} M'(x) \mid x \in \mathsf{dom}(M') \,\} \mid M' \setminus x \mid \vec{\alpha}\ .$$

If $x \in \mathsf{dom}(M')$, we have $t(\sigma \cup \sigma') \leq M'(x)(\sigma \cup \sigma')$.
Since $(M, x\colon t\sigma) \leq M'(\sigma \cup \sigma')$, we have $M \leq (M' \setminus x)(\sigma \cup \sigma')$.

Case: $C = (\mathsf{let}\ \hat{x}\colon \forall\alpha[C_1].\ \alpha\ \mathsf{in}\ C_2)$
By hypothesis:

Ⓐ $P; M_1; \sigma_1 \Vdash C_1$ Ⓑ $(P, \hat{x}\colon \langle M_1\rangle\alpha\sigma_1); M; \sigma \Vdash C_2$ Ⓒ $M \leq M_1\tilde{\sigma}_1$.

By α-renaming, we assume $\alpha \in \mathfrak{U}$.
We partition $\mathfrak{U}$ into $\{\alpha\}$, $\mathfrak{U}_1$, $\mathfrak{U}_2$, and $\mathfrak{U}_3$.
By IH from Ⓐ (using $\mathfrak{U}_1$) we have:

$$\text{Ⓓ}\ \ P \vdash C_1 \rightsquigarrow D_1 \mid M_1' \mid \vec{\alpha}_1$$
$$\text{Ⓔ}\ \ \sigma_1 \cup \sigma_1' \Vdash D_1 \qquad \text{Ⓕ}\ \ M_1 \leq M_1'(\sigma_1 \cup \sigma_1') \qquad \mathsf{dom}(\sigma_1') \subseteq \vec{\alpha}_1 \subseteq \mathfrak{U}_1$$

By Property 4.25, from Ⓔ we find $\hat{\sigma}$ and $\check{\sigma}$ such that

$$\hat{\sigma} \in \mathsf{tally}(D_1) \qquad \sigma_1 \cup \sigma_1' \simeq \check{\sigma} \circ \hat{\sigma} \qquad \mathsf{dom}(\hat{\sigma}) \subseteq \mathsf{var}(D_1)\ .$$

We show $(P, \hat{x}\colon \langle M_1'\hat{\sigma}\rangle\alpha\hat{\sigma}) \leq^{\forall} (P, \hat{x}\colon \langle M_1\rangle\alpha\sigma_1)$.
To instantiate the type scheme on the left, we use $\check{\sigma}$. We have $\alpha\hat{\sigma}\check{\sigma} \simeq \alpha(\sigma_1 \cup \sigma_1') = \alpha\sigma_1$ and $M_1 \leq M_1'(\sigma_1 \cup \sigma_1') \simeq M_1'\hat{\sigma}\check{\sigma}$.
Then, by Lemma 4.24, from Ⓑ we have Ⓖ $(P, \hat{x}\colon \langle M_1'\hat{\sigma}\rangle\alpha\hat{\sigma}); M; \sigma \Vdash C_2$.
By IH from Ⓖ (using $\mathfrak{U}_2$) we have:

$$\text{Ⓗ}\ \ (P, \hat{x}\colon \langle M_1'\hat{\sigma}\rangle\alpha\hat{\sigma}) \vdash C_2 \rightsquigarrow D_2 \mid M_2' \mid \vec{\alpha}_2$$
$$\sigma \cup \sigma_2' \Vdash D_2 \qquad M \leq M_2'(\sigma \cup \sigma_2') \qquad \mathsf{dom}(\sigma_2') \subseteq \vec{\alpha}_2 \subseteq \mathfrak{U}_2$$

Let $\vec{\beta} = \mathsf{var}(M_1'\hat{\sigma})$ and take $\vec{\gamma}$ from $\mathfrak{U}_3$.
Then from Ⓓ and Ⓗ we derive

$$P \vdash C \rightsquigarrow D_2 \mid M_1'\hat{\sigma}[\vec{\gamma}/\vec{\beta}] \wedge M_2' \mid \vec{\alpha}_2 \cup \vec{\gamma}\ .$$

We take $\sigma' = \sigma_2' \cup [\vec{\beta}\check{\sigma}\tilde{\sigma}/\vec{\gamma}]$.
By Lemma 4.27, $\vec{\gamma} \sharp D_2, M_2'$. Therefore, we have $\sigma \cup \sigma' \Vdash D_2$.
We show $M \leq (M_1'\hat{\sigma}[\vec{\gamma}/\vec{\beta}] \wedge M_2')(\sigma \cup \sigma')$.
We have $M \leq M_2'(\sigma \cup \sigma')$ because $M \leq M_2'(\sigma \cup \sigma_2')$ and $\vec{\gamma} \sharp M_2'$.
Moreover,

$$M \leq M_1\tilde{\sigma} \leq M_1'(\sigma_1 \cup \sigma_1')\tilde{\sigma} \simeq M_1'\hat{\sigma}\check{\sigma}\tilde{\sigma} = M_1'\hat{\sigma}[\vec{\gamma}/\vec{\beta}](\sigma \cup \sigma')\ . \qquad \square$$

4.4 Results and discussion

We have built an inference algorithm for the type system $\mathcal{T}^{\mathrm{i}}$ of Figure 4.1 in three steps: we have defined the reformulated type system $\mathcal{T}^{\mathrm{r}}$, defined constraints and given a declarative notion of constraint satisfaction, and finally shown how to solve constraints algorithmically. Now, we put the three steps together and state soundness and completeness for type inference for programs

(that is, closed expressions).

4.30 THEOREM (Soundness of type inference): Let e be a program and α a type variable. If $\varnothing \vdash \langle\!\langle e\colon \alpha \rangle\!\rangle \rightsquigarrow D \mid \varnothing \mid \vec{\alpha}$ and $\sigma \in \mathsf{tally}(D)$, then $\varnothing \vdash e\colon \alpha\sigma$. □

Proof: Consequence of Property 4.25, Lemma 4.28, Lemma 4.21, and Theorem 4.14. □

The λ-environment obtained by simplification is $\varnothing$ because the constraint $\langle\!\langle e\colon \alpha \rangle\!\rangle$ will not have any free x variable, since e is closed.

4.31 THEOREM (Completeness of type inference): Let e be a program and t a type such that $\varnothing \vdash e\colon t$ can be derived in $\mathcal{T}^{\mathrm{i}\setminus\wedge}$. Let α be a type variable. Then, there exist D, $\vec{\alpha}$, and σ such that $\varnothing \vdash \langle\!\langle e\colon \alpha \rangle\!\rangle \rightsquigarrow D \mid \varnothing \mid \vec{\alpha}$, that $\sigma \in \mathsf{tally}(D)$, and that, for some σ', $\alpha\sigma\sigma' \simeq t$. □

Proof: Since we can derive $\varnothing \vdash e\colon t$ in $\mathcal{T}^{\mathrm{i}\setminus\wedge}$, by Theorem 4.14 we can derive $\varnothing;\varnothing \Vdash e\colon t$ in $\mathcal{T}^{\mathrm{r}\setminus\wedge}$.

Since $t = \alpha[t/\alpha]$, by Lemma 4.22, we have $\varnothing;\varnothing;[t/\alpha] \Vdash \langle\!\langle e\colon \alpha \rangle\!\rangle$.

Then, by Lemma 4.29, we find D, M, σ, and $\vec{\alpha}$ such that

$$\varnothing \vdash \langle\!\langle e\colon \alpha \rangle\!\rangle \rightsquigarrow D \mid M \mid \vec{\alpha} \qquad [t/\alpha] \cup \sigma'' \Vdash D$$
$$\varnothing \leq M([t/\alpha] \cup \sigma'') \qquad \mathsf{dom}(\sigma'') \subseteq \vec{\alpha}\ .$$

Let $\sigma' = [t/\alpha] \cup \sigma''$.

Since $\varnothing \leq M\sigma'$, we have $M = \varnothing$.

By Property 4.25, we find $\sigma \in \mathsf{tally}(D)$ and σ''' such that $\sigma' \simeq \sigma''' \circ \sigma$. Therefore, since $\alpha\sigma' = t$, we have $\alpha\sigma\sigma''' \simeq t$. □

These two results state that type inference is sound with respect to the type system $\mathcal{T}^{\mathrm{i}}$ of Figure 4.1 and complete with respect to its restriction $\mathcal{T}^{\mathrm{i}\setminus\wedge}$ without intersection introduction.

We conjecture that type inference is also sound with respect to $\mathcal{T}^{\mathrm{i}\setminus\wedge}$ because it cannot infer intersection types for functions (since we use a single arrow type in the constraint) nor for let-bound variables (since we allow a single instantiation). To attempt to prove this, we should use a different proof technique to relate the standard and the reformulated type systems.

4.4.1 Non-determinism and lack of principal solutions

Type inference can infer more than one type for a program. Indeed, the type system $\mathcal{T}^{\mathrm{i}\setminus\wedge}$ does not have principal types.[4] As an example, assume that $\bar{e}$ is some ill-typed expression and that b_1 and b_2 are two disjoint base types. Then, the function

$$\lambda x.\, \big(\lambda y.\, y \in (b_1 \times b_1) \vee (b_2 \times b_2)\ ?\ 3 : \bar{e}\big)\ (\pi_1\, x, \pi_2\, x)$$

4 We do not know whether the system $\mathcal{T}^{\mathrm{i}}$ including $[\mathrm{T}_\wedge]$ has principal types.

can be given type $b_1 \times b_1 \to \mathsf{Int}$ or $b_2 \times b_2 \to \mathsf{Int}$, but it cannot be given any type that is more general than both. Using $[\mathrm{T}_\wedge]$, it could be given the type $(b_1 \times b_1 \to \mathsf{Int}) \wedge (b_2 \times b_2 \to \mathsf{Int})$, which is equivalent to $(b_1 \times b_1) \vee (b_2 \times b_2) \to \mathsf{Int}$. In contrast, to derive the type $(b_1 \times b_1) \vee (b_2 \times b_2) \to \mathsf{Int}$ without using $[\mathrm{T}_\wedge]$, we would need to type the body of the function as Int assuming that x has type $(b_1 \times b_1) \vee (b_2 \times b_2)$. But, under that assumption, $(\pi_1\, x, \pi_2\, x)$ has type $(b_1 \vee b_2) \times (b_1 \vee b_2)$. Therefore, we need to type $\lambda y.\, y \in (b_1 \times b_1) \vee (b_2 \times b_2)\ ?\ 3 : \bar{e}$ as $(b_1 \vee b_2) \times (b_1 \vee b_2) \to \mathsf{Int}$. We cannot do so because, since $(b_1 \vee b_2) \times (b_1 \vee b_2)$ is not a subtype of $(b_1 \times b_1) \vee (b_2 \times b_2)$, to do so we would need $\bar{e}$ to be well typed. The absence of a principal type in this case means that the algorithm must return two distinct solutions and proceed to check the rest of the program once for each of them by backtracking.

In a practical implementation, we might want to reduce non-determinism as far as possible. We outline next two modifications of the system to do so.

CONSTRAINTS FOR TYPECASES: To generate constraints for typecases, we have used disjunctive constraints to match the "either … or …" conditions in the typing rule $[\mathrm{T}_{\mathrm{case}}]$. This means that the constraints for a typecase can be solved in four possible ways, and algorithmic constraint simplification should check all of them. While this is not the only source of non-determinism (since tally can compute more than one type substitution), we could still want to use a more restrictive constraint which is simpler to solve.

We can replace the definition in Figure 4.5 with

$$\langle\!\langle (e_0 \in \mathbf{t}\ ?\ e_1 : e_2) \colon t \rangle\!\rangle = \exists \alpha.\, \langle\!\langle e_0 \colon \alpha \rangle\!\rangle \wedge \langle\!\langle e_1 \colon t \rangle\!\rangle \wedge \langle\!\langle e_2 \colon t \rangle\!\rangle\ .$$

This constraint demands that both branches be well typed. Using it, type inference accepts fewer programs: soundness (Lemma 4.21) remains valid, but completeness (Lemma 4.22) does not. To recover the same statement as Lemma 4.22, we should modify the typing rule for typecases so that it also forces the typing of every branch. We use the rule

$$\frac{P; M \Vdash e_0 \colon t_0 \qquad P; M \Vdash e_1 \colon t \qquad P; M \Vdash e_2 \colon t}{P; M \Vdash (e_0 \in \mathbf{t}\ ?\ e_1 : e_2) \colon t}$$

instead of $[\mathrm{T}^{\mathrm{r}}_{\mathrm{case}}]$ in $\mathcal{T}^{\mathrm{r}}$ and, correspondingly,

$$\frac{\Gamma \vdash e_0 \colon t_0 \qquad \Gamma \vdash e_1 \colon t \qquad \Gamma \vdash e_2 \colon t}{\Gamma \vdash (e_0 \in \mathbf{t}\ ?\ e_1 : e_2) \colon t}$$

instead of $[\mathrm{T}_{\mathrm{case}}]$ in $\mathcal{T}^{\mathrm{i}}$.

This restriction would cripple the effectiveness of intersection types to type overloaded functions, as we have remarked in Section 3.2. However, inference does not infer intersection types for functions anyway. Hence, the restriction would not be too limiting: in the type systems without $[\mathrm{T}_\wedge]$, it only affects programs with dead code (a branch of a typecase that we do not need to type in a derivation without $[\mathrm{T}_\wedge]$ can never be reached during evaluation).

INTRODUCING INTERSECTION TYPES: If we adopt the constraints for typecases that we have just described, the only source of non-determinism is the rule $[\mathrm{C}^{\mathrm{sim}}_{\mathsf{let}}]$: tally can compute more than one type substitution, and any of them can be chosen to continue simplification.

To some extent, this is unavoidable: different substitutions can make incompatible assumptions on the types of free λ-variables in the constraints. However, in some cases, the solutions are not incompatible and therefore we can use intersection types to merge them. We consider here the case of let bindings where the bound expression has no free λ-variables.

We can add one more rule for let constraints:

$$\frac{\begin{array}{c} P \vdash C_1 \rightsquigarrow D_1 \mid \varnothing \mid \vec{\alpha}_1 \\ (P, \hat{x} \colon \langle \varnothing \rangle \bigwedge_{i \in I} \alpha\sigma_i) \vdash C_2 \rightsquigarrow D_2 \mid M \mid \vec{\alpha}_2 \end{array}}{P \vdash \mathsf{let}\, \hat{x} \colon \forall \alpha[C_1].\, \alpha\ \mathsf{in}\ C_2 \rightsquigarrow D_2 \mid M \mid \vec{\alpha}_2} \begin{cases} \mathsf{tally}(D_1) = \{\, \sigma_i \mid i \in I \,\} \neq \varnothing \\ \vec{\alpha}_1 \sharp \alpha \end{cases}$$

This rule should be used instead of $[\mathrm{C}^{\mathrm{sim}}_{\mathsf{let}}]$ when the λ-environment obtained by simplifying C_1 is empty. Instead of choosing a single solution, we take the intersection of all of them. Since $\bigwedge_{i \in I} \alpha\sigma_i$ is a subtype of all $\alpha\sigma_i$, using it ensures that we find all solutions that we could find choosing any of the σ_i.

Adding this rule makes Lemma 4.28 fail: constraint simplification is no longer sound with respect to constraint satisfaction. However, we can add a corresponding rule to C^{sat} to recover soundness:

$$\frac{\forall i \in I.\ \ P; \varnothing; \sigma_i \Vdash C_1 \qquad (P, \hat{x} \colon \langle \varnothing \rangle \bigwedge_{i \in I} \alpha\sigma_i); M; \sigma \Vdash C_2}{P; M; \sigma \Vdash \mathsf{let}\, \hat{x} \colon \forall \alpha[C_1].\, \alpha\ \mathsf{in}\ C_2}\ I \neq \varnothing$$

Adding this rule makes Lemma 4.21 (soundness of C^{sat} with respect to $\mathcal{T}^{\mathrm{r}\setminus\wedge}$) fail, but the system is still sound with respect to the type system $\mathcal{T}^{\mathrm{r}}$ including $[\mathrm{T}_\wedge]$, that is, we have:

> If $P; M; \sigma \Vdash \langle\!\langle e \colon t \rangle\!\rangle$, then $P; M \Vdash e \colon t\sigma$.

This means that soundness for type inference (Theorem 4.30) still holds.

A typical program could be of the form $\mathsf{let}\ \hat{x}_1 = e_1\ \mathsf{in}\ \ldots\ \mathsf{let}\ \hat{x}_n = e_n\ \mathsf{in}\ e$: a sequence of definitions of top-level identifiers followed by an expression e to be evaluated. None of the e_i would have free λ-variables. Without this modification, the constraints for e_1 could have multiple incomparable solutions: then, we would need to try to infer types for the rest of the program once for each solution. With the modified rules, instead, backtracking might still be needed during inference for e_1, but then we choose a single typing scheme for $\hat{x}_1$ and use it for the rest of the program, making the analysis modular.

5 Adding type annotations

In this chapter, we describe how to extend type inference so that it can infer more precise types for programs that are partially annotated with type information. We do so essentially by changing constraint generation, so that we generate different constraints for an expression if it is annotated with a type; this also requires some changes to the syntax of constraints and to constraint satisfaction and solving.

In this system, adding type annotations to functions allows type inference to assign intersection types to them. For example, the annotated expression

$$\big((\lambda x.\, x \in b_{\mathsf{true}}\ ?\ \mathsf{false} : \mathsf{true}) :: (b_{\mathsf{true}} \to b_{\mathsf{false}}) \wedge (b_{\mathsf{false}} \to b_{\mathsf{true}})\big)$$

is assigned the intersection type in the annotation, which we cannot infer for the expression without the annotation, but we can derive in the declarative type system using $[\mathrm{T}_\wedge]$.

Likewise, inference can exploit annotations on $\hat{x}$ variables to derive types that are the intersection of multiple instances: for example, we can write $(\hat{x} :: (\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool}))$ when $\hat{x}$ has the typing scheme $\langle\varnothing\rangle\alpha \to \alpha$.

CHAPTER OUTLINE:

Section 5.1 We add type annotations to the syntax of expressions and show how to modify the type system to account for them.

Section 5.2 We show how to extend constraints and the notions of constraint satisfaction, generation, and solving to account for type annotations; we prove soundness and completeness properties.

Section 5.3 We summarize the results: type inference is sound and, on expressions without annotations, it enjoys the same completeness result as in the previous chapter (while being able to type more terms if we add annotations). We also point out directions for future work, in particular towards stronger completeness results.

5.1 Language syntax and type system

5.1.1 Syntax

The *annotated expressions* $\mathbf{e}$ are the terms generated inductively by the following grammar:

$$\mathbf{e} ::= \hat{x} \mid x \mid c \mid \lambda x.\, \mathbf{e} \mid \mathbf{e}\ \mathbf{e} \mid (\mathbf{e}, \mathbf{e}) \mid \pi_i\ \mathbf{e} \mid \mathbf{e} \in \mathbf{t}\ ?\ \mathbf{e} : \mathbf{e} \mid \mathsf{let}\ \vec{\alpha}\ \hat{x} = \mathbf{e}\ \mathsf{in}\ \mathbf{e} \mid (\mathbf{e} :: t)\ .$$

There are two differences with respect to the syntax of Definition 3.1. Of course, we add type ascription $(\mathbf{e} :: t)$. We also change the syntax of let constructs,

adding a *decoration* $\vec{\alpha}$, which is a vector of type variables. In $\mathsf{let}\ \vec{\alpha}\ \hat{x} = \mathbf{e}_1\ \mathsf{in}\ \mathbf{e}_2$, the $\vec{\alpha}$ variables are bound in $\mathbf{e}_1$. This decoration serves as a binder for the type variables in annotations and marks their scope. This controls whether they are polymorphic or not. For instance, $\mathsf{let}\ \alpha\ \hat{x} = ((\lambda x.\, x) :: \alpha \to \alpha)\ \mathsf{in}\ \hat{x}\ 3$ is well typed, because α is bound in the let and can be instantiated in the body of the let. Instead, $\mathsf{let}\ \epsilon\ \hat{x} = ((\lambda x.\, x) :: \alpha \to \alpha)\ \mathsf{in}\ \hat{x}\ 3$ (where ϵ is the empty vector) is ill-typed, because α is not bound in the let and cannot be instantiated when typing the body $\hat{x}\ 3$ – in practice, this means it is bound in some outer scope and is polymorphic only outside that scope.

We see the expressions of Definition 3.1 as a subset of annotated expressions by identifying $\mathsf{let}\ \epsilon\ \hat{x} = e_1\ \mathsf{in}\ e_2$ with $\mathsf{let}\ \hat{x} = e_1\ \mathsf{in}\ e_2$.

Given an annotated expression $\mathbf{e}$, we denote by $\mathsf{erase}(\mathbf{e})$ the expression in the syntax of Definition 3.1 obtained by erasing the ascriptions and decorations in $\mathbf{e}$. That is, we have

$$\mathsf{erase}(\mathsf{let}\ \vec{\alpha}\ \hat{x} = \mathbf{e}_1\ \mathsf{in}\ \mathbf{e}_2) = \mathsf{let}\ \hat{x} = \mathsf{erase}(\mathbf{e}_1)\ \mathsf{in}\ \mathsf{erase}(\mathbf{e}_2)$$
$$\mathsf{erase}((\mathbf{e} :: t)) = \mathsf{erase}(\mathbf{e})$$

and, for all other cases, $\mathsf{erase}(\cdot)$ propagates the erasure to the subterms.

5.1.2 Reformulated type system

We describe the type system of the annotated language directly following the presentation in Section 4.1. We add one more parameter to the typing relation: a set Δ of type variables. The type variables in Δ cannot be instantiated in the derivation. To type a program, we normally take Δ to be the set of free type variables in the annotations of the expression we type: as anticipated, we do not allow free type variables in annotations to be instantiated.

The typing relation $P; M; \Delta \Vdash e : t$ is defined by the rules in Figure 5.1. We write $\mathcal{T}^{\mathrm{ra}}$ to refer to this system and $\mathcal{T}^{\mathrm{ra}\setminus\wedge}$ to refer to its restriction without the rule $[\mathrm{T}^{\mathrm{ra}}_{\wedge}]$.

The interesting differences compared to $\mathcal{T}^{\mathrm{r}}$ are in the rules $[\mathrm{T}^{\mathrm{ra}}_{x}]$ and $[\mathrm{T}^{\mathrm{ra}}_{\mathsf{let}}]$. In $[\mathrm{T}^{\mathrm{ra}}_{x}]$, as anticipated, the type substitution σ cannot instantiate the variables in Δ, as imposed by the side condition $\mathsf{dom}(\sigma) \sharp \Delta$. In $[\mathrm{T}^{\mathrm{ra}}_{\mathsf{let}}]$, to type $\mathbf{e}_1$ we expand the set Δ adding $\vec{\alpha}$: this is because the $\vec{\alpha}$ variables should be kept monomorphic while typing $\mathbf{e}_1$. We ask that $\vec{\alpha}$ be chosen disjoint from Δ (this can be ensured by α-renaming) but also that it is disjoint from M_1. This is because M_1 holds the assumptions on the types of free x variables in $\mathbf{e}_1$; the type variables in $\vec{\alpha}$ cannot appear there, or they would be escaping their scope.

Two simple results relate typing in the systems $\mathcal{T}^{\mathrm{ra}}$ and $\mathcal{T}^{\mathrm{r}}$.

5.1 LEMMA: If $P; M; \Delta \Vdash \mathbf{e} : t$ can be derived in $\mathcal{T}^{\mathrm{ra}}$, then $P; M \Vdash \mathsf{erase}(\mathbf{e}) : t$ can be derived in $\mathcal{T}^{\mathrm{r}}$.

Moreover, if $P; M; \Delta \Vdash \mathbf{e} : t$ can be derived in $\mathcal{T}^{\mathrm{ra}\setminus\wedge}$, then $P; M \Vdash \mathsf{erase}(\mathbf{e}) : t$ can be derived in $\mathcal{T}^{\mathrm{r}\setminus\wedge}$. □

$$[\mathrm{T}^{\mathrm{ra}}_{\hat{x}}]\ \frac{}{P; M\sigma; \Delta \Vdash \hat{x} : t\sigma}\ \begin{cases} P(\hat{x}) = \langle M \rangle t \\ \mathsf{dom}(\sigma) \sharp \Delta \end{cases} \qquad [\mathrm{T}^{\mathrm{ra}}_{x}]\ \frac{}{P; M; \Delta \Vdash x : t}\ M(x) = t$$

$$[\mathrm{T}^{\mathrm{ra}}_{c}]\ \frac{}{P; M; \Delta \Vdash c : b_c}$$

$$[\mathrm{T}^{\mathrm{ra}}_{\lambda}]\ \frac{P; (M, x : t'); \Delta \Vdash \mathbf{e} : t}{P; M; \Delta \Vdash \lambda x.\, \mathbf{e} : t' \to t} \qquad [\mathrm{T}^{\mathrm{ra}}_{\mathrm{app}}]\ \frac{P; M; \Delta \Vdash \mathbf{e}_1 : t' \to t \qquad P; M; \Delta \Vdash \mathbf{e}_2 : t'}{P; M; \Delta \Vdash \mathbf{e}_1\ \mathbf{e}_2 : t}$$

$$[\mathrm{T}^{\mathrm{ra}}_{\mathrm{pair}}]\ \frac{P; M; \Delta \Vdash \mathbf{e}_1 : t_1 \qquad P; M; \Delta \Vdash \mathbf{e}_2 : t_2}{P; M; \Delta \Vdash (\mathbf{e}_1, \mathbf{e}_2) : t_1 \times t_2} \qquad [\mathrm{T}^{\mathrm{ra}}_{\mathrm{proj}}]\ \frac{P; M; \Delta \Vdash \mathbf{e} : t_1 \times t_2}{P; M; \Delta \Vdash \pi_i\, \mathbf{e} : t_i}$$

$$[\mathrm{T}^{\mathrm{ra}}_{\mathrm{case}}]\ \frac{\begin{array}{c} P; M; \Delta \Vdash \mathbf{e}_0 : t_0 \\ \text{either } t_0 \leq \neg\mathbf{t} \text{ or } P; M; \Delta \Vdash \mathbf{e}_1 : t \\ \text{either } t_0 \leq \mathbf{t} \text{ or } P; M; \Delta \Vdash \mathbf{e}_2 : t \end{array}}{P; M; \Delta \Vdash (\mathbf{e}_0 \in \mathbf{t}\ ?\ \mathbf{e}_1 : \mathbf{e}_2) : t}$$

$$[\mathrm{T}^{\mathrm{ra}}_{\mathrm{let}}]\ \frac{P; M_1; \Delta \cup \vec{\alpha} \Vdash \mathbf{e}_1 : t_1 \qquad (P, \hat{x} : \langle M_1 \rangle t_1); M; \Delta \Vdash \mathbf{e}_2 : t}{P; M; \Delta \Vdash \mathsf{let}\ \vec{\alpha}\ \hat{x} = \mathbf{e}_1\ \mathsf{in}\ \mathbf{e}_2 : t}\ \begin{cases} \exists\sigma.\ M \leq M_1\sigma \\ \vec{\alpha} \sharp \Delta, M_1 \end{cases}$$

$$[\mathrm{T}^{\mathrm{ra}}_{::}]\ \frac{P; M; \Delta \Vdash \mathbf{e} : t}{P; M; \Delta \Vdash (\mathbf{e} :: t) : t}$$

$$[\mathrm{T}^{\mathrm{ra}}_{\leq}]\ \frac{P; M'; \Delta \Vdash \mathbf{e} : t'}{P; M; \Delta \Vdash \mathbf{e} : t}\ \begin{cases} t' \leq t \\ M \leq M' \end{cases} \qquad [\mathrm{T}^{\mathrm{ra}}_{\wedge}]\ \frac{P; M; \Delta \Vdash \mathbf{e} : t_1 \qquad P; M; \Delta \Vdash \mathbf{e} : t_2}{P; M; \Delta \Vdash \mathbf{e} : t_1 \wedge t_2}$$

FIGURE 5.1 $\mathcal{T}^{\mathrm{ra}}$: Reformulated typing rules (with type annotations)

$$[\mathrm{C}^{\mathrm{sata}}_{\dot\leq}]\ \frac{}{P; M; \Delta; \sigma \Vdash (t_1 \mathrel{\dot\leq} t_2)}\ t_1\sigma \leq t_2\sigma$$

$$[\mathrm{C}^{\mathrm{sata}}_{x}]\ \frac{}{P; M; \Delta; \sigma \Vdash (x \mathrel{\dot\leq} t)}\ M(x) \leq t\sigma \qquad [\mathrm{C}^{\mathrm{sata}}_{\hat x}]\ \frac{}{P; M; \Delta; \sigma \Vdash (\hat x \mathrel{\dot\leq} t)} \begin{cases} P(\hat x) = \langle M_1\rangle t_1 \\ \exists \sigma_1. \begin{cases} t_1\sigma_1 \leq t\sigma \\ M \leq M_1\sigma_1 \\ \mathsf{dom}(\sigma_1) \sharp \Delta \end{cases} \end{cases}$$

$$[\mathrm{C}^{\mathrm{sata}}_{\wedge}]\ \frac{P; M; \Delta; \sigma \Vdash C_1 \qquad P; M; \Delta; \sigma \Vdash C_2}{P; M; \Delta; \sigma \Vdash C_1 \wedge C_2} \qquad [\mathrm{C}^{\mathrm{sata}}_{\vee}]\ \frac{P; M; \Delta; \sigma \Vdash C_i}{P; M; \Delta; \sigma \Vdash C_1 \vee C_2}$$

$$[\mathrm{C}^{\mathrm{sata}}_{\exists}]\ \frac{P; M; \Delta; \sigma \cup [\vec t/\vec\alpha] \Vdash C}{P; M; \Delta; \sigma \Vdash \exists \vec\alpha.\, C} \qquad [\mathrm{C}^{\mathrm{sata}}_{\mathsf{def}}]\ \frac{P; (M, x\colon t\sigma); \Delta; \sigma \Vdash C}{P; M; \Delta; \sigma \Vdash \mathsf{def}\ x\colon t\ \mathsf{in}\ C}$$

$$[\mathrm{C}^{\mathrm{sata}}_{\mathsf{let}}]\ \frac{P; M_1; \Delta \cup \vec\alpha; \sigma_1 \Vdash C_1 \qquad (P, \hat x\colon \langle M_1\rangle\alpha\sigma_1); M; \Delta; \sigma \Vdash C_2}{P; M; \Delta; \sigma \Vdash \mathsf{let}\ \hat x\colon \forall \vec\alpha; \alpha[C_1].\, \alpha\ \mathsf{in}\ C_2} \begin{cases} \exists \sigma_1'.\, M \leq M_1\sigma_1' \\ \mathsf{dom}(\sigma_1) \sharp \Delta, \vec\alpha \\ \vec\alpha \sharp \Delta, M_1 \end{cases}$$

FIGURE 5.2 C^{sata}: Constraint satisfaction rules (with type annotations)

Proof: Straightforward proof by induction on the typing derivation. □

5.2 LEMMA: If $P; M \Vdash e\colon t$, then $P; M; \varnothing \Vdash e\colon t$.

Moreover, if $P; M \Vdash e\colon t$ can be derived in $\mathcal{T}^{\mathrm{r}\backslash\wedge}$, then $P; M; \varnothing \Vdash e\colon t$ can be derived in $\mathcal{T}^{\mathrm{ra}\backslash\wedge}$. □

Proof: Straightforward induction proof. □

5.2 Constraints and constraint solving

5.2.1 Constraints and constraint satisfaction

We extend the syntax of structured constraints from the previous chapter by adding binders in let constraints to match the decorations of let expressions. The modified syntax is the following:

$$C ::= (t \mathrel{\dot\leq} t) \mid (x \mathrel{\dot\leq} t) \mid (\hat x \mathrel{\dot\leq} t) \mid C \wedge C \mid C \vee C \mid \exists \vec\alpha.\, C$$
$$\mid \mathsf{def}\ x\colon t\ \mathsf{in}\ C \mid \mathsf{let}\ \hat x\colon \forall \vec\alpha; \alpha[C].\, \alpha\ \mathsf{in}\ C\ .$$

We see the structured constraints of Definition 4.18 as a subset of these by identifying $\mathsf{let}\ \hat x\colon \forall \epsilon; \alpha[C_1].\, \alpha\ \mathsf{in}\ C_2$ with $\mathsf{let}\ \hat x\colon \forall \alpha[C_1].\, \alpha\ \mathsf{in}\ C_2$. We refer to the structured constraints of the previous chapter as structured constraints *without explicit polymorphism*.

We adapt structured-constraint satisfaction by adding the parameter Δ and adapting the rule for let constraints. The relation is defined by the rules C^{sata}

in Figure 5.2. The interesting rules are $[\mathrm{C}^{\mathrm{sata}}_{\hat{x}}]$ and $[\mathrm{C}^{\mathrm{sata}}_{\mathsf{let}}]$, which adapt $[\mathrm{C}^{\mathrm{sat}}_{\hat{x}}]$ and $[\mathrm{C}^{\mathrm{sat}}_{\mathsf{let}}]$ of Figure 4.4 as we did for $[\mathrm{T}^{\mathrm{ra}}_{\hat{x}}]$ and $[\mathrm{T}^{\mathrm{ra}}_{\mathsf{let}}]$. Note that, in $[\mathrm{C}^{\mathrm{sata}}_{\mathsf{let}}]$, we add the $\vec{\alpha}$ variables to Δ in the first sub-derivation and we require that the type substitution σ_1 does not instantiate these type variables, because they cannot be instantiated while they are in scope.

5.2.2 Constraint generation

We modify constraint generation to exploit type annotations. In particular, we want to generate different constraints for an $\hat{x}$ variable or a function when we know the type it should have. For instance, if a function is annotated as $((\lambda x.\, e) :: \bigwedge_{i\in I} t'_i \to t_i)$, then we want to generate separate constraints from e for each arrow: we break up the intersection into the set $\{\, t'_i \to t_i \mid i \in I \,\}$ and generate a constraint for each element in the set. If the type in the annotation is not syntactically an intersection of arrow, we can still try to rewrite it to an equivalent intersection (as a trivial example, we could treat the annotation $(t' \to t) \vee \mathbb{0}$ like $t' \to t$). To model this rewriting, we rely on two functions, one for constraints on variables, the other for constraints on functions, to decompose types to sets of types, breaking up intersections after (possibly) rewriting the type to some equivalent intersection type. We leave these functions unspecified except for the properties we need in the proofs.

5.3 PROPERTY: There exist two functions d and $\mathsf{d}_{\to}$ such that:

1. given a type t, $\mathsf{d}(t)$ is a finite, non-empty set of types;
2. given a type t and a set Δ of type variables, $\mathsf{d}^{\Delta}_{\to}(t)$ is a finite set of arrow types;
3. the functions satisfy the following properties, for every t and Δ:
   - $t \simeq \bigwedge_{t'\in\mathsf{d}(t)} t'$
   - $\mathsf{d}^{\Delta}_{\to}(t) = \{\, t'_i \to t_i \mid i \in I \,\} \neq \varnothing \implies \begin{cases} t \simeq \bigwedge_{i\in I} t'_i \to t_i \\ \mathsf{var}(\bigwedge_{i\in I} t'_i \to t_i) \subseteq \Delta \\ \forall i \in I.\ t'_i \simeq \mathbb{0} \implies t_i \simeq \mathbb{1} \end{cases}$

□

The function d maps a type t to some set of types whose intersection is equivalent to t. The function $\mathsf{d}_{\to}$ does similarly, but it always produces a set of arrow types, and it ensures additional properties which we discuss below; it can yield $\varnothing$ if it fails to decompose t ensuring these properties.

We can give simple syntax-based implementations for d and $\mathsf{d}_{\to}$ as follows.

- If $t = \bigwedge_{i\in I} t_i$, then $\mathsf{d}(t) = \{\, t_i \mid i \in I \,\}$; otherwise, $\mathsf{d}(t) = \{t\}$.
- If $t = \bigwedge_{i\in I} t'_i \to t_i$ and $\mathsf{var}(t) \subseteq \Delta$, then $\mathsf{d}^{\Delta}_{\to}(t) = \{\, t'_i \to t''_i \mid i \in I \,\}$, where, for each $i \in I$, we have $t''_i = \mathbb{1}$ if $t'_i \simeq \mathbb{0}$ and $t''_i = t_i$ otherwise.

$$
\begin{aligned}
\langle\!\langle \hat{x} \colon t \rangle\!\rangle^{\Delta} &= \textstyle\bigwedge_{i \in I} (\hat{x} \mathrel{\dot{\leq}} t_i) \\
&\qquad \text{where } \mathsf{d}(t) = \{\, t_i \mid i \in I \,\} \\
\langle\!\langle x \colon t \rangle\!\rangle^{\Delta} &= (x \mathrel{\dot{\leq}} t) \\
\langle\!\langle c \colon t \rangle\!\rangle^{\Delta} &= (b_c \mathrel{\dot{\leq}} t) \\
\langle\!\langle (\lambda x.\, \mathbf{e}) \colon t \rangle\!\rangle^{\Delta} &= \begin{cases} \textstyle\bigwedge_{i \in I} \left(\mathsf{def}\, x \colon t_i' \;\mathsf{in}\; \langle\!\langle \mathbf{e} \colon t_i \rangle\!\rangle^{\Delta}\right) \\ \qquad \text{if } \mathsf{d}^{\Delta}_{\rightarrow}(t) = \{\, t_i' \to t_i \mid i \in I \,\} \neq \varnothing \\ \exists \alpha_1, \alpha_2.\, (\mathsf{def}\, x \colon \alpha_1 \;\mathsf{in}\; \langle\!\langle \mathbf{e} \colon \alpha_2 \rangle\!\rangle^{\Delta}) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t) \\ \qquad \text{if } \mathsf{d}^{\Delta}_{\rightarrow}(t) = \varnothing, \text{ where } \alpha_1, \alpha_2 \sharp t, \mathbf{e}, \Delta \end{cases} \\
\langle\!\langle \mathbf{e}_1\, \mathbf{e}_2 \colon t \rangle\!\rangle^{\Delta} &= \exists \alpha.\, \langle\!\langle \mathbf{e}_1 \colon \alpha \to t \rangle\!\rangle^{\Delta} \wedge \langle\!\langle \mathbf{e}_2 \colon \alpha \rangle\!\rangle^{\Delta} \\
&\qquad \text{where } \alpha \sharp t, \mathbf{e}_1, \mathbf{e}_2, \Delta \\
\langle\!\langle (\mathbf{e}_1, \mathbf{e}_2) \colon t \rangle\!\rangle^{\Delta} &= \exists \alpha_1, \alpha_2.\, \langle\!\langle \mathbf{e}_1 \colon \alpha_1 \rangle\!\rangle^{\Delta} \wedge \langle\!\langle \mathbf{e}_2 \colon \alpha_2 \rangle\!\rangle^{\Delta} \wedge (\alpha_1 \times \alpha_2 \mathrel{\dot{\leq}} t) \\
&\qquad \text{where } \alpha_1, \alpha_2 \sharp t, \mathbf{e}_1, \mathbf{e}_2, \Delta \\
\langle\!\langle \pi_1\, \mathbf{e} \colon t \rangle\!\rangle^{\Delta} &= \langle\!\langle \mathbf{e} \colon t \times \mathbb{1} \rangle\!\rangle^{\Delta} \\
\langle\!\langle \pi_2\, \mathbf{e} \colon t \rangle\!\rangle^{\Delta} &= \langle\!\langle \mathbf{e} \colon \mathbb{1} \times t \rangle\!\rangle^{\Delta} \\
\langle\!\langle (\mathbf{e}_0 \in \mathbf{t}\ ?\ \mathbf{e}_1 : \mathbf{e}_2) \colon t \rangle\!\rangle^{\Delta} &= \exists \alpha.\, \langle\!\langle \mathbf{e}_0 \colon \alpha \rangle\!\rangle^{\Delta} \wedge \left((\alpha \mathrel{\dot{\leq}} \neg \mathbf{t}) \vee \langle\!\langle \mathbf{e}_1 \colon t \rangle\!\rangle^{\Delta}\right) \wedge \left((\alpha \mathrel{\dot{\leq}} \mathbf{t}) \vee \langle\!\langle \mathbf{e}_2 \colon t \rangle\!\rangle^{\Delta}\right) \\
&\qquad \text{where } \alpha \sharp t, \mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2, \Delta \\
\langle\!\langle (\mathsf{let}\, \vec{\alpha}\, \hat{x} = \mathbf{e}_1 \;\mathsf{in}\; \mathbf{e}_2) \colon t \rangle\!\rangle^{\Delta} &= \mathsf{let}\, \hat{x} \colon \forall \vec{\alpha}; \alpha[\langle\!\langle \mathbf{e}_1 \colon \alpha \rangle\!\rangle^{\Delta \cup \vec{\alpha}}].\, \alpha \;\mathsf{in}\; \langle\!\langle \mathbf{e}_2 \colon t \rangle\!\rangle^{\Delta} \\
&\qquad \text{where } \alpha, \vec{\alpha} \sharp \mathbf{e}_1, \Delta \\
\langle\!\langle (\mathbf{e} :: t') \colon t \rangle\!\rangle^{\Delta} &= \langle\!\langle \mathbf{e} \colon t' \rangle\!\rangle^{\Delta} \wedge (t' \mathrel{\dot{\leq}} t)
\end{aligned}
$$

FIGURE 5.3 Constraint generation (with type annotations)

These implementations are unsatisfying, since they give different results for equivalent types, but they suffice for our purpose here.

Using these functions, we define constraint generation in Figure 5.3. The set Δ is an additional parameter. It is passed through because it is used by $\mathsf{d}_{\to}$, where it is important to know which type variables are fixed and which will be instantiated by the solution of the constraints (see the remark below).

Comparing to Figure 4.5, the important differences are in the cases for $\hat{x}$ variables and functions, where we use d and $\mathsf{d}_{\to}$ and generate an intersection with one constraint for each type in $\mathsf{d}(t)$ or $\mathsf{d}^{\Delta}_{\to}(t)$.

REMARK (Decomposition of function types): The properties of $\mathsf{d}_{\to}$ include two requirements that have no analogue for d. They are needed because of the behaviour of semantic subtyping; if we did not impose them, a constraint $\langle\!\langle e \colon t \rangle\!\rangle^{\Delta}$ for an expression e without annotations could be unsatisfiable even when $\langle\!\langle e \colon t \rangle\!\rangle$ is satisfiable. Hence, completeness with respect to the algorithm of the previous chapter (Theorem 5.12) would not hold. Let us see why.

In semantic subtyping, types of the form $t \to t'$ with $t \simeq \mathbb{0}$ are supertypes of all arrow types (whatever t is). Therefore, for example, $\mathbb{0} \to \mathsf{Int} \simeq \mathbb{0} \to \mathsf{Bool}$.

If we removed the condition $\forall i \in I.\ t'_i \simeq \mathbb{0} \implies t_i \simeq \mathbb{1}$, we could have $\mathsf{d}^{\Delta}_{\to}(\mathbb{0} \to \mathsf{Bool}) = \{\mathbb{0} \to \mathsf{Bool}\}$ and

$$\langle\!\langle (\lambda x.\, 3) \colon \mathbb{0} \to \mathsf{Bool} \rangle\!\rangle^{\Delta} = \mathsf{def}\, x \colon \mathbb{0}\ \mathsf{in}\ \langle\!\langle 3 \colon \mathsf{Bool} \rangle\!\rangle^{\Delta}\ .$$

This constraint is unsatisfiable. In contrast, $\langle\!\langle (\lambda x.\, 3) \colon \mathbb{0} \to \mathsf{Bool} \rangle\!\rangle$ is

$$\exists \alpha_1, \alpha_2.\ (\mathsf{def}\, x \colon \alpha_1\ \mathsf{in}\ \langle\!\langle 3 \colon \alpha_2 \rangle\!\rangle) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} \mathbb{0} \to \mathsf{Bool})$$

and is satisfiable (mapping α_2 to Int).

The condition on type variables has the same purpose. Without it, we could have $\langle\!\langle (\lambda x.\, 3) \colon \alpha \to \mathsf{Bool} \rangle\!\rangle^{\Delta} = \mathsf{def}\, x \colon \alpha\ \mathsf{in}\ \langle\!\langle 3 \colon \mathsf{Bool} \rangle\!\rangle^{\Delta}$, which is unsatisfiable, while $\langle\!\langle (\lambda x.\, 3) \colon \alpha \to \mathsf{Bool} \rangle\!\rangle$ is satisfied by $[\mathbb{0}/\alpha]$. To avoid this, we only allow $\mathsf{d}_{\to}$ to decompose a type when the decomposition only contains variables that cannot be instantiated (in practice, variables that come from annotations). □

REMARK (Constraint generation for pairs): Here we define constraint generation for pairs as in the previous chapter. It could be interesting to allow propagation of type information for pairs by defining instead

$$\langle\!\langle (\mathbf{e}_1, \mathbf{e}_2) \colon t \rangle\!\rangle^{\Delta} = \langle\!\langle \mathbf{e}_1 \colon t_1 \rangle\!\rangle^{\Delta} \wedge \langle\!\langle \mathbf{e}_2 \colon t_2 \rangle\!\rangle^{\Delta}$$

when $t \simeq t_1 \times t_2$. However, we run in similar problems as with functions. Indeed, for this constraint to be complete, we would need to know that, if $(\mathbf{e}_1, \mathbf{e}_2)$ has type $t_1 \times t_2$, then $\mathbf{e}_1$ has type t_1 and $\mathbf{e}_2$ has type t_2. This is not true with semantic subtyping. Product types are interpreted as Cartesian products and therefore all products with an empty component are identified: for instance, we have $\mathbb{0} \times \mathsf{Int} \simeq \mathbb{0} \times \mathsf{Bool} \leq \mathsf{Int} \times \mathsf{Bool}$. As a result, if $\bar{\mathbf{e}}$ has type $\mathbb{0}$, then $(\bar{\mathbf{e}}, 3)$ can be typed as $\mathsf{Int} \times \mathsf{Bool}$

To achieve completeness, we would have to duplicate the constraints for the components, for example by defining $\langle\!\langle (\mathbf{e}_1, \mathbf{e}_2) \colon t \rangle\!\rangle^{\Delta}$ as

$$\left(\langle\!\langle \mathbf{e}_1 \colon t_1 \rangle\!\rangle^{\Delta} \wedge \langle\!\langle \mathbf{e}_2 \colon t_2 \rangle\!\rangle^{\Delta}\right) \vee \left(\exists \alpha_1, \alpha_2.\ \langle\!\langle \mathbf{e}_1 \colon \alpha_1 \rangle\!\rangle^{\Delta} \wedge \langle\!\langle \mathbf{e}_2 \colon \alpha_2 \rangle\!\rangle^{\Delta} \wedge (\alpha_1 \times \alpha_2 \mathrel{\dot{\leq}} t)\right)$$

when $t \simeq t_1 \times t_2$. □

A simple observation is that constraints generated from expressions without annotations have no explicit polymorphism (that is, they are in the syntax of Definition 4.18).

5.4 LEMMA: For every e, t, and Δ, the structured constraint $\langle\!\langle e\colon t \rangle\!\rangle^{\Delta}$ has no explicit polymorphism. □

Proof: By induction on e. If e is a let expression, its decoration is empty and therefore the let constraint we generate has no explicit polymorphism. In all other cases, we just apply the IH. □

To relate typing and constraint satisfaction, we prove two results of soundness and completeness (analogous to Lemma 4.21 and Lemma 4.22). For the latter, we consider only expressions without annotations and typing derivations that do not use intersection introduction.

5.5 LEMMA: If $P; M; \Delta; \sigma \Vdash \langle\!\langle \mathbf{e}\colon t \rangle\!\rangle^{\Delta}$ and if $\mathsf{dom}(\sigma) \sharp \Delta$ and $\mathsf{var}(\mathbf{e}) \subseteq \Delta$, then $P; M; \Delta \Vdash \mathbf{e}\colon t\sigma$. □

Proof in appendix (p. 247). Analogous to the proof of Lemma 4.21. The differences are in the cases of $\hat{x}$ variables and functions, which are simple to prove using Property 5.3.

5.6 LEMMA: If $P; M; \varnothing \Vdash e\colon t\sigma$ can be derived in $\mathcal{T}^{\mathrm{ra}\setminus\wedge}$, then $P; M; \varnothing; \sigma \Vdash \langle\!\langle e\colon t \rangle\!\rangle^{\varnothing}$. □

Proof in appendix (p. 249). Similar to the proof of Lemma 4.22.

5.2.3 Constraint solving

To solve constraints, we use tallying as in the previous chapter. In Section 4.3.1, it already allowed for a set Δ of type variables that cannot be instantiated: we always used $\varnothing$ in the previous chapter, but we will need it here.

We need to update structured-constraint simplification: we do so in Figure 5.4 (C^{sima}). We add Δ as an additional parameter and modify $[\mathrm{C}^{\mathrm{sima}}_{\hat{x}}]$ and $[\mathrm{C}^{\mathrm{sima}}_{\mathsf{let}}]$ to match the changes in structured-constraint satisfaction.

We first prove a result of soundness analogous to Lemma 4.28.

5.7 LEMMA: If $P; \Delta \vdash C \rightsquigarrow D \mid M \mid \vec{\alpha}$ and $\sigma \Vdash_{\Delta} D$, then $P; M\sigma; \Delta; \sigma|_{\setminus \vec{\alpha}} \Vdash C$. □

Proof in appendix (p. 250). Similar to the proof of Lemma 4.28.

We now prove completeness for constraints without explicit polymorphism, analogously to Lemma 4.29. To do so, we reuse Lemma 4.29 directly by proving that the new definitions of structured-constraint satisfaction and simplification

$$[\mathrm{C}^{\mathrm{sima}}_{\dot{\leq}}]\ \frac{}{P; \Delta \vdash (t_1 \mathrel{\dot{\leq}} t_2) \rightsquigarrow \{t_1 \mathrel{\dot{\leq}} t_2\} \mid \varnothing \mid \varnothing}$$

$$[\mathrm{C}^{\mathrm{sima}}_{x}]\ \frac{}{P; \Delta \vdash (x \mathrel{\dot{\leq}} t) \rightsquigarrow \varnothing \mid (x\colon t) \mid \varnothing}$$

$$[\mathrm{C}^{\mathrm{sima}}_{\hat{x}}]\ \frac{}{P; \Delta \vdash (\hat{x} \mathrel{\dot{\leq}} t) \rightsquigarrow \{t_1[\vec{\beta}/\vec{\alpha}] \mathrel{\dot{\leq}} t\} \mid M_1[\vec{\beta}/\vec{\alpha}] \mid \vec{\beta}} \begin{cases} P(\hat{x}) = \langle M_1 \rangle t_1 \\ \vec{\alpha} = \mathsf{var}(\langle M_1 \rangle t_1) \setminus \Delta \\ \vec{\beta} \sharp t, \Delta \end{cases}$$

$$[\mathrm{C}^{\mathrm{sima}}_{\wedge}]\ \frac{P; \Delta \vdash C_1 \rightsquigarrow D_1 \mid M_1 \mid \vec{\alpha}_1 \qquad P; \Delta \vdash C_2 \rightsquigarrow D_2 \mid M_2 \mid \vec{\alpha}_2}{P; \Delta \vdash C_1 \wedge C_2 \rightsquigarrow D_1 \cup D_2 \mid M_1 \wedge M_2 \mid \vec{\alpha}_1 \cup \vec{\alpha}_2} \begin{cases} \vec{\alpha}_1 \sharp \vec{\alpha}_2, C_2 \\ \vec{\alpha}_2 \sharp C_1 \end{cases}$$

$$[\mathrm{C}^{\mathrm{sima}}_{\vee}]\ \frac{P; \Delta \vdash C_i \rightsquigarrow D \mid M \mid \vec{\alpha}}{P; \Delta \vdash C_1 \vee C_2 \rightsquigarrow D \mid M \mid \vec{\alpha}} \qquad [\mathrm{C}^{\mathrm{sima}}_{\exists}]\ \frac{P; \Delta \vdash C \rightsquigarrow D \mid M \mid \vec{\alpha}'}{P; \Delta \vdash \exists \vec{\alpha}.\, C \rightsquigarrow D \mid M \mid \vec{\alpha}' \cup \vec{\alpha}}\ \vec{\alpha} \sharp \Delta, \vec{\alpha}'$$

$$[\mathrm{C}^{\mathrm{sima}}_{\mathsf{def}}]\ \frac{P; \Delta \vdash C \rightsquigarrow D \mid M \mid \vec{\alpha}}{P; \Delta \vdash \mathsf{def}\ x\colon t\ \mathsf{in}\ C \rightsquigarrow D \cup D' \mid M \setminus x \mid \vec{\alpha}} \begin{cases} D' = \begin{cases} \{t \mathrel{\dot{\leq}} M(x)\} & \text{if } x \in \mathsf{dom}(M) \\ \varnothing & \text{otherwise} \end{cases} \\ \vec{\alpha} \sharp t \end{cases}$$

$$[\mathrm{C}^{\mathrm{sima}}_{\mathsf{let}}]\ \frac{\begin{array}{c} P; \Delta \cup \vec{\alpha} \vdash C_1 \rightsquigarrow D_1 \mid M_1 \mid \vec{\alpha}_1 \\ (P, \hat{x}\colon \langle M_1\sigma_1 \rangle \alpha\sigma_1); \Delta \vdash C_2 \rightsquigarrow D_2 \mid M_2 \mid \vec{\alpha}_2 \end{array}}{P; \Delta \vdash \mathsf{let}\ \hat{x}\colon \forall \vec{\alpha}; \alpha[C_1].\, \alpha\ \mathsf{in}\ C_2 \rightsquigarrow D_2 \mid M \mid \vec{\alpha}_2 \cup \vec{\gamma}} \begin{cases} \sigma_1 \in \mathsf{tally}_{\Delta \cup \vec{\alpha}}(D_1) \\ \vec{\alpha} \sharp \Delta, M_1\sigma_1 \\ \vec{\beta} = \mathsf{var}(M_1\sigma_1) \setminus \Delta \\ M = M_1\sigma_1[\vec{\gamma}/\vec{\beta}] \wedge M_2 \\ \vec{\alpha}_1 \sharp \alpha \\ \vec{\gamma} \sharp C_1, \vec{\alpha}_2, \Delta \end{cases}$$

FIGURE 5.4 C^{sima}: Constraint simplification rules (with type annotations)

coincide with those of the previous chapter for constraints without explicit polymorphism.

5.8 LEMMA: Let C be a constraint without explicit polymorphism. Then, we have $P; M; \sigma \Vdash C$ if and only if $P; M; \varnothing; \sigma \Vdash C$. □

Proof: Straightforward proof by induction on C.
Note that, if C is a let constraint, its vector $\vec{\alpha}$ must be empty because C has no explicit polymorphism. □

5.9 LEMMA: Let C be a constraint without explicit polymorphism. Then, we have $P \vdash C \rightsquigarrow D \mid M \mid \vec{\alpha}$ if and only if $P; \varnothing \vdash C \rightsquigarrow D \mid M \mid \vec{\alpha}$. □

Proof: Straightforward proof by induction on C. □

These two lemmas, together with the result on completeness of constraint solving from the previous chapter, give us the following corollary.

5.10 COROLLARY:

$$\left.\begin{array}{l} P; M; \varnothing; \sigma \Vdash C \\ C \text{ has no explicit polymorphism} \end{array}\right\} \Longrightarrow \exists D, M', \vec{\alpha}, \sigma'. \begin{cases} P; \varnothing \vdash C \rightsquigarrow D \mid M' \mid \vec{\alpha} \\ \sigma \cup \sigma' \Vdash_{\varnothing} D \\ M \leq M'(\sigma \cup \sigma') \\ \mathsf{dom}(\sigma') \subseteq \vec{\alpha} \end{cases}$$

□

Proof: Corollary of Lemmas 5.8, 5.9 and 4.29. □

5.3 *Results and discussion*

Combining the results of the previous section, we obtain the following statement of soundness. (By "closed", for an annotated expression, we mean that it has no free expression variables and no free type variables.)

5.11 THEOREM (Soundness of type inference): Let $\mathbf{e}$ be a closed annotated expression and α a type variable.
If $\varnothing; \varnothing \vdash \langle\!\langle \mathbf{e} : \alpha \rangle\!\rangle^{\varnothing} \rightsquigarrow D \mid \varnothing \mid \vec{\alpha}$ and $\sigma \in \mathsf{tally}_{\varnothing}(D)$, then $\varnothing; \varnothing; \varnothing \Vdash \mathbf{e} : \alpha\sigma$. □

Proof: Consequence of Property 4.25, Lemma 5.7, and Lemma 5.5. □

The following theorem states that the modified type inference algorithm enjoys the same completeness property as that of the previous chapter.

5.12 THEOREM (Completeness of type inference): Let e be a closed expression and t a type such that $\varnothing \vdash e : t$ in $\mathcal{T}^{\mathrm{i}\setminus\wedge}$. Let α be a type variable.

Then, there exist D, $\vec{\alpha}$, and σ such that $\varnothing;\varnothing \vdash \langle\!\langle e : \alpha \rangle\!\rangle^{\varnothing} \rightsquigarrow D \mid \varnothing \mid \vec{\alpha}$, that $\sigma \in \mathsf{tally}_{\varnothing}(D)$, and that, for some σ', $\alpha\sigma\sigma' \simeq t$. □

Proof: Since we can derive $\varnothing \vdash e : t$ in $\mathcal{T}^{\mathrm{i}\setminus\wedge}$, by Theorem 4.14 and Lemma 5.2 we can derive $\varnothing;\varnothing;\varnothing \Vdash e : t$ in $\mathcal{T}^{\mathrm{ra}\setminus\wedge}$.

Since $t = \alpha[t/\alpha]$, by Lemma 5.6, we have $\varnothing;\varnothing;\varnothing;[t/\alpha] \Vdash \langle\!\langle e : \alpha \rangle\!\rangle^{\varnothing}$.

By Lemma 5.4, $\langle\!\langle e : \alpha \rangle\!\rangle^{\varnothing}$ has no explicit polymorphism.

Then, by Corollary 5.10, we find D, M, σ, and $\vec{\alpha}$ such that

$$\varnothing;\varnothing \vdash \langle\!\langle e : \alpha \rangle\!\rangle^{\varnothing} \rightsquigarrow D \mid M \mid \vec{\alpha} \qquad [t/\alpha] \cup \sigma'' \Vdash_{\varnothing} D$$
$$\varnothing \leq M([t/\alpha] \cup \sigma'') \qquad \mathsf{dom}(\sigma'') \subseteq \vec{\alpha}\,.$$

Let $\sigma' = [t/\alpha] \cup \sigma''$.

Since $\varnothing \leq M\sigma'$, we have $M = \varnothing$.

By Property 4.25, we find $\sigma \in \mathsf{tally}(D)$ and σ''' such that $\sigma' \simeq \sigma''' \circ \sigma$.

Therefore, since $\alpha\sigma' = t$, we have $\alpha\sigma\sigma''' \simeq t$. □

5.3.1 Towards a stronger completeness result

Theorem 5.12 is interesting because it proves that using this modified algorithm we keep the same completeness property as in the previous chapter. Moreover, we can derive intersection types using type annotations, while this was impossible in the previous algorithm: this algorithm is strictly more powerful. However, stronger properties would be desirable. For a start, we would like to ensure completeness also for expressions with annotations as long as they can be typed in $\mathcal{T}^{\mathrm{ra}\setminus\wedge}$. Moreover, we should try to ensure that, whenever an expression e has some type t in $\mathcal{T}^{\mathrm{ra}}$ (possibly derived using $[\mathrm{T}_{\wedge}]$), there is some way to annotate it (producing $\mathbf{e}$ such that $\mathsf{erase}(\mathbf{e}) = e$) so that type inference can accept it with the same type.

Achieving these results is challenging because explicit annotations reintroduce the difficulties with generalization discussed in Section 4.1.1 and avoided there thanks to the reformulated type system. The rules $[\mathrm{T}^{\mathrm{ra}}_{\mathsf{let}}]$, $[\mathrm{C}^{\mathrm{sata}}_{\mathsf{let}}]$, and $[\mathrm{C}^{\mathrm{sima}}_{\mathsf{let}}]$ all state that the variables $\vec{\alpha}$ bound by the let construct must not occur in some λ-environment. In $[\mathrm{C}^{\mathrm{sima}}_{\mathsf{let}}]$, this condition is $\vec{\alpha} \sharp M_1\sigma_1$: it depends on which type variables are introduced by σ_1. Therefore, equivalent type substitutions behave differently, which is undesirable.

We could try to solve this by modifying the tallying algorithm so that we can impose the constraint that some type variables ($\vec{\alpha}$) should not appear in the solution of other type variables (those in M_1). This has been done in other type inference systems using *unification under a mixed prefix* (Miller, 1992); for example, see the treatment of explicit type annotations in Pottier and Rémy (2003). However, it is not clear how to adapt this approach to semantic subtyping.

There is another difficulty to prove the stronger result. Assume that a

function $\lambda x.\, e$ is given the type $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$ using $[\mathrm{T}_\wedge]$. We want to annotate it so that type inference can obtain this type. In the original derivation, the body of the function is typed twice: assuming $(x\colon \mathsf{Int})$ and deriving Int, and assuming $(x\colon \mathsf{Bool})$ and deriving Bool. These two derivations might require different, possibly conflicting, annotations. Therefore, we should generalize type annotations, allowing expressions that are annotated with sets of types so that we can annotate the body of the function with all annotations needed for each typing. The typing rule $[\mathrm{T}^{\mathrm{ra}}_{::}]$ becomes then

$$\frac{P; M; \Delta \Vdash \mathbf{e}\colon t_i}{P; M; \Delta \Vdash (\mathbf{e} :: \{\, t_i \mid i \in I \,\})\colon t_i}$$

(during typing, we can choose freely which annotation to consider) and constraint generation is

$$\langle\!\langle (\mathbf{e} :: \{\, t_i \mid i \in I \,\})\colon t \rangle\!\rangle^{\Delta} = \bigvee_{i \in I} \left(\langle\!\langle \mathbf{e}\colon t_i \rangle\!\rangle^{\Delta} \wedge (t_i \mathrel{\dot{\leq}} t) \right) .$$

This solution is used in previous work on intersection type systems (Pierce, 1991; Reynolds, 1997; Davies, 2005; Dunfield, 2007). Introducing sets of type annotations can pose problems for efficiency, since each annotation must be tested in turn by backtracking. To avoid this, Dunfield (2007) augments type annotations with a fragment of the type environment which the type checker uses to select the correct annotation for each typing of the expression. We could adopt this solution also in our case.

6 Language extensions

In this chapter we describe a few possible extensions to the language of Chapter 3. We outline how to modify the semantics and the type system to account for them.

6.1 Binding typecase and pattern matching

6.1.1 Binding typecase

The typecase expression in our language has the form $e_0 \in \mathbf{t}\ ?\ e_1 : e_2$. In contrast, Frisch, Castagna, and Benzaken (2008) include a binder in their typecase: $(x = e_0) \in \mathbf{t}\ ?\ e_1 : e_2$, where x is bound in e_1 and e_2. Such a typecase is evaluated by evaluating e_0 to a value v, binding x to v, and evaluating either e_1 or e_2 according to whether v has type $\mathbf{t}$ or $\neg\mathbf{t}$.

As Castagna et al. (2014, app. E) observed, typecases with binders can be encoded as:

$$(x = e_0) \in \mathbf{t}\ ?\ e_1 : e_2 \quad \equiv \quad (\lambda x.\, x \in \mathbf{t}\ ?\ e_1 : e_2)\, e_0\ .$$

To add them as primitive, instead, we use the following two reduction rules

$$(x = v) \in \mathbf{t}\ ?\ e_1 : e_2 \ \rightsquigarrow\ e_1[v/x] \qquad \text{if } \mathsf{typeof}(v) \leq \mathbf{t}$$
$$(x = v) \in \mathbf{t}\ ?\ e_1 : e_2 \ \rightsquigarrow\ e_2[v/x] \qquad \text{if } \mathsf{typeof}(v) \leq \neg\mathbf{t}$$

and add $(x = E) \in \mathbf{t}\ ?\ e : e$ to the grammar of evaluation contexts. We use the typing rule:

$$\frac{\Gamma \vdash e_0 : t_0 \qquad \text{either } t_0 \leq \neg\mathbf{t} \text{ or } \Gamma, x : t_0 \wedge \mathbf{t} \vdash e_1 : t \qquad \text{either } t_0 \leq \mathbf{t} \text{ or } \Gamma, x : t_0 \setminus \mathbf{t} \vdash e_2 : t}{\Gamma \vdash ((x = e_0) \in \mathbf{t}\ ?\ e_1 : e_2) : t}$$

To type each branch, we assign to x a subtype of t_0: in the first branch, it is $t_0 \wedge \mathbf{t}$ because the branch will be selected only for values of type $\mathbf{t}$; in the second, correspondingly, it is $t_0 \setminus \mathbf{t}$. For example, if x has type $\mathsf{Int} \vee \mathsf{Bool}$, then $(y = x) \in \mathsf{Int}\ ?\ (y + 1) : 0$ is well typed because, in the first branch, y has type $(\mathsf{Int} \vee \mathsf{Bool}) \wedge \mathsf{Int}$ and $(\mathsf{Int} \vee \mathsf{Bool}) \wedge \mathsf{Int} \simeq \mathsf{Int}$. In contrast, the typecase without binder $x \in \mathsf{Int}\ ?\ (x + 1) : 0$ is ill-typed because x has type $\mathsf{Int} \vee \mathsf{Bool}$ also in the first branch, and + cannot be applied to an $\mathsf{Int} \vee \mathsf{Bool}$.

These rules can all be derived for the encoding. To derive the typing rule, in particular, we type $(\lambda x.\, x \in \mathbf{t}\ ?\ e_1 : e_2)$ as $(t_0 \wedge \mathbf{t} \to t) \wedge (t_0 \setminus \mathbf{t} \to t)$.

In practice, we might want to treat $x \in \mathbf{t}\ ?\ e_1 : e_2$ as syntactic sugar for $(x = x) \in \mathbf{t}\ ?\ e_1 : e_2$, with a new binding of x that shadows the previous one. This allows typecases on variables to refine the type of the variable in the branches, making $x \in \mathsf{Int}\ ?\ (x + 1) : 0$ well typed without explicit rebinding. It is a simple form of *occurrence typing* or *flow typing* (as studied, among others, by Tobin-Hochstadt and Felleisen, 2010; Pearce, 2013; Chaudhuri et al., 2017).

$$
\begin{aligned}
v/\mathbf{t} &= \begin{cases} [\,] & \text{if } \mathsf{typeof}(v) \leq \mathbf{t} \\ \mathsf{fail} & \text{otherwise} \end{cases} \\
v/x &= [v/x] \\
v/(p_1, p_2) &= \begin{cases} \varsigma_1 \cup \varsigma_2 & \text{if } v = (v_1, v_2),\ v_1/p_1 = \varsigma_1, \text{ and } v_2/p_2 = \varsigma_2 \\ \mathsf{fail} & \text{otherwise} \end{cases} \\
v/p_1 \& p_2 &= \begin{cases} \varsigma_1 \cup \varsigma_2 & \text{if } v/p_1 = \varsigma_1 \text{ and } v/p_2 = \varsigma_2 \\ \mathsf{fail} & \text{otherwise} \end{cases} \\
v/p_1 \mid p_2 &= \begin{cases} v/p_1 & \text{if } v/p_1 \neq \mathsf{fail} \\ v/p_2 & \text{otherwise} \end{cases}
\end{aligned}
$$

FIGURE 6.1 Semantics of patterns

6.1.2 Pattern matching

Typecases in our language can be used to represent a form of pattern matching. Here, we outline how we can add full-fledged pattern matching directly to the language. Similar formalizations of pattern matching for set-theoretic type systems have been described by Frisch (2004), Castagna et al. (2015b, app. E), and Castagna, Petrucciani, and Nguyễn (2016).

For simplicity, we only consider two-branch pattern matching. We extend the syntax with the match construct and with patterns:

$$
\begin{aligned}
e &::= \cdots \mid \mathsf{match}\ e\ \mathsf{with}\ p \to e \mid p \to e \\
p &::= \mathbf{t} \mid x \mid (p, p) \mid p \& p \mid p \mid p\,,
\end{aligned}
$$

with some restrictions on the variables that can appear in patterns: in (p_1, p_2) and $p_1 \& p_2$, p_1 and p_2 must have distinct variables; in $p_1 \mid p_2$, p_1 and p_2 must have the same variables.

A more familiar syntax for patterns is $p ::= \_ \mid c \mid x \mid (p, p) \mid p\ \mathsf{as}\ x \mid p \mid p$, with wildcards and constants instead of $\mathbf{t}$ types and with as-patterns "p as x" (in OCaml syntax; $x@p$ in Haskell) instead of conjunction. We can encode _ and c as $\mathbb{1}$ and b_c (both are in the grammar for $\mathbf{t}$), while "p as x" is $p \& x$, as will soon be clear.

To describe the semantics of pattern matching, we define a function $(\cdot)/(\cdot)$ that, given a value v and a pattern p, yields a result v/p which is either fail or a substitution ς mapping the variables in p to values (subterms of v). This function is defined in Figure 6.1. Then, we augment the reduction rules with

$$
\begin{aligned}
(\mathsf{match}\ v\ \mathsf{with}\ p_1 \to e_1 \mid p_2 \to e_2) &\rightsquigarrow e_1\varsigma \qquad \text{if } v/p_1 = \varsigma \\
(\mathsf{match}\ v\ \mathsf{with}\ p_1 \to e_1 \mid p_2 \to e_2) &\rightsquigarrow e_2\varsigma \qquad \text{if } v/p_1 = \mathsf{fail} \text{ and } v/p_2 = \varsigma
\end{aligned}
$$

and add $\mathsf{match}\ E\ \mathsf{with}\ p_1 \to e_1 \mid p_2 \to e_2$ to the grammar of evaluation contexts.

Set-theoretic types prove very useful to type pattern matching precisely. Given each pattern p, we can define a type $\wr p \wr$ that describes exactly the values

$$\frac{}{t/\mathbf{t} \dashv \varnothing} \qquad \frac{}{t/x \dashv (x\colon t')}\ t \le t' \qquad \frac{t_1/p_1 \dashv \Gamma_1 \qquad t_1/p_1 \dashv \Gamma_1}{t/(p_1,p_2) \dashv \Gamma_1 \cup \Gamma_2}\ t \le t_1 \times t_2$$

$$\frac{t/p_1 \dashv \Gamma_1 \qquad t/p_2 \dashv \Gamma_2}{t/p_1 \& p_2 \dashv \Gamma_1 \cup \Gamma_2} \qquad \frac{(t \wedge \wr p_1 \int)/p_1 \dashv \Gamma \qquad (t \setminus \wr p_1 \int)/p_2 \dashv \Gamma}{t/p_1 \mid p_2 \dashv \Gamma}$$

FIGURE 6.2 Environment typing for patterns

that match the pattern:

$$\wr \mathbf{t} \int = \mathbf{t} \qquad \wr x \int = \mathbb{1}$$
$$\wr (p_1,p_2) \int = \wr p_1 \int \times \wr p_2 \int \quad \wr p_1 \& p_2 \int = \wr p_1 \int \wedge \wr p_2 \int \quad \wr p_1 \mid p_2 \int = \wr p_1 \int \vee \wr p_2 \int$$

It can be shown that, for every well-typed v and every p, we have $v/p \neq \mathsf{fail}$ if and only if $\varnothing \vdash v\colon \wr p \int$. This allows us to formalize the exhaustiveness and redundancy checks that are often performed on pattern matching purely at the level of types. The typing rule for match is the following.

$$\frac{\begin{array}{c}\Gamma \vdash e_0\colon t_0 \\ \text{either } t_0 \le \neg \wr t_1 \int \text{ or } \Gamma, \Gamma_1 \vdash e_1\colon t \\ \text{either } t_0 \le \wr t_1 \int \text{ or } \Gamma, \Gamma_2 \vdash e_2\colon t\end{array}}{\Gamma \vdash \mathsf{match}\ e_0\ \mathsf{with}\ p_1 \to e_1 \mid p_2 \to e_2\colon t} \begin{cases} t_0 \le \wr p_1 \int \vee \wr p_2 \int \\ (t_0 \wedge \wr p_1 \int)/p_1 \dashv \Gamma_1 \\ (t_0 \setminus \wr p_1 \int)/p_2 \dashv \Gamma_2 \end{cases}$$

The "either … or …" conditions have the same purpose as for typecases. The side condition $t_0 \le \wr p_1 \int \vee \wr p_2 \int$ ensures that matching is exhaustive: any value produced by e_0 has type t_0 and therefore matches either p_1 or p_2. The other side conditions rely on the relation $t/p \dashv \Gamma$, defined in Figure 6.2. This relation describes which types we can assume for the variables in p when a value of type t is matched against p and matching succeeds. In particular, the following holds for every t, p, and v: if $t/p \dashv \Gamma$ and $\varnothing \vdash v\colon t$ and $v/p = \varsigma$, then, for every variable x in p, $\varnothing \vdash x\varsigma\colon \Gamma(x)$.

6.2 Polymorphic variants

Polymorphic variants are a feature of OCaml that provides a limited form of union types within a Hindley-Milner type system without subtyping. In contrast to normal variant types (in OCaml terminology; also called sum or disjoint union types) which require explicit type declarations like, for instance, `type t = A of int | B of bool` and require different types to have distinct labels, polymorphic variants do not require type declarations and allow different types to share some labels, thus providing a form of subtyping. In OCaml, we can build a polymorphic variant value simply as the pair of a tag and an argument: `` `A 3 `` or `` `B true ``, for example. We do not need to declare types, and we can write functions that are defined on some arbitrary tags. For example,

```
let f = function `A x -> (x mod 2 = 0) | `B x -> x
```

defines a function whose domain is (intuitively) the union (`A of int)∨(`B of bool) and which returns Booleans. Another function could be defined on the same tags, on more or fewer, or could associate the same tags to different types.

Polymorphic variants are typed in OCaml with a system described by Garrigue (2002, 2015), following the earlier work of Ohori (1995). This formalization avoids the introduction of true subtyping, but encodes a form of union subtyping into a unification-based setting. Other formalizations in Hindley-Milner type systems exist, for example, see Rémy (1989) and Blume, Acar, and Chae (2006), based on row polymorphism.

Polymorphic variant types and values can be added to our setting easily, since they are just a restricted form of union type. If we assume that constants include tags like `A and `B, then polymorphic variants can be encoded as pairs of a tag and a value. Otherwise, we can them add primitively as a new production `tag($t$) (for each tag `tag) in the grammar of types, whose interpretation can be derived from the encoding. We then add `tag($e$) to the syntax of expressions. Destructors can then be added by extending pattern matching, adding patterns of the form `tag(p).

In Castagna, Petrucciani, and Nguyễn (2016), we have described polymorphic variants at length. We have modelled the fragment of the type system of OCaml that concerns polymorphic variants by extending the work of Garrigue (2002, 2015). We prove that set-theoretic types allow us to give a more expressive type system and avoid some of the problematic and arguably unintuitive behaviour of polymorphic variants in OCaml.[1]

6.3 Records

Record types and expressions can be added to the language as follows. We add record types by representing them as finite functions from *fields* f, drawn from a set Field, to types: we write such functions as $\{\mathsf{f}_i : t_i \mid i \in I\}$. We add

$$t ::= \cdots \mid \{\mathsf{f}_i : t_i \mid i \in I\}$$

to the syntax of types, and we also add records to the domain of interpretation as finite functions from fields to domain elements (with, as usual, a label L):

$$d ::= \cdots \mid \{\mathsf{f}_i : d_i \mid i \in I\}^L .$$

Then, we extend the interpretation to have

$$[\![\{\mathsf{f}_i : t_i \mid i \in I\}]\!] = \left\{ \{\mathsf{f}_i : d_i \mid i \in I \cup J\}^L \;\middle|\; \forall i \in I.\, d_i \in [\![t_i]\!] \right\} :$$

the interpretation of a record type contains records with at least the labels specified in the type. This interpretation can be obtained by extending the relation $(d : t)$ as follows:

$$(\{\mathsf{f}_i : d_i \mid i \in I \cup J\}^L : \{\mathsf{f}_i : t_i \mid i \in I\}) = \forall i \in I.\, (d_i : t_i) .$$

1 We refer the reader to Castagna, Petrucciani, and Nguyễn (2016) for some examples of this behaviour.

In the language syntax, we add record expressions and values representing them once more as finite functions. We also add record field access.

$$e ::= \cdots \mid \{\mathsf{f}_i : e_i \mid i \in I\} \mid e.\mathsf{f} \qquad\qquad v ::= \cdots \mid \{\mathsf{f}_i : v_i \mid i \in I\}$$

We add the reduction rule

$$\{\mathsf{f}_i : v_i \mid i \in I\}.\mathsf{f}_{i_0} \rightsquigarrow v_{i_0} \qquad \text{if } i_0 \in I$$

and we add records to evaluation contexts (in order to keep evaluation deterministic, we need to choose some ordering on field names).

The typing rules are:

$$\frac{\forall i \in I.\ \ \Gamma \vdash e_i : t_i}{\Gamma \vdash \{\mathsf{f}_i : e_i \mid i \in I\} : \{\mathsf{f}_i : t_i \mid i \in I\}} \qquad\qquad \frac{\Gamma \vdash e : \{\mathsf{f} : t\}}{\Gamma \vdash e.\mathsf{f} : t}$$

Record types in semantic subtyping have been described first in a monomorphic setting in Frisch's PhD thesis (Frisch, 2004). Frisch uses quasi-constant functions instead of finite functions to consider also records with infinite domain. Such records could be constructed by specifying a default initializer for every field except those mentioned explicitly: for example, allowing an expression $\{\mathsf{f}_1 : e_1, \ldots, \mathsf{f}_n : e_n, \_ : e\}$, where e is the default. Record types as described here can be recovered by having a value undef that can be used for e, signifying that only the fields f_i are defined. We have used finite relations to give a concise and familiar description, but quasi-constant functions with an explicit domain element for undefined fields are useful to represent more notions uniformly. For example, they can represent also *closed* record types, which allow depth subtyping but not width subtyping.[2]

6.3.1 Polymorphic typing of record operations

The definitions in this section allow polymorphic typing of record access. For example, the function $\lambda x.\,(x.\mathsf{f})$ can be given the type scheme $\forall\alpha.\,\{\mathsf{f} : \alpha\} \to \alpha$. The type states that the function can be applied to records with a field f and any number of other fields, and captures correctly the dependence between the input and output types.

However, it seems that we cannot describe precise polymorphic typing of record update operations. For instance, consider an operator $\setminus\mathsf{f}$ which, applied to a record value, removes the field f it it is present. Then, we would like the function $\lambda x.\,(x \setminus \mathsf{f})$ to be applicable to any record. We would like its type scheme to express this behaviour, that is, that the output record has all fields in the input record (with the same types) except for f. Such a type scheme cannot be expressed in our system, unlike, for instance, in other systems (e.g., Rémy, 1989, 1993; Blume, Acar, and Chae, 2006) using row polymorphism or similar features. It remains to be seen whether and how such features can be integrated with semantic subtyping.

2 Such types are available in Flow, for example, as *exact object types*.

7 Discussion

In the four previous chapters, we have studied how to use set-theoretic types and semantic subtyping, as defined in Chapter 2, for implicitly typed languages with type inference. Initially, in Chapter 3, we have given a declarative presentation of the type system (relying on the structural rules [$T_{\leq}$] and [$T_{\wedge}$]), which is simple to understand but does not directly yield an algorithm. Then, in Chapter 4, we have described a type inference algorithm for the type system. In Chapter 5, we have described how to make type inference more effective in the presence of type annotations. Finally, in Chapter 6, we have outlined how to extend the language with more features.

The main technical contribution of Chapter 3 is to show how to use polymorphic set-theoretic types for implicitly typed languages. The main difficulty is the proof of type soundness: as a consequence of the presence of semantic subtyping and negation types, it required us to extend the type system with a novel rule [$T_{\lambda\neg}$] to derive negation types for functions. We prove soundness for the type system extended with that rule ($\mathcal{T}^{\lambda\neg}$); as a consequence, the system $\mathcal{T}$ without that rule is sound too (since it allows fewer derivations).

In Chapter 4, the main contribution is the description of type inference, with its results of soundness and completeness. To achieve these results, we have reworked techniques from previous work on inference with subtyping (notably, the reformulated type system) to solve difficulties in the treatment of generalization for let bindings.

In this chapter, we discuss the relationship between this work and related work concerning subtyping, union and intersection types, and type inference. We also point out some directions for future work.

7.1 Related work

SET-THEORETIC TYPES AND SEMANTIC SUBTYPING: This work builds upon those of Frisch, Castagna, and Benzaken (2008) and Castagna et al. (2014, 2015b) on typing functional languages with set-theoretic types.

Frisch, Castagna, and Benzaken (2008) only consider monomorphic typing. In contrast, Castagna et al. (2014) describe a type system with prenex polymorphism for an explicitly typed language: every function must be annotated with its type and the instantiation of polymorphic functions is explicit too. The semantics is complex because it must propagate instantiations of polymorphic functions during reduction (intersection types make this more difficult). Castagna et al. (2015b) show how to add local type inference to infer instantiations, while functions remain explicitly typed. They also outline how to do full type inference, but without any result of completeness; inference for let-polymorphism is treated only cursorily.

Here, we move to an implicitly typed setting. This, together with our restriction on typecases (forbidding arrow types apart from $\mathbb{0} \to \mathbb{1}$), allows us to give a standard operational semantics which does not depend on static types (it only assumes some form of runtime tagging for values, as found in dynamic languages). Moreover, we develop type inference for programs without annotations fully, proving completeness with respect to the system without intersection introduction.

ALGEBRAIC SUBTYPING: A notable recent work on subtyping and type inference is that of Dolan and Mycroft (2017), already mentioned in Chapter 4. Dolan and Mycroft define subtyping and use it, together with let-polymorphism, in the type system of a language for which they prove soundness, completeness, and principality of type inference. We also combine subtyping and let-polymorphism in our type system, and we prove soundness and completeness; principality, however, does not hold (as exemplified in Section 4.4.1).

An important aspect of their work is the algebraic definition of subtyping. This yields a subtyping relation with very different behaviour as compared to semantic subtyping. Important differences include the following.

- Algebraic subtyping is based on an open-world assumption and strives to ensure extensibility. As a result, for example, subtyping does not identify with the bottom type some intersection types that we can expect to be uninhabited (e.g., $(\mathsf{Int} \to \mathsf{Bool}) \wedge (\mathsf{Int} \times \mathsf{Bool})$). The reasoning is that this makes the behaviour of subtyping simpler and more regular, but also that it makes subtyping more amenable to extensions – a language might introduce a new value that acts as both a function and a pair, for example. This precludes reasoning on negation as in semantic subtyping; indeed, Dolan and Mycroft do not include negation types.

- Algebraic subtyping does not seem to be suited to a system with ad-hoc polymorphism in the form of overloaded functions (as discussed by Dolan, 2016, Section 10.2.3) nor, presumably, for systems with intersection introduction. This is because the following equivalence holds:
$$(t_1 \to t_2) \wedge (t_1' \to t_2') \simeq (t_1 \vee t_1') \to (t_2 \wedge t_2') \,.$$
In a system with the rule $[\mathrm{T}_\wedge]$, instead, we expect $\lambda x.\, x$ to have type $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$ but not $(\mathsf{Int} \vee \mathsf{Bool}) \to (\mathsf{Int} \wedge \mathsf{Bool})$.

- Dolan and Mycroft prove that type inference infers principal types and, moreover, that these types are *polar*: by this they mean, in a nutshell, that union types never appear in contravariant position and intersection types never appear in covariant position. This simplifies the form of constraints that they must solve. This property seems unachievable in our system because of the typing that we want to allow for typecases. Indeed, the principal type of a function like $\lambda x.\, x \in \mathsf{Int}\ ?\ x + 1 : \neg x$ should use the union $\mathsf{Int} \vee \mathsf{Bool}$ in the domain.

In brief, this work and that of Dolan and Mycroft should be seen as very different approaches to adding subtyping to implicitly typed languages with

type inference. However, it would be interesting to study whether the algebraic construction of Dolan and Mycroft could be adapted to describe a subtyping relation closer to ours (notably, without the equivalence on intersections of arrows shown above) without losing its advantages in terms of extensibility and more regular behaviour.

OTHER SYSTEMS WITH UNION AND INTERSECTION TYPES: Much work on intersection types for the λ-calculus, including that of Coppo and Dezani-Ciancaglini (1980), Barendregt, Coppo, and Dezani-Ciancaglini (1983), and Barbanera, Dezani-Ciancaglini, and de'Liguoro (1995) and the work of Reynolds (1997) on the Forsythe language, does not allow intersections that correspond to overloading as in our system (the theory of Barbanera, Dezani-Ciancaglini, and de'Liguoro (1995) satisfies the equivalence $(t_1 \to t_2) \wedge (t_1' \to t_2') \simeq (t_1 \vee t_1') \to (t_2 \wedge t_2')$ that we have discussed above). Instead, the work on refinement types with datatype refinements (Freeman and Pfenning, 1991; Davies, 2005; Dunfield, 2007) uses intersection types in a way that is more similar to ours, though the arrows in an intersection must all refine a single ML type.

Recently, Muehlboeck and Tate (2018) have described a way to integrate union and intersection types in an existing subtyping relation. This approach has been used in the Ceylon programming language by Red Hat. However, their work concerns specifically the definition of subtyping and its decision procedure: it is therefore more closely related to the work we have as background than to the new results in this thesis.

TYPE INFERENCE FOR SUBTYPING: The addition of subtyping to a language presents a significant challenge for type inference, and there is a long line of work on this problem (Fuh and Mishra, 1988; Mitchell, 1991; Aiken and Wimmers, 1993; Pottier, 2001), the aforementioned work of Dolan and Mycroft (2017) being a recent result. There is also a long history of work on type inference with intersection types (Ronchi Della Rocca, 1988; Kfoury and Wells, 2004) and union types, including in the work on soft typing (Cartwright and Fagan, 1991; Aiken, Wimmers, and Lakshman, 1994), as well as both combined (Aiken and Wimmers, 1993). These challenges are intertwined because intersection and union types (or at least meet and join meta-operations on types) are needed to describe type inference and to simplify type constraints.

TYPE INFERENCE WITH TYPE ANNOTATIONS: The combination of ML-style type inference with explicit type annotations has often been studied in order to add higher-order polymorphism to ML. For instance, Odersky and Läufer (1996) describe type inference and reduce it to unification under a mixed prefix (Miller, 1992). Peyton Jones et al. (2007) build on that work, combining it with local type inference (Pierce and Turner, 2000) to reduce the number of required annotations. We have a different goal – to use annotations to allow intersection introduction – but annotations with explicit polymorphism also seem to require some analogue to unification under a mixed prefix.

Much work on type inference for partially annotated programs has considered local type inference, often using *bidirectional type checking*. This normally means that the types of function parameters are not inferred from their use, though it is not necessarily so: Dunfield and Krishnaswami (2013) and Peyton Jones et al. (2007), for instance, propose bidirectional type checking algorithms that can also infer types for function parameters. Local type inference techniques have also seen wide use in industry – for instance, in Scala (Odersky, Zenger, and Zenger, 2001), C♯ (Bierman, Meijer, and Torgersen, 2007), and TypeScript (Bierman, Abadi, and Torgersen, 2014). We have preferred to add annotations while keeping the structure of constraint-based type inference, but we could also attempt to restructure our algorithm in a way inspired by these presentations.

7.2 Future work

Possible directions for future work include improving the description of type inference, considering different strategies for type checking partially annotated programs, and studying precise typing of record operations.

SOUNDNESS OF TYPE INFERENCE: Chapter 4 presents results of soundness and completeness of type inference. A limitation is that we cannot prove that type inference is sound with respect to the type system $\mathcal{T}^{\mathrm{i}\setminus\wedge}$ (that without the intersection-introduction rule [$\mathrm{T}_\wedge$]), though we conjecture it. To solve this, we should find a different proof of equivalence between the standard and the reformulated type systems, one that does not rely on [$\mathrm{T}_\wedge$]. Moreover, the current proof is quite convoluted, relying as it does on tracking the instantiations of typing schemes. A better proof could be easier to extend, notably to have an equivalent result for the language with type annotations. Dolan and Mycroft have suggested an alternative proof technique (see footnote 3 on p. 93) which is still to be explored.

TYPE INFERENCE WITH ANNOTATIONS: The development of type inference in the presence of type annotations in Chapter 5 is a first step, but much more can be done. In Section 5.3.1, we have outlined how we can work towards stronger results of completeness. It would be interesting to prove that any expression typed using [$\mathrm{T}_\wedge$] can be annotated so that type inference can accept it with the same type. It would also be useful to characterize which expressions and typing derivations require annotations and which do not, especially to ensure that the system can be used effectively without having to write an excessive amount of annotations.

In Chapter 5 we try to derive intersection types for expressions only when they are annotated. We could try to study techniques to infer intersection types also for some functions without annotations. For example, we can try to exploit the information in typecases inside the function to find out how many and which arrows we should check.

LOCAL TYPE INFERENCE: In Chapters 4 and 5 we have studied global type inference: our algorithm tries to infer the type of the parameter of a function from its uses in the body. In contrast, local type inference techniques often do not do so (see, e.g., Pierce and Turner, 2000): they infer types for parameters only if they are known from the context. As a result, these systems are often simpler to describe, to implement efficiently, and to extend with more features, while requiring only a modest amount of type annotations. It would be worthwhile to study how such an approach can be used to type check our language. Castagna et al. (2015b) have already studied local type inference for polymorphic set-theoretic types, but they did not consider any form of bidirectional propagation, meaning that all functions had to be annotated and only the instantiations of polymorphic functions were inferred.

RECORD TYPING: We have sketched a simple treatment of record types in Section 6.3. However, as we have mentioned, it does not allow precise polymorphic typing of record operations including field update, field deletion, and record concatenation. It would be interesting to explore how to provide this additional expressiveness in our framework, possibly by integrating some form of row polymorphism.

Part II

Gradual typing

8 Introduction

This part of the thesis is devoted to *gradual typing*, an approach that combines the safety guarantees of static typing with the flexibility of dynamic typing (Siek and Taha, 2006). The initial goal of this work was to study how gradual typing could be used in polymorphic type systems with set-theoretic types. It has led, however, to a novel approach to the definition of gradual type systems, independent of the idea of set-theoretic types. Therefore, we first illustrate our approach in a Hindley-Milner type system with implicit parametric polymorphism but no subtyping. Then, we study the extension with set-theoretic types.

The core idea of gradual typing is to introduce an *unknown* type, denoted by "?", used to inform the compiler that additional type checks may be needed at run time. Programmers can add type annotations to a program *gradually* and control precisely how much checking is done statically versus dynamically. The type checker ensures that the parts of the program that are typed with *static types* (i.e., types that do not contain ?) enjoy the type safety guarantees of static typing – well-typed expressions never get stuck – while the parts annotated with *gradual types* (i.e., types in which the dynamic type ? occurs) enjoy the same property modulo the possibility to fail on some dynamic type check inserted by the type-driven compilation.

8.1 Gradual typing with polymorphic set-theoretic types

Some practical benefits of combining gradual typing with union and intersection types were presented by Castagna and Lanvin (2017) in a monomorphic setting. With this work we extend such benefits to a polymorphic setting with type inference.

For a glimpse of what can be done in this setting, consider the following ML-like code snippet adapted from Siek and Vachharajani (2008):

```
let mymap (condition) (f) (x: ?) =
  if condition then Array.map f x else List.map f x
```

According to the value of the argument condition, the function mymap applies either the array version or the list version of map to the other two arguments. This example cannot be typed using only simple types: the type of x and the return type of mymap change depending on the value of condition. By annotating x with the gradual type ?, the type inference system for gradual types of Siek and Vachharajani (2008) can type this function with the type $\mathsf{Bool} \to (\alpha \to \beta) \to ? \to ?$. That is, inference recognizes that the parameter condition must be bound to a Boolean value, and the compilation process adds dynamic checks to ensure that the value bound to x will be, according to the

case, either an array or a list whose elements are of a type compatible with the actual input type of f.

This type however is still imprecise. For example, if we pass a value that is neither an array nor a list as the last argument to mymap, then the application is well typed, even though its execution will always fail, independently of the value of condition. Moreover, the type gives no useful information about the result of mymap, even though it will always be either a list or an array of β elements. These problems can be remedied by using set-theoretic types:

```
let mymap (condition) (f) (x: (α array ∨ α list) ∧ ?) =
  if condition then Array.map f x else List.map f x
```

The union indicates that a value of this type is *either* an array *or* a list, both of α elements. The intersection indicates that x has *both* type α array $\vee$ α list *and* type ?. Intuitively, this type annotation means that the function mymap accepts for x a value of any type (which is indicated by ?), as long as this value is also either an array or a list of α elements (α being the domain of the f argument). The use of the intersection of a union type with ? to type a parameter corresponds to a programming style in which the programmer asks the system to enforce *statically* that the function will be applied only to arguments in the union type and delegates to the system any *dynamic* check regarding the use of the parameter in the body of the function. A system like that in Chapter 10 could deduce for this definition the type:

$$\mathsf{Bool} \to (\alpha \to \beta) \to ((\alpha\ \mathsf{array} \vee \alpha\ \mathsf{list}) \wedge ?) \to (\beta\ \mathsf{array} \vee \beta\ \mathsf{list})$$

This type forces the last argument of mymap to be either an array or a list of elements whose type is the input type of the argument bound to f. Note that the return type of mymap is no longer gradual: the union type allows us to define it without any loss of precision and to capture its correlation with the return type of the argument bound to f. The derivation of this type is used by the compiler to insert dynamic type checks that ensure type soundness. In particular, the compilation process described in Section 9.2.4 inserts in the body of mymap the casts that check dynamically that the first occurrence of x is bound to an array of elements of the appropriate type, and that the second occurrence of x is bound to a list of such elements, producing code like

```
let mymap (condition) (f) (x: (α array ∨ α list) ∧ ?) =
  if condition then Array.map f (x ⟨α array⟩) else List.map f (x ⟨α list⟩)
```

where $e\langle t \rangle$ is a type cast expression that checks dynamically whether the result of e has type t.

This kind of type discipline is out of reach of current systems. Castagna and Lanvin (2017) have described it only in a monomorphic setting and without type inference. A similar discipline is allowed by the gradual unions of Toro and Tanter (2017), but they too do not consider polymorphism and type inference. To obtain the system we aim for, we want gradual typing to coexist with polymorphic set-theoretic types with semantic subtyping.

8.2 Our approach

Standard presentations of gradual typing rely on the *consistency* relation $\sim$. Given two gradual types τ_1 and τ_2, $\tau_1 \sim \tau_2$ holds when τ_1 and τ_2 are equal everywhere except where they contain ?. For example:

$$? \sim \mathsf{Int} \qquad \mathsf{Int} \sim ? \qquad \mathsf{Int} \not\sim \mathsf{Bool} \qquad ? \to \mathsf{Int} \sim (\mathsf{Int} \to \mathsf{Bool}) \to ? \,.$$

Consistency is reflexive and symmetric but not transitive; its transitive closure is the total relation on gradual types, because every type is consistent with ?. Since it is not transitive, consistency cannot be added to a type system by a subsumption-like structural rule,[1] as that would yield a system which accepts every program, even those that do not contain ? and would be ill-typed in a sound type system. Therefore, consistency is normally added by embedding it in elimination rules: for instance, by replacing the normal rule for application in the simply typed λ-calculus with the following two rules.

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}\ \tau' \sim \tau_2 \qquad \frac{\Gamma \vdash e_1 : ? \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : ?}$$

Adding consistency to the rules is not necessarily an ad-hoc process: there has been work on formalizing the transition from a static to a gradual type system (e.g., Cimini and Siek, 2016; Garcia, Clark, and Tanter, 2016).

The presentation of these type systems resembles that of algorithmic type systems for languages with subtyping: instead of a structural rule for subsumption, subtyping judgments are embedded in several rules. This style of presentation describes effectively the behaviour of a type checker. However, adding subtyping by a single structural rule gives a concise way to describe the relation between a system with subtyping and one without, and also to compare different algorithmic type systems. We show that this holds for gradual typing as well, and we describe what is to our knowledge the first presentation of a gradual type system that relies entirely on a single structural rule to "gradualize" an existing static type system.

This structural rule does not, of course, use consistency. It uses instead the relation that we call *materialization* and denote by $\sqsubseteq$. Given τ_1 and τ_2, $\tau_1 \sqsubseteq \tau_2$ holds when τ_2 is more precise than τ_1, that is, when τ_2 is obtained from τ_1 by replacing some occurrences of ? by gradual types.[2] Materialization is a preorder and can therefore be used in a structural rule. Adding the rule

$$[\mathrm{T}_\sqsubseteq]\ \frac{\Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau}\ \tau' \sqsubseteq \tau$$

1 In logic, logical rules refer to a particular connective (here, a type constructor, that is, either $\to$, or $\times$, or b), while identity rules (e.g., axioms and cuts) and structural rules (e.g., weakening and contraction) do not.

2 This is the relation that Siek and Vachharajani (2008) name "less or equally informative". A fitting and concise name would be "precision": we avoid it because it is already used for the inverse relation, with ? at the top, by Garcia (2013) and others.

In Castagna et al. (2019), the symbol used is $\preccurlyeq$. We use $\sqsubseteq$, following Siek and Vachharajani (2008), to make it more distinguishable from $\leq$.

is enough to add gradual typing to a static type system. A type system defined using consistency corresponds to a particular strategy of building derivations using $[\mathrm{T}_{\sqsubseteq}]$.

To have both gradual typing and subtyping in the same type system, it suffices to have both $[\mathrm{T}_{\sqsubseteq}]$ and a standard subsumption rule. However, to do so we must extend an existing subtyping relation $\leq$ on static types to be defined on gradual types; we denote the relation on gradual types by $\leq^?$. How should it treat the unknown type? We follow previous approaches (notably Siek and Taha, 2007) in having subtyping treat ? simply as a new base or abstract type: that is, we have $? \leq^? ?$ but not, for instance, $? \leq^? \mathsf{Int}$ or $\mathsf{Int} \leq^? ?$. Subtyping and materialization then have clearly separated purposes. Subtyping and gradual typing are added to the type system as two separate structural rules, without affecting the other typing rules. This stands in contrast with previous work, including that by Siek and Taha (2007), that uses both subtyping and consistency or combines them to obtain a non-transitive *consistent-subtyping* relation (e.g., Siek and Taha, 2007; Garcia, Clark, and Tanter, 2016; Castagna and Lanvin, 2017).

Defining a suitable subtyping relation for gradual set-theoretic types is challenging. It turns out that we cannot give a set-theoretic interpretation to the unknown type directly: we will see that we cannot treat $? \setminus ?$ as an empty type, like we would expect with a set-theoretic subtyping relation. Instead, we define subtyping on gradual types in terms of subtyping on static types by replacing the occurrences of ? with type variables, an operation that we name *discrimination*. To define subtyping, we distinguish whether ? occurs under a negation type or not, in order to ensure that the problematic judgment $? \setminus ? \leq^? \mathbb{0}$ does not hold.

This idea of interpreting gradual types by replacing occurrences of ? with static types originated as a way to define subtyping, but it informs our entire approach. It gives us a way to define materialization in terms of discrimination and type substitutions, which is useful because it works for both inductively and coinductively defined types. It also allows us to describe type inference for gradual typing by reusing directly the algorithms for static type system – unification in the absence of subtyping and tallying (as in Section 4.3) with set-theoretic types – by adding pre- and post-processing steps that turn occurrences of ? to variables and back to ?.

Finally, our approach to defining gradual typing using materialization sheds some light on the logical meaning of gradual typing. It is well known that there is a strong correspondence between systems with subtyping and systems without subtyping but with explicit coercions: every usage of the subsumption rule in the former corresponds to the insertion of an explicit coercion in the latter. Our definition of materialization yields an analogous correspondence between a gradually typed language and the cast calculus to which the language is compiled: every usage of the materialization rule in the former corresponds to the insertion of an explicit cast in the latter. As such, the cast calculus looks like an important ingredient for a Curry-Howard isomorphism for gradual

typing disciplines. An intriguing direction for future work is to study the logic associated with these expressions.

8.3 Overview

Our first step, in Chapter 9, is to use our approach to add gradual typing to ML-like languages. We define the type system of a gradually typed language in declarative form, using the structural rule [$T_{\sqsubseteq}$] for materialization. Then, we define an associated cast language and compilation. We study type inference and prove it sound and complete with respect to declarative typing. Finally, we outline how the declarative type system could be extended with subtyping, considering a simple syntactic subtyping relation without set-theoretic types.

Then, in Chapter 10, we study how to apply our approach with set-theoretic types in order to obtain a system that allows the typing discipline discussed in Section 8.1. We describe two challenges. One is to define a suitable subtyping relation on gradual set-theoretic types. The other is to adapt type inference to subtyping. For the latter, we prove soundness of type inference, but not completeness.

We conclude in Chapter 11 by discussing our results, comparing our approach to previous work, and pointing out directions for future research.

In the work on which this part of the thesis is based (Castagna et al., 2019), we define operational semantics for the cast calculi of Chapter 9 and Chapter 10 and prove soundness for their type systems. Here, we only give a quick overview of the semantics, whereas the full definition is given in appendix and, for the proofs of the result, we refer the reader to the cited work. There are several reasons for this omission, which we have already anticipated in Chapter 1. Notably, the challenges and concerns in the definition of the semantics of a cast calculus (one with set-theoretic types, in particular, since the cast calculus of Chapter 9 has a simple and standard semantics) are quite different from those studied in the rest of the thesis, which concentrates on typing (the study of subtyping, declarative typing, and type inference) and considers simple semantics that are independent of typing (like those in Chapters 3 and 13).

9 *Gradual typing for Hindley-Milner systems*

In this chapter we add gradual typing to a language with ML-style polymorphism, following the approach that we have introduced.

CHAPTER OUTLINE:

Section 9.1 We describe the syntax of types and of the source language and the declarative type system. We explain the relationship between our presentation and standard gradual type systems.

Section 9.2 We describe the cast language and how to compile expressions.

Section 9.3 We describe type inference for the source language and prove it sound and complete.

Section 9.4 We outline how we can add subtyping to the declarative system. However, we do not study how to extend type inference for subtyping: we will do so in the next chapter when we consider set-theoretic types.

9.1 Source language

9.1.1 Types and expressions

Let α, β, and γ range over a countable set TVar of *type variables*. Let b range over a set Base of *base types* (e.g., $\mathsf{Base} = \{\mathsf{Int}, \mathsf{Bool}\}$). Let c range over a set Const of *constants*.

Static and *gradual types* are inductively defined by the following two grammars

$$\begin{array}{lll} \mathsf{SType} \ni t ::= \alpha \mid b \mid t \times t \mid t \to t & & \text{static types} \\ \mathsf{GType} \ni \tau ::= ? \mid \alpha \mid b \mid \tau \times \tau \mid \tau \to \tau & & \text{gradual types} \end{array}$$

and *source language expressions* by

$$e ::= x \mid c \mid \lambda x.\, e \mid \lambda x\colon \tau.\, e \mid e\ e \mid (e, e) \mid \pi_i\ e \mid \mathsf{let}\ \vec{\alpha}\ x = e\ \mathsf{in}\ e\ .$$

(For simplicity, in this chapter we do not consider recursive types.)

Static types SType (ranged over by t) are the types of an ML-like language: type variables, base types, products, and arrows. Gradual types GType (ranged over by τ) add the unknown type ? to them.

The source language is a fairly standard λ-calculus with constants, pairs (e, e), projections for the elements of a pair $\pi_i\ e$ (where $i \in \{1, 2\}$), plus a

let construct. It is similar to the language in Chapter 3 without the typecase construct, but there are two aspects to point out.

One is that there are two forms of λ-abstraction: $\lambda x.\, e$ and $\lambda x\colon \tau.\, e$. In the latter, the annotation τ fixes the type of the argument, whereas in the former the type can be chosen during typing (and will in practice be computed by inference). Furthermore, the type τ in the annotation is gradual, while in $\lambda x.\, e$ the inferred type of the parameter must be a static type t (cf. Figure 9.1, rule $[\mathrm{T}_\lambda]$). This is the same restriction imposed by Garcia and Cimini (2015) to properly reject some ill-typed programs. For example, without this restriction $\lambda x.\, (x + 1, \neg x)$ would be well typed since, by inferring the type ? for x, we can deduce for $\lambda x.\, (x + 1, \neg x)$ the type $? \to \mathsf{Int} \times \mathsf{Bool}$. But $\lambda x.\, (x + 1, \neg x)$ is not a well-typed term in ML, therefore by the principles of gradual typing (see Theorem 1 of Siek et al., 2015) it must be rejected unless its parameter is explicitly annotated by a type in which ? occurs (here, annotated by ? itself).

The second non-standard element of this syntax is that the let binding is decorated with a vector $\vec{\alpha}$ of type variables, as in $\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2$. This *decoration* (we reserve the word *annotation* for types annotating parameters in λ-abstractions) serves as a binder for the type variables that appear in annotations occurring in e_1. For instance, $\mathsf{let}\ \alpha\ z = \lambda x\colon \alpha.\, x\ \mathsf{in}\ e$ and $\mathsf{let}\ z = \lambda x.\, x\ \mathsf{in}\ e$ are equivalent, while $\mathsf{let} z = \lambda x\colon \alpha.\, x\ \mathsf{in}\ e$ means that α was introduced in an outer expression such as $\lambda y\colon \alpha.\, \mathsf{let}\ z = \lambda x\colon \alpha.\, x\ \mathsf{in}\ e$. The normal let from ML can be recovered as the case where $\vec{\alpha}$ is empty (which would always be the case if, as in ML, function parameters never had type annotations). We have used analogous decorations for the same purpose in Chapter 5.

As customary, we consider expressions modulo α-renaming of bound variables. In $\lambda x.\, e$ and $\lambda x\colon \tau.\, e$, x is bound in e; in $\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2$, x is bound in e_2 and the $\vec{\alpha}$ variables are bound in e_1. Following standard usage, we refer to the source language also as the *gradually typed language*.

9.1.2 Type system

We describe the declarative type system of the source language.

We use the standard notion for type schemes and type environments. A type scheme has the form $\forall \vec{\alpha}.\, \tau$, where $\vec{\alpha}$ is a vector of distinct variables. We identify type schemes with an empty $\vec{\alpha}$ with gradual types. A type environment Γ is a finite function from variables to type schemes.

The type system $\mathcal{T}_?$ is defined by the rules in Figure 9.1.

The first eight rules are almost those of a standard Hindley-Milner type system. In $[\mathrm{T}_c]$, we use b_c to denote the base type for a constant c (e.g., $b_3 = \mathsf{Int}$). One important aspect to note is that the types used to instantiate the type scheme in $[\mathrm{T}_x]$ and the type used for the domain in $[\mathrm{T}_\lambda]$ must all be static types, as forced by the use of the metavariable t.

The other non-standard aspect is the rule for let. To type $\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2$, we type e_1 with some type τ_1; then, we type e_2 in the expanded environment in which x has type $\forall \vec{\alpha}, \vec{\beta}.\, \tau_1$. The first side condition ($\vec{\alpha}, \vec{\beta} \sharp \Gamma$) asks that all the variables we generalize do not occur free in Γ; this is standard. The second

$$[\mathrm{T}_x]\ \frac{}{\Gamma \vdash x : \tau[\vec{t}/\vec{\alpha}]}\ \Gamma(x) = \forall\vec{\alpha}.\,\tau \qquad [\mathrm{T}_c]\ \frac{}{\Gamma \vdash c : b_c}$$

$$[\mathrm{T}_\lambda]\ \frac{\Gamma, x : t \vdash e : \tau}{\Gamma \vdash (\lambda x.\, e) : t \to \tau} \qquad [\mathrm{T}_{\lambda:}]\ \frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash (\lambda x : \tau'.\, e) : \tau' \to \tau}$$

$$[\mathrm{T}_{\mathrm{app}}]\ \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$[\mathrm{T}_{\mathrm{pair}}]\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad [\mathrm{T}_{\mathrm{proj}}]\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\ e : \tau_i}$$

$$[\mathrm{T}_{\mathsf{let}}]\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \forall\vec{\alpha},\vec{\beta}.\,\tau_1 \vdash e_2 : \tau}{\Gamma \vdash (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2) : \tau}\ \begin{cases} \vec{\alpha},\vec{\beta} \sharp \Gamma \\ \vec{\beta} \sharp e_1 \end{cases}$$

$$[\mathrm{T}_\sqsubseteq]\ \frac{\Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau}\ \tau' \sqsubseteq \tau$$

FIGURE 9.1 $\mathcal{T}_?$: Typing rules of the source language

condition ($\vec{\beta} \sharp e_1$) states that the type variables $\vec{\beta}$ must not occur free in e_1. This means that the type variables that are explicitly introduced by the programmer (by using them in annotations) can only be generalized at the level of a let binding by explicitly specifying them in the decoration. In contrast, type variables introduced by the type system (i.e., the fresh variables in the type t in the rule $[\mathrm{T}_\lambda]$) can be generalized at any let (implicitly, that is, by the type system), provided they do not occur in the environment. Note that we recover the standard Hindley-Milner rule for let bindings when expressions do not contain annotations and decorations are empty.

As anticipated, the type system does not need to deal with gradual types explicitly except in one rule. Indeed, the first eight rules do not check anything regarding gradual types (they only impose restrictions that some types must be static). The last rule, $[\mathrm{T}_\sqsubseteq]$, is a subsumption-like rule that allows us to make any gradual type more precise by replacing occurrences of ? with arbitrary gradual types. This is accomplished by the *materialization* relation $\sqsubseteq$ defined below.

MATERIALIZATION: Intuitively, $\tau_1 \sqsubseteq \tau_2$ holds when τ_2 can be obtained from τ_1 by replacing some occurrences of ? with arbitrary gradual types, possibly different for every occurrence. This relation can be defined easily by the following inductive rules, which merely add the reflexive case for type

variables to the rules of Siek and Vachharajani (2008):[1]

$$\frac{}{? \sqsubseteq \tau} \qquad \frac{}{\alpha \sqsubseteq \alpha} \qquad \frac{}{b \sqsubseteq b} \qquad \frac{\tau_1 \sqsubseteq \tau_1' \qquad \tau_2 \sqsubseteq \tau_2'}{\tau_1 \times \tau_2 \sqsubseteq \tau_1' \times \tau_2'} \qquad \frac{\tau_1 \sqsubseteq \tau_1' \qquad \tau_2 \sqsubseteq \tau_2'}{\tau_1 \to \tau_2 \sqsubseteq \tau_1' \to \tau_2'}$$

However, this definition is intrinsically tied to the syntax of types. Instead, we want the definition of materialization to remain valid also when we extend the language of types we use (notably with recursive types, which preclude giving an inductive definition in this way). Therefore, we give a definition based on our view, anticipated earlier, of occurrences of ? as type variables.

First, let us define a new sort of types, *type frames*, as follows:

$$\mathsf{TFrame} \ni T ::= X \mid \alpha \mid b \mid T \times T \mid T \to T$$

where X ranges over a set FVar of *frame variables* disjoint from TVar. Type frames are like gradual types except that, instead of ?, they have frame variables. We write TFrame for the set of all type frames.

We introduce some additional notation that we will use later. We write Var for $\mathsf{TVar} \cup \mathsf{FVar}$ and use A to range over it. We write $\mathsf{var}(T)$ for the set of variables in a type frame T, and we write $\mathsf{tvar}(T)$ and $\mathsf{fvar}(T)$ respectively for $\mathsf{var}(T) \cap \mathsf{TVar}$ and $\mathsf{var}(T) \cap \mathsf{FVar}$. We use $\mathsf{var}(\cdot)$ also for static and gradual types, as well as for type schemes and type environments (to denote the set of free type variables).

Given a type frame T, we write $T^\dagger$ for the gradual type obtained by replacing all frame variables in T with ?. The reverse operation, which we call *discrimination*, is defined as follows.

9.1 DEFINITION (Discrimination of a gradual type): Given a gradual type τ, the set $\star(\tau)$ of its *discriminations* is defined as:

$$\star(\tau) \stackrel{\text{def}}{=} \{\, T \in \mathsf{TFrame} \mid T^\dagger = \tau \,\} \,. \qquad \square$$

The definition of materialization, stated formally below, says that τ_2 materializes τ_1 if it can be obtained from τ_1 by first replacing all occurrences of ? with arbitrary variables in FVar and then applying a substitution which replaces those variables with gradual types.

9.2 DEFINITION (Materialization): The *materialization* relation on gradual types $\tau_1 \sqsubseteq \tau_2$ ("τ_2 materializes τ_1") is defined as follows:

$$\tau_1 \sqsubseteq \tau_2 \iff^{\text{def}} \exists T \in \star(\tau_1), \sigma : \mathsf{FVar} \to \mathsf{GType}.\ T\sigma = \tau_2 \,. \qquad \square$$

In the above, $\sigma : \mathsf{FVar} \to \mathsf{GType}$ is a type substitution (i.e., a mapping that is the identity on a cofinite set of variables) from frame variables to gradual types. We use $\mathsf{dom}(\sigma)$ to denote the set of variables for which σ is not the identity (i.e., $\mathsf{dom}(\sigma) = \{X \mid X\sigma \neq X\}$).

It is not difficult to prove that the materialization relation of Definition 9.2 and the one defined by the inductive rules we have given are equivalent, and that they are inverses of the precision relation (Garcia, 2013) and of naive subtyping (Wadler and Findler, 2009).

1 Henglein (1994) defines an equivalent relation for monomorphic types (called "subtyping") but with different rules.

$$\frac{}{x \sqsubseteq x} \qquad \frac{}{c \sqsubseteq c} \qquad \frac{e \sqsubseteq e'}{(\lambda x.\, e) \sqsubseteq (\lambda x.\, e')} \qquad \frac{e \sqsubseteq e'}{(\lambda x\colon \tau.\, e) \sqsubseteq (\lambda x\colon \tau'.\, e')}\ \tau \sqsubseteq \tau' \qquad \frac{e_1 \sqsubseteq e_1' \qquad e_2 \sqsubseteq e_2'}{e_1\ e_2 \sqsubseteq e_1'\ e_2'}$$

$$\frac{e_1 \sqsubseteq e_1' \qquad e_2 \sqsubseteq e_2'}{(e_1, e_2) \sqsubseteq (e_1', e_2')} \qquad \frac{e \sqsubseteq e'}{\pi_i\ e \sqsubseteq \pi_i\ e'} \qquad \frac{e_1 \sqsubseteq e_1' \qquad e_2 \sqsubseteq e_2'}{\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2 \sqsubseteq \mathsf{let}\ \vec{\alpha}\ x = e_1'\ \mathsf{in}\ e_2'}$$

FIGURE 9.2 Lifting of the materialization relation to expressions

9.1.3 Static gradual guarantee

The presence of $[\mathrm{T}_{\sqsubseteq}]$ in $\mathcal{T}_?$ yields the static gradual guarantee property of Siek et al. (2015) for free. We lift the materialization relation to terms as usual by relating type annotations via materialization. The relation is defined by the rules in Figure 9.2.

The static gradual guarantee states that if $\varnothing \vdash e\colon \tau$ and $e' \sqsubseteq e$, then $\varnothing \vdash e'\colon \tau$. Making the annotations in a program less precise preserves its type.

To prove the static gradual guarantee, we show a weakening property. First, we define an order of generality on type schemes that considers instantiation and materialization. Given two type schemes $S_1 = \forall \vec{\alpha}_1.\, \tau_1$ and $S_2 = \forall \vec{\alpha}_2.\, \tau_2$, we write $S_1 \sqsubseteq^{\forall} S_2$ when, for every instance $\tau_2[\vec{t}_2/\vec{\alpha}_2]$ of S_2, there exists an instance $\tau_1[\vec{t}_1/\vec{\alpha}_1]$ such that $\tau_1[\vec{t}_1/\vec{\alpha}_1] \sqsubseteq \tau_2[\vec{t}_2/\vec{\alpha}_2]$. We extend this definition to type environments: when Γ_1 and Γ_2 are two environments with the same domain, we write $\Gamma_1 \sqsubseteq^{\forall} \Gamma_2$ when, for every $x \in \mathsf{dom}(\Gamma_1)$, $\Gamma_1(x) \sqsubseteq^{\forall} \Gamma_2(x)$.

We have the following results.

9.3 LEMMA: Let $S = \forall \vec{\alpha}.\, \tau$. The following hold:

- for every instance $\tau[\vec{t}/\vec{\alpha}]$ of S, $\mathsf{var}(S) \subseteq \mathsf{var}(\tau[\vec{t}/\vec{\alpha}])$;
- there exists an instance $\tau[\vec{t}/\vec{\alpha}]$ of S such that $\mathsf{var}(S) = \mathsf{var}(\tau[\vec{t}/\vec{\alpha}])$. □

Proof: For the first point, just observe that $\mathsf{var}(\tau[\vec{t}/\vec{\alpha}]) \supseteq \mathsf{var}(\tau) \setminus \vec{\alpha} = \mathsf{var}(S)$. For the second, take any instance in which $\vec{t}$ is a vector of closed types. □

9.4 LEMMA: If $\tau_1 \sqsubseteq \tau_2$, then $\mathsf{var}(\tau_1) \subseteq \mathsf{var}(\tau_2)$. □

Proof: Since $\tau_1 \sqsubseteq \tau_2$, we have $T_1\sigma = \tau_2$ with T_1 such that $T_1^{\dagger} = \tau_1$ and with $\sigma\colon \mathsf{FVar} \to \mathsf{GType}$. Since σ only maps frame variables, every type variable $\alpha \in \mathsf{var}(\tau_1)$, which occurs in T_1, must also occur in $T_1\sigma$. □

9.5 LEMMA: If $S_1 \sqsubseteq^{\forall} S_2$, then $\mathsf{var}(S_1) \subseteq \mathsf{var}(S_2)$. If $\Gamma_1 \sqsubseteq^{\forall} \Gamma_2$, then $\mathsf{var}(\Gamma_1) \subseteq \mathsf{var}(\Gamma_2)$. □

Proof: Let $S_1 = \forall \vec{\alpha}_1.\, \tau_1$ and $S_2 = \forall \vec{\alpha}_2.\, \tau_2$ be such that $S_1 \sqsubseteq^{\forall} S_2$. By Lemma 9.3, we can find an instance $\tau_2[\vec{t}_2/\vec{\alpha}_2]$ of S_2 such that $\mathsf{var}(\tau_2[\vec{t}_2/\vec{\alpha}_2]) = \mathsf{var}(S_2)$. By definition of $S_1 \sqsubseteq^{\forall} S_2$, there exists an instance $\tau_1[\vec{t}_1/\vec{\alpha}_1]$ of S_1 such that $\tau_1[\vec{t}_1/\vec{\alpha}_1] \sqsubseteq \tau_2[\vec{t}_2/\vec{\alpha}_2]$. By Lemma 9.3, we have $\mathsf{var}(S_1) \subseteq \mathsf{var}(\tau_1[\vec{t}_1/\vec{\alpha}_1])$. By Lemma 9.4, we have $\mathsf{var}(\tau_1[\vec{t}_1/\vec{\alpha}_1]) \subseteq \mathsf{var}(\tau_2[\vec{t}_2/\vec{\alpha}_2])$. Hence, $\mathsf{var}(S_1) \subseteq \mathsf{var}(S_2)$.

The result on type environments is a straightforward corollary. □

9.6 LEMMA: If $\Gamma_2 \vdash e : \tau$ and $\Gamma_1 \sqsubseteq^{\forall} \Gamma_2$, then $\Gamma_1 \vdash e : \tau$. □

Proof in appendix (p. 251).

Using weakening, we can prove the static gradual guarantee easily.

9.7 PROPOSITION (Static gradual guarantee): If $\varnothing \vdash e : \tau$ and $e' \sqsubseteq e$, then $\varnothing \vdash e' : \tau$. □

Proof: We prove the stronger claim that, for every Γ, e, e', and τ, if $\Gamma \vdash e : \tau$ and $e' \sqsubseteq e$, then $\Gamma \vdash e' : \tau$. The proof is by induction on the typing derivation of $\Gamma \vdash e : \tau$ and by case analysis on the last rule applied. All cases are straightforward except that for $[\mathrm{T}_{\lambda:}]$.

In that case, we have $e = (\lambda x : \tau_1.\, e_1)$, $\tau = \tau_1 \to \tau_2$, and $\Gamma, x : \tau_1 \vdash e_1 : \tau_2$. Since $e' \sqsubseteq e$, we have $e' = (\lambda x : \tau_1'.\, e_1')$ with $\tau_1' \sqsubseteq \tau_1$ and $e_1' \sqsubseteq e_1$. By IH, $\Gamma, x : \tau_1 \vdash e_1' : \tau_2$. By Lemma 9.6, $\Gamma, x : \tau_1' \vdash e_1' : \tau_2$. By $[\mathrm{T}_{\lambda:}]$ we derive that $\Gamma \vdash e' : \tau_1' \to \tau_2$. By $[\mathrm{T}_{\sqsubseteq}]$, we conclude $\Gamma \vdash e' : \tau$. □

9.1.4 Relationship with standard gradual type systems

The type system $\mathcal{T}_?$ is *declarative* in the sense that all auxiliary relations (here materialization) are handled by structural rules (here $[\mathrm{T}_{\sqsubseteq}]$) added to an existing set of logical and identity rules. In a declarative system, every term may have different types and derivations; removing the structural rules corresponds to finding an algorithmic system that for every well-typed term chooses one particular derivation and, thus, one type of the declarative system. This is usually obtained by moving the checks of the auxiliary relations into the elimination rules: this yields a system that is easier to implement but less understandable. This is exactly what current gradual type systems do. It is possible to show that the set of typable terms of our declarative system is the same as the set of typable terms of the existing gradual type systems that use consistency.

Consistency for our types is defined by the following inductive rules.

$$\frac{}{? \sim \tau} \qquad \frac{}{\tau \sim ?} \qquad \frac{}{\alpha \sim \alpha} \qquad \frac{}{b \sim b} \qquad \frac{\tau_1 \sim \tau_1' \quad \tau_2 \sim \tau_2'}{\tau_1 \times \tau_2 \sim \tau_1' \times \tau_2'} \qquad \frac{\tau_1 \sim \tau_1' \quad \tau_2 \sim \tau_2'}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}$$

As remarked by Siek and Vachharajani (2008), the following result holds.

$$\mathsf{GType} \ni \tau ::= ? \mid b \mid \tau \to \tau \qquad \text{gradual types}$$
$$e ::= x \mid c \mid \lambda x\colon \tau.\, e \mid e\ e \qquad \text{source language expressions}$$

$$[\mathrm{T}_x]\ \frac{}{\Gamma \vdash_1 x\colon t}\ \Gamma(x) = t \qquad [\mathrm{T}_c]\ \frac{}{\Gamma \vdash_1 c\colon b_c}$$

$$[\mathrm{T}_{\lambda:}]\ \frac{\Gamma, x\colon \tau' \vdash_1 e\colon \tau}{\Gamma \vdash_1 (\lambda x\colon \tau'.\, e)\colon \tau' \to \tau} \qquad [\mathrm{T}_{\mathrm{app}}]\ \frac{\Gamma \vdash_1 e_1\colon \tau' \to \tau \qquad \Gamma \vdash_1 e_2\colon \tau'}{\Gamma \vdash_1 e_1\ e_2\colon \tau}$$

$$[\mathrm{T}_{\sqsubseteq}]\ \frac{\Gamma \vdash_1 e\colon \tau'}{\Gamma \vdash_1 e\colon \tau}\ \tau' \sqsubseteq \tau$$

FIGURE 9.3 Monomorphic restriction of the implicative fragment of $\mathcal{T}_?$

9.8 PROPOSITION: For every two types τ_1 and τ_2,

$$\tau_1 \sim \tau_2 \iff \exists \tau.\ \tau_1 \sqsubseteq \tau \text{ and } \tau_2 \sqsubseteq \tau\ . \qquad \square$$

Proof in appendix (p. 252).

The relation between our system $\mathcal{T}_?$ and the gradual type system of Siek and Taha (2006) can be stated formally. Let $\vdash_{\mathsf{ST}}$ denote the typing judgment of Siek and Taha (2006). Let $\vdash_1$ denote the monomorphic restriction of the implicative fragment of $\mathcal{T}_?$, that is, our gradual types without type variables and products and the typing rules of the simply typed λ-calculus plus materialization: see Figure 9.3. Then we have the following result.

9.9 PROPOSITION: If $\Gamma \vdash_{\mathsf{ST}} e\colon \tau$, then $\Gamma \vdash_1 e\colon \tau$. Conversely, if $\Gamma \vdash_1 e\colon \tau$, then there exists a type τ' such that $\Gamma \vdash_{\mathsf{ST}} e\colon \tau'$ and $\tau' \sqsubseteq \tau$. $\square$

Proof sketch (full proof in appendix, p. 253): Both implications can be shown by induction on the typing derivation. In the proof that $\Gamma \vdash_{\mathsf{ST}} e\colon \tau$ implies $\Gamma \vdash_1 e\colon \tau$, the interesting case is that for the rule [GApp2] of Siek and Taha (2006):

$$[\mathrm{GApp2}]\ \frac{\Gamma \vdash_{\mathsf{ST}} e_1\colon \tau' \to \tau \qquad \Gamma \vdash_{\mathsf{ST}} e_2\colon \tau_2}{\Gamma \vdash_{\mathsf{ST}} e_1\ e_2\colon \tau}\ \tau_2 \sim \tau'$$

This rule is derivable in $\mathcal{T}_?$. By Proposition 9.8, $\tau_2 \sim \tau'$ implies that there is some τ_3 such that $\tau_2 \sqsubseteq \tau_3$ and $\tau' \sqsubseteq \tau_3$. Then, we have $\Gamma \vdash_1 e_1\colon \tau_3 \to \tau$ and $\Gamma \vdash_1 e_2\colon \tau_3$ by two uses of $[\mathrm{T}_{\sqsubseteq}]$. We apply $[\mathrm{T}_{\mathrm{app}}]$ to conclude. $\square$

The (polymorphic) implicative fragment of $\mathcal{T}_?$ (i.e., $\mathcal{T}_?$ without products), denoted by $\vdash_{\rightarrow}$ and presented in Figure 9.4, is yet another well-known gradual

$$
\begin{array}{rll}
\mathsf{GType} \ni \tau & ::= ? \mid \alpha \mid b \mid \tau \to \tau & \text{gradual types} \\
\mathsf{SType} \ni t & ::= \alpha \mid b \mid \tau \to \tau & \text{static types} \\
e & ::= x \mid c \mid \lambda x.\, e \mid \lambda x\colon \tau.\, e \mid e\ e & \text{source language expressions}
\end{array}
$$

$$
[\mathrm{T}_x]\ \frac{}{\Gamma \vdash_{\rightarrow} x\colon \tau[\vec{t}/\vec{\alpha}]}\ \Gamma(x) = \forall \vec{\alpha}.\, \tau \qquad [\mathrm{T}_c]\ \frac{}{\Gamma \vdash_{\rightarrow} c\colon b_c}
$$

$$
[\mathrm{T}_\lambda]\ \frac{\Gamma, x\colon t \vdash_{\rightarrow} e\colon \tau}{\Gamma \vdash_{\rightarrow} (\lambda x.\, e)\colon t \to \tau} \qquad [\mathrm{T}_{\lambda:}]\ \frac{\Gamma, x\colon \tau' \vdash_{\rightarrow} e\colon \tau}{\Gamma \vdash_{\rightarrow} (\lambda x\colon \tau'.\, e)\colon \tau' \to \tau}
$$

$$
[\mathrm{T}_{\mathrm{app}}]\ \frac{\Gamma \vdash_{\rightarrow} e_1\colon \tau' \to \tau \qquad \Gamma \vdash_{\rightarrow} e_2\colon \tau'}{\Gamma \vdash_{\rightarrow} e_1\ e_2\colon \tau} \qquad [\mathrm{T}_{\sqsubseteq}]\ \frac{\Gamma \vdash_{\rightarrow} e\colon \tau'}{\Gamma \vdash_{\rightarrow} e\colon \tau}\ \tau' \sqsubseteq \tau
$$

FIGURE 9.4 Polymorphic restriction of the implicative fragment of $\mathcal{T}_?$

type system: it coincides with the ITGL type system of Garcia and Cimini (2015), denoted by $\vdash_{\mathsf{GC}}$, as stated by the following result.

9.10 PROPOSITION: If $\Gamma \vdash_{\mathsf{GC}} e\colon \tau$ then $\Gamma \vdash_{\rightarrow} e\colon \tau$. Conversely, if $\Gamma \vdash_{\rightarrow} e\colon \tau$, then there exists a type τ' such that $\Gamma \vdash_{\mathsf{GC}} e\colon \tau'$ and $\tau' \sqsubseteq \tau$. □

Proof: The proof is mostly the same as the proof of Proposition 9.9. The main difference is the presence of the rule for untyped λ-abstractions $[\mathrm{T}_\lambda]$, which is however identical to the rule $[\mathrm{U}\lambda]$ of Garcia and Cimini (2015). □

In other words, the relationship between our new declarative approach and the standard ones that use consistency is analogous to the usual relationship between a declarative type system with subtyping (i.e., one with a subsumption rule) and an algorithmic type system.

9.2 *Cast language*

As customary with gradual typing, the semantics of the gradually typed language is given by translating its well-typed expressions into a *cast language* or *cast calculus*, which we define next. As anticipated, we do not describe the semantics here, but we refer to the appendix for its definition and to Castagna et al. (2019) for the proofs.

9.2.1 Syntax

The syntax of the cast language is defined as follows:

$$
\begin{aligned}
E ::= {} & x \mid c \mid \lambda^{\tau \to \tau} x.\, E \mid E\ E \mid (E, E) \mid \pi_i\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ E \\
& \mid \Lambda \vec{\alpha}.\, E \mid E\ [\vec{t}\,] \mid E\langle \tau \overset{p}{\Rightarrow} \tau \rangle
\end{aligned}
$$

This is an explicitly typed λ-calculus similar to the source language with a few differences and the addition of explicit casts.

There is now just one kind of λ-abstraction, which is annotated with its arrow type (rather than just the parameter type as in $\lambda x\colon \tau.\, e$).[2]

The let construct no longer binds type variables; instead, there are explicit type abstractions $\Lambda\vec{\alpha}.\, E$ and applications $E[\vec{t}]$. For example, the source language expression $\mathsf{let}\ \alpha\ z = \lambda x\colon \alpha.\, \lambda y.\, x\ \mathsf{in}\ z\ 42$, of type $\beta \to \mathsf{Int}$, is translated into the cast calculus as $\mathsf{let}\ z = \Lambda\alpha\beta.\, \lambda^{\alpha\to\beta\to\alpha} x.\, \lambda^{\beta\to\alpha} y.\, x\ \mathsf{in}\ z\ [\mathsf{Int}, \beta]\ 42$. Despite the presence of type abstractions, the cast calculus does not support first-class polymorphism; the syntax of types remains unchanged from Section 9.1.1 and does not include universally quantified types.

Finally, the most important addition to the calculus are explicit casts of the form $E\langle \tau \overset{p}{\Rightarrow} \tau' \rangle$ where, as usual, p ranges over a set of blame labels. Such an expression dynamically checks whether E, of static type τ, produces a value of type τ'; if the cast fails, then the label p is used to blame the cast. These casts are inserted during compilation to perform runtime checks in dynamically typed code: for instance, the function $\lambda x\colon ?.\, x + 1$ will be compiled into $\lambda^{?\to\mathsf{Int}} x.\, x\langle ? \overset{p}{\Rightarrow} \mathsf{Int}\rangle + 1$, which checks at run time whether the function parameter is bound to an integer value (and if not blames the label p). As customary blame labels have a polarity, and we follow the standard convention of using ℓ to range over positive labels and $\bar{\ell}$ for negative ones.

9.2.2 Type system

The typing rules for the cast language are presented in Figure 9.5. Type environments associate variables to type schemes of the form $\forall\vec{\alpha}.\, \tau$ (rule $[\mathrm{T}_x]$) and we use the standard rules for the introduction $[\mathrm{T}_\Lambda]$ and elimination $[\mathrm{T}_{[\,]}]$ of type schemes.

Our typing rules for casts are more precise than the current literature: they capture invariants that are typically captured by a separate "safe-for" relation which is used to establish the *blame theorem* (Tobin-Hochstadt and Felleisen, 2006; Wadler and Findler, 2009). Our casts are well-typed if they go from the type of the casted expression τ' to either a more precise (positive label) or a less precise (negative label) gradual type τ (rules $[\mathrm{T}_{\langle\rangle\sqsubseteq}]$ and $[\mathrm{T}_{\langle\rangle\sqsupseteq}]$, respectively). Blame safety usually involves two subtyping relations, called *positive subtyping* ($<:^+$) and *negative subtyping* ($<:^-$), characterizing respectively casts that cannot yield positive blame and casts that cannot yield negative blame. By the factoring theorem for naive subtyping (Wadler and Findler, 2009), $\tau' \sqsubseteq \tau$ implies $\tau' <:^+ \tau$, so a cast that satisfies rule $[\mathrm{T}_{\langle\rangle\sqsubseteq}]$ is safe for ℓ. Conversely, $\tau \sqsubseteq \tau'$ implies $\tau' <:^- \tau$, so a cast that satisfies rule $[\mathrm{T}_{\langle\rangle\sqsupseteq}]$ is also safe for ℓ.

2 We need to have the arrow type, rather than just the domain, for the operational semantics of the cast language with set-theoretic types (cf. the operator "type" in Appendix B.2).

$$[\mathrm{T}_x]\ \frac{}{\Gamma \vdash x\colon \forall\vec{\alpha}.\,\tau}\ \Gamma(x) = \forall\vec{\alpha}.\,\tau \qquad [\mathrm{T}_c]\ \frac{}{\Gamma \vdash c\colon b_c}$$

$$[\mathrm{T}_\lambda]\ \frac{\Gamma, x\colon \tau' \vdash E\colon \tau}{\Gamma \vdash (\lambda^{\tau'\to\tau} x.\,E)\colon \tau' \to \tau} \qquad [\mathrm{T}_{\mathrm{app}}]\ \frac{\Gamma \vdash E_1\colon \tau' \to \tau \qquad \Gamma \vdash E_2\colon \tau'}{\Gamma \vdash E_1\,E_2\colon \tau}$$

$$[\mathrm{T}_{\mathrm{pair}}]\ \frac{\Gamma \vdash E_1\colon \tau_1 \qquad \Gamma \vdash E_2\colon \tau_2}{\Gamma \vdash (E_1, E_2)\colon \tau_1 \times \tau_2} \qquad [\mathrm{T}_{\mathrm{proj}}]\ \frac{\Gamma \vdash E\colon \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\,E\colon \tau_i}$$

$$[\mathrm{T}_{\mathsf{let}}]\ \frac{\Gamma \vdash E_1\colon \forall\vec{\alpha}.\,\tau_1 \qquad \Gamma, x\colon \forall\vec{\alpha}.\,\tau_1 \vdash E_2\colon \tau}{\Gamma \vdash (\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2)\colon \tau}$$

$$[\mathrm{T}_\Lambda]\ \frac{\Gamma \vdash E\colon \tau}{\Gamma \vdash \Lambda\vec{\alpha}.\,E\colon \forall\vec{\alpha}.\,\tau}\ \vec{\alpha} \sharp \Gamma \qquad [\mathrm{T}_{[\,]}]\ \frac{\Gamma \vdash E\colon \forall\vec{\alpha}.\,\tau}{\Gamma \vdash E\,[\vec{t}]\colon \tau[\vec{t}/\vec{\alpha}]}$$

$$[\mathrm{T}_{\langle\rangle\sqsubseteq}]\ \frac{\Gamma \vdash E\colon \tau'}{\Gamma \vdash E\langle\tau' \overset{\ell}{\Rightarrow} \tau\rangle\colon \tau}\ \tau' \sqsubseteq \tau \qquad [\mathrm{T}_{\langle\rangle\sqsupseteq}]\ \frac{\Gamma \vdash E\colon \tau'}{\Gamma \vdash E\langle\tau' \overset{\bar{\ell}}{\Rightarrow} \tau\rangle\colon \tau}\ \tau \sqsubseteq \tau'$$

FIGURE 9.5 $\mathcal{T}_?^{\langle\rangle}$: Typing rules of the cast language

9.2.3 Semantics

As anticipated, we describe the operational semantics of the cast calculus in appendix. Here we just summarise it briefly. The (strict) semantics is defined as a small-step reduction relation $\hookrightarrow$ by which a cast language expression can reduce to another cast language expression or to a cast error, written $\mathsf{blame}\ p$ to indicate the label that is blamed.

The reduction rules closely follow the presentation of Siek, Thiemann, and Wadler (2015). The reductions for the application of casts to a value use the technique by Wadler and Findler (2009) that consists in checking whether a cast is performed between two types with the same top-level constructor and failing when this is not the case. To do so, we use the notion of *ground type* of Wadler and Findler (2009), albeit employing a different notation that is more convenient when we extend the system to set-theoretic types. A ground type is a type different from ? and whose strict subterms are all ?: for example, $? \to ?$ and Int are ground, but $\mathsf{Int} \to ?$ is not.[3]

The soundness of the cast calculus is proved via progress and subject reduction. These are not proved directly; rather, the results are shown for the cast calculus with set-theoretic types, which is shown to be a conservative extension of this. The same holds for the property of *blame safety*. For reference,

3 This notion of "ground type" is unrelated to the usage of "ground", synonymous with "closed", for a type without type variables. In this Part we always use "closed" for the latter to avoid confusion. Note that α is a ground type, but of course it is not closed.

the properties are the following.

Soundness: For every term E such that $\varnothing \vdash E \colon \forall \vec{\alpha}.\, \tau$, there exists a value V such that $E \hookrightarrow^* V$, or there exists a label p such that $E \hookrightarrow^* \mathsf{blame}\, p$, or E diverges.

Blame safety: For every term E such that $\varnothing \vdash E \colon \forall \vec{\alpha}.\, \tau$ and every blame label ℓ, $E \not\hookrightarrow^* \mathsf{blame}\, \bar{\ell}$.

The statement of blame safety is unlike that of Wadler and Findler (2009) because the typing rules enforce a correspondence between the polarity of the label of a cast and the direction of materialization. That is, we only have casts of the form $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle$ where $\tau' \sqsubseteq \tau$ (i.e., $\tau <:_n \tau'$) for a negative p and $\tau \sqsubseteq \tau'$ (i.e., $\tau' <:_n \tau$) for a positive p. Therefore, only negative labels can cause blame. Since all this information is encoded in the typing rules, blame safety is a corollary of subject reduction and can be stated without resorting to positive and negative subtyping.

9.2.4 Compilation

The final ingredient of the declarative definition of the system is to show how to compile a well-typed expression of the source language into an expression of the cast calculus and prove that compilation preserves types. This result, combined with the soundness of the cast language, implies the soundness of the gradually typed language: a well-typed expression is compiled into an expression that can only either return a value of the same type, return a cast error, or diverge.

Compilation is driven by the derivation of the type for the source language expression. Conceptually, compilation is straightforward: every time the derivation uses the $[\mathrm{T}_\sqsubseteq]$ rule on some sub-expression for a relation $\tau_1 \sqsubseteq \tau_2$, a cast $\langle \tau_1 \overset{\ell}{\Rightarrow} \tau_2 \rangle$ must be added to that sub-expression. Technically, we achieve this by enriching the judgments of typing derivations with a compilation part: $\Gamma \vdash e \rightsquigarrow E \colon \tau$ means that the source language expression e of type τ compiles to the cast language expression E. These judgments are derived by the same rules as those given for the source language in Figure 9.1 to whose judgments we add the compilation part. The modified rules are in Figure 9.6.

The only rules that are modified in a non-trivial way are $[\mathrm{T}_x]$, $[\mathrm{T}_\lambda]$, $[\mathrm{T}_{\mathsf{let}}]$, and $[\mathrm{T}_\sqsubseteq]$. In $[\mathrm{T}_x]$, we compile occurrences of polymorphic variables by adding a type application corresponding to the instantiation. In $[\mathrm{T}_\lambda]$, we explicitly annotate the function with the type deduced by inference. In $[\mathrm{T}_{\mathsf{let}}]$, we introduce a type abstraction for the type variables that are generalized. Finally, the core of compilation is given by the rule $[\mathrm{T}_\sqsubseteq]$, which corresponds to the insertion of an explicit cast (with a positive fresh label ℓ). All the remaining rules are straightforward modifications of the rules in Figure 9.1 insofar as their conclusions simply compose the compiled expressions in the premises.

Compilation is defined for all well-typed expressions and preserves typing.

9.11 PROPOSITION: If $\Gamma \vdash e \colon \tau$, then there is an E such that $\Gamma \vdash e \rightsquigarrow E \colon \tau$. □

$$[\mathrm{T}_x]\ \frac{}{\Gamma \vdash x \rightsquigarrow x\,[\vec{t}] : \tau[\vec{t}/\vec{\alpha}]}\ \Gamma(x) = \forall\vec{\alpha}.\,\tau \qquad [\mathrm{T}_c]\ \frac{}{\Gamma \vdash c \rightsquigarrow c : b_c}$$

$$[\mathrm{T}_\lambda]\ \frac{\Gamma, x : t \vdash e \rightsquigarrow E : \tau}{\Gamma \vdash (\lambda x.\,e) \rightsquigarrow (\lambda^{t\to\tau} x.\,E) : t \to \tau} \qquad [\mathrm{T}_{\lambda:}]\ \frac{\Gamma, x : \tau' \vdash e \rightsquigarrow E : \tau}{\Gamma \vdash (\lambda x : \tau'.\,e) \rightsquigarrow (\lambda^{\tau'\to\tau} x.\,E) : \tau' \to \tau}$$

$$[\mathrm{T}_{\mathrm{app}}]\ \frac{\Gamma \vdash e_1 \rightsquigarrow E_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 \rightsquigarrow E_2 : \tau'}{\Gamma \vdash e_1\ e_2 \rightsquigarrow E_1\ E_2 : \tau}$$

$$[\mathrm{T}_{\mathrm{pair}}]\ \frac{\Gamma \vdash e_1 \rightsquigarrow E_1 : \tau_1 \qquad \Gamma \vdash e_2 \rightsquigarrow E_2 : \tau_2}{\Gamma \vdash (e_1, e_2) \rightsquigarrow (E_1, E_2) : \tau_1 \times \tau_2} \qquad [\mathrm{T}_{\mathrm{proj}}]\ \frac{\Gamma \vdash e \rightsquigarrow E : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\ e \rightsquigarrow \pi_i\ E : \tau_i}$$

$$[\mathrm{T}_{\mathsf{let}}]\ \frac{\Gamma \vdash e_1 \rightsquigarrow E_1 : \tau_1 \qquad \Gamma, x : \forall\vec{\alpha},\vec{\beta}.\,\tau_1 \vdash e_2 \rightsquigarrow E_2 : \tau}{\Gamma \vdash (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2) \rightsquigarrow (\mathsf{let}\ x = \Lambda\vec{\alpha},\vec{\beta}.\,E_1\ \mathsf{in}\ E_2) : \tau}\ \begin{cases} \vec{\alpha},\vec{\beta} \sharp \Gamma \\ \vec{\beta} \sharp e_1 \end{cases}$$

$$[\mathrm{T}_\sqsubseteq]\ \frac{\Gamma \vdash e \rightsquigarrow E : \tau'}{\Gamma \vdash e \rightsquigarrow E\langle \tau' \overset{\ell}{\Rightarrow} \tau\rangle : \tau}\ \tau' \sqsubseteq \tau$$

FIGURE 9.6 $\mathcal{T}_?^{\rightsquigarrow}$: Compilation from the source language to the cast language

Proof: By induction on the derivation of $\Gamma \vdash e : \tau$. □

9.12 PROPOSITION: If $\Gamma \vdash e \rightsquigarrow E : \tau$, then $\Gamma \vdash e : \tau$ and $\Gamma \vdash E : \tau$. □

Proof: By induction on the derivation of $\Gamma \vdash e \rightsquigarrow E : \tau$ and by case analysis on last rule applied. Showing $\Gamma \vdash e : \tau$ is trivial.

Showing $\Gamma \vdash E : \tau$ is also straightforward. If the last rule is $[\mathrm{T}_x]$, we use $[\mathrm{T}_x]$ and $[\mathrm{T}_{[\,]}]$. If the last rule is $[\mathrm{T}_c]$, $[\mathrm{T}_{\mathrm{app}}]$, $[\mathrm{T}_{\mathrm{pair}}]$, or $[\mathrm{T}_{\mathrm{proj}}]$, we use the same rule. If is is $[\mathrm{T}_\lambda]$ or $[\mathrm{T}_{\lambda:}]$, we use $[\mathrm{T}_\lambda]$. If it is $[\mathrm{T}_\sqsubseteq]$, we use $[\mathrm{T}_{\langle\rangle\sqsubseteq}]$.

Finally, if the last rule is $[\mathrm{T}_{\mathsf{let}}]$, from the premise $\Gamma \vdash e_1 \rightsquigarrow E_1 : \tau_1$ we get, by IH, $\Gamma \vdash E_1 : \tau_1$. Then (since $\vec{\alpha},\vec{\beta} \sharp \Gamma$) we get $\Gamma \vdash \Lambda\vec{\alpha},\vec{\beta}.\,E_1 : \forall\vec{\alpha},\vec{\beta}.\,\tau_1$ by $[\mathrm{T}_\Lambda]$. From the premise $\Gamma, x : \forall\vec{\alpha},\vec{\beta}.\,\tau_1 \vdash e_2 \rightsquigarrow E_2 : \tau$ we get, by IH, $\Gamma, x : \forall\vec{\alpha},\vec{\beta}.\,\tau_1 \vdash E_2 : \tau$. We apply $[\mathrm{T}_{\mathsf{let}}]$ to conclude. □

9.13 COROLLARY: If $\Gamma \vdash e : \tau$, then there exists an E such that $\Gamma \vdash e \rightsquigarrow E : \tau$ and $\Gamma \vdash E : \tau$. □

Proof: Corollary of Propositions 9.11 and 9.12. □

9.3 Type inference

In this section we show how to decide whether a given term of the source language is well typed or not: we define a type inference algorithm that is

sound and complete with respect to the system of Section 9.1.2. The algorithm is mostly based on the work of Pottier and Rémy (2005) and of Castagna, Petrucciani, and Nguyễn (2016), adapted for gradual typing. Our algorithm differs from that of Garcia and Cimini (2015) in that ours literally reduces the inference problem to unification. To infer the type of an expression, we generate constraints that specify the conditions that must hold for the expression to be well typed; then, we solve these constraints via unification to obtain a solution (a type substitution).

Our presentation proceeds as follows. We first introduce *type constraints* (Section 9.3.1) and show how to solve sets of type constraints using standard unification (Section 9.3.2). Then we show how to generate constraints for a given expression (Section 9.3.3). To keep constraint generation separated from solving, generation uses more complex *structured constraints* (this is essentially due to the presence of let-polymorphism) which are then solved by simplifying them into the simpler type constraints (Section 9.3.4). Finally, we present our results of soundness and completeness of type inference.

As compared to the work in Chapters 4 and 5, the form of structured constraints we use is very similar. However, instead of defining the reformulated type system and describing separate notions of constraint satisfaction and simplification, we work directly on the original type system and only describe constraint simplification. The approach of Chapters 4 and 5 is meant for subtyping and cannot be used as is for a Hindley-Milner system.

9.3.1 Type constraints and solutions

A *type constraint* has either the form $(t_1 \mathrel{\dot{\leq}} t_2)$ or the form $(\tau \mathrel{\dot{\sqsubseteq}} \alpha)$; we describe their meaning below. *Type-constraint sets* (ranged over by the metavariable D) are finite sets of type constraints.

We write $\mathsf{var}(D)$ for the set of type variables appearing in the type constraints in D. We write $\mathsf{var}_{\dot{\sqsubseteq}}(D)$ for the set of type variables appearing in the gradual types in materialization constraints in D: that is, $\mathsf{var}_{\dot{\sqsubseteq}}(D) = \bigcup_{(\tau \mathrel{\dot{\sqsubseteq}} \alpha) \in D} \mathsf{var}(\tau)$. When $\overline{\alpha} \subseteq \mathsf{TVar}$ is a set of type variables and σ is a type substitution, we define the application $\overline{\alpha}\sigma$ of σ to $\overline{\alpha}$ to be the set of type variables $\bigcup_{\alpha \in \overline{\alpha}} \mathsf{var}(\alpha\sigma)$.

A type substitution $\sigma : \mathsf{TVar} \to \mathsf{GType}$ is a *solution* of a type-constraint set D (with respect to a finite set $\Delta \subseteq \mathsf{TVar}$), written $\sigma \Vdash_\Delta D$, if:

- for every $(t_1 \mathrel{\dot{\leq}} t_2) \in D$, we have $t_1\sigma = t_2\sigma$;
- for every $(\tau \mathrel{\dot{\sqsubseteq}} \alpha) \in D$, we have $\tau\sigma \sqsubseteq \alpha\sigma$ and, for all $\beta \in \mathsf{var}(\tau)$, $\beta\sigma$ is a static type;
- $\mathsf{dom}(\sigma) \cap \Delta = \emptyset$.

A subtyping constraint $(t_1 \mathrel{\dot{\leq}} t_2)$ forces the substitution to unify t_1 and t_2. We use $\mathrel{\dot{\leq}}$ instead of, say $\doteq$, to have uniform syntax with the later section on subtyping (Section 9.4).

A materialization constraint $(\tau \mathrel{\dot{\sqsubseteq}} \alpha)$ imposes two distinct requirements: the solution must make α a materialization of τ and must map all variables in τ to static types. These two conditions might be separated but in practice

they must always be imposed together, and their combination simplifies the description of constraint solving. Note that the constraint $(\alpha \dot{\sqsubseteq} \alpha)$ forces $\alpha\sigma$ to be static (since the other requirement, $\alpha\sigma \sqsubseteq \alpha\sigma$, is trivial).

Finally, the set Δ is used to force the solution *not* to instantiate certain type variables.

9.3.2 Type-constraint solving

We solve a type-constraint set in three steps: we convert the type constraints to unification constraints between type frames (by changing every occurrence of ? into a different frame variable); then we compute a unifier; finally, we convert the unifier into a solution (by renaming some variables and then changing frame variables back to ?).

We define this process as an algorithm $\mathsf{solve}_{(\cdot)}(\cdot)$ which, given a type-constraint set D and a finite set $\Delta \subseteq \mathsf{TVar}$, computes a set of type substitutions $\mathsf{solve}_\Delta(D)$. This set is either empty, indicating failure, or a singleton set containing the solution (which is unique up to variable renaming).[4]

We do not describe a unification algorithm explicitly; rather, we rely on properties satisfied by standard implementations (e.g., that by Martelli and Montanari (1982)). We use unification on type frames: its input is a finite set $\overline{T^1 \doteq T^2}$ of equality constraints of the form $T^1 \doteq T^2$. We also include as input a finite set $\Delta \subseteq \mathsf{TVar}$ that specifies the variables that unification must *not* instantiate (i.e., that should be treated as constants). We write $\mathsf{unify}_\Delta(\overline{T^1 \doteq T^2})$ for the result of the algorithm, which is either fail or a type substitution $\sigma : \mathsf{Var} \longrightarrow \mathsf{TFrame}$.

We assume that unify satisfies standard properties of soundness and completeness, and that it computes idempotent substitutions. In particular, we assume that the following holds.

- If $\mathsf{unify}_\Delta(\overline{T^1 \doteq T^2}) = \sigma$, then:
  - $\mathsf{dom}(\sigma) \subseteq \mathsf{var}(\overline{T^1 \doteq T^2}) \setminus \Delta$;
  - $\mathsf{var}(\sigma) \subseteq \mathsf{var}(\overline{T^1 \doteq T^2}) \setminus \mathsf{dom}(\sigma)$;
  - for every $(T^1 \doteq T^2) \in \overline{T^1 \doteq T^2}$, we have $T^1\sigma = T^2\sigma$.
- If σ' is a unifier for $\overline{T^1 \doteq T^2}$ and $\mathsf{dom}(\sigma') \cap \Delta = \varnothing$, then there exists σ such that $\mathsf{unify}_\Delta(\overline{T^1 \doteq T^2}) = \sigma$ and $\sigma' = \sigma' \circ \sigma$.

(As in Section 2.2.1, we use $\mathsf{var}(\sigma)$ for the set of variables appearing in the type in the range of σ: that is, $\mathsf{var}(\sigma) = \bigcup_{A \in \mathsf{dom}(\sigma)} \mathsf{var}(A\sigma)$, where A ranges over both type and frame variables.)

Unification is the main ingredient of our type-constraint solving algorithm, but we need some extra steps to handle materialization constraints.

Let D be of the form $\{ (t_i^1 \mathrel{\dot{\leq}} t_i^2) \mid i \in I \} \cup \{ (\tau_j \dot{\sqsubseteq} \alpha_j) \mid j \in J \}$: then $\mathsf{solve}_\Delta(D)$ is defined as follows.

4 We use a set because, in the extension with subtyping, constraint solving can produce multiple incomparable solutions (it relies on tallying, described in Section 4.3.1).

1. Let $\overline{T^1 \doteq T^2}$ be $\{ (t_i^1 \doteq t_i^2) \mid i \in I \} \cup \{ (T_j \doteq \alpha_j) \mid j \in J \}$ where the T_j are chosen to ensure:
   a. for every $j \in J$, $T_j^\dagger = \tau_j$;
   b. every frame variable X occurs in at most one of the T_j, at most once.
2. Compute $\mathsf{unify}_\Delta(\overline{T^1 \doteq T^2})$:
   a. if $\mathsf{unify}_\Delta(\overline{T^1 \doteq T^2}) = \mathsf{fail}$, return $\emptyset$;
   b. if $\mathsf{unify}_\Delta(\overline{T^1 \doteq T^2}) = \sigma_0$, return $\{(\sigma_0' \circ \sigma_0)^\dagger|_{\mathsf{TVar}}\}$ where:
      i. $\sigma_0' = [\vec{\alpha}'/\vec{X}] \cup [\vec{X}'/\vec{\alpha}]$
      ii. $\vec{X} = \mathsf{FVar} \cap \mathsf{var}_{\dot{\sqsubseteq}}(D)\sigma_0$
      iii. $\vec{\alpha} = \mathsf{var}(D) \setminus (\Delta \cup \mathsf{dom}(\sigma_0) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)\sigma_0)$
      iv. $\vec{\alpha}'$ and $\vec{X}'$ are vectors of fresh variables

In step 1, we convert D to a set of type frame equality constraints. To do so, we convert all gradual types in materialization constraints by replacing each occurrence of ? with a different frame variable. In step 2, we compute a unifier for these constraints. If a unifier σ_0 exists (step 2b), we use it to build our solution: however, we need a post-processing step to ensure that α and X variables are treated correctly. For example, a unifier could map α to X when $(\alpha \dot{\sqsubseteq} \alpha) \in D$: then, converting type frames back to gradual types would mean mapping α to ?, which is not a solution because α is mapped to a gradual type, but a static type is required. Therefore, to obtain the result we first compose σ_0 with a renaming substitution σ_0'; then, we apply $\dagger$ to change type frames back to gradual types, and we restrict the domain to TVar. The renaming introduces fresh variables to replace some frame variables with type variables ($[\vec{\alpha}'/\vec{X}]$) and some type variables with frame variables ($[\vec{X}'/\vec{\alpha}]$). It has two purposes. One is to ensure that the variables in $\mathsf{var}_{\dot{\sqsubseteq}}(D)$ are mapped to static types, which we need for $\sigma \Vdash_\Delta D$ to hold. The other is to have the substitution introduce as few type variables as possible.

The following soundness property holds.

9.14 PROPOSITION (Soundness of solve): If $\sigma \in \mathsf{solve}_\Delta(D)$, then the following hold:

- $\sigma \Vdash_\Delta D$;
- $\mathsf{dom}(\sigma) \subseteq \mathsf{var}(D)$;
- $\mathsf{var}(D)\sigma \subseteq \mathsf{var}_{\dot{\sqsubseteq}}(D)\sigma \cup \Delta$. □

Proof in appendix (p. 254).

The last property states that a solution σ returned by solve introduces as few variables as possible. In particular, the variables it introduces in D are only those in Δ and those that appear in the solution of variables in $\mathsf{var}_{\dot{\sqsubseteq}}(D)$ (whose solution must be static). To ensure this, we perform the substitution $[\vec{X}'/\vec{\alpha}]$. This avoids useless materializations of ? to type variables (and thus the insertion of useless casts at compilation): for example, it ensures that, in

let $y = x$ in e, if x has type ?, then y is given type ? too. In the declarative system, it can be typed also as $\forall\alpha.\,\alpha$, but then the compiled expression has a cast: let $y = \Lambda\alpha.\,x\langle ? \overset{\ell}{\Rightarrow} \alpha\rangle$ in E. We prefer the compilation without this cast, which is why we replace as many type variables as possible with ?.

We prove also a result of completeness. It relies on the following lemma.

9.15 LEMMA: Let $\sigma\colon \mathsf{TVar} \to \mathsf{GType}$ and $\sigma'\colon \mathsf{Var} \to \mathsf{TFrame}$ be two type substitutions such that $\forall\alpha \in \mathsf{TVar}.\,(\alpha\sigma')^\dagger = \alpha\sigma$. For every T, we have $T^\dagger\sigma \sqsubseteq (T\sigma')^\dagger$. □

Proof in appendix (p. 255).

9.16 PROPOSITION (Completeness of solve): If $\sigma \Vdash_\Delta D$, then there exist two type substitutions σ' and σ'' such that:

- $\sigma' \in \mathsf{solve}_\Delta(D)$;
- $\mathsf{dom}(\sigma'') \subseteq \mathsf{var}(\sigma') \setminus \mathsf{var}(D)$;
- for every α, $\alpha\sigma'(\sigma \cup \sigma'') \sqsubseteq \alpha(\sigma \cup \sigma'')$;
- for every α such that $\alpha\sigma'$ is static, $\alpha\sigma'(\sigma \cup \sigma'') = \alpha(\sigma \cup \sigma'')$. □

Proof in appendix (p. 256).

As compared to a standard statement of completeness for unification, instead of having $\alpha\sigma'(\sigma \cup \sigma'') = \alpha(\sigma \cup \sigma'')$ for every α, we have a weaker condition that allows for materialization, except when $\alpha\sigma'$ is static.

9.3.3 Structured constraints and constraint generation

As discussed in Section 4.2, without let-polymorphism we can define type inference using type constraints alone; with let-polymorphism, instead, we would need either to mix constraint generation and solving or to copy constraints for let-bound expressions multiple times. To avoid this, we introduce structured constraints like those in Section 4.2.

A *structured constraint* is a term generated by the following grammar:

$$C ::= (t \dot{\leq} t) \mid (\tau \dot{\sqsubseteq} \alpha) \mid (x \dot{\sqsubseteq} \alpha) \mid C \wedge C \mid \exists\vec{\alpha}.\,C$$
$$\mid \mathsf{def}\ x\colon \tau\ \mathsf{in}\ C \mid \mathsf{let}\ x\colon \forall\vec{\alpha}; \alpha[C]^{\vec{\alpha}}.\,\alpha\ \mathsf{in}\ C$$

Structured constraints are considered equal up to α-renaming of bound variables. In $\exists\vec{\alpha}.\,C$, the $\vec{\alpha}$ variables are bound in C. In $\mathsf{let}\ x\colon \forall\vec{\alpha}; \alpha[C_1]^{\vec{\alpha}'}.\,\alpha\ \mathsf{in}\ C_2$, α and the $\vec{\alpha}$ variables are bound in C_1.

Structured constraints include type constraints and five other forms. A constraint $(x \dot{\sqsubseteq} \alpha)$ asks that the type scheme for x has an instance that materializes to the solution of α. Existential constraints $\exists\vec{\alpha}.\,C$ bind the type variables $\vec{\alpha}$ occurring in C; this simplifies freshness conditions, as in Chapter 4. $C \wedge C$ is simply the conjunction of two constraints; in Chapter 4 we included disjunction as well, but we do not need it here. The def and let constraint forms are generated to type λ-abstractions and let-expressions.

$$
\begin{aligned}
\langle\!\langle x\colon t\rangle\!\rangle &= \exists\alpha.\,(x \mathrel{\dot\sqsubseteq} \alpha) \wedge (\alpha \mathrel{\dot\leq} t) && \alpha \sharp t\\
\langle\!\langle c\colon t\rangle\!\rangle &= (b_c \mathrel{\dot\leq} t)\\
\langle\!\langle (\lambda x.\,e)\colon t\rangle\!\rangle &= \exists\alpha_1,\alpha_2.\,(\mathsf{def}\ x\colon \alpha_1\ \mathsf{in}\ \langle\!\langle e\colon \alpha_2\rangle\!\rangle) \wedge (\alpha_1 \mathrel{\dot\sqsubseteq} \alpha_1) \wedge (\alpha_1{\to}\alpha_2 \mathrel{\dot\leq} t) && \alpha_1,\alpha_2 \sharp t, e\\
\langle\!\langle (\lambda x\colon \tau.\,e)\colon t\rangle\!\rangle &= \exists\alpha_1,\alpha_2.\,(\mathsf{def}\ x\colon \tau\ \mathsf{in}\ \langle\!\langle e\colon \alpha_2\rangle\!\rangle) \wedge (\tau \mathrel{\dot\sqsubseteq} \alpha_1) \wedge (\alpha_1{\to}\alpha_2 \mathrel{\dot\leq} t) && \alpha_1,\alpha_2 \sharp t, \tau, e\\
\langle\!\langle e_1\ e_2\colon t\rangle\!\rangle &= \exists\alpha.\,\langle\!\langle e_1\colon \alpha\to t\rangle\!\rangle \wedge \langle\!\langle e_2\colon \alpha\rangle\!\rangle && \alpha \sharp t, e_1, e_2\\
\langle\!\langle (e_1,e_2)\colon t\rangle\!\rangle &= \exists\alpha_1,\alpha_2.\,\langle\!\langle e_1\colon \alpha_1\rangle\!\rangle \wedge \langle\!\langle e_2\colon \alpha_2\rangle\!\rangle \wedge (\alpha_1\times\alpha_2 \mathrel{\dot\leq} t) && \alpha_1,\alpha_2 \sharp t, e_1, e_2\\
\langle\!\langle \pi_i\ e\colon t\rangle\!\rangle &= \exists\alpha_1,\alpha_2.\,\langle\!\langle e\colon \alpha_1\times\alpha_2\rangle\!\rangle \wedge (\alpha_i \mathrel{\dot\leq} t) && \alpha_1,\alpha_2 \sharp t, e\\
\langle\!\langle \mathsf{let}\ \vec\alpha\ x = e_1\ \mathsf{in}\ e_2\colon t\rangle\!\rangle &= \mathsf{let}\ x\colon \forall\vec\alpha;\alpha[\langle\!\langle e_1\colon \alpha\rangle\!\rangle]^{\mathsf{var}(e_1)\setminus\vec\alpha}.\,\alpha\ \mathsf{in}\ \langle\!\langle e_2\colon t\rangle\!\rangle && \alpha \sharp \vec\alpha, e_1
\end{aligned}
$$

FIGURE 9.7 Constraint generation

Figure 9.7 defines a function $\langle\!\langle (\cdot)\colon (\cdot)\rangle\!\rangle$ such that, for every expression e and every static type t, $\langle\!\langle e\colon t\rangle\!\rangle$ is a structured constraint that expresses the conditions that must hold for e to have type $t\sigma$ for some substitution σ.

We point out some peculiarities of the rules. For variables, we generate a constraint combining materialization and subtyping. This allows us to use the form $(x \mathrel{\dot\sqsubseteq} \alpha)$ instead of $(x \mathrel{\dot\sqsubseteq} t)$; more importantly, it means that the same definition for constraint generation can be reused when we add subtyping. For a λ-abstraction, constraint generation wraps the constraint for the body in a def constraint to introduce the type of the parameter. In the absence of annotations, the constraint $(\alpha_1 \mathrel{\dot\sqsubseteq} \alpha_1)$ is used to ensure that the parameter will have a static type. For annotated functions, the constraint $(\tau \mathrel{\dot\sqsubseteq} \alpha_1)$ allows the domain of the function to be materialized. This is needed, for example, to obtain solvable constraints for the abstraction $(\lambda x\colon ?.\,x)$ in a context expecting $\mathsf{Int}\to\mathsf{Int}$. For let, we build a let constraint including the constraints of the two expressions and recording the variables that *must* be generalized ($\vec\alpha$) and those that must *not* be ($\mathsf{var}(e_1)\setminus\vec\alpha$).[5] In all rules, the side conditions force the choice of fresh variables.

9.3.4 Constraint solving

In Chapter 4, we gave both a declarative definition of constraint satisfaction and a constraint solving algorithm. Here, we define directly constraint solving. We use a *constraint simplification* system, defined in Figure 9.8, to convert a structured constraint to a type-constraint set; then, we compute a solution using the algorithm solve of Section 9.3.2. Because of let-polymorphism, constraint simplification also uses type-constraint solving internally to compute partial solutions. Constraint simplification is similar to that of Chapter 4, but there are differences in the treatment of type environments (since we use a single type environment Γ instead of distinguishing between λ- and let-environments).

5 We include the latter for convenience: actually, they can be recomputed from the rest since $\mathsf{var}(e_1) = \mathsf{var}(\langle\!\langle e_1\colon \alpha\rangle\!\rangle)\setminus\{\alpha\}$.

$$\frac{}{\Gamma;\Delta \vdash (t_1 \mathrel{\dot\leq} t_2) \rightsquigarrow \{t_1 \mathrel{\dot\leq} t_2\} \mid \varnothing} \qquad \frac{}{\Gamma;\Delta \vdash (\tau \mathrel{\dot\sqsubseteq} \alpha) \rightsquigarrow \{\tau \mathrel{\dot\sqsubseteq} \alpha\} \mid \varnothing}$$

$$\frac{}{\Gamma;\Delta \vdash (x \mathrel{\dot\sqsubseteq} \alpha) \rightsquigarrow \{\tau[\vec\beta/\vec\alpha] \mathrel{\dot\sqsubseteq} \alpha\} \mid \vec\beta} \begin{cases} \Gamma(x) = \forall\vec\alpha.\,\tau \\ \vec\beta \sharp \Gamma \end{cases} \qquad \frac{(\Gamma, x\colon \tau);\Delta \vdash C \rightsquigarrow D \mid \vec\alpha}{\Gamma;\Delta \vdash \mathsf{def}\, x\colon \tau\, \mathsf{in}\, C \rightsquigarrow D \mid \vec\alpha}$$

$$\frac{\Gamma;\Delta \vdash C \rightsquigarrow D \mid \vec\alpha'}{\Gamma;\Delta \vdash (\exists\vec\alpha.\, C) \rightsquigarrow D \mid \vec\alpha' \cup \vec\alpha}\ \vec\alpha \sharp \Gamma, \vec\alpha' \qquad \frac{\Gamma;\Delta \vdash C_1 \rightsquigarrow D_1 \mid \vec\alpha_1 \qquad \Gamma;\Delta \vdash C_2 \rightsquigarrow D_2 \mid \vec\alpha_2}{\Gamma;\Delta \vdash C_1 \wedge C_2 \rightsquigarrow D_1 \cup D_2 \mid \vec\alpha_1 \cup \vec\alpha_2}\ \vec\alpha_1 \sharp \vec\alpha_2$$

$$\frac{\begin{array}{c}\Gamma;\Delta \cup \vec\alpha \vdash C_1 \rightsquigarrow D_1 \mid \vec\alpha_1 \\ (\Gamma, x\colon \forall\vec\alpha,\vec\beta.\,\alpha\sigma_1);\Delta \vdash C_2 \rightsquigarrow D_2 \mid \vec\alpha_2\end{array}}{\Gamma;\Delta \vdash \mathsf{let}\, x\colon \forall\vec\alpha;\alpha[C_1]^{\vec\alpha'}.\,\alpha\ \mathsf{in}\ C_2 \rightsquigarrow D_2 \cup \mathsf{equiv}(\sigma_1, D_1) \mid \vec\alpha_3} \begin{cases} \sigma_1 \in \mathsf{solve}_{\Delta\cup\vec\alpha}(D_1) \\ \vec\alpha \sharp \Gamma\sigma_1 \\ \vec\beta = \mathsf{var}(\alpha\sigma_1) \setminus (\mathsf{var}(\Gamma\sigma_1) \cup \vec\alpha \cup \vec\alpha') \\ \sharp \left\{\{\alpha\}, \vec\alpha, \vec\alpha_1, \vec\alpha_2, (\mathsf{var}(\sigma_1) \setminus \mathsf{var}(D_1))\right\} \\ \alpha, \vec\alpha \sharp \Gamma, \Delta \\ \vec\alpha_3 = \{\alpha\} \cup \vec\alpha \cup \vec\alpha_1 \cup \vec\alpha_2 \\ \qquad\qquad \cup (\mathsf{var}(\sigma_1) \setminus \mathsf{var}(D_1)) \end{cases}$$

$$\text{where}\ \ \mathsf{equiv}(\sigma, D) \stackrel{\text{def}}{=} \left\{ (\alpha \mathrel{\dot\sqsubseteq} \alpha) \,\middle|\, \alpha \in \mathsf{var}_{\mathrel{\dot\sqsubseteq}}(D) \cup \mathsf{var}(D)\sigma \right\} \cup \bigcup_{\substack{\alpha \in \mathsf{dom}(\sigma) \\ \alpha\sigma\ \text{static}}} \left\{(\alpha \mathrel{\dot\leq} \alpha\sigma), (\alpha\sigma \mathrel{\dot\leq} \alpha)\right\}$$

FIGURE 9.8 $C_?^{\text{sim}}$: Constraint simplification rules

Constraint simplification is a relation $\Gamma;\Delta \vdash C \rightsquigarrow D \mid \vec\alpha$. Γ is a type environment used to assign types to the variables in constraints of the form $(x \mathrel{\dot\sqsubseteq} \alpha)$. Δ is a finite subset of TVar used to record variables that must not be instantiated. When simplifying constraints for a whole program, we take Γ to be empty and Δ to be the set of free type variables in the program (presumably empty as well). Finally, C is the constraint to be simplified, D the result of simplification, and $\vec\alpha$ are the fresh variables introduced during the process. We will often omit $\vec\alpha$ and write $\Gamma;\Delta \vdash C \rightsquigarrow D$ when we are not interested in keeping track of these variables (in particular, in the proof of soundness).

The rules are syntax-directed and deterministic (modulo the choice of fresh variables). Subtyping and materialization constraints are left unchanged. Variable constraints ($x \mathrel{\dot\sqsubseteq} \alpha$) are converted to materialization constraints by replacing x with a fresh instance of its type scheme. To simplify a def constraint, we update the environment and simplify the inner constraint. For $\exists\vec\alpha.\, C$, we simplify C after performing α-renaming, if needed, to ensure that $\vec\alpha$ is fresh. To simplify $C_1 \wedge C_2$, we simplify C_1 and C_2 and take the union of the resulting sets.

Finally, the rule for let constraints is of course the most complicated. To simplify a constraint $\mathsf{let}\, x\colon \forall\vec\alpha;\alpha[C_1]^{\vec\alpha'}.\,\alpha\ \mathsf{in}\ C_2$, we perform five steps:

1. we simplify the constraint C_1 to obtain a set D_1;
2. we apply the solve algorithm to D_1 to obtain a solution σ_1, if one exists;

3. we compute the type scheme for x by generalizing the type given by the solution;
4. we simplify the constraint C_2 in the expanded environment to obtain a set D_2;
5. finally, we add to D_2 the set $\mathsf{equiv}(\sigma_1, D_1)$, whose purpose is to constrain the solution to be an instantiation of σ_1 and to yield static types where needed.

In steps 1 and 2, we add $\vec{\alpha}$ to Δ to ensure that the $\vec{\alpha}$ variables are not instantiated while solving C_1, otherwise we could not generalize them later. The type $\alpha\sigma_1$ for x is generalized by quantifying over the $\vec{\alpha}$ variables (checking that they are not introduced in the environment by σ_1) as well as over $\vec{\beta}$, which contains all variables in $\alpha\sigma_1$ that do not appear in any of $\Gamma\sigma_1$, $\vec{\alpha}$, or $\vec{\alpha}'$. Recall that we record in $\vec{\alpha}'$ the variables that cannot be generalized (typically because they appeared in the expression but not in the decoration of the let construct).

We use the set $\mathsf{equiv}(\sigma_1, D_1)$ to constrain a solution σ to adhere to σ_1 in two ways. First, σ must map to static types all variables in $\mathsf{var}_{\doteq}(D_1)$ (which σ_1 had to map to static types) and all variables introduced by σ_1. Also, σ must satisfy $\alpha\sigma_1\sigma = \alpha\sigma$ whenever $\alpha\sigma_1$ is a static type. To ensure the latter, we add the two subtyping constraints $(\alpha \doteq \alpha\sigma_1)$ and $(\alpha\sigma_1 \doteq \alpha)$. Adding both is redundant here (both require equality), but they are needed when we add subtyping.

COMPILATION: The results of type inference can also be used for compilation. When e is an expression, $\mathcal{D}$ is a derivation of $\Gamma; \Delta \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$, and $\sigma \Vdash_\Delta D$, we can compute a cast language expression $\{\!|e|\!\}^{\mathcal{D}}_{\sigma}$. Figure 9.9 defines this compilation algorithm. It is defined by induction on e. For each case, we deconstruct the derivation $\mathcal{D}$ to obtain the sub-derivations used to compile the sub-expressions of e. We write the derivation $\mathcal{D}$ in a compressed form where we collapse applications of the rules for definition, existential, and conjunctive constraints. We write $\mathcal{D} :: \Gamma; \Delta \vdash C \rightsquigarrow D$ to denote a derivation of $\Gamma; \Delta \vdash C \rightsquigarrow D$ that we name $\mathcal{D}$. The definition is lengthy, but straightforward: to compile a variable, we insert the appropriate type application and cast; to compile other expressions, we just compile their sub-expressions; annotated λ-abstractions require a cast. The compilation of let constructs is a bit more involved because there are two different type substitutions to consider: σ and the intermediate solution σ_1; to compose them, we use another substitution ρ to ensure that they are distinct from the variables introduced by σ.

PROPERTIES OF TYPE INFERENCE: This concludes our description of type inference. The remainder of this section presents the proofs of soundness and completeness. The statements we will obtain are the following.

Soundness Let $\mathcal{D}$ be a derivation of $\Gamma; \mathsf{var}(e) \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$. Let σ be a type substitution such that $\sigma \Vdash_{\mathsf{var}(e)} D$. Then, we have $\Gamma\sigma \vdash e \rightsquigarrow\!\!\!\!\rightsquigarrow \{\!|e|\!\}^{\mathcal{D}}_{\sigma} : t\sigma$.

Completeness If $\Gamma \vdash e : \tau$, then, for every fresh type variable α, there exist D and σ such that $\Gamma; \mathsf{var}(e) \vdash \langle\!\langle e : \alpha \rangle\!\rangle \rightsquigarrow D$ and $[\tau/\alpha] \cup \sigma \Vdash_{\mathsf{var}(e)} D$.

$$\{\!|x|\!\}^{\mathcal{D}}_{\sigma} = x\,[\vec{\beta}\sigma]\langle\tau[\vec{\beta}/\vec{\alpha}]\sigma \overset{\ell}{\Rightarrow} \alpha\sigma\rangle$$
with ℓ fresh
where $\mathcal{D} = \dfrac{}{\Gamma;\Delta \vdash \langle\!\langle x\colon t\rangle\!\rangle \rightsquigarrow \{(\tau[\vec{\beta}/\vec{\alpha}] \mathrel{\dot{\sqsubseteq}} \alpha), (\alpha \mathrel{\dot{\leq}} t)\}}$

$$\{\!|c|\!\}^{\mathcal{D}}_{\sigma} = c$$

$$\{\!|\lambda x.\, e|\!\}^{\mathcal{D}}_{\sigma} = \lambda^{(\alpha_1\to\alpha_2)\sigma} x.\, \{\!|e|\!\}^{\mathcal{D}'}_{\sigma}$$
where $\mathcal{D} = \dfrac{\mathcal{D}' :: (\Gamma, x\colon \alpha_1);\Delta \vdash \langle\!\langle e\colon \alpha_2\rangle\!\rangle \rightsquigarrow D'}{\Gamma;\Delta \vdash \langle\!\langle (\lambda x.\, e)\colon t\rangle\!\rangle \rightsquigarrow D' \cup \{(\alpha_1 \mathrel{\dot{\sqsubseteq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\}}$

$$\{\!|\lambda x\colon \tau.\, e|\!\}^{\mathcal{D}}_{\sigma} = (\lambda^{(\tau\to\alpha_2)\sigma} x.\, \{\!|e|\!\}^{\mathcal{D}'}_{\sigma})\langle(\tau \to \alpha_2)\sigma \overset{\ell}{\Rightarrow} (\alpha_1 \to \alpha_2)\sigma\rangle$$
with ℓ fresh
where $\mathcal{D} = \dfrac{\mathcal{D}' :: (\Gamma, x\colon \tau);\Delta \vdash \langle\!\langle e\colon \alpha_2\rangle\!\rangle \rightsquigarrow D'}{\Gamma;\Delta \vdash \langle\!\langle (\lambda x\colon \tau.\, e)\colon t\rangle\!\rangle \rightsquigarrow D' \cup \{(\tau \mathrel{\dot{\sqsubseteq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\}}$

$$\{\!|e_1\ e_2|\!\}^{\mathcal{D}}_{\sigma} = \{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma}\ \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}$$
where $\mathcal{D} = \dfrac{\mathcal{D}_1 :: \Gamma;\Delta \vdash \langle\!\langle e_1\colon \alpha \to t\rangle\!\rangle \rightsquigarrow D_1 \qquad \mathcal{D}_2 :: \Gamma;\Delta \vdash \langle\!\langle e_2\colon \alpha\rangle\!\rangle \rightsquigarrow D_2}{\Gamma;\Delta \vdash \langle\!\langle e_1\ e_2\colon t\rangle\!\rangle \rightsquigarrow D_1 \cup D_2}$

$$\{\!|(e_1, e_2)|\!\}^{\mathcal{D}}_{\sigma} = (\{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma}, \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma})$$
where $\mathcal{D} = \dfrac{\mathcal{D}_1 :: \Gamma;\Delta \vdash \langle\!\langle e_1\colon \alpha_1\rangle\!\rangle \rightsquigarrow D_1 \qquad \mathcal{D}_2 :: \Gamma;\Delta \vdash \langle\!\langle e_2\colon \alpha_2\rangle\!\rangle \rightsquigarrow D_2}{\Gamma;\Delta \vdash \langle\!\langle (e_1, e_2)\colon t\rangle\!\rangle \rightsquigarrow D_1 \cup D_2 \cup \{\alpha_1 \times \alpha_2 \mathrel{\dot{\leq}} t\}}$

$$\{\!|\pi_i\ e|\!\}^{\mathcal{D}}_{\sigma} = \pi_i\ \{\!|e|\!\}^{\mathcal{D}'}_{\sigma}$$
where $\mathcal{D} = \dfrac{\mathcal{D}' :: \Gamma;\Delta \vdash \langle\!\langle e\colon \alpha_1 \times \alpha_2\rangle\!\rangle \rightsquigarrow D'}{\Gamma;\Delta \vdash \langle\!\langle \pi_i\ e\colon t\rangle\!\rangle \rightsquigarrow D' \cup \{\alpha_i \mathrel{\dot{\leq}} t\}}$

$$\{\!|\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2|\!\}^{\mathcal{D}}_{\sigma} = \mathsf{let}\ x = \Lambda\vec{\alpha}_1,\vec{\beta}_1.\ \{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma_1}\rho\sigma\ \mathsf{in}\ \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}$$
where $\mathcal{D} = \dfrac{\mathcal{D}_1 :: \Gamma;\Delta \cup \vec{\alpha} \vdash C_1 \rightsquigarrow D_1 \qquad \mathcal{D}_2 :: (\Gamma, x\colon \forall\vec{\alpha},\vec{\beta}.\,\alpha\sigma_1);\Delta \vdash C_2 \rightsquigarrow D_2}{\Gamma;\Delta \vdash \langle\!\langle \mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2\colon t\rangle\!\rangle \rightsquigarrow D_2 \cup \mathsf{equiv}(\sigma_1, D_1)}$
and $\sigma_1 \in \mathsf{solve}_{\Delta\cup\vec{\alpha}}(D_1)$ $\quad\vec{\alpha}_1,\vec{\beta}_1$ fresh $\quad\rho = [\vec{\alpha}_1/\vec{\alpha},\vec{\beta}_1/\vec{\beta}]$

FIGURE 9.9 Algorithmic compilation

The latter result, combined with completeness of solve, ensures that inference can compute most general types for all expressions. In particular, starting from a program (i.e., a closed expression) e, we pick a fresh variable α and generate $\langle\!\langle e : \alpha \rangle\!\rangle$. Completeness ensures that, if the program is well typed, we can find a derivation $\mathcal{D}$ for $\varnothing; \varnothing \vdash \langle\!\langle e : \alpha \rangle\!\rangle \rightsquigarrow D$ and D has a solution. Since solve is complete, we can compute the principal solution σ of D. Then, $\alpha\sigma$ is the most general type for the program and $\{\!|e|\!\}^{\mathcal{D}}_{\sigma}$ is its compilation driven by the derivation $\mathcal{D}$.

9.3.5 Soundness of type inference

We say that a type substitution σ is *static* if it maps type variables to static types. When $\overline{\alpha}$ is a set of type variables, we say that σ is *static on* $\overline{\alpha}$, and we write $\mathsf{static}(\sigma, \overline{\alpha})$, to mean that $\alpha\sigma$ is static for every $\alpha \in \overline{\alpha}$.

The following lemma states that typing is preserved by static type substitutions. It is not necessarily preserved by non-static substitutions, because the typing rules require some types to be static (the parameters of functions without annotations and the types used to instantiate type schemes). For example, $\lambda x.\, x$ has type $\alpha \to \alpha$ but not $? \to ?$: typing is not preserved by the substitution $[?/\alpha]$.

9.17 LEMMA (Stability of typing under type substitution): If $\Gamma \vdash e \rightsquigarrow E : \tau$, then, for every static type substitution σ, we have $\Gamma\sigma \vdash e\sigma \rightsquigarrow E\sigma : \tau\sigma$. □

Proof in appendix (p. 257).

The following two results are straightforward. In the first, the hypothesis $\mathsf{static}(\sigma', \mathsf{var}(D)\sigma)$ is not needed. We include it to highlight the fact that it holds when we apply it: it will be important when we add subtyping, because then the proof of the result will require it.

9.18 LEMMA: Let σ and σ' be two type substitutions such that $\sigma \Vdash_{\Delta} D$ and $\mathsf{static}(\sigma', \mathsf{var}(D)\sigma)$. If $(t_1 \doteq t_2) \in D$, then $t_1\sigma\sigma' = t_2\sigma\sigma'$. □

Proof: By definition of $\sigma \Vdash_{\Delta} D$, we have $t_1\sigma = t_2\sigma$. Then, $t_1\sigma\sigma' = t_2\sigma\sigma'$. □

9.19 LEMMA: Let σ and σ' be two type substitutions. If $\sigma \Vdash_{\Delta} D$ and $(\tau \mathrel{\dot{\sqsubseteq}} \alpha) \in D$, then $\tau\sigma\sigma' \sqsubseteq \alpha\sigma\sigma'$. □

Proof: By definition of $\sigma \Vdash_{\Delta} D$, we have $\tau\sigma \sqsubseteq \alpha\sigma$. Then, $\tau\sigma\sigma' \sqsubseteq \alpha\sigma\sigma'$. □

The following two results give an inversion principle on the constraint simplification relation and characterize which variables appear in the constraints obtained by simplification.

9.20 LEMMA: Let $\mathcal{D}$ be a derivation of $\Gamma; \Delta \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$. Then:

- if $e = x$, then $\Gamma(x) = \forall\vec{\alpha}.\,\tau$ and $D = \{(\tau[\vec{\beta}/\vec{\alpha}] \mathrel{\dot{\sqsubseteq}} \alpha), (\alpha \mathrel{\dot{\leq}} t)\}$ (for some τ, α, $\vec{\alpha}$, $\vec{\beta}$);
- if $e = c$, then $D = \{b_c \mathrel{\dot{\leq}} t\}$;
- if $e = \lambda x.\,e'$, then $\mathcal{D}$ contains a sub-derivation of $(\Gamma, x\colon \alpha_1); \Delta \vdash \langle\!\langle e'\colon \alpha_2 \rangle\!\rangle \rightsquigarrow D'$, and $D = D' \cup \{(\alpha_1 \mathrel{\dot{\sqsubseteq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\}$;
- if $e = \lambda x\colon \tau.\,e'$, then $\mathcal{D}$ contains a sub-derivation of $(\Gamma, x\colon \tau); \Delta \vdash \langle\!\langle e'\colon \alpha_2 \rangle\!\rangle \rightsquigarrow D'$, and $D = D' \cup \{(\tau \mathrel{\dot{\sqsubseteq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\}$;
- if $e = e_1 e_2$, then $\mathcal{D}$ contains two sub-derivations of $\Gamma; \Delta \vdash \langle\!\langle e_1\colon \alpha \to t \rangle\!\rangle \rightsquigarrow D_1$ and $\Gamma; \Delta \vdash \langle\!\langle e_2\colon \alpha \rangle\!\rangle \rightsquigarrow D_2$ (for some α, D_1, and D_2), and $D = D_1 \cup D_2$;
- if $e = (e_1, e_2)$, then $\mathcal{D}$ contains two sub-derivations of $\Gamma; \Delta \vdash \langle\!\langle e_1\colon \alpha_1 \rangle\!\rangle \rightsquigarrow D_1$ and $\Gamma; \Delta \vdash \langle\!\langle e_2\colon \alpha_2 \rangle\!\rangle \rightsquigarrow D_2$ (for some α_1, α_2, D_1, and D_2), and $D = D_1 \cup D_2 \cup \{\alpha_1 \times \alpha_2 \mathrel{\dot{\leq}} t\}$;
- if $e = \pi_i e'$, then $\mathcal{D}$ contains a sub-derivation of $\Gamma; \Delta \vdash \langle\!\langle e'\colon \alpha_1 \times \alpha_2 \rangle\!\rangle \rightsquigarrow D'$, and $D = D' \cup \{\alpha_i \mathrel{\dot{\leq}} t\}$;
- if $e = (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2)$, then $\mathcal{D}$ contains two sub-derivations of $\Gamma; \Delta \cup \vec{\alpha} \vdash \langle\!\langle e_1\colon \alpha \rangle\!\rangle \rightsquigarrow D_1$ and $(\Gamma, x\colon \forall\vec{\alpha},\vec{\beta}.\,\alpha\sigma_1); \Delta \vdash \langle\!\langle e_2\colon t \rangle\!\rangle \rightsquigarrow D_2$, and the following hold:

$$D = D_2 \cup \mathsf{equiv}(\sigma_1, D_1) \qquad \sigma_1 \in \mathsf{solve}_{\Delta\cup\vec{\alpha}}(D_1)$$
$$\vec{\alpha} \mathrel{\sharp} \mathsf{var}(\Gamma\sigma_1) \qquad \vec{\beta} = \mathsf{var}(\alpha\sigma_1) \setminus (\mathsf{var}(\Gamma\sigma_1) \cup \vec{\alpha} \cup \mathsf{var}(e_1)) \qquad \square$$

Proof: Straightforward, since the constraint simplification rules are syntax-directed. □

9.21 LEMMA: If $\Gamma; \Delta \vdash C \rightsquigarrow D$, then $\mathsf{var}(\Gamma) \cap \mathsf{var}(D) \subseteq \mathsf{var}(C) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)$. □

Proof in appendix (p. 258).

We prove that the solutions obtained by solve map the variables in the type environment to static types. This is important because the variables in the type environment can correspond to the parameters of functions without annotations, which must be static types.

9.22 LEMMA:

$$\forall\Gamma, \Delta, e, \alpha, D, \sigma.\ \left.\begin{array}{l} \Gamma; \Delta \vdash \langle\!\langle e\colon \alpha \rangle\!\rangle \rightsquigarrow D \\ \sigma \in \mathsf{solve}_\Delta(D) \\ \mathsf{var}(e) \subseteq \Delta \\ \alpha \notin \mathsf{var}(\Gamma) \end{array}\right\} \implies \mathsf{static}(\sigma, \mathsf{var}(\Gamma))$$

□

Proof in appendix (p. 259).

The following result states that the definition of the set equiv is sound. The statement is quite involved because it considers four different substitutions. In

essence, it states that, if σ satisfies $\mathsf{equiv}(\sigma_1, D_1)$, then it behaves on the type environment Γ like the composition of itself with $\rho \circ \sigma_1$. The statement could be simplified here by removing σ': we use this form because it matches the one we need in the extension with subtyping.

9.23 LEMMA:

$$\forall \Gamma, \Delta, D_1, \sigma_1, \rho, \sigma, \sigma'. \left.\begin{array}{l} \sigma \Vdash_\Delta \mathsf{equiv}(\sigma_1, D_1) \\ \mathsf{dom}(\rho) \sharp \Gamma\sigma_1 \\ \mathsf{static}(\sigma', \mathsf{var}(\mathsf{equiv}(\sigma_1, D_1))\sigma) \\ \mathsf{static}(\sigma_1, \mathsf{var}(\Gamma)) \end{array}\right\} \Longrightarrow \Gamma\sigma\sigma' = \Gamma\sigma_1\rho\sigma\sigma'$$

□

Proof in appendix (p. 259).

Finally, we prove soundness itself.

9.24 THEOREM (Soundness of type inference): Let $\mathcal{D}$ be a derivation of $\Gamma; \mathsf{var}(e) \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$. Let σ be a type substitution such that $\sigma \Vdash_{\mathsf{var}(e)} D$. Then, we have $\Gamma\sigma \vdash e \rightsquigarrow \{\!|e|\!\}^{\mathcal{D}}_{\sigma} : t\sigma$. □

Proof in appendix (p. 260). The proof is by structural induction on e. It relies on Lemmas 9.18 to 9.20 and, when e is a let expression, on Lemmas 9.22 and 9.23. The induction hypothesis must consider an additional substitution σ' to deal with let expressions, where $\mathcal{D}$ includes a substitution computed by solve which is different from σ.

9.3.6 Completeness of type inference

We first state an inversion principle for the declarative typing relation.

9.25 LEMMA: Let $\Gamma \vdash e : \tau$. Then:

- if $e = x$ then $\Gamma(x) = \forall \vec{\alpha}. \tau_x$ and $\tau_x[\vec{t}/\vec{\alpha}] \sqsubseteq \tau$;
- if $e = c$, then $\tau = b_c$;
- if $e = \lambda x. e_1$ then $\tau = t \to \tau_1$ and $\Gamma, x : t \vdash e_1 : \tau_1$;
- if $e = \lambda x : \tau'. e_1$ then $\tau = \tau'_1 \to \tau_1$, $\tau' \sqsubseteq \tau'_1$, and $\Gamma, x : \tau' \vdash e_1 : \tau_1$;
- if $e = e_1\ e_2$, then $\Gamma \vdash e_1 : \tau' \to \tau$ and $\Gamma \vdash e_2 : \tau'$;
- if $e = (e_1, e_2)$, then $\tau = \tau_1 \times \tau_2$, $\Gamma \vdash e_1 : \tau_1$, and $\Gamma \vdash e_2 : \tau_2$;
- if $e = \pi_i\ e'$, then $\Gamma \vdash e' : \tau_1 \times \tau_2$ and $\tau = \tau_i$;
- if $e = (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2)$, then $\Gamma \vdash e_1 : \tau_1$, $\Gamma, x : \forall \vec{\alpha}, \vec{\beta}. \tau_1 \vdash e_2 : \tau$, $\vec{\alpha}, \vec{\beta} \sharp \Gamma$, and $\vec{\beta} \sharp e_1$. □

Proof: The derivation of $\Gamma \vdash e : \tau$ must end with the rule corresponding to the shape of e, possibly followed by applications of $[\mathrm{T}_\sqsubseteq]$. We proceed by case

analysis on the derivation, possibly applying $[\mathrm{T}_{\sqsubseteq}]$ to the derivations in the premises to obtain the needed results. □

The following are three auxiliary results used to prove completeness.

9.26 LEMMA: If $\Gamma; \Delta \vdash C \rightsquigarrow D \mid \vec{\alpha}$, then $\mathsf{var}(D) \subseteq \mathsf{var}(\Gamma) \cup \mathsf{var}(C) \cup \vec{\alpha}$. □

Proof in appendix (p. 263).

9.27 LEMMA: If $\Gamma; \Delta \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D \mid \vec{\alpha}$, then $\mathsf{var}(t) \subseteq \mathsf{var}(D)$. □

Proof in appendix (p. 264).

9.28 LEMMA: Let σ and $\sigma_1, \ldots, \sigma_n$ be type substitutions, such that the σ_i are pairwise disjoint and every σ_i is disjoint from σ. Let $D_1, \ldots, D_n$ be type constraint sets such that, for every $i_1 \neq i_2$, $\sigma_{i_1} \not\sharp \mathsf{var}(D_{i_2})$.

If, for every $i \in \{1, \ldots, n\}$, we have $\sigma \cup \sigma_i \Vdash_\Delta D_i$, then $\sigma \cup \bigcup_{i=1}^n \sigma_i \Vdash_\Delta \bigcup_{i=1}^n D_i$. □

Proof: Straightforward since, because of the disjointness conditions, for every i_0 and every $\alpha \in \mathsf{var}(D_{i_0})$, we have $\alpha(\sigma \cup \bigcup_{i=1}^n \sigma_i) = \alpha(\sigma \cup \sigma_{i_0})$. □

Finally, we give the statement of completeness of type inference.

9.29 THEOREM (Completeness of type inference): If $\Gamma \vdash e : \tau$, then, for every fresh type variable α, there exist D and σ such that $\Gamma; \mathsf{var}(e) \vdash \langle\!\langle e : \alpha \rangle\!\rangle \rightsquigarrow D$ and $[\tau/\alpha] \cup \sigma \Vdash_{\mathsf{var}(e)} D$. □

Proof in appendix (p. 264).

Throughout the proof of completeness, we use *variable pools* to choose fresh variables: a set $\mathfrak{U} \subseteq \mathsf{Var}$ is a variable pool if both $\mathfrak{U} \cap \mathsf{TVar}$ and $\mathfrak{U} \cap \mathsf{FVar}$ are countably infinite. We can partition variable pools to obtain new pools. For example, we write $\mathfrak{U} = \{\alpha\} \uplus \mathfrak{U}_1 \uplus \mathfrak{U}_2$ to mean that we partition $\mathfrak{U}$ into three sets: a singleton set α and two variable pools $\mathfrak{U}_1$ and $\mathfrak{U}_2$.

9.3.7 An example of type inference

Let e be the term $\mathsf{let}\ \alpha\ x = (\lambda y : \alpha.\ y)\ \mathsf{in}\ 1 + \big(x\ ((\lambda z : ?.\ z)\ 3)\big)$ (we assume to have a + operator in the language). Since $x\ ((\lambda z : ?.\ z)\ 3)$ is used as a number, to be well typed it should be given type Int. In the declarative system, $\lambda z : ?.\ z$ has type $? \to ?$, which can be materialized to $\mathsf{Int} \to \mathsf{Int}$; then its application to 3 has type Int; therefore applying the identity function x, we also get type Int.

Inference can find this solution, as follows. We use a type variable β as the expected type, and we generate the constraints below. We have:

$$\begin{aligned}\langle\!\langle e\colon \beta\rangle\!\rangle &= \langle\!\langle \mathsf{let}\ \alpha\ x = (\lambda y\colon \alpha.\ y)\ \mathsf{in}\ 1 + \big(x\ ((\lambda z\colon ?.\ z)\ 3)\big)\colon \beta\rangle\!\rangle \\ &= \mathsf{let}\ x\colon \forall\alpha; \alpha_1[C_1]^{\epsilon}.\ \alpha_1\ \mathsf{in}\ C_2\end{aligned}$$

where

$$\begin{aligned}C_1 &= \langle\!\langle (\lambda y\colon \alpha.\ y)\colon \alpha_1\rangle\!\rangle \\ &= \exists\alpha_2, \alpha_3.\ \big(\mathsf{def}\ y\colon \alpha\ \mathsf{in}\ \langle\!\langle y\colon \alpha_3\rangle\!\rangle\big) \wedge (\alpha \mathrel{\dot{\sqsubseteq}} \alpha_2) \wedge (\alpha_2 \to \alpha_3 \mathrel{\dot{\leq}} \alpha_1) \\ C_2 &= \langle\!\langle 1 + \big(x\ ((\lambda z\colon ?.\ z)\ 3)\big)\colon \beta\rangle\!\rangle = (\mathsf{Int} \mathrel{\dot{\leq}} \beta) \wedge \langle\!\langle x\ ((\lambda z\colon ?.\ z)\ 3)\colon \mathsf{Int}\rangle\!\rangle \\ &= (\mathsf{Int} \mathrel{\dot{\leq}} \beta) \wedge \big(\exists\alpha_4.\ \langle\!\langle x\colon \alpha_4 \to \mathsf{Int}\rangle\!\rangle \\ &\qquad \wedge \big(\exists\alpha_5.\ \langle\!\langle (\lambda z\colon ?.\ z)\colon \alpha_5 \to \alpha_4\rangle\!\rangle \wedge (b_3 \mathrel{\dot{\leq}} \alpha_5)\big)\big)\end{aligned}$$

and

$$\begin{aligned}\langle\!\langle y\colon \alpha_3\rangle\!\rangle &= \exists\alpha_6.\ (y \mathrel{\dot{\sqsubseteq}} \alpha_6) \wedge (\alpha_6 \mathrel{\dot{\leq}} \alpha_3) \\ \langle\!\langle x\colon \alpha_4 \to \mathsf{Int}\rangle\!\rangle &= \exists\alpha_7.\ (x \mathrel{\dot{\sqsubseteq}} \alpha_7) \wedge (\alpha_7 \mathrel{\dot{\leq}} \alpha_4 \to \mathsf{Int}) \\ \langle\!\langle (\lambda z\colon ?.\ z)\colon \alpha_5 \to \alpha_4\rangle\!\rangle &= \exists\alpha_8, \alpha_9.\ \big(\mathsf{def}\ z\colon ?\ \mathsf{in}\ \exists\alpha_{10}.\ (z \mathrel{\dot{\sqsubseteq}} \alpha_{10}) \wedge (\alpha_{10} \mathrel{\dot{\leq}} \alpha_9)\big) \\ &\qquad \wedge (? \mathrel{\dot{\sqsubseteq}} \alpha_8) \wedge (\alpha_8 \to \alpha_9 \mathrel{\dot{\leq}} \alpha_5 \to \alpha_4)\end{aligned}$$

We simplify $\langle\!\langle e\colon \beta\rangle\!\rangle$ in the empty environment with $\Delta = \emptyset$. To do this, we first simplify C_1: we have

$$\emptyset; \{\alpha\} \vdash C_1 \rightsquigarrow \big\{(\alpha \mathrel{\dot{\sqsubseteq}} \alpha_6), (\alpha_6 \mathrel{\dot{\leq}} \alpha_3), (\alpha \mathrel{\dot{\sqsubseteq}} \alpha_2), (\alpha_2 \to \alpha_3 \mathrel{\dot{\leq}} \alpha_1)\big\}\ .$$

By unification we obtain the solution $\sigma_1 = [(\alpha \to \alpha)/\alpha_1, \alpha/\alpha_2, \alpha/\alpha_3, \alpha/\alpha_6]$. We obtain the expanded environment $x\colon \forall\alpha.\ \alpha \to \alpha$. Then, we simplify C_2. We have $(x\colon \forall\alpha.\ \alpha \to \alpha); \emptyset \vdash C_2 \rightsquigarrow D_2$ with

$$\begin{aligned}D_2 = \big\{&(\gamma \to \gamma \mathrel{\dot{\sqsubseteq}} \alpha_7), (\alpha_7 \mathrel{\dot{\leq}} \alpha_4 \to \mathsf{Int}), \\ &(? \mathrel{\dot{\sqsubseteq}} \alpha_{10}), (\alpha_{10} \mathrel{\dot{\leq}} \alpha_9), (? \mathrel{\dot{\sqsubseteq}} \alpha_8), (\alpha_8 \to \alpha_9 \mathrel{\dot{\leq}} \alpha_5 \to \alpha_4), (b_3 \mathrel{\dot{\leq}} \alpha_5)\big\}\ .\end{aligned}$$

The final constraint set is $D = D_2 \cup \mathsf{equiv}(\sigma_1, D_1)$, with

$$\begin{aligned}\mathsf{equiv}(\sigma_1, D_1) = \{&(\alpha \mathrel{\dot{\sqsubseteq}} \alpha), (\alpha_1 \mathrel{\dot{\leq}} \alpha \to \alpha), (\alpha \to \alpha \mathrel{\dot{\leq}} \alpha_1), \\ &(\alpha_2 \mathrel{\dot{\leq}} \alpha), (\alpha \mathrel{\dot{\leq}} \alpha_2), (\alpha_3 \mathrel{\dot{\leq}} \alpha), (\alpha \mathrel{\dot{\leq}} \alpha_3), (\alpha_6 \mathrel{\dot{\leq}} \alpha), (\alpha \mathrel{\dot{\leq}} \alpha_6)\}\ .\end{aligned}$$

A solution to D is

$$\sigma = \sigma_1 \cup [\mathsf{Int}/\alpha_4, \mathsf{Int}/\alpha_5, (\mathsf{Int} \to \mathsf{Int})/\alpha_7, \mathsf{Int}/\alpha_8, \mathsf{Int}/\alpha_9, \mathsf{Int}/\alpha_{10}, \mathsf{Int}/\beta, \mathsf{Int}/\gamma]\ .$$

Let $\mathcal{D}$ be the derivation of constraint simplification that we have described. Then, the compiled expression $[\![e]\!]_{\sigma}^{\mathcal{D}}$ is (omitting identity casts)

$$\begin{aligned}&\mathsf{let}\ x = (\Lambda\alpha.\ \lambda^{\alpha\to\alpha} y.\ y)\ \mathsf{in} \\ &\qquad \big(x\ [\mathsf{Int}]\big)\big((\lambda^{?\to\mathsf{Int}} z.\ z\langle ? \overset{\ell_1}{\Rightarrow} \mathsf{Int}\rangle)\langle ? \to \mathsf{Int} \overset{\ell_2}{\Rightarrow} \mathsf{Int} \to \mathsf{Int}\rangle\ 3\big)\ .\end{aligned}$$

9.4 Adding subtyping

In this section we explain how to add subtyping to the system of the previous sections. We outline the necessary additions in brief. We present only the declarative system and not the type inference algorithm. The extension of type inference with subtyping is, of course, challenging; as we explain in Section 9.4.2, it requires some form of union and intersection operations on types. Therefore, we postpone it to the next chapter, where we add set-theoretic types to the language.

9.4.1 Declarative system

SUBTYPING: We add subtyping to the language by defining a preorder $\leq^?$ on gradual types. In the absence of set-theoretic type connectives and recursive types, subtyping can be defined with simple inductive rules. We start from a preorder $\leq^?$ on Base (e.g., Nat $\leq^?$ Int $\leq^?$ Real) and extend it to GType by the inductive application of the following inference rules:

$$\frac{}{? \leq^? ?} \qquad \frac{}{\alpha \leq^? \alpha} \qquad \frac{\tau_1 \leq^? \tau_1' \qquad \tau_2 \leq^? \tau_2'}{\tau_1 \times \tau_2 \leq^? \tau_1' \times \tau_2'} \qquad \frac{\tau_1' \leq^? \tau_1 \qquad \tau_2 \leq^? \tau_2'}{\tau_1 \to \tau_2 \leq^? \tau_1' \to \tau_2'}$$

These rules are standard: covariance for products, co-contravariance for arrows. Just notice that, from the point of view of subtyping, the dynamic type ? is only related to itself, just like a type variable (cf. Siek and Taha, 2007).

TYPE SYSTEM: The extension of the source gradual language with subtyping could not be simpler: it suffices to add to the declarative typing rules of Figure 9.1 the standard subsumption rule [T$_\leq$].

$$[\mathrm{T}_\leq]\ \frac{\Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau}\ \tau' \leq^? \tau$$

The definition of the dynamic semantics does not require any essential change, either. The cast calculus is the same as in Section 9.2, except that the [T$_\leq$] rule above must be added to its typing rules and that two cast reduction rules (in appendix) that use type equality must be generalized to subtyping. The definition of the compilation of the source language into the "new" cast calculus does not change either (subsumption is neutral for compilation). The proof that compilation preserves types stays essentially the same, since we have just added the subsumption rule to both systems.

9.4.2 Type inference

The changes required to add subtyping to the declarative system are minimal: define the subtyping relation, add the subsumption rule, and recheck the proofs since they need slight modifications. On the contrary, defining algorithms to decide the relations we have just defined is more complicated. As we saw in Section 9.3, this amounts to generating and solving constraints.

Constraint generation is not problematic. The form of the constraints and the generation algorithm given in Section 9.3 already account for the extension with subtyping: hence, they do not need to be changed, neither here nor in the next chapter. Constraint resolution, instead, is a different matter. In the previous section, constraints of the form $\alpha \doteq t$ were actually equality constraints (i.e., $\alpha \doteq t$) that could be solved by unification. The same constraints now denote subtyping, and their resolution requires the computation of intersections and unions.

To see why, consider the following OCaml code snippet (that does not involve any gradual typing):

```
fun x → if (fst x) then (1 + snd x) else x
```

We want our system to deduce for this definition the following type:[6]

$$(\mathsf{Bool} \times \mathsf{Int}) \to (\mathsf{Int} \vee (\mathsf{Bool} \times \mathsf{Int}))$$

To that end, a constraint generation system like ours could assign to the function the type $\alpha \to \beta$ and generate the following set of four constraints:

$$\{(\alpha \mathrel{\dot{\leq}} \mathsf{Bool} \times \mathbb{1}), (\alpha \mathrel{\dot{\leq}} \mathbb{1} \times \mathsf{Int}), (\mathsf{Int} \mathrel{\dot{\leq}} \beta), (\alpha \mathrel{\dot{\leq}} \beta)\}$$

where $\mathbb{1}$ denotes the top type (that is, the supertype of all types). The first constraint is generated because $\mathsf{fst}\,\mathsf{x}$ is used in a position where a Boolean is expected; the second comes from the use of $\mathsf{snd}\,\mathsf{x}$ in an integer position; the last two constraints are produced to type the result of the conditional branch with a supertype of the types of both branches. To compute the solution of two constraints of the form $\alpha \mathrel{\dot{\leq}} t_1$ and $\alpha \mathrel{\dot{\leq}} t_2$, the resolution algorithm must compute the greatest lower bound of t_1 and t_2 (or an approximation thereof); likewise for two constraints of the form $s_1 \mathrel{\dot{\leq}} \beta$ and $s_2 \mathrel{\dot{\leq}} \beta$ the best solution is the least upper bound of s_1 and s_2. This yields $\mathsf{Bool} \times \mathsf{Int}$ for the domain (i.e., the intersection of the upper bounds for α) and $\mathsf{Int} \vee (\mathsf{Bool} \times \mathsf{Int})$ for the codomain (i.e., the union of the lower bounds for β).

In summary, to perform type reconstruction in the presence of subtyping, one must be able to compute unions and intersections of types. In some cases, as for the domain in the example above, the solution of these operations is a type of ML (or of the language at issue): then the operations can be meta-operations computed by the type checker but not exposed to the programmer. In other cases, as for the codomain in the example, the solution is a type which might not already exist in the language: therefore, the only solution to type the expression precisely is to add the corresponding set-theoretic operations to the types of the language.

The full range of these options can be found in the literature. For instance, Pottier (2001) defines intersection and union as meta-operations, and it is not possible to simplify the constraints to derive a type like the one above. Other systems include both intersections and unions in the types, starting from the earliest work by Aiken and Wimmers (1993) to more recent work by Dolan and Mycroft (2017). In the next chapter, we add set-theoretic connectives to gradual types, and we show how to adapt type inference to that setting.

6 Using set-theoretic types, we could give a more precise type: $(\mathsf{Bool} \times \mathsf{Int}) \wedge \alpha \to \mathsf{Int} \vee \alpha$. For instance, using this type (with the instantiation $[(\mathsf{Bool} \times \mathsf{Nat})/\alpha]$) we can predict that the application of the function to an expression of type $\mathsf{Bool} \times \mathsf{Nat}$ has type $\mathsf{Int} \vee (\mathsf{Bool} \times \mathsf{Nat})$ instead of $\mathsf{Int} \vee (\mathsf{Bool} \times \mathsf{Int})$.

10 Gradual typing for set-theoretic types

In this chapter, we study how to apply our approach to gradual typing in order to define a gradual type system featuring set-theoretic types and semantic subtyping. In Section 9.4, we have outlined how to add subtyping to the type system. To extend the declarative presentation of typing, we only need to define a suitable subtyping relation on gradual types. The relation in Section 9.4.1 is straightforward – it treats ? just like a type variable – but its extension to set-theoretic types is more difficult. Adding set-theoretic types also makes it more complex to define the operational semantics of the cast calculus but, as in the previous chapter, we will only introduce the problem because we concentrate on typing. Finally, the extension of type inference to set-theoretic types can be done by replacing the type-constraint solving algorithm with one adapted to set-theoretic types and subtyping constraints. We show how to do so and prove soundness of type inference; however, completeness does not hold, the main difficulty being the treatment of recursive types.

CHAPTER OUTLINE:

Section 10.1 We define type frames, static, and gradual types including set-theoretic type connectives. Type frames use the subtyping relation of Chapter 2.

Section 10.2 We define subtyping on set-theoretic gradual types by translating them to type frames. We consider different possible characterizations and prove their equivalence. Finally, we study some properties of the subtyping relation and of its interaction with materialization.

Section 10.3 We describe how to update the syntax and type systems of the source and cast languages of Sections 9.1 and 9.2 to add set-theoretic types. We introduce briefly the needed changes in the semantics.

Section 10.4 We describe type inference with set-theoretic types, relying on the tallying algorithm of Castagna et al. (2015b) already used in Part I.

10.1 Type frames, static types, and gradual types

We start by defining the different sorts of types that we will use.

As in the previous chapter, we distinguish two sorts of variables: we use *type variables* to express polymorphism and *frame variables* to replace ? in type frames in the definition of materialization and, as we will see, also in that of subtyping. We consider a countable set Var, partitioned into two countable sets: the set TVar of type variables and the set FVar of frame variables. We use

the metavariable A to range over Var, α (and also β and γ) to range over TVar, and X (and also Y) to range over FVar.

As in Section 2.2, we also consider a set Const of *language constants* (ranged over by c), a set Base of *base types* (ranged over by b), and two functions

$$b_{(\cdot)} \colon \mathsf{Const} \to \mathsf{Base} \qquad \mathbb{B}(\cdot) \colon \mathsf{Base} \to \mathcal{P}(\mathsf{Const})$$

that map constants to base types and base types to sets of constants. We assume that each constant has an associated singleton type: that is, for every $c \in \mathsf{Const}$, we assume that $\mathbb{B}(b_c) = \{c\}$.

We define type frames with both type and frame variables, static types with type variables only, and gradual types with type variables and ?.

10.1 DEFINITION (Type frames, static types, and gradual types): The sets TFrame of *type frames*, SType of *static types*, and GType of *gradual types*, are the sets of terms T, t, and τ, respectively, generated coinductively by the following grammars:

$$\begin{aligned}
\mathsf{TFrame} \ni T &::= A \mid b \mid T \times T \mid T \to T \mid T \vee T \mid \neg T \mid \mathbb{0} && \text{type frames} \\
\mathsf{SType} \ni t &::= \alpha \mid b \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid \mathbb{0} && \text{static types} \\
\mathsf{GType} \ni \tau &::= ? \mid \alpha \mid b \mid \tau \times \tau \mid \tau \to \tau \mid \tau \vee \tau \mid \neg \tau \mid \mathbb{0} && \text{gradual types}
\end{aligned}$$

(where A ranges over Var, α over TVar, and b over Base) and that satisfy the following two conditions:

(regularity) the term has finitely many distinct subterms;

(contractivity) every infinite path in the term contains infinitely many occurrences of the $\times$ or $\to$ constructors. □

We introduce the usual abbreviations

$$T_1 \wedge T_2 \stackrel{\text{def}}{=} \neg(\neg T_1 \vee \neg T_2) \qquad T_1 \setminus T_2 \stackrel{\text{def}}{=} T_1 \wedge (\neg T_2) \qquad \mathbb{1} \stackrel{\text{def}}{=} \neg \mathbb{0}$$

for type frames and likewise for static and gradual types.

Given a type frame T, we write $\mathsf{var}(T)$ for the set of variables occurring in T. We write $\mathsf{tvar}(T)$ for $\mathsf{var}(T) \cap \mathsf{TVar}$ and $\mathsf{fvar}(T)$ for $\mathsf{var}(T) \cap \mathsf{FVar}$. We use this notation also for static and gradual types – of course, for these, $\mathsf{var}(\cdot)$ and $\mathsf{tvar}(\cdot)$ coincide and $\mathsf{fvar}(\cdot)$ is always empty.

Note that static types are included in both gradual types and type frames; in particular, $\mathsf{SType} = \{\, T \in \mathsf{TFrame} \mid \mathsf{fvar}(T) = \varnothing \,\}$.

Type substitutions are defined as in Section 2.2.1. For gradual types, we have $?\sigma = ?$ for every type substitution σ. Substitutions can instantiate both type and frame variables, and their range can include any of these sorts of types. When $\mathcal{V}$ is a set of type variables and $\mathcal{T}$ a set of types, we write $\sigma \colon \mathcal{V} \to \mathcal{T}$ to mean that $\mathsf{dom}(\sigma) \subseteq \mathcal{V}$ and to restrict which types can be in the range of σ. For instance, we write $\sigma_1 \colon \mathsf{TVar} \to \mathsf{SType}$ if σ_1 maps α variables to static types and $\sigma_2 \colon \mathsf{FVar} \to \mathsf{GType}$ if σ_2 maps X variables to gradual types.

10.1.1 Subtyping on type frames and static types

Type frames and static types use the subtyping relation $\leq$ and the equivalence relation $\simeq$ defined in Section 2.3. We do not repeat the definitions here: they are exactly as in Section 2.3 except that we replace TVar with Var everywhere, since type frames include both α and X variables. The following property holds (proven as Proposition 2.11).

10.2 PROPOSITION: If $T_1 \leq T_2$, then $T_1\sigma \leq T_2\sigma$ for any type substitution σ. □

10.1.2 Materialization

As in the previous chapter, we write $T^\dagger$ for the gradual type obtained from T by replacing all frame variables with ?. We define the set $\star(\tau)$ of the *discriminations* of τ as

$$\star(\tau) \stackrel{\text{def}}{=} \{\, T \in \mathsf{TFrame} \mid T^\dagger = \tau \,\}\,.$$

To define materialization, nothing needs to change. Definition 9.2, which defines materialization as

$$\tau_1 \sqsubseteq \tau_2 \iff^{\text{def}} \exists T_1 \in \star(\tau_1), \sigma : \mathsf{FVar} \to \mathsf{GType}.\ T_1\sigma = \tau_2$$

using discrimination and type substitutions, is equally valid here though we have changed the syntax of types. In contrast, an inductive definition would no longer work because types are defined coinductively.

Like static subtyping, materialization is preserved by type substitutions.

10.3 PROPOSITION: If $\tau_1 \sqsubseteq \tau_2$, then $\tau_1\sigma \sqsubseteq \tau_2\sigma$ for any type substitution σ. □

Proof: By definition of $\tau_1 \sqsubseteq \tau_2$, we have $T_1\sigma_1 = \tau_2$ for a T_1 such that $T_1^\dagger = \tau_1$ and a $\sigma_1 : \mathsf{FVar} \to \mathsf{GType}$.

Choose a $\sigma' : \mathsf{TVar} \to \mathsf{TFrame}$ such that, for every α, $(\alpha\sigma')^\dagger = \alpha\sigma$ and that $\mathsf{fvar}(\sigma') \cap \mathsf{dom}(\sigma_1) = \emptyset$.

Then we have $(T_1\sigma')^\dagger = \tau_1\sigma$ and therefore $T_1\sigma' \in \star(\tau_1\sigma)$.

Consider $\sigma_1' = [X\sigma_1\sigma/X]_{X\in\mathsf{dom}(\sigma_1)} \cup [?/X]_{X\in\mathsf{fvar}(\sigma')}$.

We have $T_1\sigma'\sigma_1' = T_1\sigma_1\sigma$ because:

- for every $\alpha \in \mathsf{var}(T_1)$, if $\alpha \in \mathsf{dom}(\sigma)$, then $\alpha\sigma'\sigma_1' = (\alpha\sigma')^\dagger = \alpha\sigma = \alpha\sigma_1\sigma$, and, if $\alpha \notin \mathsf{dom}(\sigma)$, then $\alpha\sigma'\sigma_1' = \alpha = \alpha\sigma_1\sigma$;
- for every $X \in \mathsf{var}(T_1)$, we must have $X \in \mathsf{dom}(\sigma_1)$ (otherwise, $T_1\sigma_1$ would not be a gradual type): then $X\sigma'\sigma_1' = X\sigma_1' = X\sigma_1\sigma$.

Since $T_1\sigma_1\sigma = \tau_2\sigma$, we have $\tau_1\sigma \sqsubseteq \tau_2\sigma$. □

10.2 *Subtyping on gradual set-theoretic types*

In Section 9.4 we defined the subtyping relation $\leq^?$ on gradual types by treating ? exactly like a type variable. This ensured that subtyping could not convert

between ? and static types (in contrast to the *consistent-subtyping* relation of other formalizations): that role is performed by materialization, and we want to keep the two separate.

We might be tempted to do the same here. Then, $\tau_1 \leq^? \tau_2$ would hold if and only if $T_1 \leq T_2$, where each T_i is obtained from the corresponding τ_i by replacing every occurrence of ? with a distinguished frame variable X°.

This relation is not satisfactory. Indeed, note that it would satisfy $? \setminus ? \leq^? \mathbb{0}$ (because $X^\circ \setminus X^\circ \leq \mathbb{0}$). As a consequence, combined with materialization, it would imply that the declarative type system can type *every* program, even fully static and nonsensical ones (inserting casts that always fail). This is because any type could be converted to any other: for example,

$$\mathsf{Int} \leq^? \mathsf{Int} \setminus (? \setminus ?) \sqsubseteq \mathsf{Int} \setminus (\mathsf{Int} \setminus ?) \leq^? \mathbb{0} \leq^? \mathsf{Bool}\ .$$

This is undesirable, of course: a gradual type system must reject programs that do not use ? and are ill-typed in a static type system.

This indicates that a well-behaved subtyping relation on gradual set-theoretic types cannot give a set-theoretic interpretation to ? directly, since that would make $? \setminus ?$ an empty type, which we do not want. To define subtyping, then, we keep our idea of replacing ? with type variables, but we take care to distinguish occurrences that appear below negation from those that do not. There are different ways to perform such a replacement.

Using discrimination, we could try to define subtyping as

$$\tau_1 \leq^? \tau_2 \iff \exists T_1 \in \star(\tau_1), T_2 \in \star(\tau_2).\ T_1 \leq T_2\ .$$

Of course, this has the same problem as using just one frame variable: $? \setminus ? \leq^? \mathbb{0}$ holds. So we need to restrict the possible choices of T_1 and T_2. To define subtyping, below, we will ask T_1 and T_2 to be *polarized*: by this we mean that no frame variable occurs in them both positively (under an even number of negations) and negatively (under an odd number of negations). This implies that, if τ_1 is $? \setminus ?$, we cannot choose as T_1 the type frame $X \setminus X$, but only a type frame with two distinct variables (for example, $X \setminus Y$); therefore, $? \setminus ? \leq^? \mathbb{0}$ does not hold.

In the remainder of this section, we define some terminology to describe the position of variables in types and characterize different particular discriminations of a gradual type. We use them to give several characterizations of subtyping. Then, we prove that they are all equivalent. These different characterizations are suitable to obtain different results; in particular, we use one to prove that subtyping commutes with materialization.

10.2.1 Polarity, parity, and variance

Given a type frame and an occurrence of a type or frame variable in it, we can represent the path from the root of the type frame to that occurrence of the variable as a string on the alphabet $\{\times_\mathsf{L}, \times_\mathsf{R}, \to_\mathsf{L}, \to_\mathsf{R}, \vee_\mathsf{L}, \vee_\mathsf{R}, \neg\}$ describing the constructors and connectives traversed along the path and the direction of traversal (to the left or to the right) for binary ones. For example, the path to

X in $(\mathsf{Int} \to X) \vee \mathsf{Bool}$ is $\vee_{\mathsf{L}}\to_{\mathsf{R}}$. A variable can have multiple occurrences in a type at different paths – even infinitely many of them, if the type is recursive. For example, there are infinitely many occurrences of α in the type T described by the equation $T = (\alpha \times T) \vee b$; their paths are all the strings described by the regular expression $(\vee_{\mathsf{L}}\times_{\mathsf{R}})^*\vee_{\mathsf{L}}\times_{\mathsf{L}}$.

We distinguish three characteristics of occurrences according to their path.

Polarity: an occurrence is *positive* if $\neg$ occurs an even number of times in its path; it is *negative* otherwise.[1]

Parity: an occurrence is *even* if $\to_{\mathsf{L}}$ occurs an even number of times in its path; it is *odd* otherwise.

Variance: an occurrence is *covariant* if it is both positive and even or both negative and odd; it is *contravariant* otherwise.

The notion of variance coincides with the normal notion of variance for subtyping: descending below a negation or to the left of an arrow flips the variance.

We introduce some notation to refer to the variables that occur in specific positions in a type frame. We write $\mathsf{var}^+(T)$, $\mathsf{var}^-(T)$, $\mathsf{var}^{\mathsf{cov}}(T)$, $\mathsf{var}^{\mathsf{cnt}}(T)$, $\mathsf{var}^{\mathsf{even}}(T)$, and $\mathsf{var}^{\mathsf{odd}}(T)$ to denote the sets of variables that have at least one occurrence in T in the specified position – respectively, positive, negative, covariant, contravariant, even, or odd. We use the same notation also for $\mathsf{tvar}(\cdot)$ and $\mathsf{fvar}(\cdot)$. All the notions here also apply to static and gradual types.

Given a type frame T, we say

- that T is *polarized* if no frame variable has both positive and negative occurrences in it, that is, if $\mathsf{fvar}^+(T) \cap \mathsf{fvar}^-(T) = \varnothing$;
- that T is *variance-polarized* if no frame variable has both covariant and contravariant occurrences in it, that is, if $\mathsf{fvar}^{\mathsf{cov}}(T) \cap \mathsf{fvar}^{\mathsf{cnt}}(T) = \varnothing$.

We write $\mathsf{TFrame}^{\mathsf{pol}}$ and $\mathsf{TFrame}^{\mathsf{var}}$, respectively, for the sets of polarized and variance-polarized type frames.

10.2.2 Subtyping using polarized discriminations

We define two subsets of the set $\star(\tau)$ of the discriminations of τ:

$$\star^{\mathsf{pol}}(\tau) \stackrel{\text{def}}{=} \star(\tau) \cap \mathsf{TFrame}^{\mathsf{pol}} \qquad \textit{polarized discriminations}$$

$$\star^{\mathsf{var}}(\tau) \stackrel{\text{def}}{=} \star(\tau) \cap \mathsf{TFrame}^{\mathsf{var}} \qquad \textit{variance-polarized discriminations.}$$

We use the first of these to define subtyping.

10.4 DEFINITION (Subtyping on gradual types): We define the *subtyping* relation $\leq^?$ and the *subtype equivalence* relation $\simeq^?$ on gradual types as:

$$\tau_1 \leq^? \tau_2 \iff \exists T_1 \in \star^{\mathsf{pol}}(\tau_1), T_2 \in \star^{\mathsf{pol}}(\tau_2).\ T_1 \leq T_2$$

$$\tau_1 \simeq^? \tau_2 \iff (\tau_1 \leq^? \tau_2) \wedge (\tau_2 \leq^? \tau_1)\,. \qquad \square$$

1 This notion of polarity is unrelated to the polarity of blame labels in the cast calculus and to the notions of positive and negative subtyping: it only concerns negation in types.

We could alternatively characterize subtyping using variance instead of polarity, having $\tau_1 \leq^? \tau_2$ hold if and only if

$$\exists T_1 \in \star^{\mathsf{var}}(\tau_1), T_2 \in \star^{\mathsf{var}}(\tau_2).\ T_1 \leq T_2\ .$$

We will prove that the two definitions are equivalent. The former is interesting because it makes it explicit that we only need to use distinct variables because of negation types. The latter, however, is more convenient to use for some proofs.

10.2.3 Avoiding existential quantification

The definition of subtyping could be computationally problematic because of the existential quantification. However, it turns out that we do not need to check every discrimination. It is enough to use the discrimination in which just two frame variables appear (thus eliminating the existential quantification): one to replace all positive occurrences of ? and another for all negative ones. Equivalently, one variable could be used for all covariant occurrences and another for all contravariant occurrences. We introduce some terminology to describe these alternative definitions.

In the following, we assume that X^1 and X^0 are two distinguished variables in FVar. We define four subsets of type frames as follows.

$$\mathsf{TFrame}^{\mathsf{pol1}} \stackrel{\text{def}}{=} \left\{ T \in \mathsf{TFrame} \,\middle|\, \mathsf{fvar}^+(T) \subseteq \{X^1\} \text{ and } \mathsf{fvar}^-(T) \subseteq \{X^0\} \right\}$$
$$\mathsf{TFrame}^{\mathsf{pol0}} \stackrel{\text{def}}{=} \left\{ T \in \mathsf{TFrame} \,\middle|\, \mathsf{fvar}^+(T) \subseteq \{X^0\} \text{ and } \mathsf{fvar}^-(T) \subseteq \{X^1\} \right\}$$
$$\mathsf{TFrame}^{\mathsf{var1}} \stackrel{\text{def}}{=} \left\{ T \in \mathsf{TFrame} \,\middle|\, \mathsf{fvar}^{\mathsf{cov}}(T) \subseteq \{X^1\} \text{ and } \mathsf{fvar}^{\mathsf{cnt}}(T) \subseteq \{X^0\} \right\}$$
$$\mathsf{TFrame}^{\mathsf{var1}} \stackrel{\text{def}}{=} \left\{ T \in \mathsf{TFrame} \,\middle|\, \mathsf{fvar}^{\mathsf{cov}}(T) \subseteq \{X^0\} \text{ and } \mathsf{fvar}^{\mathsf{cnt}}(T) \subseteq \{X^1\} \right\}$$

We refer to type frames in these sets as being, respectively, *strongly polarized*, *strongly negatively polarized*, *strongly variance-polarized*, and *strongly negatively variance-polarized*.

Given a gradual type τ, there is a unique type frame in $\star(\tau)$ that is strongly polarized; likewise for the other forms of polarization. We define notation to refer to such specific discriminations of a gradual type τ.

$\tau^{\oplus}$	*positive discrimination*	unique element of $\star(\tau) \cap \mathsf{TFrame}^{\mathsf{pol1}}$
$\tau^{\ominus}$	*negative discrimination*	unique element of $\star(\tau) \cap \mathsf{TFrame}^{\mathsf{pol0}}$
$\tau^{\oslash}$	*covariant discrimination*	unique element of $\star(\tau) \cap \mathsf{TFrame}^{\mathsf{var1}}$
$\tau^{\otimes}$	*contravariant discrimination*	unique element of $\star(\tau) \cap \mathsf{TFrame}^{\mathsf{var0}}$

We have the following equalities

$$\begin{aligned} ?^{\oplus} &= X^1 & ?^{\ominus} &= X^0 \\ \alpha^{\oplus} &= \alpha & \alpha^{\ominus} &= \alpha \\ b^{\oplus} &= b & b^{\ominus} &= b \\ (\tau_1 \times \tau_2)^{\oplus} &= \tau_1^{\oplus} \times \tau_2^{\oplus} & (\tau_1 \times \tau_2)^{\ominus} &= \tau_1^{\ominus} \times \tau_2^{\ominus} \\ (\tau_1 \to \tau_2)^{\oplus} &= \tau_1^{\ominus} \to \tau_2^{\oplus} & (\tau_1 \to \tau_2)^{\ominus} &= \tau_1^{\oplus} \to \tau_2^{\ominus} \\ (\tau_1 \vee \tau_2)^{\oplus} &= \tau_1^{\oplus} \vee \tau_2^{\oplus} & (\tau_1 \vee \tau_2)^{\ominus} &= \tau_1^{\ominus} \vee \tau_2^{\ominus} \\ (\neg\tau)^{\oplus} &= \neg(\tau^{\ominus}) & (\neg\tau)^{\ominus} &= \neg(\tau^{\oplus}) \\ \mathbb{0}^{\oplus} &= \mathbb{0} & \mathbb{0}^{\ominus} &= \mathbb{0} \end{aligned}$$

and similar equalities for $\tau^{\mathbb{O}}$ and $\tau^{\mathbb{V}}$, except that on the left of arrows we switch from $(\cdot)^{\mathbb{O}}$ to $(\cdot)^{\mathbb{V}}$.

Note that, for every T, we have:

$$\begin{aligned} T \in \mathsf{TFrame}^{\mathsf{pol1}} &\implies (T^{\dagger})^{\oplus} = T & T \in \mathsf{TFrame}^{\mathsf{pol0}} &\implies (T^{\dagger})^{\ominus} = T \\ T \in \mathsf{TFrame}^{\mathsf{var1}} &\implies (T^{\dagger})^{\mathbb{O}} = T & T \in \mathsf{TFrame}^{\mathsf{var0}} &\implies (T^{\dagger})^{\mathbb{V}} = T \end{aligned}$$

These definitions allow us to give several different characterizations of subtyping. We will prove that, for any τ_1 and τ_2, all the following statements are equivalent

$$\tau_1^{\oplus} \leq \tau_2^{\oplus} \qquad \tau_1^{\ominus} \leq \tau_2^{\ominus} \qquad \tau_1^{\mathbb{O}} \leq \tau_2^{\mathbb{O}} \qquad \tau_1^{\mathbb{V}} \leq \tau_2^{\mathbb{V}}$$

and that they are equivalent to subtyping as defined in Definition 10.4.

The equivalence of the first and second statements are straightforward: they are the same up to the type substitution switching X^1 and X^0; likewise for the equivalence of the third and the fourth. The equivalence between $\tau_1^{\oplus} \leq \tau_2^{\oplus}$ and $\tau_1^{\mathbb{O}} \leq \tau_2^{\mathbb{O}}$ is non-trivial, but the intuition is that it does not matter whether or not we switch between the two variables on the left of arrows, because subtyping never compares two subterms of the types unless they are to the left of the same number of arrows.

The equivalence between these notions and Definition 10.4 is tricky to establish. Clearly, $\tau_1^{\oplus} \leq \tau_2^{\oplus}$ implies $\exists T_1 \in \star^{\mathsf{pol}}(\tau_1), T_2 \in \star^{\mathsf{pol}}(\tau_2).\ T_1 \leq T_2$, because, for every τ, $\tau^{\oplus} \in \star^{\mathsf{pol}}(\tau)$. For the other direction, assume to have $T_1 \in \star^{\mathsf{pol}}(\tau_1)$ and $T_2 \in \star^{\mathsf{pol}}(\tau_2)$ such that $T_1 \leq T_2$; we want $\tau_1^{\oplus} \leq \tau_2^{\oplus}$. If no frame variable appears with opposite polarity in T_1 and T_2, then from T_1 and T_2 we can obtain $\tau_1^{\oplus}$ and $\tau_2^{\oplus}$ by applying a type substitution, so we conclude by Proposition 10.2. The difficulty is when some frame variables occur in positive position in T_1 and in negative position in T_2. For example, a variable X might occur positively in T_1 and negatively in T_2. Then, we cannot obtain $\tau_1^{\oplus}$ and $\tau_2^{\oplus}$ by applying a single substitution to T_1 and T_2: we will prove that we can apply two different substitutions while preserving subtyping.

These equivalences, which we show in the next section, are useful because they allow us to avoid quantification. In particular, they show that subtyping on gradual types can be computed using the same algorithm used for subtyping on static types, simply by performing a type substitution.

Note that, while for subtyping we do not need to consider quantification, the same does not hold for materialization, where we must consider discriminations using more variables (to allow, for instance $? \to ? \sqsubseteq \mathsf{Int} \to \mathsf{Bool}$). However, the problem is otherwise simpler because, rather than subtyping, it considers syntactic equality up to a single type substitution.

10.2.4 Equivalence of the different characterizations of subtyping

We introduce additional notation to refer to the variables in specific positions in type frames. We write $\mathsf{var}^{+\mathsf{cov}}(T)$, $\mathsf{var}^{+\mathsf{cnt}}(T)$, $\mathsf{var}^{-\mathsf{cov}}(T)$, and $\mathsf{var}^{-\mathsf{cnt}}(T)$ to denote the sets of variables that have at least one occurrence in T that is in *both* specified positions – respectively, both positive and covariant, both positive and contravariant, both negative and covariant, or both negative and contravariant. (We use these also for $\mathsf{tvar}(\cdot)$ and $\mathsf{fvar}(\cdot)$ and also for static and gradual types.)

For brevity, we will often write $\mathsf{var}(T_1, \dots, T_n)$ for $\mathsf{var}(T_1) \cup \dots \cup \mathsf{var}(T_n)$ and similarly for $\mathsf{fvar}(\cdot)$, $\mathsf{fvar}^+(\cdot)$, etc.

The following lemma and its corollaries state that, given a type frame (or a gradual type in the last corollary), we can obtain another type frame by renaming each occurrence of each variable in it according to the polarity, parity, and variance of the occurrence.

10.5 LEMMA: Let T be a type frame with $\mathsf{var}(T) = \{ A_i \mid i \in I \}$. There exists a type frame T' such that the four sets

$$\mathsf{var}^{+\mathsf{cov}}(T') \subseteq \{ A_i^{+\wedge} \mid i \in I \} \qquad \mathsf{var}^{+\mathsf{cnt}}(T') \subseteq \{ A_i^{+\vee} \mid i \in I \}$$
$$\mathsf{var}^{-\mathsf{cov}}(T') \subseteq \{ A_i^{-\wedge} \mid i \in I \} \qquad \mathsf{var}^{-\mathsf{cnt}}(T') \subseteq \{ A_i^{-\vee} \mid i \in I \}$$

are pairwise disjoint and that

$$T = T'\big([A_i/A_i^{+\wedge}]_{i\in I} \cup [A_i/A_i^{+\vee}]_{i\in I} \cup [A_i/A_i^{-\wedge}]_{i\in I} \cup [A_i/A_i^{-\vee}]_{i\in I}\big)\ . \qquad \square$$

Proof in appendix (p. 269).

10.6 COROLLARY: Let T be a type frame with $\mathsf{fvar}(T) = \{X_1, \dots, X_n\}$. There exists a type frame T', with $\mathsf{fvar}^{\mathsf{cov}}(T') \subseteq \{X_1, \dots, X_n\}$ disjoint from $\mathsf{fvar}^{\mathsf{cnt}}(T') \subseteq \{X'_1, \dots, X'_n\}$, such that $T = T'[X_i/X'_i]_{i=1}^n$. $\square$

Proof: Consequence of Lemma 10.5. We apply the lemma to find a type where type and frame variables are renamed according to their position (polarity and variance); then, we apply a substitution to unify the positions we do not want to distinguish. $\square$

10.7 COROLLARY: Let T be a type frame with $\mathsf{fvar}(T) = \{X_1, \dots, X_n\}$. There exists a type frame T', with $\mathsf{fvar}^{\mathsf{even}}(T') \subseteq \{X_1, \dots, X_n\}$ disjoint from $\mathsf{fvar}^{\mathsf{odd}}(T') \subseteq \{X'_1, \dots, X'_n\}$, such that $T = T'[X_i/X'_i]_{i=1}^n$. $\square$

Proof: Consequence of Lemma 10.5, similarly to Corollary 10.6. □

10.8 COROLLARY: Let τ be a gradual type with $\mathsf{var}(\tau) = \{\alpha_1, \ldots, \alpha_n\}$. There exists a gradual type τ', with $\mathsf{var}^+(\tau') \subseteq \{\alpha_1, \ldots, \alpha_n\}$ disjoint from $\mathsf{var}^-(\tau') \subseteq \{\alpha'_1, \ldots, \alpha'_n\}$, such that $\tau = \tau'[\alpha_i/\alpha'_i]_{i=1}^n$. □

Proof: Consequence of Lemma 10.5, similarly to Corollary 10.6. We first choose a T such that $T^\dagger = \tau$; then, we apply the lemma and a substitution to unify the positions that we do not need to distinguish; finally, we apply $\dagger$ to obtain a gradual type. □

The following lemma is one of the key ingredients for the proof of equivalence. Given a type frame T such that $T \not\leq \mathbb{0}$, we do not normally know whether $T[X/Y] \not\leq \mathbb{0}$ holds or not (conversely, if $T \leq \mathbb{0}$, then $T[X/Y] \leq \mathbb{0}$ holds by Proposition 10.2). However, if X and Y always occur with the same polarity in T, then we can prove that $T[X/Y] \not\leq \mathbb{0}$ must hold. For example, we have $X \setminus Y \not\leq \mathbb{0}$ and $X \setminus X \leq \mathbb{0}$, but X and Y occur with opposite polarity in $X \setminus Y$. When they occur with the same polarity (as in $X \wedge Y$ or $X \times Y$) the substitution $[X/Y]$ cannot make the type empty.

10.9 LEMMA:

$$\left.\begin{array}{l} T \not\leq \mathbb{0} \\ \text{either } \{X, Y\} \sharp \mathsf{fvar}^-(T) \text{ or } \{X, Y\} \sharp \mathsf{fvar}^+(T) \end{array}\right\} \implies T[X/Y] \not\leq \mathbb{0}$$

□

Proof in appendix (p. 270).

The following lemma is a consequence of the one above, proved by using the equivalence $T_1 \leq T_2 \iff T_1 \setminus T_2 \leq \mathbb{0}$ and the contrapositive of the result above.

10.10 LEMMA:

$$\left.\begin{array}{l} T_1 \leq T_2 \\ X \in \mathsf{fvar}^+(T_1) \implies X \notin \mathsf{fvar}^+(T_2) \\ X \in \mathsf{fvar}^-(T_1) \implies X \notin \mathsf{fvar}^-(T_2) \\ Y \sharp T_1, T_2, X \end{array}\right\} \implies T_1[Y/X] \leq T_2$$

□

Proof in appendix (p. 274).

We generalize the lemma above to consider more than two variables. This shows that, when some variables occur with one polarity in a type and the opposite polarity in the other, we can rename them in one of the types while preserving subtyping.

10.11 LEMMA:

$$\left.\begin{array}{l} T_1 \leq T_2 \\ \forall X \in \vec{X}.\ \begin{cases} X \in \mathsf{fvar}^+(T_1) \implies X \notin \mathsf{fvar}^+(T_2) \\ X \in \mathsf{fvar}^-(T_1) \implies X \notin \mathsf{fvar}^-(T_2) \end{cases} \\ \vec{Y} \sharp T_1, T_2, \vec{X} \end{array}\right\} \implies T_1[\vec{Y}/\vec{X}] \leq T_2$$

□

Proof: By induction on $\vec{X}$. If $\vec{X}$ is empty, there is nothing to prove.

Otherwise, we have $\vec{X} = X_0\vec{X}'$ and $\vec{Y} = Y_0\vec{Y}'$. By Lemma 10.10, we have $T_1[Y_0/X_0] \leq T_2$. Then, by IH, we have $T_1[Y_0/X_0][\vec{Y}'/\vec{X}'] \leq T_2$ and we conclude since $T_1[Y_0/X_0][\vec{Y}'/\vec{X}'] = T_1[\vec{Y}/\vec{X}]$. □

The following lemma is similar to Lemma 10.9. In that case, the condition under which the substitution does not make T empty is that the two variables occur with different parity in T (X is never even and Y never odd). We generalize this result too to multiple variables.

10.12 LEMMA:

$$\left.\begin{array}{l} T \not\leq \mathbb{0} \\ X \notin \mathsf{fvar}^{\mathsf{even}}(T) \\ Y \notin \mathsf{fvar}^{\mathsf{odd}}(T) \end{array}\right\} \implies T[X/Y] \not\leq \mathbb{0}$$

□

Proof in appendix (p. 275).

10.13 LEMMA:

$$\left.\begin{array}{l} T \not\leq \mathbb{0} \\ \vec{X} \sharp \mathsf{fvar}^{\mathsf{even}}(T) \\ \vec{Y} \sharp \mathsf{fvar}^{\mathsf{odd}}(T), \vec{X} \end{array}\right\} \implies T[\vec{X}/\vec{Y}] \not\leq \mathbb{0}$$

□

Proof: By induction on $\vec{X}$. If $\vec{X}$ is empty, there is nothing to prove.

Otherwise, we have $\vec{X} = X_0\vec{X}'$ and $\vec{Y} = Y_0\vec{Y}'$. By Lemma 10.12, we have $T[X_0/Y_0] \not\leq \mathbb{0}$. Then, by IH, we have $T[X_0/Y_0][\vec{X}'/\vec{Y}'] \not\leq \mathbb{0}$ and we conclude since $T[X_0/Y_0][\vec{X}'/\vec{Y}'] = T[\vec{X}/\vec{Y}]$. □

The next lemma, which relies on Lemma 10.12, proves that if T is empty then we can find a type frame T' which is also empty and in which no frame variable appears with both parities. The following lemma is a consequence of this; we use it to prove that the subtyping relation can be defined equivalently using polarity or variance.

10.14 LEMMA:

$$T \leq \mathbb{0} \implies \exists T', \vec{X}, \vec{Y}.\ \begin{cases} T' \leq \mathbb{0} \\ T = T'[\vec{X}/\vec{Y}] \\ \mathsf{fvar}^{\mathsf{even}}(T') \sharp \mathsf{fvar}^{\mathsf{odd}}(T') \end{cases}$$

□

Proof in appendix (p. 277).

10.15 LEMMA:

$$T_1 \leq T_2 \implies \exists T_1', T_2', \vec{X}, \vec{Y}.\ \begin{cases} T_1' \leq T_2' \\ T_1 = T_1'[\vec{X}/\vec{Y}] \\ T_2 = T_2'[\vec{X}/\vec{Y}] \\ \mathsf{fvar}^{\mathsf{even}}(T_1', T_2') \sharp \mathsf{fvar}^{\mathsf{odd}}(T_1', T_2') \end{cases}$$

□

Proof: Let $T = T_1 \setminus T_2$. We have $T \leq \mathbb{0}$ by definition of subtyping.

By Lemma 10.14, we find T', $\vec{X}$, and $\vec{Y}$ such that

$$T' \leq \mathbb{0} \qquad T = T'[\vec{X}/\vec{Y}] \qquad \mathsf{fvar}^{\mathsf{even}}(T') \sharp \mathsf{fvar}^{\mathsf{odd}}(T')\ .$$

Since T' is empty, it cannot be a type variable or a frame variable. Then, we must have $T' = T_1' \setminus T_2'$ for two types such that $T_1 = T_1'[\vec{X}/\vec{Y}]$ and $T_2 = T_2'[\vec{X}/\vec{Y}]$.

We have $T_1' \leq T_2'$ by definition of subtyping.

We have $\mathsf{fvar}^{\mathsf{even}}(T_1', T_2') = \mathsf{fvar}^{\mathsf{even}}(T')$ and $\mathsf{fvar}^{\mathsf{odd}}(T_1', T_2') = \mathsf{fvar}^{\mathsf{odd}}(T')$, therefore the two sets are disjoint. □

To relate the different definitions of subtyping, we define one more specific discrimination of gradual types. Let $X^{+\wedge}$, $X^{+\vee}$, $X^{-\wedge}$, and $X^{-\vee}$ be four distinguished variables in FVar. Given a gradual type τ, we define $\tau^\bullet$ as the unique type frame T such that $T^\dagger = \tau$, $\mathsf{fvar}^{+\mathsf{cov}}(T) \subseteq \{X^{+\wedge}\}$, $\mathsf{fvar}^{+\mathsf{cnt}}(T) \subseteq \{X^{+\vee}\}$, $\mathsf{fvar}^{-\mathsf{cov}}(T) \subseteq \{X^{-\wedge}\}$, and $\mathsf{fvar}^{-\mathsf{cnt}}(T) \subseteq \{X^{-\vee}\}$.

The following result is straightforward: if $T_1 \leq T_2$ holds for two discriminations of τ_1 and τ_2 which have distinct variables in different positions, then $\tau_1^\bullet \leq \tau_2^\bullet$ holds too. This is because we can obtain $\tau_i^\bullet$ from T_i by performing a type substitution.

10.16 LEMMA:

$$\left.\begin{array}{l} T_1 \leq T_2 \\ T_1^\dagger = \tau_1 \text{ and } T_2^\dagger = \tau_2 \\ \mathsf{fvar}^{+\mathsf{cov}}(T_1, T_2),\ \mathsf{fvar}^{+\mathsf{cnt}}(T_1, T_2),\ \mathsf{fvar}^{-\mathsf{cov}}(T_1, T_2), \\ \qquad \text{and } \mathsf{fvar}^{-\mathsf{cnt}}(T_1, T_2) \text{ are pairwise disjoint} \end{array}\right\} \implies \tau_1^\bullet \leq \tau_2^\bullet$$

□

Proof: We define

$$\sigma = [X^{+\wedge}/X]_{X\in\mathsf{fvar}^{+\mathsf{cov}}(T_1,T_2)} \cup [X^{+\vee}/X]_{X\in\mathsf{fvar}^{+\mathsf{cnt}}(T_1,T_2)} \cup [X^{-\vee}/X]_{X\in\mathsf{fvar}^{-\mathsf{cov}}(T_1,T_2)} \cup [X^{-\wedge}/X]_{X\in\mathsf{fvar}^{-\mathsf{cnt}}(T_1,T_2)} .$$

It is well defined because the four sets are disjoint. We have $T_1\sigma = \tau_1^\bullet$ and $T_2\sigma = \tau_2^\bullet$. We have $T_1\sigma \leq T_2\sigma$ by Proposition 10.2. □

Finally, we prove that the different notions of subtyping that we have proposed are all equivalent.

10.17 LEMMA: If $\tau_1 \leq^? \tau_2$, then $\tau_1^\bullet \leq \tau_2^\bullet$. □

Proof in appendix (p. 278).

10.18 LEMMA: Let τ_1 and τ_2 be two gradual types. Let $T_1 \in \star^{\mathsf{var}}(\tau_1)$ and $T_2 \in \star^{\mathsf{var}}(\tau_2)$ be such that $T_1 \leq T_2$. Then, $\tau_1^\bullet \leq \tau_2^\bullet$. □

Proof in appendix (p. 279).

10.19 PROPOSITION: Let τ_1 and τ_2 be two gradual types. The following statements are all equivalent:

$$\text{(A)}\ \tau_1 \leq^? \tau_2 \qquad \text{(B)}\ \tau_1^\oplus \leq \tau_2^\oplus \qquad \text{(C)}\ \tau_1^\ominus \leq \tau_2^\ominus$$
$$\text{(D)}\ \exists T_1 \in \star^{\mathsf{var}}(\tau_1), T_2 \in \star^{\mathsf{var}}(\tau_2).\ T_1 \leq T_2$$
$$\text{(E)}\ \tau_1^{\oslash} \leq \tau_2^{\oslash} \qquad \text{(F)}\ \tau_1^{\otimes} \leq \tau_2^{\otimes} \qquad \text{(G)}\ \tau_1^\bullet \leq \tau_2^\bullet$$ □

Proof: We have (A) $\implies$ (G) by Lemma 10.17 and (D) $\implies$ (G) by Lemma 10.18.

The equivalences (B) $\iff$ (C) and (E) $\iff$ (F) are shown trivially by Proposition 10.2 since, for every τ, we have $\tau^\oplus = \tau^\ominus[X^1/X^0, X^0/X^1]$ and $\tau^{\oslash} = \tau^{\otimes}[X^1/X^0, X^0/X^1]$

We can show (G) $\implies$ (B) $\wedge$ (E) by Proposition 10.2. If $\tau_1^\bullet \leq \tau_2^\bullet$, then $\tau_1^\bullet\sigma \leq \tau_2^\bullet\sigma$ holds for every type substitution σ. To show (B), we choose $\sigma = [X^1/X^{+\wedge}, X^1/X^{+\vee}, X^0/X^{-\wedge}, X^0/X^{-\vee}]$ and have $\tau_1^\bullet\sigma = \tau_1^\oplus$ and $\tau_2^\bullet\sigma = \tau_2^\oplus$. We prooced analogously to show (E).

The implication (B) $\implies$ (A) holds because, for any τ, $\tau^\oplus \in \star^{\mathsf{pol}}(\tau)$. Likewise for the implication (E) $\implies$ (D). All other implications follow by transitivity. □

10.2.5 Properties of subtyping

In this section we study some properties of subtyping on gradual types and of its interaction with materialization.

Subtyping on static types and type frames is preserved by type substitutions (Proposition 10.2). In contrast, subtyping on gradual types is not: we have

$\alpha \setminus \alpha \leq^? \mathbb{0}$ but $? \setminus ? \not\leq^? \mathbb{0}$, though $? \setminus ? = (\alpha \setminus \alpha)[?/\alpha]$. However, we can prove that subtyping on gradual types is preserved by *static* type substitutions, that is, by substitutions that map type variables to static types.

10.20 PROPOSITION: If $\tau_1 \leq^? \tau_2$ then, for any static type substitution σ, we have $\tau_1\sigma \leq^? \tau_2\sigma$. □

Proof: If $\tau_1 \leq^? \tau_2$, then by Proposition 10.19 we have $\tau_1^{\oplus} \leq \tau_2^{\oplus}$. Then, $\tau_1^{\oplus}\sigma \leq \tau_2^{\oplus}\sigma$ by Proposition 10.2. We have $\tau_1^{\oplus}\sigma = (\tau_1\sigma)^{\oplus}$ because $(\tau_1^{\oplus})^{\dagger} = \tau_1\sigma$ and because $\tau_1^{\oplus}\sigma$ is strongly polarized (since σ does not introduce frame variables). Similarly, we have $\tau_2^{\oplus}\sigma = (\tau_2\sigma)^{\oplus}$. Therefore, $\tau_1\sigma \leq^? \tau_2\sigma$. □

We show next that we can commute applications of subtyping and materialization to apply materialization first. This is interesting in order to study the inversion of the typing relation. To type expressions in the declarative type system, we can apply the rule $[\mathrm{T}_{\leq}]$ and $[\mathrm{T}_{\sqsubseteq}]$ as many times as and in whichever order we want. It is useful to show that this chain of applications can always be collapsed to one application of $[\mathrm{T}_{\sqsubseteq}]$ followed by one of $[\mathrm{T}_{\leq}]$.

We first prove an auxiliary result. When $\tau_1 \sqsubseteq \tau_2$, by definition of $\sqsubseteq$ we have $T\sigma = \tau_2$ for T and σ such that $T^{\dagger} = \tau_1$ and $\sigma\colon \mathsf{fvar}(T) \to \mathsf{GType}$. We prove that we can always choose T so that no frame variable has both covariant and contravariant occurrences in it.

10.21 LEMMA: If $\tau_1 \sqsubseteq \tau_2$, then there exist a T and a $\sigma\colon \mathsf{fvar}(T) \to \mathsf{GType}$ such that $T^{\dagger} = \tau_1$, that $T\sigma = \tau_2$, and that $\mathsf{fvar}^{\mathsf{cov}}(T) \nparallel \mathsf{fvar}^{\mathsf{cnt}}(T)$. □

Proof: By definition of $\tau_1 \sqsubseteq \tau_2$, there exist a T_1 and a $\sigma_1\colon \mathsf{FVar} \to \mathsf{GType}$ such that $T_1^{\dagger} = \tau_1$ and that $T_1\sigma_1 = \tau_2$. Let $\mathsf{fvar}(T_1) = \{X_1, \ldots, X_n\}$.

By Corollary 10.6, we can find a T such that $\mathsf{fvar}^{\mathsf{cov}}(T) \subseteq \{X_1, \ldots, X_n\}$ is disjoint from $\mathsf{fvar}^{\mathsf{cnt}}(T) \subseteq \{X'_1, \ldots, X'_n\}$ and such that $T_1 = T[X_i/X'_i]_{i=1}^n$. Clearly, $T^{\dagger} = T_1^{\dagger} = \tau_1$.

We take σ to be $[X_i\sigma_1/X_i]_{i=1}^n \cup [X_i\sigma_1/X'_i]_{i=1}^n$ restricted to $\mathsf{fvar}(T)$. We have:

$$T\sigma = T([X_i\sigma_1/X_i]_{i=1}^n \cup [X_i\sigma_1/X'_i]_{i=1}^n) = T[X_i/X'_i]_{i=1}^n\sigma_1 = T_1\sigma_1 = \tau_2\,. \qquad \square$$

We also give the following result on type substitutions. Given two type substitutions σ_1 and σ_2, we write $\sigma_1 \leq \sigma_2$ when, for every A, $A\sigma_1 \leq A\sigma_2$. When $\overline{A} \subseteq \mathsf{Var}$, we define $\sigma|_{\overline{A}}$ as the type substitution such that $A\sigma|_{\overline{A}} = A\sigma$ if $A \in \overline{A}$ and $A\sigma|_{\overline{A}} = A$ otherwise (as in Section 2.2.1). The following lemma states that $T\sigma_1 \leq T\sigma_2$ holds when $A\sigma_1 \leq A\sigma_2$ for every A that is covariant in T and $A\sigma_2 \leq A\sigma_1$ for every A that is contravariant in T.

10.22 PROPOSITION:

$$\forall T, \sigma_1, \sigma_2.\quad \left.\begin{array}{l} \sigma_1|_{\mathsf{var}^{\mathsf{cov}}(T)} \leq \sigma_2|_{\mathsf{var}^{\mathsf{cov}}(T)} \\ \sigma_2|_{\mathsf{var}^{\mathsf{cnt}}(T)} \leq \sigma_1|_{\mathsf{var}^{\mathsf{cnt}}(T)} \end{array}\right\} \implies T\sigma_1 \leq T\sigma_2$$

□

Proof in appendix (p. 279).

Now we show that materialization can always be applied before subtyping.

10.23 LEMMA: If $\tau_1 \leq^? \tau_2 \sqsubseteq \tau_3$, then, for some τ_2', we have $\tau_1 \sqsubseteq \tau_2' \leq^? \tau_3$. □

Proof in appendix (p. 281).

We write $\preccurlyeq^?$ for the preorder on gradual types that combines subtyping and materialization, defined inductively by the following rules.

$$\frac{}{\tau \preccurlyeq^? \tau} \qquad \frac{\tau_1 \leq^? \tau_2 \qquad \tau_2 \preccurlyeq^? \tau_3}{\tau_1 \preccurlyeq^? \tau_3} \qquad \frac{\tau_1 \sqsubseteq \tau_2 \qquad \tau_2 \preccurlyeq^? \tau_3}{\tau_1 \preccurlyeq^? \tau_3}$$

Then, we obtain the following corollary of the result above.

10.24 COROLLARY: If $\tau_1 \preccurlyeq^? \tau_2$, then there exists a type τ such that $\tau_1 \sqsubseteq \tau \leq^? \tau_2$. □

Proof: By induction on the derivation of $\tau_1 \preccurlyeq^? \tau_2$.

If $\tau_1 = \tau_2$, then $\tau_1 \sqsubseteq \tau_1 \leq^? \tau_2$.

If $\tau_1 \leq^? \tau' \preccurlyeq^? \tau_2$, then by IH we find τ'' such that $\tau' \sqsubseteq \tau'' \leq^? \tau_2$, by Lemma 10.23 we find τ such that $\tau_1 \sqsubseteq \tau \leq^? \tau''$, and finally (by transitivity of $\leq^?$) we have $\tau_1 \sqsubseteq \tau \leq^? \tau_2$.

If $\tau_1 \sqsubseteq \tau' \preccurlyeq^? \tau_2$, then by IH we find τ'' such that $\tau' \sqsubseteq \tau'' \leq^? \tau_2$, and (by transitivity of $\sqsubseteq$) we have $\tau_1 \sqsubseteq \tau'' \leq^? \tau_2$. □

There are two observations we can make about this corollary. One is that it justifies the constraint we use for variables:

$$\langle\!\langle x \colon t \rangle\!\rangle = \exists \alpha.\, (x \dot{\sqsubseteq} \alpha) \wedge (\alpha \dot{\leq} t)\,.$$

Our use of a materialization constraint and a subtyping constraint is justified by the fact that a typing derivation for a variable can always be reduced to the application of three rules: $[\mathrm{T}_x]$, $[\mathrm{T}_\sqsubseteq]$, and $[\mathrm{T}_\leq]$, in this order; instantiation, done by $[\mathrm{T}_x]$, is merged into materialization constraints. This result would therefore be useful to prove completeness of type inference (though we do not achieve completeness for other reasons that we will discuss).

Another observation is that this corollary proves that, for any static type t and any τ, if $t \preccurlyeq^? \tau$ then $t \leq \tau$. Using subtyping, we can go from static types to gradual types (e.g., $t \leq t \vee ?$), but then materialization on these gradual types is not useful, because subtyping could be used in its place. This is important because it shows that undesirable judgments like $\mathsf{Int} \preccurlyeq^? \mathsf{Bool}$, for example, do not hold (while it would if we did not consider polarity when we turn ? to frame variables). This ensures that the gradual type system still behaves like a static type system when no type annotation contains ?.

Finally, we prove a result analogous to Lemma 2.13: type substitutions that are pointwise equivalent according to $\simeq^?$ map the same type to equivalent types. This holds also if the substitutions are not static.

10.25 PROPOSITION: Let τ be a gradual type and σ_1 and σ_2 two substitutions such that $\forall \alpha \in \mathsf{var}(\tau).\ \alpha\sigma_1 \simeq^? \alpha\sigma_2$. Then, $\tau\sigma_1 \simeq^? \tau\sigma_2$. □

Proof in appendix (p. 281).

10.3 Source and cast languages

10.3.1 Syntax and typing

To add set-theoretic types to the source language and to the cast language, we do not need to change their syntax, except, of course, by allowing set-theoretic types in the syntax (wherever types appear: in annotations, casts, and type applications). The type system is also defined using the same rules as before (those of Figure 9.1 for the source language and those of Figure 9.5 for the cast language) except that we add the subsumption rule

$$[\mathrm{T}_{\leq}]\ \frac{\Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau}\ \tau' \leq^? \tau$$

using the subtyping relation of Definition 10.4. Compilation also stays the same as in Section 9.2.4 except that we add compilation for the rule $[\mathrm{T}_{\leq}]$ as follows.

$$[\mathrm{T}_{\leq}]\ \frac{\Gamma \vdash e \rightsquigarrow E : \tau'}{\Gamma \vdash e \rightsquigarrow E : \tau}\ \tau' \leq^? \tau$$

10.3.2 Semantics

The definition of the operational semantics is challenging, but here we just outline the main difficulties. The full definition is in Appendix B.2.

The addition of set-theoretic type connectives makes the form of casts more complicated. Previously, either a cast had ? as its source or target type (like $\langle ? \overset{p}{\Rightarrow} \mathsf{Int} \to \mathsf{Int} \rangle$ and $\langle \mathsf{Int} \to \mathsf{Int} \overset{p}{\Rightarrow} ? \rangle$) or it acted only under a type constructor (like $\langle \mathsf{Int} \to \mathsf{Int} \overset{p}{\Rightarrow} ? \to ? \rangle$). Here, casts can act under type connectives: for example, $\langle (\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool}) \overset{p}{\Rightarrow} (\mathsf{Int} \to \mathsf{Int}) \wedge ? \rangle$. The notion of ground type must be generalized to deal with such casts.

The reduction of applications and projections with casts is challenging. Consider the case of applications as an example. Without set-theoretic types, the reduction rule

$$\big(V\langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2' \rangle\big)\ V' \ \hookrightarrow\ \big(V\ (V'\langle \tau_1' \overset{\bar{p}}{\Rightarrow} \tau_1 \rangle)\big)\langle \tau_2 \overset{p}{\Rightarrow} \tau_2' \rangle$$

can be used to reduce the application of a value with a cast, splitting the cast into two casts, one on the argument and one on the result of the application. We can split a cast in this way only if both its source and its target types are arrows. With set-theoretic types, function types can be union or intersection of arrows, in which case the rule cannot be applied directly. To reduce an application $(V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle)\ V'$, we must compute two arrow types that approximate τ and τ',

to replicate the same construction of the rule above. This approximation is performed by an operator $\circ$, whose result depends on the cast and on the type of the argument V'.

The cast language satisfies the soundness and blame safety properties already stated in Section 9.2.3 for the cast language without set-theoretic types. Moreover, it is a conservative extension of the latter: indeed, the proof of soundness for the cast language of Section 9.2.3 follows by conservativity from the proof for this extension.

10.4 *Type inference*

We describe what must be changed to adapt the type inference system in Section 9.3 to set-theoretic types. The description of that system was meant to be extended here; this motivated some design choices, including the use of subtyping constraints. We must redefine type-constraint solving; on the other hand, the definition of constraint simplification remains unchanged.

10.4.1 Type constraints and solutions

We keep the same definition for type constraints except, of course, for the different definition of types. However, the conditions for a type substitution σ to be a solution of a type-constraint set D in Δ must be changed: subtyping constraints now require subtyping instead of equality. We now write $\sigma \Vdash_\Delta D$ to mean that:

- for every $(t_1 \mathrel{\dot{\leq}} t_2) \in D$, we have $t_1\sigma \leq^? t_2\sigma$;
- for every $(\tau \mathrel{\dot{\sqsubseteq}} \alpha) \in D$, we have $\tau\sigma \sqsubseteq \alpha\sigma$ and, for all $\beta \in \mathsf{var}(\tau)$, $\beta\sigma$ is a static type;
- $\mathsf{dom}(\sigma) \cap \Delta = \varnothing$.

10.4.2 Type-constraint solving

To solve type-constraint sets, we replace unification with an algorithm designed for set-theoretic types and semantic subtyping: the *tallying* algorithm of Castagna et al. (2015b), which we have already described in Section 4.3.1. Given a set $\overline{T^1 \mathrel{\dot{\leq}} T^2}$ of subtyping constraints between type frames, tallying computes a finite set Σ of type substitutions such that, for every $\sigma \in \Sigma$ and $(T^1 \mathrel{\dot{\leq}} T^2) \in \overline{T^1 \mathrel{\dot{\leq}} T^2}$, we have $T^1\sigma \leq T^2\sigma$. Tallying can compute a set containing more than one type substitution, because some constraints do not have a single type substitution that is their principal solution. Tallying verifies soundness and completeness properties described in Property 4.25.

We want to use tallying to define an algorithm to solve type constraints. Previously, we converted materialization constraints $(\tau \mathrel{\dot{\sqsubseteq}} \alpha)$ to equality constraints $(T \doteq \alpha)$ and used unification. To do the same here, we first need to extend tallying to handle such equality constraints. This is easy to do in our case by adding simple pre- and post-processing steps. We are only interested

in using this when the equality constraints are those we will generate from materialization constraints. Therefore, we give an algorithm $\mathsf{tally}^{\doteq}$ tailored to this situation, which fails unless certain conditions are satisfied. In practice, these conditions should never occur when solving the constraints we generate in our system. We do not prove this here, though, because the proof would only be needed to show completeness for type inference, and completeness does not hold anyway, as we will explain.

The algorithm $\mathsf{tally}^{\doteq}_{\Delta}(\{\,(t_i^1 \mathrel{\dot{\leq}} t_i^2) \mid i \in I\,\} \cup \{\,(T_j \doteq \alpha_j) \mid j \in J\,\})$ is the following.

1. If any of the following conditions holds, return $\varnothing$:
   - there exist $j_1, j_2 \in J$ such that $\alpha_{j_1} = \alpha_{j_2}$ and $T_{j_1} \neq T_{j_2}$;
   - there exist $j_1, j_2 \in J$ such that $\alpha_{j_1} \in \mathsf{var}(T_{j_2})$;
   - there exists $j \in J$ such that $\alpha_j \in \Delta$.
2. Compute $\Sigma = \mathsf{tally}_{\Delta}(\{\,(t_i^1[T_j/\alpha_j]_{j\in J} \mathrel{\dot{\leq}} t_i^2[T_j/\alpha_j]_{j\in J}) \mid i \in I\,\})$.
3. Return $\{\,\sigma_0 \cup [T_j\sigma_0/\alpha_j]_{j\in J} \mid \sigma_0 \in \Sigma\,\}$.

In step 1 the algorithm fails if some conditions are met. These should never occur when the algorithm is used for type inference, because α in a constraint $(\tau \mathrel{\dot{\sqsubseteq}} \alpha)$ (which will become $(T \doteq \alpha)$) is always chosen fresh. As anticipated, we do not prove this formally.

The algorithm works by inlining the equality constraints in the subtyping constraints and relying on tallying to find a solution. Then, in step 3, the solutions found by tallying are extended with mappings for the variables in the equality constraints. The union in step 3 is well defined because σ_0 is not defined on the α_j, since they do not appear in the input to tally.

The algorithm satisfies the following property.

10.26 PROPOSITION (Soundness of $\mathsf{tally}^{\doteq}$):

$$\forall \sigma \in \mathsf{tally}^{\doteq}_{\Delta}\left(\overline{t^1 \mathrel{\dot{\leq}} t^2} \cup \overline{T \doteq \alpha}\right).\quad \begin{cases} \forall (t^1 \mathrel{\dot{\leq}} t^2) \in \overline{t^1 \mathrel{\dot{\leq}} t^2}.\ t^1\sigma \leq t^2\sigma \\ \forall (T \doteq \alpha) \in \overline{T \doteq \alpha}.\ T\sigma = \alpha\sigma \\ \mathsf{dom}(\sigma) \subseteq \mathsf{var}\left(\overline{t^1 \mathrel{\dot{\leq}} t^2} \cup \overline{T \doteq \alpha}\right) \setminus \Delta \end{cases}$$

□

Proof in appendix (p. 282).

Using $\mathsf{tally}^{\doteq}$, we can define the version of solve for set-theoretic types following the same approach as before. However, there are two difficulties.

The main difficulty is the presence of recursive types and their behaviour with respect to materialization. Consider the recursive type defined by the equation $\tau = (? \times \tau) \vee b$, where b is some base type. It corresponds to the type of lists of elements of type ?, terminated by a constant in b. Since recursive types in our definition are infinite regular trees (and not finite trees with explicit binders), $\tau = (? \times \tau) \vee b$ and $\tau' = (? \times ((? \times \tau') \vee b)) \vee b$ denote exactly the same type. What types can τ materialize to? Clearly, both $\tau_1 = (\mathsf{Int} \times \tau_2) \vee b$

and $\tau_2 = (\mathsf{Int} \times ((\mathsf{Bool} \times \tau_2) \vee b)) \vee b$ are possible. Indeed, ? occurs infinitely many times in τ. Materialization could in principle allow us to change each occurrence to a different type. However, since types must be regular trees, only a finite number of occurrences can be replaced with different types (otherwise, the resulting tree would not be a gradual type). While finite, this number is unbounded.

Recall that step 1 of solve (in Section 9.3.2, p. 165) picked a discrimination T_j of each τ_j such that no frame variable appeared more than once in T_j. If we consider the recursive type τ above, there is no T such that $T^\dagger = \tau$ and that T has no repeated frame variables: it would need to have infinitely many frame variables and thus be non-regular. While we will never need infinitely many variables, we do not know in advance (in this pre-processing step) how many we will need.

A solution to this would be to change the tallying algorithm so that discrimination is performed during tallying. Then, it could be done lazily, introducing as many frame variables as needed. However, this sacrifices the modularity of our current approach.

Currently, we give a definition where no constraint is placed on how many frame variables are used to replace ?. Of course, a sensible choice is to use different variables as much as possible except for the infinitely many occurrences of ? in a recursive loop.

There is a second difficulty. For a subtyping constraint $(t_1 \mathrel{\dot\leq} t_2)$, a substitution σ computed by tallying ensures $t_1\sigma \leq t_2\sigma$. However, what we want is rather $(t_1\sigma)^\dagger \leq^? (t_2\sigma)^\dagger$. This does not necessarily hold unless the type frames $t_1\sigma$ and $t_2\sigma$ are polarized. For example, if the constraint is $(\alpha \mathrel{\dot\leq} \mathbb{0})$ and the substitution is $[(X \setminus X)/\alpha]$, we have $X \setminus X \leq \mathbb{0}$ but $? \setminus ? \not\leq^? \mathbb{0}$. We define solve so that it ensures polarization in these cases by adjusting the variable renaming step we already had.

Having described these differences, we can give the definition of the algorithm. Let D be of the form $\{ (t_i^1 \mathrel{\dot\leq} t_i^2) \mid i \in I \} \cup \{ (\tau_j \mathrel{\dot\sqsubseteq} \alpha_j) \mid j \in J \}$: then $\mathsf{solve}_\Delta(D)$ is defined as follows.

1. Let $\overline{T \doteq \alpha}$ be $\{ (T_j \doteq \alpha_j) \mid j \in J, \tau_j \neq \alpha_j \}$ where, for each $j \in J$, $T_j^\dagger = \tau_j$;
2. Compute $\Sigma = \mathsf{tally}^{\dot\leq}_\Delta\big(\{ (t_i^1 \mathrel{\dot\leq} t_i^2) \mid i \in I \} \cup \overline{T \doteq \alpha}\big)$;
3. Return $\{ (\sigma_0' \circ \sigma_0)^\dagger|_{\mathsf{TVar}} \mid \sigma_0 \in \Sigma \}$,
   where, for every $\sigma_0 \in \Sigma$, σ_0' is computed as follows:
   a. $\sigma_0' = [\vec\alpha'/\vec X] \cup [\vec X'/\vec\alpha]$
   b. $\overline{A} = \mathsf{var}_{\dot\sqsubseteq}(D)\sigma_0 \cup \bigcup_{i\in I} \left(\mathsf{var}^\pm(t_i^1\sigma_0) \cup \mathsf{var}^\pm(t_i^2\sigma_0)\right)$
   c. $\vec X = \mathsf{FVar} \cap \overline{A}$
   d. $\vec\alpha = \mathsf{var}(D) \setminus (\Delta \cup \mathsf{dom}(\sigma_0) \cup \overline{A})$
   e. $\vec\alpha'$ and $\vec X'$ are vectors of fresh variables

In step 3b, we write $\mathsf{var}^\pm(T)$ for $\mathsf{var}^+(T) \cap \mathsf{var}^-(T)$: the set of variables that have at least both positive and negative occurrences in T. A type frame T is

polarized when $\mathsf{var}^{\pm}(T) \cap \mathsf{FVar} = \emptyset$: the renaming substitution σ_0' is constructed to ensure this for all type frames $t_i^1\sigma_0\sigma_0'$ and $t_i^2\sigma_0\sigma_0'$.

The following result states soundness for solve.

10.27 PROPOSITION: If $\sigma \in \mathsf{solve}_\Delta(D)$, then $\sigma \Vdash_\Delta D$ and $\mathsf{dom}(\sigma) \subseteq \mathsf{var}(D)$. □

Proof in appendix (p. 282).

10.4.3 Structured constraints, generation, and simplification

The syntax of structured constraints can be kept unchanged except for the change in the syntax of types. Constraint generation is also unchanged. Constraint simplification still uses the same rules, but it relies on the new solve algorithm. Soundness still holds, with the same statement as Theorem 9.24.

> Let $\mathcal{D}$ be a derivation of $\Gamma; \mathsf{var}(e) \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$. Let σ be a type substitution such that $\sigma \Vdash_{\mathsf{var}(e)} D$. Then, we have $\Gamma\sigma \vdash e \rightsquigarrow \{\!|e|\!\}_\sigma^{\mathcal{D}} : t\sigma$.

However, completeness no longer holds. The main obstacle to completeness is the aforementioned problem with materialization of ? in recursive types. Therefore, the first step to attempt to recover completeness for inference would be to study how to change the solve algorithm to make it complete. This probably requires a modification of tallying to handle materialization directly. In that case, $\mathsf{tally}^{\doteq}$ would no longer be needed; indeed, while sufficient for our purpose here, its awkward definition would complicate a proof of completeness because it relies very much on the specifics of the constraints we generate.

There is one further obstacle to achieve completeness. In this presentation, we have used the same general structure for type inference for subtyping as without. However, this means that a proof of completeness would run into the same problems as those described in Sections 4.1.1 and 5.3.1. Therefore, while we have chosen here a uniform presentation, the best path towards completeness is probably to adapt the work in Part I for this setting.

10.4.4 Soundness of type inference

Here we develop the proof of soundness. The intermediate results we need are mostly the same as in Section 9.3.5. We begin with standard properties of stability under type substitutions and of weakening for declarative typing and compilation.

10.28 LEMMA (Stability of typing under type substitution): If $\Gamma \vdash e \rightsquigarrow E : \tau$, then, for every static type substitution σ, we have $\Gamma\sigma \vdash e\sigma \rightsquigarrow E\sigma : \tau\sigma$. □

Proof in appendix (p. 284).

Given two type schemes S_1 and S_2, we write $S_1 \leq^? S_2$ when the schemes have the same quantified variables and their types are in the subtyping relation: that is, $\forall\vec{\alpha}.\,\tau_1 \leq^? \forall\vec{\alpha}.\,\tau_2$ if and only if $\tau_1 \leq^? \tau_2$. We write $\Gamma_1 \leq^? \Gamma_2$ when $\mathsf{dom}(\Gamma_1) = \mathsf{dom}(\Gamma_2)$ and, for all $x \in \mathsf{dom}(\Gamma_1)$, $\Gamma_1(x) \leq^? \Gamma_2(x)$.

10.29 LEMMA (Weakening): Let Γ_1 and Γ_2 be two type environments such that $\Gamma_1 \leq^? \Gamma_2$. If $\Gamma_2 \vdash e \rightsquigarrow E : \tau$, then $\Gamma_1 \vdash e \rightsquigarrow E : \tau$. □

Proof in appendix (p. 285).

We prove the following six auxiliary results and finally the proof of soundness of type inference (Theorem 10.36). The statements and general proof technique correspond closely to those in Section 9.3.5. For the first lemma, note that the hypothesis $\mathsf{static}(\sigma', \mathsf{var}(D)\sigma)$ is important because subtyping is only preserved by type substitutions that map type variables to static types.

10.30 LEMMA: Let σ and σ' be two type substitutions such that $\sigma \Vdash_\Delta D$ and $\mathsf{static}(\sigma', \mathsf{var}(D)\sigma)$. If $(t_1 \mathrel{\dot\leq} t_2) \in D$, then $t_1\sigma\sigma' \leq^? t_2\sigma\sigma'$. □

Proof: By definition of $\sigma \Vdash_\Delta D$, we have $t_1\sigma \leq^? t_2\sigma$. Since $\mathsf{var}(t_1) \cup \mathsf{var}(t_2) \subseteq \mathsf{var}(D)$, we have $\mathsf{var}(t_1\sigma) \cup \mathsf{var}(t_2\sigma) \subseteq \mathsf{var}(D)\sigma$. Because $\mathsf{static}(\sigma', \mathsf{var}(D)\sigma)$, the restriction of σ' to $\mathsf{var}(t_1\sigma) \cup \mathsf{var}(t_2\sigma)$ is a static substitution. By Proposition 10.20, $t_1\sigma\sigma' \leq^? t_2\sigma\sigma'$. □

10.31 LEMMA: Let σ and σ' be two type substitutions. If $\sigma \Vdash_\Delta D$ and $(\tau \mathrel{\dot\sqsubseteq} \alpha) \in D$, then $\tau\sigma\sigma' \sqsubseteq \alpha\sigma\sigma'$. □

Proof: By definition of $\sigma \Vdash_\Delta D$, we have $\tau\sigma \sqsubseteq \alpha\sigma$. Then, $\tau\sigma\sigma' \sqsubseteq \alpha\sigma\sigma'$ follows by Proposition 10.3. □

10.32 LEMMA: Let $\mathcal{D}$ be a derivation of $\Gamma; \Delta \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$. Then:

- if $e = x$, then $\Gamma(x) = \forall\vec{\alpha}.\,\tau$ and $D = \{(\tau[\vec{\beta}/\vec{\alpha}] \mathrel{\dot\sqsubseteq} \alpha), (\alpha \mathrel{\dot\leq} t)\}$ (for some τ, α, $\vec{\alpha}$, $\vec{\beta}$);
- if $e = c$, then $D = \{b_c \mathrel{\dot\leq} t\}$;
- if $e = \lambda x.\,e'$, then $\mathcal{D}$ contains a sub-derivation of $(\Gamma, x : \alpha_1); \Delta \vdash \langle\!\langle e' : \alpha_2 \rangle\!\rangle \rightsquigarrow D'$, and $D = D' \cup \{(\alpha_1 \mathrel{\dot\sqsubseteq} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot\leq} t)\}$;
- if $e = \lambda x : \tau.\,e'$, then $\mathcal{D}$ contains a sub-derivation of $(\Gamma, x : \tau); \Delta \vdash \langle\!\langle e' : \alpha_2 \rangle\!\rangle \rightsquigarrow D'$, and $D = D' \cup \{(\tau \mathrel{\dot\sqsubseteq} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot\leq} t)\}$;
- if $e = e_1 e_2$, then $\mathcal{D}$ contains two sub-derivations of $\Gamma; \Delta \vdash \langle\!\langle e_1 : \alpha \to t \rangle\!\rangle \rightsquigarrow D_1$ and $\Gamma; \Delta \vdash \langle\!\langle e_2 : \alpha \rangle\!\rangle \rightsquigarrow D_2$ (for some α, D_1, and D_2), and $D = D_1 \cup D_2$;
- if $e = (e_1, e_2)$, then $\mathcal{D}$ contains two sub-derivations of $\Gamma; \Delta \vdash \langle\!\langle e_1 : \alpha_1 \rangle\!\rangle \rightsquigarrow D_1$ and $\Gamma; \Delta \vdash \langle\!\langle e_2 : \alpha_2 \rangle\!\rangle \rightsquigarrow D_2$ (for some α_1, α_2, D_1, and D_2), and $D = D_1 \cup D_2 \cup \{\alpha_1 \times \alpha_2 \mathrel{\dot\leq} t\}$;

- if $e = \pi_i e'$, then $\mathcal{D}$ contains a sub-derivation of $\Gamma; \Delta \vdash \langle\!\langle e' : \alpha_1 \times \alpha_2 \rangle\!\rangle \rightsquigarrow D'$, and $D = D' \cup \{\alpha_i \mathrel{\dot{\leq}} t\}$;
- if $e = (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2)$, then $\mathcal{D}$ contains two sub-derivations of $\Gamma; \Delta \cup \vec{\alpha} \vdash \langle\!\langle e_1 : \alpha \rangle\!\rangle \rightsquigarrow D_1$ and $(\Gamma, x : \forall \vec{\alpha}, \vec{\beta}.\, \alpha\sigma_1); \Delta \vdash \langle\!\langle e_2 : t \rangle\!\rangle \rightsquigarrow D_2$, and the following hold:

$$D = D_2 \cup \mathsf{equiv}(\sigma_1, D_1) \qquad \sigma_1 \in \mathsf{solve}_{\Delta \cup \vec{\alpha}}(D_1)$$
$$\vec{\alpha} \sharp \mathsf{var}(\Gamma\sigma_1) \qquad \vec{\beta} = \mathsf{var}(\alpha\sigma_1) \setminus (\mathsf{var}(\Gamma\sigma_1) \cup \vec{\alpha} \cup \mathsf{var}(e_1)) \qquad \square$$

Proof: Straightforward, since the constraint simplification rules are syntax-directed. □

10.33 LEMMA: If $\Gamma; \Delta \vdash C \rightsquigarrow D$, then $\mathsf{var}(\Gamma) \cap \mathsf{var}(D) \subseteq \mathsf{var}(C) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)$. □

Proof in appendix (p. 286).

10.34 LEMMA:

$$\forall \Gamma, \Delta, e, \alpha, D, \sigma.\ \left.\begin{array}{l} \Gamma; \Delta \vdash \langle\!\langle e : \alpha \rangle\!\rangle \rightsquigarrow D \\ \sigma \in \mathsf{solve}_\Delta(D) \\ \mathsf{var}(e) \subseteq \Delta \\ \alpha \notin \mathsf{var}(\Gamma) \end{array}\right\} \implies \mathsf{static}(\sigma, \mathsf{var}(\Gamma))$$

□

Proof in appendix (p. 287).

10.35 LEMMA:

$$\forall \Gamma, \Delta, D_1, \sigma_1, \rho, \sigma, \sigma'.\ \left.\begin{array}{l} \sigma \Vdash_\Delta \mathsf{equiv}(\sigma_1, D_1) \\ \mathsf{dom}(\rho) \sharp \Gamma\sigma_1 \\ \mathsf{static}(\sigma', \mathsf{var}(\mathsf{equiv}(\sigma_1, D_1))\sigma) \\ \mathsf{static}(\sigma_1, \mathsf{var}(\Gamma)) \end{array}\right\} \implies \Gamma\sigma\sigma' \leq^? \Gamma\sigma_1\rho\sigma\sigma'$$

□

Proof in appendix (p. 287).

10.36 THEOREM (Soundness of type inference): Let $\mathcal{D}$ be a derivation of $\Gamma; \mathsf{var}(e) \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$. Let σ be a type substitution such that $\sigma \Vdash_{\mathsf{var}(e)} D$. Then, we have $\Gamma\sigma \vdash e \rightsquigarrow \{\!|e|\!\}^{\mathcal{D}}_{\sigma} : t\sigma$. □

Proof in appendix (p. 288).

11 Discussion

The original goal of the work described in this part of the thesis was to combine polymorphic gradual typing and set-theoretic types. The difficulty lies in the intrinsic differences between the two: gradual typing is essentially syntactic ("?" is a syntactic placeholder), while subtyping for set-theoretic types is defined using a semantic-oriented interpretation of types. To overcome this discrepancy, we have sought to interpret gradual types indirectly, using the operation of discrimination, so that we could rely on the existing interpretation of static types. Discrimination is a key ingredient of our approach because we use it to define subtyping, materialization, and even type inference; for the latter, it allows us to reuse existing algorithms for constraint solving.

Finally, our approach led us to realize that gradual typing could be perceived and captured neatly by a subsumption-like rule using the preorder on types that we refer to as materialization. Since this preorder is orthogonal to subtyping, the two can be coupled in a type system without much interference (but a lot of interplay). Despite our new definition, materialization was already well known by several names (less or equally informative, precision, naive subtyping). However, it had never been singled out in a dedicated structural rule. We have done so, and thereby we have demonstrated how adding the rule $[\mathrm{T}_{\sqsubseteq}]$ alone is enough to endow a declarative type system with graduality. We believe that this declarative formulation is a valuable contribution to the understanding of gradual typing and complements the algorithmic systems that were the focus of previous work. As an example, materialization gives a new meaning to the cast calculus: its expressions encode the proofs of the declarative systems, and casts spot the places where $[\mathrm{T}_{\sqsubseteq}]$ was used.

That said, it is not all a bed of roses. Despite this novel presentation, subtyping and type inference for gradual set-theoretic types rely on long and tedious proofs that have to deal with the syntax of types. The same applies to the definition of the semantics for the cast language, which we have omitted from this presentation. Nevertheless, we believe that our declarative formalization makes graduality more intelligible and that our work raises new questions and opens fresh perspectives. We discuss at the end of this chapter two directions for future work that are particularly relevant for the topics discussed in this thesis.

11.1 Related work

The contributions of this work include the replacement of consistency with materialization to define gradual type systems and the integration of gradual typing with set-theoretic types (intersection, union, negation, recursion) and Hindley-Milner polymorphism (with type inference). The integration of all of

these features is novel, but prior work has studied the combination of subsets of these features.

Castagna and Lanvin (2017) study the combination of gradual typing with set-theoretic types, but without polymorphism. They employ the approach of Garcia, Clark, and Tanter (2016) that uses abstract interpretation to guide the design of the operations on types. Compared to the work of Castagna and Lanvin (2017), our work adds Hindley-Milner polymorphism with type inference and gives a new operational semantics that includes blame tracking and better lines up with the prior work on gradual typing. Ortin and García (2011) also investigate the combination of intersection and union types with gradual typing, but without higher-order functions and polymorphism. Toro and Tanter (2017) introduce a new kind of union type inspired by gradual typing, that provides implicit downcasts from a union to any of its constituent types. There is some overlap in the intended use cases of these gradual union types and our design, though there are considerable differences as well, given that our work handles polymorphism and the full range of set-theoretic types. A similar overlap exists with the work by Jafery and Dunfield (2017), who introduce gradual sum types, yet, with the same kind of limitations as Toro and Tanter (2017). Ângelo and Florido (2018) study the combination of gradual typing and intersection types, but in a somewhat limited form, as the design does not support subtyping or the other set-theoretic types.

As discussed in Chapter 8, Siek and Vachharajani (2008) showed how to do unification-based inference in a gradually typed language. Garcia and Cimini (2015) took this a step further, providing inference for Hindley-Milner polymorphism and proving that their algorithm yields principal types. The present work builds on this prior work and contributes the additional insight that a special-purpose constraint solver is not needed to handle gradual typing, but an off-the-shelf unification algorithm can be used in combination of some pre- and post-processing of the solution. In another line of work, Rastogi, Chaudhuri, and Hosmer (2012) develop a flow-based type inference algorithm for ActionScript to facilitate type specialization and the removal of runtime checks as part of their optimizing compiler. Campora et al. (2017) improve the support for migrating from dynamic to static typing by integrating gradual typing with variational types. They define a constraint-based type inference algorithm that accounts for the combination of these two features.

The combination of gradual typing with subtyping has been studied by many authors in the context of object-oriented languages. Siek and Taha (2007) showed how to augment an object calculus with gradual typing. Their declarative type system uses consistency in the elimination rules and has a subsumption rule to support subtyping. Their algorithmic type system combines consistency and subtyping into a single relation, consistent-subtyping. Many subsequent works adapted consistent-subtyping to different settings (Ina and Igarashi, 2011; Bierman, Abadi, and Torgersen, 2014; Swamy et al., 2014; Maidl, Mascarenhas, and Ierusalimschy, 2014; Garcia, Clark, and Tanter, 2016; Lehmann and Tanter, 2017; Xie, Bi, and Oliveira, 2018).

Ours is not the first line of work that tries to attack the syntactic hegemony currently ruling the gradual types community. The first and, alas hitherto unique, other example of this is the already cited work of Garcia, Clark, and Tanter (2016) on "Abstracting Gradual Typing" (AGT) (and its several follow-ups) which was a source of inspiration both for our work and for Castagna and Lanvin (2017). AGT uses abstract interpretation to relate gradual types to sets of static types. This is done via two functions: a *concretization* function that maps a gradual type τ into the set of static types obtained by replacing static types for all occurrences of ? in τ; an *abstraction* function that maps a set of static types to the gradual type whose concretization best approximates the set. Like AGT, we map gradual types to sets of static types, although they are different from those obtained by concretization, since we use type variables rather than arbitrary static types. As long as only concretization is involved, we can follow and reproduce the AGT approach in ours: (1) AGT concretizations of a type τ can be defined in our system as the set of static types to which τ can materialize; (2) this definition can be used to give a different characterization of the AGT consistency relation; and (3) by using that characterization we can show consistency to be decidable, define consistent-subtyping, and show that the problem of deciding consistent-subtyping in AGT reduces in linear time to deciding semantic subtyping. But then it is not possible to follow the approach further, because the AGT definition of the abstraction function is inherently syntactic and, thus, is unfit to handle type connectives whose definition is fundamentally of semantic nature. In other terms, we have no idea about whether – let alone how – AGT could handle set-theoretic types and this is why we had to find new characterizations of constructions that in AGT are smoothly obtained by a simple application of the abstraction function.

11.2 Future work

This work lays a foundation for integrating gradual typing and polymorphic set-theoretic types. As such, it opens new questions and issues. There are two main issues that it would be important to address in the future.

TYPE INFERENCE WITH SET-THEORETIC TYPES: In the description of type inference in Section 10.4, we have tried to rely on the existing algorithm for tallying, defining solve by adding pre- and post-processing steps to it. This is not appropriate to handle recursive types, as we have discussed. Therefore, it would be interesting to study how to extend tallying with materialization constraints in order to obtain a complete algorithm for type-constraint solving.

This would be an important step towards achieving completeness for type inference as a whole. However, the difficulties that we have described for inference without gradual typing – the treatment of generalization (Section 4.1.1) and of explicit polymorphism from annotations (Section 5.3.1) – exist here too, though we have not met them because we have only proven soundness. We have chosen to describe constraint simplification in a uniform way both without and with subtyping. However, to obtain a more robust description

and possibly achieve completeness, we should reframe the type system in the reformulated form of Section 4.1 and define constraint simplification following the definition in Section 4.3.

INTERSECTION TYPES FOR FUNCTIONS: We have not included in our type system an intersection-introduction rule like $[\mathrm{T}_\wedge]$ in Chapter 3. This was an early design choice of this work, motivated by several reasons. The presence of such a rule would complicate the dynamic semantics of the cast calculus (see Castagna and Lanvin (2017), where this restriction is not present), especially when combined with a typecase construct, which we need to use intersection types for overloading as in Part I. Moreover, in the study of type inference, we would have considered the restriction of the system without $[\mathrm{T}_\wedge]$ anyway.

The drawback is, of course, that function types are not as expressive as they could be. For instance, consider the type deduced for mymap in Chapter 8:

$$\mathsf{Bool} \to (\alpha \to \beta) \to ((\alpha\ \mathsf{array} \vee \alpha\ \mathsf{list}) \wedge ?) \to (\beta\ \mathsf{array} \vee \beta\ \mathsf{list})\ .$$

This type is not completely satisfactory: it does not capture the precise correlation between input and output. As a matter of fact, the following program (which transforms lists into arrays and vice versa) would get the same type as mymap:

```
let mymap2 (condition) (f) (x: (α array ∨ α list) ∧ ?) =
  if condition then Array.to_list (Array.map f x) else Array.of_list (List.map f x)
```

We plan to study how to add typecases to the language and intersection introduction to the type system so that this restriction can be removed and that we can derive intersection types at least for annotated functions. Then, (an annotated version of) mymap could be given the type

$$\mathsf{Bool} \to (\alpha \to \beta) \to \big((\alpha\ \mathsf{array} \wedge ?) \to \beta\ \mathsf{array}\big) \wedge \big((\alpha\ \mathsf{list} \wedge ?) \to \beta\ \mathsf{list}\big)$$

whereas mymap2 would have the different type

$$\mathsf{Bool} \to (\alpha \to \beta) \to \big((\alpha\ \mathsf{array} \wedge ?) \to \beta\ \mathsf{list}\big) \wedge \big((\alpha\ \mathsf{list} \wedge ?) \to \beta\ \mathsf{array}\big)\ .$$

Part III

Non-strict languages

12 Introduction

Semantic subtyping has been developed for languages with strict, call-by-value semantics. The type systems described in previous work and in the first two parts of this thesis (for instance, the system of Chapter 3) are unsound for non-strict languages. In this part, we show how to adapt the semantic subtyping approach to obtain soundness for non-strict semantics – specifically, for call-by-need.

To do so, we introduce an explicit representation for divergence in the types: a type $\bot$ which is distinct from the type $\mathbb{0}$ associated to diverging expressions in call-by-value semantic subtyping. We modify the type system so that it keeps track of divergence, albeit with a very coarse approximation. As a result, we recover soundness while maintaining much of the behaviour of subtyping from the call-by-value case.

In this chapter, we show why existing type systems with semantic subtyping are unsound for non-strict languages. Then, we introduce the approach we use to design a sound type system, and we motivate our choice of studying call-by-need instead of call-by-name.

12.1 Semantic subtyping for non-strict languages

This work started as an attempt to design a type system for the Nix Expression Language (Dolstra and Löh, 2008), an untyped, purely functional, and lazily evaluated language for Unix/Linux package management. Since Nix is untyped, some programming idioms it encourages require advanced type system features to be analyzed properly. Notably, the possibility of writing functions that use type tests to have an overloaded-like behaviour made intersection types and semantic subtyping a good fit for the language. However, existing semantic subtyping relations are unsound for non-strict semantics; this was already observed by Frisch, Castagna, and Benzaken (2008) and no adaptation has been proposed later.

Current semantic subtyping systems are unsound for non-strict semantics because of how they deal with the bottom type $\mathbb{0}$. The type $\mathbb{0}$ corresponds to the empty set of values; accordingly, we have $[\![\mathbb{0}]\!] = \varnothing$ (cf. Section 2.1). The intuition is that a reducible expression e can be safely given a type t only if all results (i.e., values) it can return are of type t. Thus, $\mathbb{0}$ can only be assigned to expressions that are statically known to diverge (i.e., that never return a result). For example, the ML expression `let rec f x = f x in f ()` can be given type $\mathbb{0}$. Let us use $\bar{e}$ to denote any diverging expression that, like this, can be given type $\mathbb{0}$. Consider the following typing derivations, which are valid in current

semantic subtyping systems (for example, the system of Chapter 3).

$$\dfrac{\dfrac{\vdash (\bar{e}, 3) : \mathbb{0} \times \mathsf{Int}}{\vdash (\bar{e}, 3) : \mathbb{0} \times \mathsf{Bool}}\,[\simeq]}{\vdash \pi_2\,(\bar{e}, 3) : \mathsf{Bool}} \qquad \dfrac{\dfrac{\vdash \lambda x.\,3 : \mathbb{0} \to \mathsf{Int}}{\vdash \lambda x.\,3 : \mathbb{0} \to \mathsf{Bool}}\,[\simeq] \qquad \vdash \bar{e} : \mathbb{0}}{\vdash (\lambda x.\,3)\,\bar{e} : \mathsf{Bool}}$$

Both $\pi_2\,(\bar{e}, 3)$ and $(\lambda x.\,3)\,\bar{e}$ diverge in call-by-value semantics (since $\bar{e}$ must be evaluated first), while they both reduce to 3 in call-by-name or call-by-need. The derivations are therefore sound for call-by-value, while they are clearly unsound with non-strict evaluation.

Why are these derivations valid? The crucial steps are those marked with $[\simeq]$, which convert between types that have the same interpretation. With semantic subtyping (as defined in Chapter 2, for example), $\mathbb{0} \times \mathsf{Int} \simeq \mathbb{0} \times \mathsf{Bool}$ holds because all types of the form $\mathbb{0} \times t$ are equivalent to $\mathbb{0}$ itself: none of these types contains any value (indeed, product types are interpreted as Cartesian products and therefore the product with the empty set is itself empty). The equivalence $\mathbb{0} \to \mathsf{Int} \simeq \mathbb{0} \to \mathsf{Bool}$ holds too. Intuitively, we interpret a type $t_1 \to t_2$ as the set of functions which, on arguments of type t_1, either diverge or return results in type t_2. There are no arguments of type $\mathbb{0}$ (because, in call-by-value, arguments are always values); hence, all types of the form $\mathbb{0} \to t$ are equivalent: they all contain every well-typed function. (As we have discussed in Chapter 2, arrow types are not really interpreted as sets of functions, but the actual interpretation behaves as if they were.)

12.2 Our approach

The intuition behind our solution is that, with non-strict semantics, it is not appropriate to see a type as the set of the values that have that type. In a call-by-value language, operations like application or projection occur on values: thus, we can identify two types (and, in some sense, the expressions they type) if they contain (and their expressions may produce) the same values. In non-strict languages, though, operations also occur on partially evaluated results: these, like $(\bar{e}, 3)$ in our example, can contain diverging sub-expressions below their top-level constructor.

As a consequence, it is unsound, for example, to type $(\bar{e}, 3)$ as $\mathbb{0} \times \mathsf{Int}$ and at the same time to have $\mathbb{0} \times \mathsf{Int} \simeq \mathbb{0} \times \mathsf{Bool}$. It is also unsound to have a notion of subtyping on arrow types that assumes implicitly that every argument to a function must be a value.

One approach to solve this problem would be to change the interpretation of $\mathbb{0}$ so that it is non-empty. However, the existence of types with an empty interpretation is important for the internal machinery of semantic subtyping. Notably, the decision procedure for subtyping relies on them (checking whether $t_1 \leq t_2$ holds is reduced to checking whether the type $t_1 \wedge \neg t_2$ is empty). Therefore, we keep the interpretation $[\![\mathbb{0}]\!] = \varnothing$, but we change the type system so that this type is *never* derivable, not even for diverging expressions. We keep it as a purely "internal" type useful to describe subtyping, but never used to type expressions.

We introduce instead a separate type $\bot$ as the type of diverging expressions. This type is non-empty but disjoint from the types of constants, functions, and pairs: $[\![\bot]\!]$ is a singleton whose unique element represents divergence.

Introducing the $\bot$ type means that we track termination in types. In particular, we distinguish two classes of types: those that are disjoint from $\bot$ (for example, Int, $\mathsf{Int} \to \mathsf{Bool}$, or $\mathsf{Int} \times \mathsf{Bool}$) and those that include $\bot$ (since the interpretation of $\bot$ is a singleton, no type can contain a proper subset of it). Intuitively, the former correspond to computations that are guaranteed to terminate: for example, Int is the type of terminating expressions producing an integer result. Conversely, the types of diverging expressions must always contain $\bot$ and, as a result, they can always be written in the form $t \vee \bot$, for some type t. Subtyping verifies $t \leq t \vee \bot$ for any t: this ensures that a terminating expression can always be used when a possibly diverging one of the same type is expected.

This subdivision of types suggests that $\bot$ is used to approximate the set of diverging well-typed expressions: an expression whose type contains $\bot$ is an expression that *may* diverge; an expression of type $\bot$ is one that *surely* diverges. Actually, the type system we propose performs a rather gross approximation. We derive "terminating types" (i.e., subtypes of $\neg\bot$) only for expressions that are already results and cannot be reduced: constants, functions, or pairs thereof. Applications and projections, instead, are always typed by assuming that they might diverge. The typing rules are written to handle and propagate the $\bot$ type. For example, we type applications using the following rule.

$$\frac{\Gamma \vdash e_1 : (t' \to t) \vee \bot \qquad \Gamma \vdash e_2 : t'}{\Gamma \vdash e_1\, e_2 : t \vee \bot}$$

This rule allows the expression e_1 to be possibly diverging: we require it to have the type $(t' \to t) \vee \bot$ instead of the usual $t' \to t$. We type the whole application as $t \vee \bot$ to signify that it can diverge even if the codomain t does not include $\bot$, since e_1 can diverge.

This system avoids the problems we have seen with semantic subtyping: no expression can be assigned the empty type, which was the type on which subtyping behaved incorrectly. The new type $\bot$ does not cause the same problems because $[\![\bot]\!]$ is non-empty. For example, the type of expressions like $(\bar{e}, 3)$ – where $\bar{e}$ is diverging – is now $\bot \times \mathsf{Int}$. This type is not equivalent to $\bot \times \mathsf{Bool}$: the two interpretations are different because the interpretation of types includes an element ($[\![\bot]\!]$) to represent divergence.

Typing all applications as possibly diverging – even very simple ones like $(\lambda x.\, 3)\, e$ – is a very coarse approximation which can seem unsatisfactory. We could try to amend the rule to say that if e_1 has type $t' \to t$, then $e_1\, e_2$ has type t instead of $t \vee \bot$. However, we prefer to keep the simpler rules since they achieve our goal of giving a sound type system that still enjoys most benefits of semantic subtyping.

An advantage of the simpler system is that it allows us to treat $\bot$ as an internal type that does not need to be written explicitly by programmers. Since the language is explicitly typed, if $\bot$ were to be treated more precisely,

programmers would presumably need to include it or exclude it explicitly from function signatures. This would make the type system significantly different from conventional ones where divergence is not explicitly expressed in the types. In the present system, instead, we can assume that programmers annotate programs using standard set-theoretic types and $\bot$ is introduced only behind the scenes and, thus, is transparent to programmers.

We define this type system for a call-by-need variant of the language studied by Frisch, Castagna, and Benzaken (2008), and we prove its soundness in terms of progress and subject reduction. The language is similar to that of Chapter 3, but, for simplicity, we use explicitly typed functions and do not consider polymorphism.

The choice of call-by-need rather than call-by-name stems from the behaviour of semantic subtyping on intersections of arrow types. Our type system would actually be unsound for call-by-name if the language were extended with constructs that can reduce non-deterministically to different answers. For example, the expression $\mathsf{rnd}(t)$ of Frisch, Castagna, and Benzaken (2008) that returns a random result of type t could not be added while keeping soundness. This is because in call-by-name, if such an expression is duplicated, each occurrence could reduce differently; in call-by-need, instead, its evaluation would be shared. Intersection and union types make the type system precise enough to expose this difference. In the absence of such non-deterministic constructs, call-by-name and call-by-need can be shown to be observationally equivalent, so that soundness should hold for both; however, call-by-need also simplifies the technical work to prove soundness.

We show an example of this, though we will return on this point later. The example is similar to the one we have discussed in Section 3.3.1. Consider the following derivation, where $\bar{e}$ is an expression of type $\mathsf{Int} \vee \mathsf{Bool}$.

$$\dfrac{[\leq]\ \dfrac{\vdash \lambda x.\,(x,x)\colon (\mathsf{Int} \to \mathsf{Int} \times \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool} \times \mathsf{Bool})}{\vdash \lambda x.\,(x,x)\colon \mathsf{Int} \vee \mathsf{Bool} \to (\mathsf{Int} \times \mathsf{Int}) \vee (\mathsf{Bool} \times \mathsf{Bool})} \qquad \vdash \bar{e}\colon \mathsf{Int} \vee \mathsf{Bool}}{\vdash (\lambda x.\,(x,x))\,\bar{e}\colon (\mathsf{Int} \times \mathsf{Int}) \vee (\mathsf{Bool} \times \mathsf{Bool})}$$

We type $\lambda x.\,(x,x)$ with the intersection $(\mathsf{Int} \to \mathsf{Int} \times \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool} \times \mathsf{Bool})$ using, for instance, the rule $[\mathrm{T}_\wedge]$ of Figure 3.2. Then, the step marked with $[\leq]$ applies subsumption, which is possible because the intersection type is a subtype of $(\mathsf{Int} \vee \mathsf{Bool}) \to ((\mathsf{Int} \times \mathsf{Int}) \vee (\mathsf{Bool} \times \mathsf{Bool}))$. We obtain that the application $(\lambda x.\,(x,x))\,\bar{e}$ is well typed with type $(\mathsf{Int} \times \mathsf{Int}) \vee (\mathsf{Bool} \times \mathsf{Bool})$. In call-by-name, it reduces to $(\bar{e},\bar{e})$: therefore, for the system to satisfy subject reduction, we must be able to type $(\bar{e},\bar{e})$ as $(\mathsf{Int} \times \mathsf{Int}) \vee (\mathsf{Bool} \times \mathsf{Bool})$ too. But, intuitively, this type would be unsound for $(\bar{e},\bar{e})$ if each occurrence of $\bar{e}$ could reduce independently and non-deterministically either to an integer or to a Boolean. Using a typecase we can actually exhibit a term that breaks subject reduction (we return on this in Section 13.4.1).

There are several ways to approach this problem. We could try to make $(\mathsf{Int} \vee \mathsf{Bool}) \to ((\mathsf{Int} \times \mathsf{Int}) \vee (\mathsf{Bool} \times \mathsf{Bool}))$ no longer derivable for $\lambda x.\,(x,x)$, by changing the type system or the subtyping relation. However, this would

curtail the expressive power of intersection types as used in the semantic subtyping approach. We could instead assume explicitly that the semantics is deterministic. In this case, intuitively the typing would not be unsound, but a proof of subject reduction would be difficult: we should give a complex union disjunction rule to type $(\bar{e}, \bar{e})$. We choose instead to consider a call-by-need semantics because it solves both problems. With call-by-need, non-determinism poses no difficulty because of sharing. We still need a union disjunction rule, but it is simpler to state since we only need it to type the let bindings that we will introduce to represent shared computations.

12.3 Contributions

The main contribution of this part of the thesis is the development of a type system for non-strict languages based on semantic subtyping; to our knowledge, this had not been studied before.

Although the idea of our solution is simple – to track divergence – its technical development is not trivial. Our work highlights how a type system featuring union and intersection types is sensitive to the difference between strict and non-strict semantics and also, in the presence of non-determinism, to that between call-by-name and call-by-need. This shows once more how union and intersection types can express very fine properties of programs. The main technical contribution is the description of sound typing in the presence of union types for the let bindings which many formalizations of call-by-need use to represent shared computations. Finally, this work shows how to integrate the $\bot$ type, which is an explicit representation for divergence, in a semantic subtyping system. It can thus also be seen as a first step towards the definition of a type system based on semantic subtyping that performs a non-trivial form of termination analysis.

12.4 Related work

Previous work on semantic subtyping does not discuss non-strict semantics. Castagna and Frisch (2005) describe how to add a type constructor $\mathsf{lazy}(t)$ to semantic subtyping systems, but this is meant to have lazily constructed expressions within a call-by-value language.

Many type systems for functional languages (the simply typed λ-calculus or Hindley-Milner typing, for example) are sound for both strict and non-strict semantics. However, difficulties similar to ours are found in work on refinement types. Vazou et al. (2014) study how to adapt refinement types for Haskell. Their types contain logical predicates as refinements: for instance, the type of positive integers is $\{\, v \colon \mathsf{Int} \mid v > 0 \,\}$. They observe that the standard approach to type checking in these systems (checking implication between predicates with an SMT solver) is unsound for non-strict semantics. In their system, a type like $\{\, v \colon \mathsf{Int} \mid \mathsf{false} \,\}$ is analogous to $\mathbb{0}$ in our system insofar as it is not inhabited by any value. These types can be given to diverging expressions, and their introduction into the environment causes unsoundness.

To avoid this problem, they stratify types, with types divided into diverging and non-diverging ones. This corresponds in a way to our use of a type $\bot$ in types of possibly diverging expressions. As for ours, their type system can track termination to a certain extent. Partial correctness properties can be verified even without precise termination analysis. However, with their kind of analysis (which goes beyond what is expressible with set-theoretic types) there is a significant practical benefit to tracking termination more precisely. Hence, they also study how to check termination of recursive functions.

The notion of a stratification of types to keep track of divergence can also be found in work of a more theoretical strain. For instance, Constable and Smith (1987) use it to model partial functions in constructive type theory. This stratification can be understood as a monad for partiality, as it is treated by Capretta (2005). Our type system can also be seen, intuitively, as following this monadic structure. Notably, the rule for applications in a sense lifts the usual rule for application in this partiality monad. Injection in this monad is performed implicitly by subtyping via the judgment $t \leq t \vee \bot$. However, we have not developed this intuition formally.

The fact that a type system with union and intersection types can require changes to account for non-strict semantics is also discussed in work on refinement types. Dunfield and Pfenning (2003, p. 8, footnote 3) remark how a union elimination rule cannot be used to eliminate unions in function arguments if arguments are passed by name: this is analogous to the aforementioned difficulties which led to our choice of call-by-need (their system uses a dedicated typing rule for what our system handles by subtyping). Dunfield (2007, Section 8.1.5) proposes as future work to adapt a subset of the type system he considers (of refinement types for a call-by-value effectful language) to call-by-name. He notes some of the difficulties and advocates studying call-by-need as a possible way to face them. In this work we show, indeed, that a call-by-need semantics can be used to have the type system handle union and intersection types expressively without requiring complex rules.

13 A call-by-need language with set-theoretic types

This chapter presents the technical development of our approach. We first define a language with a non-strict, call-by-need semantics and a monomorphic type system for it that uses semantic subtyping. Then, we show that the type system is sound, highlighting the technical difficulties and how our approach deals with them.

CHAPTER OUTLINE:

Section 13.1 We define types, their interpretation, and subtyping. The definitions are very close to those in Chapter 2, but we add the new type $\bot$ and do not include type variables.

Section 13.2 We define the syntax and operational semantics of the language we study. The syntax is similar to that of Chapter 3 but with explicitly typed functions.

Section 13.3 We describe the type system, which is unlike that of Chapter 3 because it keeps track of divergence in the typing rules.

Section 13.4 We develop the proof of soundness. We discuss in more detail our choice of call-by-need for the semantics and how it impacts the proof.

13.1 Types and subtyping

In this section, we describe the types and the subtyping relation of our system. The definitions here are very similar to those in Sections 2.2 and 2.3, so we give them with minimal comment. The differences are that types do not include type variables (because the type system we define is monomorphic) and that they include the new type $\bot$.

As in Section 2.2, we start from a set Const of *language constants* (ranged over by c), a set Base of *base types* (ranged over by b), and two functions

$$b_{(\cdot)} \colon \mathsf{Const} \to \mathsf{Base} \qquad \mathbb{B}(\cdot) \colon \mathsf{Base} \to \mathcal{P}(\mathsf{Const})$$

mapping constants to base types and base types to sets of constants. We assume that base types include singleton types for constants: therefore, for every $c \in \mathsf{Const}$, we assume that $\mathbb{B}(b_c) = \{c\}$.

13.1 DEFINITION (Types): The set Type of *types* is the set of terms t generated coinductively by the following grammar

$$t ::= \bot \mid b \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid \mathbb{0}$$

and that satisfy the following two conditions:

(regularity) the term has finitely many distinct subterms;

(contractivity) every infinite path in the term contains infinitely many occurrences of the $\times$ or $\to$ constructors. □

We introduce the usual abbreviations:

$$t_1 \wedge t_2 \stackrel{\text{def}}{=} \neg(\neg t_1 \vee \neg t_2) \qquad t_1 \setminus t_2 \stackrel{\text{def}}{=} t_1 \wedge (\neg t_2) \qquad \mathbb{1} \stackrel{\text{def}}{=} \neg\mathbb{0}\,.$$

To define subtyping, we first introduce the interpretation domain. As in Section 2.3, we pick a symbol Ω outside Const to represent type errors.

13.2 DEFINITION: The *interpretation domain* Domain is the set of finite terms d generated inductively by the following grammar

$$d ::= \bot \mid c \mid (d, d) \mid \{(d, d_\Omega), \ldots, (d, d_\Omega)\} \qquad d_\Omega ::= d \mid \Omega$$

where c ranges over Const. □

Compared to Definition 2.3, we add $\bot$ to represent divergence explicitly in the domain, and we remove labels on the elements because they were only needed to describe subtyping with type variables.

We want the interpretation of types $[\![\cdot]\!]$ to satisfy the following equalities:

$$\begin{aligned}
[\![\bot]\!] &= \{\bot\} \\
[\![b]\!] &= \mathbb{B}(b) \\
[\![t_1 \times t_2]\!] &= [\![t_1]\!] \times [\![t_2]\!] \\
[\![t_1 \to t_2]\!] &= \left\{ \{ (d^i, d^i_\Omega) \mid i \in I \} \;\middle|\; \forall i \in I.\, d^i \in [\![t_1]\!] \implies d^i_\Omega \in [\![t_2]\!] \right\} \\
[\![t_1 \vee t_2]\!] &= [\![t_1]\!] \cup [\![t_2]\!] \\
[\![\neg t]\!] &= \mathsf{Domain} \setminus [\![t]\!] \\
[\![\mathbb{0}]\!] &= \varnothing
\end{aligned}$$

We proceed as in Section 2.3 to define $[\![\cdot]\!]$ accounting for recursive types.

13.3 DEFINITION (Set-theoretic interpretation of types): We define a binary predicate $(d : t)$, where $d \in \mathsf{Domain}$ and $t \in \mathsf{Type}$, by induction on the pair (d, t) ordered lexicographically. The predicate is defined as follows:

$$\begin{aligned}
(\bot : \bot) &= \text{true} \\
(c : b) &= c \in \mathbb{B}(b) \\
((d_1, d_2) : t_1 \times t_2) &= (d_1 : t_1) \wedge (d_2 : t_2) \\
(\{ (d^i, d^i_\Omega) \mid i \in I \} : t_1 \to t_2) &= \forall i \in I.\, (d^i : t_1) \implies (d^i_\Omega \neq \Omega) \wedge (d^i_\Omega : t_2) \\
(d : t_1 \vee t_2) &= (d : t_1) \vee (d : t_2) \\
(d : \neg t) &= \neg(d : t) \\
(d : t) &= \text{false} \qquad \text{otherwise}
\end{aligned}$$

We define the *set-theoretic interpretation* $[\![\cdot]\!] : \mathsf{Type} \to \mathcal{P}(\mathsf{Domain})$ as

$$[\![t]\!] = \{\, d \in \mathsf{Domain} \mid (d : t) \,\}\,.$$ □

Finally, we define subtyping and subtype equivalence as usual.

13.4 DEFINITION (Subtyping): We define the *subtyping* relation $\leq$ and the *subtype equivalence* relation $\simeq$ on types as:

$$t_1 \leq t_2 \stackrel{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!] \qquad t_1 \simeq t_2 \stackrel{\text{def}}{\iff} (t_1 \leq t_2) \wedge (t_2 \leq t_1)\,. \qquad \square$$

We have intentionally stayed very close to the original definitions of semantic subtyping. This allows us to reuse existing results, including the algorithm to decide subtyping (since $\bot$ is added just like a new base type). To ensure soundness, instead of changing subtyping, we change the type system. A drawback of this approach is that the interpretation of types is not as appropriate for non-strict languages as it is for strict ones: in Section 14.1, we will discuss this extensively and point out directions for further improvement.

13.1.1 Properties of subtyping

We collect here some properties of subtyping on arrow types that we rely on later. In particular, we show how we can compute from an intersection of arrow types an equivalent intersection where all arrows have disjoint domains: this is convenient to describe the typing of functions.

13.5 LEMMA:

$$\bigwedge_{i\in I} t'_i \to t_i \leq \bigvee_{j\in J} t'_j \to t_j \iff$$
$$\exists j_0 \in J.\ \left(t'_{j_0} \leq \bigvee_{i\in I} t'_i\right) \wedge \left(\forall I' \subsetneq I.\ \left(t'_{j_0} \leq \bigvee_{i\in I'} t'_i\right) \vee \left(\bigwedge_{i\in I\setminus I'} t_i \leq t_{j_0}\right)\right)$$

$\square$

Proof: Analogous to the proof of Lemma 2.16. $\square$

13.6 COROLLARY: Let $\bigwedge_{i\in I} t'_i \to t_i$ (with $|I| > 0$) be such that, for every $i_1, i_2 \in I$, if $i_1 \neq i_2$ then $t'_{i_1} \wedge t'_{i_2} \simeq \mathbb{0}$. Then:

$$\bigwedge_{i\in I} t'_i \to t_i \leq t' \to t \implies \left(t' \leq \bigvee_{i\in I} t'_i\right) \wedge \left(\forall i \in I.\ (t'_i \wedge t' \not\simeq \mathbb{0}) \implies (t_i \leq t)\right)$$

$\square$

Proof in appendix (p. 291).

13.7 COROLLARY: Let $\bar{t} = (\bigwedge_{i\in I} t'_i \to t_i) \wedge (\bigwedge_{j\in J} \neg(t'_j \to t_j))$. If $\bar{t} \not\simeq \mathbb{0}$ and $\bar{t} \leq t' \to t$, then $(\bigwedge_{i\in I} t'_i \to t_i) \leq t' \to t$. $\square$

Proof in appendix (p. 292).

13.8 LEMMA: For every finite set J and every set $\{ t_j \mid j \in J \}$,

$$\bigvee_{J' \subseteq J} \left(\bigwedge_{j \in J'} t_j \wedge \bigwedge_{j \in J \setminus J'} \neg t_j \right) \simeq \mathbb{1}$$

(with the convention that an intersection over an empty set is $\mathbb{1}$). □

Proof in appendix (p. 292).

13.9 LEMMA: Let $\mathbb{I} = \bigwedge_{i \in I} t'_i \to t_i$ (with $|I| > 0$) be a type. Then:

$\mathbb{I} \simeq \bigwedge_{\varnothing \subsetneq I' \subseteq I} s_{I'} \to u_{I'}$ where $s_{I'} \stackrel{\text{def}}{=} \bigwedge_{i \in I'} t'_i \wedge \bigwedge_{i \in I \setminus I'} \neg t'_i$ and $u_{I'} \stackrel{\text{def}}{=} \bigwedge_{i \in I'} t_i$

(with the convention: $\bigwedge_{i \in \varnothing} \neg t'_i = \mathbb{1}$). □

Proof in appendix (p. 292).

13.2 Language syntax and semantics

We consider a language based on that studied by Frisch, Castagna, and Benzaken (2008): a λ-calculus with recursive explicitly annotated functions, pair constructors and destructors, and a typecase construct. Compared to the language in Chapter 3, the main difference is that here functions are explicitly annotated with their type: an *interface* $\mathbb{I}$, which is an intersection of arrow types. Moreover, functions are recursive (with a binder for the recursion parameter), and typecases include a binder (as in Section 6.1.1).

We actually define two languages: a *source language* in which programs will be written and a slightly different *internal language* on which we define the semantics. The internal language adds a let construct; this is a form of explicit substitution used to model call-by-need semantics in a small-step operational style, following a standard approach (Ariola et al., 1995; Ariola and Felleisen, 1997; Maraist, Odersky, and Wadler, 1998). Typecases are also defined slightly differently in the two languages (to simplify the semantics), so we show how to compile source programs to the internal language. The let construct used here is unlike that of Part I: it is only used in the semantics and is not a binder used in programs for polymorphic definition (the type system is monomorphic).

As anticipated, we want $\bot$ to be an internal type, used in the description of the type system but not by programmers explicitly. To do so, we introduce two restricted grammars of types (**T** and **t**, below) where $\bot$ does not appear explicitly. Programs will only contain types from these grammars.

First, we introduce the abbreviations:

$$\langle t \rangle \stackrel{\text{def}}{=} t \vee \bot \qquad t_1 \mathrel{\not\to} t_2 \stackrel{\text{def}}{=} \langle t_1 \rangle \to \langle t_2 \rangle \qquad t_1 \mathrel{\not\times} t_2 \stackrel{\text{def}}{=} \langle t_1 \rangle \times \langle t_2 \rangle \,.$$

These are compact notations for types including $\bot$. The first, $\langle t \rangle$, is an abbreviated way to write the type of possibly diverging expressions whose result has type t. The latter two are used in type annotations: programmers use $\not\to$

and $\barwedge$ instead of $\to$ and $\times$, so that $\bot$ is introduced implicitly. The $\to$ and $\times$ constructors are never written directly in programs.

We define the following restricted grammars of types

$$\mathbf{T} ::= b \mid \mathbf{T} \barwedge \mathbf{T} \mid \mathbf{T} \bararrow \mathbf{T} \mid \mathbf{T} \vee \mathbf{T} \mid \neg\mathbf{T} \mid \mathbb{0}$$
$$\mathbf{t} ::= b \mid \mathbf{t} \barwedge \mathbf{t} \mid \mathbb{0} \to \mathbb{1} \mid \mathbf{t} \vee \mathbf{t} \mid \neg\mathbf{t} \mid \mathbb{0}$$

both of which are interpreted coinductively, with the same restrictions of regularity and contractivity as in the definition of types. The types defined by these grammars will be the only ones which appear in programs.

In particular, functions will be annotated with $\mathbf{T}$ types, where the use of $\barwedge$ and $\bararrow$ ensures that every type below a constructor is of the form $t \vee \bot$.

Typecases, instead, will check $\mathbf{t}$ types. The only arrow type that can appear in them is $\mathbb{0} \to \mathbb{1}$, which is the top type of functions. This is the same restriction that we have imposed in Section 3.1: typecases cannot test function types. We impose it here mostly for uniformity: with explicitly typed functions and no polymorphism, the restriction can be lifted without difficulty.

13.2.1 Source language

The *source language expressions* are the terms $\mathbf{e}$ produced inductively by the grammar

$$\mathbf{e} ::= x \mid c \mid \mu f\colon \mathbb{I}.\, \lambda x.\, \mathbf{e} \mid \mathbf{e}\ \mathbf{e} \mid (\mathbf{e}, \mathbf{e}) \mid \pi_i\ \mathbf{e} \mid (x = \mathbf{e}) \in \mathbf{t}\ ?\ \mathbf{e} : \mathbf{e}$$
$$\mathbb{I} ::= \bigwedge_{i \in I} \mathbf{T}'_i \bararrow \mathbf{T}_i \qquad |I| > 0$$

where f and x range over a set EVar of *expression variables*, c over the set Const of constants, i in $\pi_i\ \mathbf{e}$ over $\{1, 2\}$, and where $\mathbf{t}$ in $(x = \mathbf{e}) \in \mathbf{t}\ ?\ \mathbf{e} : \mathbf{e}$ is such that $\mathbf{t} \not\simeq \mathbb{0}$ and $\mathbf{t} \not\simeq \mathbb{1}$.

A λ-abstraction $\mu f\colon \mathbb{I}.\, \lambda x.\, \mathbf{e}$ is a possibly recursive function, with recursion parameter f and argument x, both of which are bound in the body; the function is explicitly annotated with its *interface* $\mathbb{I}$, which is a finite intersection of types of the form $\mathbf{T}' \bararrow \mathbf{T}$.

A typecase expression $(x = \mathbf{e}_0) \in \mathbf{t}\ ?\ \mathbf{e}_1 : \mathbf{e}_2$ has the following intended semantics: $\mathbf{e}_0$ is evaluated until it can be determined whether it has type $\mathbf{t}$ or not, then the selected branch ($\mathbf{e}_1$ if the result of $\mathbf{e}_0$ has type $\mathbf{t}$, $\mathbf{e}_2$ if it has type $\neg\mathbf{t}$: one of the two cases always occurs) is evaluated in an environment where x is bound to the result of $\mathbf{e}_0$. Actually, to simplify the presentation, we will give a non-deterministic semantics in which we allow to evaluate $\mathbf{e}_0$ more than what is needed to ascertain whether it has type $\mathbf{t}$.

In the syntax definition above we have restricted the types $\mathbf{t}$ in typecases by requiring $\mathbf{t} \not\simeq \mathbb{1}$ and $\mathbf{t} \not\simeq \mathbb{0}$. A typecase checking the type $\mathbb{1}$ is useless: since all expressions have type $\mathbb{1}$, it immediately reduces to its first branch. Likewise, a typecase checking the type $\mathbb{0}$ reduces directly to the second branch. Therefore, the two cases are uninteresting to consider. We forbid them because this allows us to give a simpler typing rule for typecases. Allowing them is just a matter of adding two (trivial) typing rules specific to these cases, as we show later.

As customary, we consider expressions up to renaming of bound variables. In $\mu f\colon \mathbb{I}.\, \lambda x.\, \mathbf{e}$, f and x are bound in $\mathbf{e}$. In $(x = \mathbf{e}_0) \in \mathbf{t}\ ?\ \mathbf{e}_1 : \mathbf{e}_2$, x is bound in $\mathbf{e}_1$ and $\mathbf{e}_2$.

We do not provide mechanisms to define cyclic data structures. For example, we do not have a direct syntactic construct to define the infinitely nested pair $(1, (1, \ldots))$. We can define it by writing a fixpoint operator or by defining and applying a recursive function which constructs the pair. A general letrec construct (as in Ariola and Felleisen, 1997) might be useful in practice (for efficiency or to provide greater sharing) but we omit it here since we are only concerned with typing.

13.2.2 Internal language

The *internal language expressions* are the terms e produced inductively by the grammar

$$e ::= x \mid c \mid \mu f\colon \mathbb{I}.\, \lambda x.\, e \mid e\ e \mid (e, e) \mid \pi_i\ e \mid (x = \varepsilon) \in \mathbf{t}\ ?\ e : e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e$$
$$\varepsilon ::= x \mid c \mid \mu f\colon \mathbb{I}.\, \lambda x.\, e \mid (\varepsilon, \varepsilon)$$

where metavariables and conventions are as in the source language.

There are two differences with respect to the source language. One is the introduction of the construct $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, which is a binder used to model call-by-need semantics (in $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, x is bound in e_2). The other difference is that typecases cannot check arbitrary expressions, but only expressions of the restricted form given by ε.

A source language expression $\mathbf{e}$ can be compiled to an internal language expression $\lceil \mathbf{e} \rceil$ as follows. Compilation is straightforward for all expressions apart from typecases:

$$\lceil x \rceil = x \qquad \lceil c \rceil = c$$
$$\lceil \mu f\colon \mathbb{I}.\, \lambda x.\, \mathbf{e} \rceil = \mu f\colon \mathbb{I}.\, \lambda x.\, \lceil \mathbf{e} \rceil \qquad \lceil \mathbf{e}_1\ \mathbf{e}_2 \rceil = \lceil \mathbf{e}_1 \rceil\ \lceil \mathbf{e}_2 \rceil$$
$$\lceil (\mathbf{e}_1, \mathbf{e}_2) \rceil = (\lceil \mathbf{e}_1 \rceil, \lceil \mathbf{e}_2 \rceil) \qquad \lceil \pi_i\ \mathbf{e} \rceil = \pi_i\ \lceil \mathbf{e} \rceil$$

and for typecases it introduces a let binder to ensure that the checked expression is a variable:

$$\lceil (x = \mathbf{e}_0) \in \mathbf{t}\ ?\ \mathbf{e}_1 : \mathbf{e}_2 \rceil = (\mathsf{let}\ y = \lceil \mathbf{e}_0 \rceil\ \mathsf{in}\ (x = y) \in \mathbf{t}\ ?\ \lceil \mathbf{e}_1 \rceil : \lceil \mathbf{e}_2 \rceil)$$

where y is chosen to avoid variables free in $\mathbf{e}_1$ and $\mathbf{e}_2$. (The other forms for ε can appear during reduction.)

13.2.3 Semantics

We define the operational semantics of the internal language as a small-step reduction relation using call-by-need. The semantics of the source language is then given indirectly through the compilation. The choice of call-by-need rather than call-by-name was briefly motivated in Chapter 12 and will be discussed more extensively in Section 13.3.

We first define the sets of *answers* (ranged over by a) and of *values* (ranged over by v) as the subsets of expressions produced by the following grammars:

$$a ::= c \mid \mu f{:}\mathbb{I}.\, \lambda x.\, e \mid (e, e) \mid \mathsf{let}\ x = e\ \mathsf{in}\ a$$
$$v ::= c \mid \mu f{:}\mathbb{I}.\, \lambda x.\, e$$

Answers are the results of evaluation. They correspond to expressions which are fully evaluated up to their top-level constructor (constant, function, or pair) but which may include arbitrary expressions below that constructor (so we have (e, e) rather than (a, a)). Since they also include let bindings, they represent closures in which variables can be bound to arbitrary expressions. Values are a subset of answers treated specially in a reduction rule.

The semantics uses evaluation contexts to direct the order of evaluation. A *context* C is an expression with a hole (written [ ]) in it. We write $C[e]$ for the expression obtained by replacing the hole in C with e. We write $C\lceil e \rceil$ for $C[e]$ when the free variables of e are not bound by C: for example, $\mathsf{let}\ x = e_1\ \mathsf{in}\ x$ is of the form $C[x]$ – with $C \equiv (\mathsf{let}\ x = e_1\ \mathsf{in}\ [\,])$ – but not of the form $C\lceil x \rceil$; conversely, $\mathsf{let}\ x = e_1\ \mathsf{in}\ y$ is both of the form $C[y]$ and $C\lceil y \rceil$.

Evaluation contexts E are the subset of contexts generated by the following grammar:

$$E ::= [\,] \mid E\, e \mid \pi_i\, E \mid (x = F) \in \mathbf{t}\ ?\ e : e \mid \mathsf{let}\ x = e\ \mathsf{in}\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ E\lceil x \rceil$$
$$F ::= [\,] \mid (F, \varepsilon) \mid (\varepsilon, F)$$

Evaluation contexts allow reduction to occur on the left of applications and below projections, but not on the right of applications and below pairs. For typecases alone, the contexts allow reduction also below pairs, since this reduction might be necessary to be able to determine whether the expression has type $\mathbf{t}$ or not. This is analogous to the behaviour of pattern matching in lazy languages, which can force evaluation below constructors. The contexts for let are from standard presentations of call-by-need (Ariola and Felleisen, 1997; Maraist, Odersky, and Wadler, 1998). They always allow reduction of the body of the let, while they only allow reduction of the bound expression when it is required to continue evaluating the body: this is enforced by requiring the body to have the form $E\lceil x \rceil$.

Figure 13.1 presents the reduction rules. They rely on the typeof function, defined as

$$\mathsf{typeof}(\varepsilon) \stackrel{\text{def}}{=} \begin{cases} \mathbb{1} & \text{if } \varepsilon = x \\ b_c & \text{if } \varepsilon = c \\ \mathbb{0} \to \mathbb{1} & \text{if } \varepsilon = \mu f{:}\mathbb{I}.\, \lambda x.\, e \\ \mathsf{typeof}(\varepsilon_1) \times \mathsf{typeof}(\varepsilon_2) & \text{if } \varepsilon = (\varepsilon_1, \varepsilon_2) \end{cases}$$

that assigns types to expressions in the grammar for ε.

The rule[$\mathrm{R}_{\mathrm{app}}$] is the standard application rule for call-by-need: the application $(\mu f{:}\mathbb{I}.\, \lambda x.\, e)\ e'$ reduces to e prefixed by two let bindings that bind the recursion variable f to the function itself and the parameter x to the argument e'. [$\mathrm{R}^{\mathsf{let}}_{\mathrm{app}}$] instead deals with applications with a let expression in function

$$
\begin{array}{rrcll}
[\mathrm{R}_{\mathrm{app}}] & (\mu f{:}\mathbb{I}.\,\lambda x.\,e)\,e' & \rightsquigarrow & \mathsf{let}\ f = (\mu f{:}\mathbb{I}.\,\lambda x.\,e)\ \mathsf{in}\ \mathsf{let}\ x = e'\ \mathsf{in}\ e & \\
[\mathrm{R}^{\mathsf{let}}_{\mathrm{app}}] & (\mathsf{let}\ x = e\ \mathsf{in}\ a)\,e' & \rightsquigarrow & \mathsf{let}\ x = e\ \mathsf{in}\ a\,e' & \\
[\mathrm{R}_{\mathrm{proj}}] & \pi_i\,(e_1, e_2) & \rightsquigarrow & e_i & \\
[\mathrm{R}^{\mathsf{let}}_{\mathrm{proj}}] & \pi_i\,(\mathsf{let}\ x = e\ \mathsf{in}\ a) & \rightsquigarrow & \mathsf{let}\ x = e\ \mathsf{in}\ \pi_i\,a & \\
[\mathrm{R}^{v}_{\mathsf{let}}] & \mathsf{let}\ x = v\ \mathsf{in}\ E[\![x]\!] & \rightsquigarrow & (E[\![x]\!])[v/x] & \\
[\mathrm{R}^{\mathrm{pair}}_{\mathsf{let}}] & \mathsf{let}\ x = (e_1, e_2)\ \mathsf{in}\ E[\![x]\!] & \rightsquigarrow & \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ (E[\![x]\!])[(x_1, x_2)/x] & \\
[\mathrm{R}^{\mathsf{let}}_{\mathsf{let}}] & \mathsf{let}\ x = (\mathsf{let}\ y = e\ \mathsf{in}\ a)\ \mathsf{in}\ E[\![x]\!] & \rightsquigarrow & \mathsf{let}\ y = e\ \mathsf{in}\ \mathsf{let}\ x = a\ \mathsf{in}\ E[\![x]\!] & \\
[\mathrm{R}^{1}_{\mathrm{case}}] & (x = \varepsilon) \in \mathbf{t}\ ?\ e_1 : e_2 & \rightsquigarrow & \mathsf{let}\ x = \varepsilon\ \mathsf{in}\ e_1 & \text{if } \mathsf{typeof}(\varepsilon) \leq \mathbf{t} \\
[\mathrm{R}^{2}_{\mathrm{case}}] & (x = \varepsilon) \in \mathbf{t}\ ?\ e_1 : e_2 & \rightsquigarrow & \mathsf{let}\ x = \varepsilon\ \mathsf{in}\ e_2 & \text{if } \mathsf{typeof}(\varepsilon) \leq \neg\mathbf{t} \\
[\mathrm{R}_{\mathrm{ctx}}] & E[e] & \rightsquigarrow & E[e'] & \text{if } e \rightsquigarrow e'
\end{array}
$$

FIGURE 13.1 Reduction rules

position: it moves the application below the let. The rule is necessary to prevent loss of sharing: substituting the binding of x to e in a would duplicate e. Symmetrically, there are two rules for pair projections, $[\mathrm{R}_{\mathrm{proj}}]$ and $[\mathrm{R}^{\mathsf{let}}_{\mathrm{proj}}]$.

There are three rules for let expressions. They rewrite expressions of the form $\mathsf{let}\ x = a\ \mathsf{in}\ E[\![x]\!]$: that is, let bindings where the bound expression is an answer and the body is an expression whose evaluation requires the evaluation of x. If a is a value v, $[\mathrm{R}^{v}_{\mathsf{let}}]$ applies and the expression is reduced by replacing v for x in the body. If a is a pair, $[\mathrm{R}^{\mathrm{pair}}_{\mathsf{let}}]$ applies: the occurrences of x in the body are replaced with a pair of variables (x_1, x_2) and each x_i is bound to e_i by new let bindings (replacing x directly by (e_1, e_2) would duplicate expressions). Finally, the $[\mathrm{R}^{\mathsf{let}}_{\mathsf{let}}]$ rule moves one let binding out of another.

There are two rules for typecases, $[\mathrm{R}^{1}_{\mathrm{case}}]$ and $[\mathrm{R}^{2}_{\mathrm{case}}]$, by which a typecase construct $(x = \varepsilon) \in \mathbf{t}\ ?\ e_1 : e_2$ can be reduced to either branch, introducing a new binding of x to ε. The rules apply only if either of $\mathsf{typeof}(\varepsilon) \leq \mathbf{t}$ or $\mathsf{typeof}(\varepsilon) \leq \neg\mathbf{t}$ holds. If neither holds, then the two rules do not apply, but the $[\mathrm{R}_{\mathrm{ctx}}]$ rule can be used to continue the evaluation of ε.

EXAMPLES OF THE EVALUATION OF TYPECASES: We start with a simple example. Let $\bar{e}_1$ be the expression $(x = \mathsf{true}) \in b_{\mathsf{true}}\ ?\ 1 : 2$, where b_{true} denotes the singleton type of true. This typecase corresponds to the conditional expression $\mathsf{if\ true\ then\ 1\ else\ 2}$. Since $\mathsf{typeof}(\mathsf{true}) = b_{\mathsf{true}} \leq b_{\mathsf{true}}$, we can apply $[\mathrm{R}^{1}_{\mathrm{case}}]$ and reduce $\bar{e}_1$ to $\mathsf{let}\ x = \mathsf{true}\ \mathsf{in}\ 1$.

As a more complex example, consider the expression $\bar{e} \equiv (\mathsf{let}\ y = \bar{e}_1\ \mathsf{in}\ \bar{e}_2)$, where $\bar{e}_1$ is defined as before and $\bar{e}_2$ is $(z = (y, 2)) \in (\mathsf{Int} \between \mathsf{Int})\ ?\ \mathsf{true} : \mathsf{false}$.

Note that $\mathsf{typeof}((y, 2)) = \mathbb{1} \times b_2$ (where b_2 is the singleton type of 2) and that neither of $\mathbb{1} \times b_2 \leq \mathsf{Int} \between \mathsf{Int}$ or $\mathbb{1} \times b_2 \leq \neg(\mathsf{Int} \between \mathsf{Int})$ holds. Hence, the typecase cannot reduce directly. However, $\bar{e}$ is of the form $\mathsf{let}\ y = E_1[\bar{e}_1]\ \mathsf{in}\ E_2[\![y]\!]$, taking E_1 to be $[\,]$ and E_2 to be $(z = ([\,], 2)) \in (\mathsf{Int} \between \mathsf{Int})\ ?\ \mathsf{true} : \mathsf{false}$. Therefore, it can

be evaluated as follows:

$$
\begin{aligned}
\bar{e} &\rightsquigarrow \mathsf{let}\ y = (\mathsf{let}\ x = \mathsf{true}\ \mathsf{in}\ 1)\ \mathsf{in}\ \bar{e}_2 && \text{by } [\mathrm{R}_{\mathrm{ctx}}] \text{ and } [\mathrm{R}^{1}_{\mathrm{case}}] \\
&\rightsquigarrow \mathsf{let}\ x = \mathsf{true}\ \mathsf{in}\ \mathsf{let}\ y = 1\ \mathsf{in}\ \bar{e}_2 && \text{by } [\mathrm{R}^{\mathsf{let}}_{\mathsf{let}}] \\
&\rightsquigarrow \mathsf{let}\ x = \mathsf{true}\ \mathsf{in}\ \bar{e}_2[1/y] && \text{by } [\mathrm{R}_{\mathrm{ctx}}] \text{ and } [\mathrm{R}^{v}_{\mathsf{let}}] \\
&\equiv \mathsf{let}\ x = \mathsf{true}\ \mathsf{in}\ (z = (1,2)) \in (\mathsf{Int} \between \mathsf{Int})\ ?\ \mathsf{true} : \mathsf{false} \\
&\rightsquigarrow \mathsf{let}\ x = \mathsf{true}\ \mathsf{in}\ \mathsf{let}\ z = (1,2)\ \mathsf{in}\ \mathsf{true} && \text{by } [\mathrm{R}_{\mathrm{ctx}}] \text{ and } [\mathrm{R}^{1}_{\mathrm{case}}]
\end{aligned}
$$

The answer we obtain has useless let bindings. We did not include a garbage collection rule to get rid of these, though it could be added without difficulty.

COMPARISON TO OTHER PRESENTATIONS OF CALL-BY-NEED: These reduction rules mirror those from standard presentations of call-by-need (Ariola et al., 1995; Ariola and Felleisen, 1997; Maraist, Odersky, and Wadler, 1998). A difference is that, in $[\mathrm{R}^{v}_{\mathsf{let}}]$ or $[\mathrm{R}^{\mathrm{pair}}_{\mathsf{let}}]$, we replace *all* occurrences of x in $E\lceil x \rceil$ at once, whereas in the cited presentations only the occurrence in the hole is replaced: for example, in $[\mathrm{R}^{v}_{\mathsf{let}}]$ they reduce to $E\lceil v \rceil$ instead of $(E\lceil x \rceil)[v/x]$. Our $[\mathrm{R}^{v}_{\mathsf{let}}]$ rule is mentioned as a variant by Maraist, Odersky, and Wadler (1998, p. 38). We use it because it simplifies the proof of subject reduction while maintaining an equivalent semantics.

NON-DETERMINISM IN THE RULES: The semantics is not deterministic. There are two sources of non-determinism, both related to typecases. One is that the contexts F include both (F, ε) and (ε, F) and thereby impose no constraint on the order with which pairs are examined. The second is that the contexts for typecases allow us to reduce the bindings of variables in the checked expression even when we can already apply $[\mathrm{R}^{1}_{\mathrm{case}}]$ or $[\mathrm{R}^{2}_{\mathrm{case}}]$.

For example, take $\mathsf{let}\ x = e\ \mathsf{in}\ (y = (3, x)) \in (\mathsf{Int} \between \mathbb{1})\ ?\ e_1 : e_2$. It can be immediately reduced to $\mathsf{let}\ x = e\ \mathsf{in}\ \mathsf{let}\ y = (3, x)\ \mathsf{in}\ e_1$ by applying $[\mathrm{R}_{\mathrm{ctx}}]$ and $[\mathrm{R}^{1}_{\mathrm{case}}]$, because $\mathsf{typeof}((3, x)) = b_3 \times \mathbb{1} \leq \mathsf{Int} \between \mathbb{1}$. However, we can also use $[\mathrm{R}_{\mathrm{ctx}}]$ to reduce e, if it is reducible: we do so by writing the expression as $\mathsf{let}\ x = e\ \mathsf{in}\ E\lceil x \rceil$, where E is $(y = (3, [\,]))) \in (\mathsf{Int} \between \mathbb{1})\ ?\ e_1 : e_2$. To model a lazy implementation more faithfully, we should forbid this reduction and state that $(x = F) \in \mathbf{t}\ ?\ e : e$ is a context only if it cannot be reduced by $[\mathrm{R}^{1}_{\mathrm{case}}]$ or $[\mathrm{R}^{2}_{\mathrm{case}}]$.

In both cases, we have chosen a non-deterministic semantics because it is less restrictive: as a consequence, the soundness result will also hold for semantics that fix an order.

13.3 Type system

We define here the typing relations for the two languages.

A *type environment* Γ is a finite mapping of type variables to types. We write $\varnothing$ for the empty environment. We say that a type environment Γ is *well formed* if, for all $(x\colon t) \in \Gamma$, we have $t \not\simeq \mathbb{0}$. Since we want to ensure that the empty type is never derivable, we will only consider well-formed type environments in the soundness proof.

$$[\mathrm{T}^{\mathrm{s}}_x]\ \frac{}{\Gamma \vdash x : t}\ \Gamma(x) = t \qquad\qquad [\mathrm{T}^{\mathrm{s}}_c]\ \frac{}{\Gamma \vdash c : b_c}$$

$$[\mathrm{T}^{\mathrm{s}}_\lambda]\ \frac{\forall i \in I.\quad \Gamma, f : \mathbb{I}, x : \langle \mathbf{T}'_i \rangle \vdash \mathbf{e} : \langle \mathbf{T}_i \rangle}{\Gamma \vdash (\mu f : \mathbb{I}.\, \lambda x.\, \mathbf{e}) : \mathbb{I}}\ \mathbb{I} = \bigwedge_{i \in I} \mathbf{T}'_i \mathrel{\not\to} \mathbf{T}_i$$

$$[\mathrm{T}^{\mathrm{s}}_{\mathrm{app}}]\ \frac{\Gamma \vdash \mathbf{e}_1 : \langle t' \to t \rangle \qquad \Gamma \vdash \mathbf{e}_2 : t'}{\Gamma \vdash \mathbf{e}_1\ \mathbf{e}_2 : \langle t \rangle}$$

$$[\mathrm{T}^{\mathrm{s}}_{\mathrm{pair}}]\ \frac{\Gamma \vdash \mathbf{e}_1 : t_1 \qquad \Gamma \vdash \mathbf{e}_2 : t_2}{\Gamma \vdash (\mathbf{e}_1, \mathbf{e}_2) : t_1 \times t_2} \qquad\qquad [\mathrm{T}^{\mathrm{s}}_{\mathrm{proj}}]\ \frac{\Gamma \vdash \mathbf{e} : \langle t_1 \times t_2 \rangle}{\Gamma \vdash \pi_i\ \mathbf{e} : \langle t_i \rangle}$$

$$[\mathrm{T}^{\mathrm{s}}_{\mathrm{case}}]\ \frac{\begin{array}{c}\Gamma \vdash \mathbf{e}_0 : \langle t' \rangle \\ \text{either } t' \le \neg\mathbf{t} \text{ or } \Gamma, x : (t' \wedge \mathbf{t}) \vdash \mathbf{e}_1 : t \qquad \text{either } t' \le \mathbf{t} \text{ or } \Gamma, x : (t' \setminus \mathbf{t}) \vdash \mathbf{e}_2 : t\end{array}}{\Gamma \vdash \big((x = \mathbf{e}_0) \in \mathbf{t}\ ?\ \mathbf{e}_1 : \mathbf{e}_2\big) : \langle t \rangle}$$

$$[\mathrm{T}^{\mathrm{s}}_{\le}]\ \frac{\Gamma \vdash \mathbf{e} : t'}{\Gamma \vdash \mathbf{e} : t}\ t' \le t$$

FIGURE 13.2 $\mathcal{T}^{\mathrm{s}}_\perp$: Typing rules of the source language

13.3.1 Type system of the source language

Figure 13.2 presents the typing rules $\mathcal{T}^{\mathrm{s}}_\perp$ of the source language. The rules $[\mathrm{T}^{\mathrm{s}}_x]$ and $[\mathrm{T}^{\mathrm{s}}_c]$ for variables and constants are standard.

The $[\mathrm{T}^{\mathrm{s}}_\lambda]$ rule for functions is also straightforward. Function interfaces have the form $\bigwedge_{i \in I} \mathbf{T}'_i \mathrel{\not\to} \mathbf{T}_i$, that is, $\bigwedge_{i \in I} \langle \mathbf{T}'_i \rangle \to \langle \mathbf{T}_i \rangle$ (by definition of $\not\to$). To type a function $\mu f : \mathbb{I}.\, \lambda x.\, \mathbf{e}$, we check that it has all the arrow types in $\mathbb{I}$. Namely, for every arrow $\mathbf{T}'_i \mathrel{\not\to} \mathbf{T}_i$ (i.e., $\langle \mathbf{T}'_i \rangle \to \langle \mathbf{T}_i \rangle$), we assume that x has type $\langle \mathbf{T}'_i \rangle$ and that the recursion variable f has type $\mathbb{I}$, and we check that the body has type $\langle \mathbf{T}_i \rangle$.

The $[\mathrm{T}^{\mathrm{s}}_{\mathrm{app}}]$ rule is the first one that deals with $\perp$ in a non-trivial way, instead of being the standard *modus ponens* rule of call-by-value semantic subtyping systems (as in Parts I and II). We allow the function term $\mathbf{e}_1$ to have the type $\langle t' \to t \rangle$ (i.e., $(t' \to t) \vee \perp$) to allow it to be possibly diverging. We use $\langle t \rangle$ as the type of the whole application, signifying that it might diverge. As anticipated, we do not try to predict whether applications will converge.

The rule $[\mathrm{T}^{\mathrm{s}}_{\mathrm{pair}}]$ for pairs is standard; $[\mathrm{T}^{\mathrm{s}}_{\mathrm{proj}}]$ handles $\perp$ as in applications.

$[\mathrm{T}^{\mathrm{s}}_{\mathrm{case}}]$ is the most complex one, but it's very similar to the $[\mathrm{T}_{\mathrm{case}}]$ in Figure 3.2, and even more so to that of Frisch, Castagna, and Benzaken (2008). We type the checked expression $\mathbf{e}_0$ and then, possibly, one branch or both, depending on the conditions $t' \le \neg\mathbf{t}$ and $t' \le \mathbf{t}$, which hold when we know statically that the first or second branch, respectively, cannot be selected. Compared to

the rule in Figure 3.2, here we type the branches in an extended environment because the checked expression is bound in the body. We treat $\bot$ as in $[\mathrm{T}^{\mathrm{s}}_{\mathrm{app}}]$ and $[\mathrm{T}^{\mathrm{s}}_{\mathrm{proj}}]$.

The subsumption rule $[\mathrm{T}^{\mathrm{s}}_{\leq}]$ is used to apply subtyping. Notably, it allows expressions with surely converging types (like a pair with type $\mathsf{Int} \times \mathsf{Bool}$) to be used where diverging types are expected: $t \leq \langle t \rangle$ holds for every t (since $[\![t]\!] \subseteq [\![t]\!] \cup \{\bot\} = [\![t \vee \bot]\!] = [\![\langle t \rangle]\!]$).

As anticipated, in the syntax we have restricted the type $\mathbf{t}$ in typecases requiring $\mathbf{t} \not\simeq \mathbb{1}$ and $\mathbf{t} \not\simeq \mathbb{0}$. Typecases where these conditions do not hold are uninteresting, since they do not actually check anything. The rule $[\mathrm{T}^{\mathrm{s}}_{\mathrm{case}}]$ would be unsound for them because these typecases can reduce to one branch even if $\mathbf{e}_0$ is a diverging expression that does not evaluate to an answer. For instance, if $\bar{\mathbf{e}}$ has type $\bot$ (that is, $\langle \mathbb{0} \rangle$), then $(x = \bar{\mathbf{e}}) \in \mathbb{1}\ ?\ 1 : 2$ could be given any type, including unsound ones like $\langle \mathsf{Bool} \rangle$. To allow these typecases, we could add the side condition "$\mathbf{t} \not\simeq \mathbb{1}$ and $\mathbf{t} \not\simeq \mathbb{0}$" to $[\mathrm{T}^{\mathrm{s}}_{\mathrm{case}}]$ and give two specialized rules as follows:

$$\frac{\Gamma \vdash \mathbf{e}_0 : t' \qquad \Gamma, x : t' \vdash \mathbf{e}_1 : t}{\Gamma \vdash \big((x = \mathbf{e}_0) \in \mathbf{t}\ ?\ \mathbf{e}_1 : \mathbf{e}_2\big) : \langle t \rangle}\ \mathbf{t} \simeq \mathbb{1} \qquad \frac{\Gamma \vdash \mathbf{e}_0 : t' \qquad \Gamma, x : t' \vdash \mathbf{e}_2 : t}{\Gamma \vdash \big((x = \mathbf{e}_0) \in \mathbf{t}\ ?\ \mathbf{e}_1 : \mathbf{e}_2\big) : \langle t \rangle}\ \mathbf{t} \simeq \mathbb{0}$$

13.3.2 Type system of the internal language

Figure 13.3 presents the typing rules $\mathcal{T}^{\mathrm{i}}_{\bot}$ of the internal language. These include a new rule for let expressions and a modified rule for λ-abstractions; the other rules are the same as those for the source language (except for the different syntax of typecases).

The rule $[\mathrm{T}_{\lambda}]$ for the internal language differs from that of the source language because it allows us to derive negations of arrow types. It is taken directly from Frisch, Castagna, and Benzaken (2008). We have discussed why such a rule is needed in Section 3.3 (that was for call-by-value, but the situation is similar for call-by-need). The explicitly typed and monomorphic setting makes it easier to define it here than in Chapter 3. Note that the negated arrows in t can be chosen freely providing that the intersection $\mathbb{I} \wedge t$ remains non-empty. This can look surprising. For example, it allows us to type $\mu f : (\mathsf{Int} \nrightarrow \mathsf{Int}).\ \lambda x.\ x$ as $(\mathsf{Int} \nrightarrow \mathsf{Int}) \wedge \neg(\mathsf{Bool} \to \mathsf{Bool})$ even though, disregarding the interface, the function does map Booleans to Booleans. But the language is explicitly typed, and thus we can't ignore interfaces (indeed, the function cannot be given the type $\mathsf{Bool} \to \mathsf{Bool}$).

The $[\mathrm{T}_{\mathsf{let}}]$ rule combines a standard rule for (monomorphic) binders with a union disjunction rule: it lets us decompose the type of e_1 as a union and type the body of the let once for each summand in the union. The purpose of this rule was hinted at in Section 12.2 and will be discussed again in Section 13.4, where we show that this rule – combined with the property on union types above – is central to this work: it is the key technical feature that ensures the soundness of the system (see in particular Section 13.4.2 later on). For the time being, just note that the type of e_1 can be decomposed in arbitrarily complex ways by applying

$$[\mathrm{T}_x]\ \frac{}{\Gamma \vdash x : t}\ \Gamma(x) = t \qquad [\mathrm{T}_c]\ \frac{}{\Gamma \vdash c : b_c}$$

$$[\mathrm{T}_\lambda]\ \frac{\forall i \in I.\quad \Gamma, f : \mathbb{I}, x : \langle \mathbf{T}'_i \rangle \vdash e : \langle \mathbf{T}_i \rangle}{\Gamma \vdash (\mu f{:}\mathbb{I}.\, \lambda x.\, e) : \mathbb{I} \wedge t}\ \begin{cases} \mathbb{I} = \bigwedge_{i \in I} \mathbf{T}'_i \mathrel{\not\leftrightarrow} \mathbf{T}_i \\ t = \bigwedge_{j \in J} \neg(t'_j \to t_j) \\ \mathbb{I} \wedge t \not\simeq \mathbb{0} \end{cases}$$

$$[\mathrm{T}_{\mathrm{app}}]\ \frac{\Gamma \vdash e_1 : \langle t' \to t \rangle \qquad \Gamma \vdash e_2 : t'}{\Gamma \vdash e_1\ e_2 : \langle t \rangle}$$

$$[\mathrm{T}_{\mathrm{pair}}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad [\mathrm{T}_{\mathrm{proj}}]\ \frac{\Gamma \vdash e : \langle t_1 \times t_2 \rangle}{\Gamma \vdash \pi_i\ e : \langle t_i \rangle}$$

$$[\mathrm{T}_{\mathrm{case}}]\ \frac{\begin{array}{c}\Gamma \vdash \varepsilon : \langle t' \rangle \\ \text{either } t' \leq \neg\mathbf{t} \text{ or } \Gamma, x : (t' \wedge \mathbf{t}) \vdash e_1 : t \qquad \text{either } t' \leq \mathbf{t} \text{ or } \Gamma, x : (t' \setminus \mathbf{t}) \vdash e_2 : t\end{array}}{\Gamma \vdash \big((x = \varepsilon) \in \mathbf{t}\ ?\ e_1 : e_2\big) : \langle t \rangle}$$

$$[\mathrm{T}_{\mathsf{let}}]\ \frac{\Gamma \vdash e_1 : \bigvee_{i \in I} t_i \qquad \forall i \in I.\ \Gamma, x : t_i \vdash e_2 : t}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : t}$$

$$[\mathrm{T}_{\leq}]\ \frac{\Gamma \vdash e : t'}{\Gamma \vdash e : t}\ t' \leq t$$

FIGURE 13.3 $\mathcal{T}_\perp^{\mathrm{i}}$: Typing rules of the internal language

subsumption. For example, if e_1 is a pair of type $(\mathsf{Int} \vee \mathsf{Bool}) \times (\mathsf{Int} \vee \mathsf{Bool})$, by applying $[\mathrm{T}_{\leq}]$ we can type it as $(\mathsf{Int}\times\mathsf{Int}) \vee (\mathsf{Int}\times\mathsf{Bool}) \vee (\mathsf{Bool}\times\mathsf{Int}) \vee (\mathsf{Bool}\times\mathsf{Bool})$ and then type e_2 once for each of the four summands.

The $[\mathrm{T}_\lambda]$ and $[\mathrm{T}_{\mathsf{let}}]$ rules introduce non-determinism respectively in the choice of the negations to introduce and of how to decompose types as unions. This would not complicate a practical implementation, since a type checker would only need to check the source language.

13.4 Proving type soundness

In this section, we prove the soundness property for our type system. We want to obtain the following familiar statement for the internal language.

> Let e be a well-typed, closed expression (i.e., $\varnothing \vdash e : t$ holds for some t). If $e \rightsquigarrow^* e'$ and e' cannot reduce, then e' is an answer and $\varnothing \vdash e' : t$.

Soundness for the source language then follows from this proposition.

13.10 PROPOSITION: If $\Gamma \vdash \mathbf{e} : t$, then $\Gamma \vdash \lceil \mathbf{e} \rceil : t$. □

Proof: Straightforward proof by induction on the typing derivation. □

We prove soundness using the two results of progress and subject reduction for the internal language, stated as follows.

Progress: Let Γ be a well-formed type environment. Let e be an expression that is well typed in Γ (that is, $\Gamma \vdash e : t$ holds for some t). Then either e is an answer, or e is of the form $E\lceil\lfloor x \rfloor\rceil$, or $\exists e'.\ e \rightsquigarrow e'$.

Subject reduction: Let Γ be a well-formed type environment. If $\Gamma \vdash e : t$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : t$.

The statement of progress is adapted to call-by-need: it applies also to expressions that are typed in a non-empty Γ, and it allows a well-typed expression to have the form $E\lceil\lfloor x \rfloor\rceil$. We recover the usual statement in empty environments because $E\lceil\lfloor x \rfloor\rceil$ can only be well typed in a non-empty environment.

We introduced the $\bot$ type for diverging expressions because assigning the type $\mathbb{0}$ to any expression causes unsoundness. We must hence ensure that no expression can be assigned the type $\mathbb{0}$. In well-formed type environments, we can prove this easily by induction.

13.11 LEMMA: If $\Gamma \vdash e : t$ and Γ is well formed, then $t \not\simeq \mathbb{0}$. □

Proof: By induction on the derivation of $\Gamma \vdash e : t$ and by case analysis on the last typing rule applied.

Case: $[\mathrm{T}_x]$ Straightforward since Γ is well formed.

Case: $[\mathrm{T}_c]$, $[\mathrm{T}_\lambda]$, $[\mathrm{T}_{\mathrm{app}}]$, $[\mathrm{T}_{\mathrm{proj}}]$, $[\mathrm{T}_{\mathrm{case}}]$ Straightforward.

Case: $[\mathrm{T}_{\mathrm{pair}}]$
By IH, t_1 and t_2 are non-empty.
Then, by definition of subtyping, $t_1 \times t_2$ is non-empty as well.

Case: $[\mathrm{T}_{\mathsf{let}}]$
By IH we derive that $\bigvee_{i \in I} t_i$ is non-empty.
Therefore, there exists an $i_0 \in I$ such that t_{i_0} is non-empty.
Then, $\Gamma, x\colon t_{i_0}$ is well formed, and, by IH, $t \not\simeq \mathbb{0}$.

Case: $[\mathrm{T}_{\leq}]$ Direct by IH. □

Set-theoretic types and semantic subtyping make proving subject reduction challenging. These difficulties have also motivated our choice of using call-by-need. We review and discuss in more detail this choice in order to explain the main challenges in the proof.

13.4.1 Call-by-name and call-by-need

In Section 12.2, we gave two reasons for our choice of call-by-need rather than call-by-name. One is that the system is only sound for call-by-name if we make assumptions on the semantics that might not hold in an extended language: for example, introducing an expression that can reduce non-deterministically either to an integer or to a Boolean would break soundness. The other reason is that, even when these assumptions hold (and when presumably call-by-name and call-by-need are observationally equivalent), call-by-need is better suited to the soundness proof.

Let us review the example from Section 12.2. Consider the source language function $\mu f\colon \mathbb{I}.\, \lambda x.\, (x, x)$, where $\mathbb{I} = (\mathsf{Int} \mathrel{\bar{\to}} \mathsf{Int} \mathbin{\bar{\times}} \mathsf{Int}) \wedge (\mathsf{Bool} \mathrel{\bar{\to}} \mathsf{Bool} \mathbin{\bar{\times}} \mathsf{Bool})$. It is well typed with type $\mathbb{I}$. By subsumption, we can also derive the type $(\mathsf{Int} \vee \mathsf{Bool}) \mathrel{\bar{\to}} (\mathsf{Int} \mathbin{\bar{\times}} \mathsf{Int}) \vee (\mathsf{Bool} \mathbin{\bar{\times}} \mathsf{Bool})$, which is a supertype of $\mathbb{I}$: in general we have $(t_1' \to t_1) \wedge (t_2' \to t_2) \leq (t_1' \vee t_2') \to (t_1 \vee t_2)$ and therefore $(t_1' \mathrel{\bar{\to}} t_1) \wedge (t_2' \mathrel{\bar{\to}} t_2) \leq (t_1' \vee t_2') \mathrel{\bar{\to}} (t_1 \vee t_2)$.

Therefore, if $\bar{\mathbf{e}}$ has type $\mathsf{Int} \vee \mathsf{Bool} \vee \bot$, the application $(\mu f\colon \mathbb{I}.\, \lambda x.\, (x, x))\ \bar{\mathbf{e}}$ is well typed with type $(\mathsf{Int} \mathbin{\bar{\times}} \mathsf{Int}) \vee (\mathsf{Bool} \mathbin{\bar{\times}} \mathsf{Bool}) \vee \bot$. Assume that $\bar{\mathbf{e}}$ can reduce either to an integer or to a Boolean: for instance, assume that both $\bar{\mathbf{e}} \rightsquigarrow 3$ and $\bar{\mathbf{e}} \rightsquigarrow \mathsf{true}$ can occur.

With call-by-name, $(\mu f\colon \mathbb{I}.\, \lambda x.\, (x, x))\ \bar{\mathbf{e}}$ reduces to $(\bar{\mathbf{e}}, \bar{\mathbf{e}})$; then, the two occurrences of $\bar{\mathbf{e}}$ reduce independently. It is intuitively unsound to type $(\bar{\mathbf{e}}, \bar{\mathbf{e}})$ as $(\mathsf{Int} \mathbin{\bar{\times}} \mathsf{Int}) \vee (\mathsf{Bool} \mathbin{\bar{\times}} \mathsf{Bool}) \vee \bot$: there is no guarantee that the two components of the pair will be of the same type once they are reduced. We can find terms that break subject reduction. Assume for example that there exists a Boolean "and" operation; then this typecase is well typed (as $\langle \mathsf{Bool} \rangle$) but unsafe:

$$(y = (\mu f\colon \mathbb{I}.\, \lambda x.\, (x, x))\ \bar{\mathbf{e}}) \in (\mathsf{Int} \mathbin{\bar{\times}} \mathsf{Int})\ ?\ \mathsf{true} : (\pi_1\ y\ \mathsf{and}\ \pi_2\ y)\ .$$

Since the application has type $\langle (\mathsf{Int} \mathbin{\bar{\times}} \mathsf{Int}) \vee (\mathsf{Bool} \mathbin{\bar{\times}} \mathsf{Bool}) \rangle$, to type the second branch of the typecase we can assume the type $((\mathsf{Int} \mathbin{\bar{\times}} \mathsf{Int}) \vee (\mathsf{Bool} \mathbin{\bar{\times}} \mathsf{Bool})) \setminus (\mathsf{Int} \mathbin{\bar{\times}} \mathsf{Int})$ for y. This is a subtype of $\mathsf{Bool} \mathbin{\bar{\times}} \mathsf{Bool}$ (it is actually equivalent to

(Bool ⊼ Bool) \ (⊥ × ⊥)). Therefore, both $\pi_1\ y$ and $\pi_2\ y$ have type ⟨Bool⟩. We deduce then that ($\pi_1\ y$ and $\pi_2\ y$) has type ⟨Bool⟩ as well (we assume that "and" is defined so as to handle arguments of type ⊥ correctly).

A possible reduction in a call-by-name semantics would be the following:

$$\begin{aligned}&(y = (\mu f\colon \mathbb{I}.\, \lambda x.\, (x, x))\ \bar{\mathbf{e}}) \in (\mathsf{Int} \barwedge \mathsf{Int})\ ?\ \mathsf{true} : (\pi_1\ y\ \mathsf{and}\ \pi_2\ y)\\ \rightsquigarrow\ &(y = (\bar{\mathbf{e}}, \bar{\mathbf{e}})) \in (\mathsf{Int} \barwedge \mathsf{Int})\ ?\ \mathsf{true} : (\pi_1\ y\ \mathsf{and}\ \pi_2\ y)\end{aligned}$$

(the typecase must force the evaluation of $(\bar{\mathbf{e}}, \bar{\mathbf{e}})$ to know which branch should be selected)

$$\rightsquigarrow^* (y = (\mathsf{true}, \bar{\mathbf{e}})) \in (\mathsf{Int} \barwedge \mathsf{Int})\ ?\ \mathsf{true} : (\pi_1\ y\ \mathsf{and}\ \pi_2\ y)$$

(now we know that the first branch is impossible, so the second is chosen)

$$\rightsquigarrow \pi_1\ (\mathsf{true}, \bar{\mathbf{e}})\ \mathsf{and}\ \pi_2\ (\mathsf{true}, \bar{\mathbf{e}}) \rightsquigarrow \mathsf{true}\ \mathsf{and}\ \bar{\mathbf{e}} \rightsquigarrow \bar{\mathbf{e}} \rightsquigarrow 3$$

The integer 3 is not a Bool: this disproves subject reduction for call-by-name if the language contains expressions like $\bar{\mathbf{e}}$. No such expressions exist in our language, but they could be introduced if we extended it with non-deterministic constructs like rnd(t) in the work of Frisch, Castagna, and Benzaken (2008).

Since we use a call-by-need semantics, instead, expressions such as $\bar{\mathbf{e}}$ do not pose problems for soundness. With call-by-need, $(\mu f\colon \mathbb{I}.\, \lambda x.\, (x, x))\ \bar{\mathbf{e}}$ reduces to $\mathsf{let}\ f = \mu f\colon \mathbb{I}.\, \lambda x.\, (x, x)\ \mathsf{in}\ \mathsf{let}\ x = \bar{\mathbf{e}}\ \mathsf{in}\ (x, x)$. The occurrences of x in the pair are only substituted when $\bar{\mathbf{e}}$ has been reduced to an answer, so they cannot reduce independently.

To ensure subject reduction, we allow the rule for let bindings to split union types which occur in the type of the bound term. This means that the following derivation is allowed.

$$\frac{\Gamma \vdash \bar{\mathbf{e}}\colon \mathsf{Int} \vee \mathsf{Bool} \quad \Gamma, x\colon \mathsf{Int} \vdash (x, x)\colon \mathsf{Int} \barwedge \mathsf{Int} \quad \Gamma, x\colon \mathsf{Bool} \vdash (x, x)\colon \mathsf{Bool} \barwedge \mathsf{Bool}}{\Gamma \vdash \mathsf{let}\ x = \bar{\mathbf{e}}\ \mathsf{in}\ (x, x)\colon (\mathsf{Int} \barwedge \mathsf{Int}) \vee (\mathsf{Bool} \barwedge \mathsf{Bool})}$$

13.4.2 Proving subject reduction: challenges

While the typing rule for let bindings is simple to describe, proving subject reduction for the two reduction rules that perform substitutions – $[\mathrm{R}^{v}_{\mathsf{let}}]$ and $[\mathrm{R}^{\mathrm{pair}}_{\mathsf{let}}]$ – is challenging.

For the reduction $\mathsf{let}\ x = v\ \mathsf{in}\ E^{\ulcorner}_{\llcorner}[x^{\urcorner}] \rightsquigarrow (E^{\ulcorner}_{\llcorner}[x^{\urcorner}])[v/x]$, we prove

> If $\Gamma \vdash v\colon \bigvee_{i\in I} t_i$, then there exists an $i_0 \in I$ such that $\Gamma \vdash v\colon t_{i_0}$. (⋆)

from a proposition corresponding to that discussed in Section 3.3.1:

> Let v be a value that is well typed in Γ (i.e., $\Gamma \vdash v\colon t'$ holds for some t'). Then, for every type t, we have either $\Gamma \vdash v\colon t$ or $\Gamma \vdash v\colon \neg t$.

Consider for example the reduction $\mathsf{let}\ x = v\ \mathsf{in}\ (x, x) \rightsquigarrow (v, v)$. If v has type Int ∨ Bool, then $\mathsf{let}\ x = v\ \mathsf{in}\ (x, x)$ has type (Int ⊼ Int) ∨ (Bool ⊼ Bool) as in the derivation above. Without the result (⋆), for (v, v) we could only derive the

type $(\mathsf{Int} \vee \mathsf{Bool}) \times (\mathsf{Int} \vee \mathsf{Bool})$, which is not a subtype of the type deduced for the redex. Applying the result ($\star$), we deduce that v has either type Int or Bool; in both cases (v, v) can be given the type $(\mathsf{Int} \mathbin{\text{⁂}} \mathsf{Int}) \vee (\mathsf{Bool} \mathbin{\text{⁂}} \mathsf{Bool})$.

The problem is similar to that for strict languages, and the solution is the same: ensuring that we can derive negations of arrow types for functions (in Chapter 3, type variables also posed difficulties, but we do not have them here). Since functions are explicitly typed here, we can reuse the typing rule from Frisch, Castagna, and Benzaken (2008) instead of the more involved approach from Chapter 3.

For the reduction

$$\mathsf{let}\ x = (e_1, e_2)\ \mathsf{in}\ E^{\lceil}_{\lfloor}[x]^{\rceil}_{\lfloor} \rightsquigarrow \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ (E^{\lceil}_{\lfloor}[x]^{\rceil}_{\lfloor})[(x_1, x_2)/x]\,,$$

instead, we use the following result.

> If $\Gamma \vdash (e_1, e_2) \colon \bigvee_{i \in I} t_i$, then there exist two types $\bigvee_{j \in J} t_j$ and $\bigvee_{k \in K} t_k$ such that $\Gamma \vdash e_1 \colon \bigvee_{j \in J} t_j$ and $\Gamma \vdash e_2 \colon \bigvee_{k \in K} t_k$ and
> $$\forall j \in J.\, \forall k \in K.\, \exists i \in I.\ t_j \times t_k \leq t_i\,.$$

This is the result we need for the proof: $\mathsf{let}\ x = (e_1, e_2)\ \mathsf{in}\ E^{\lceil}_{\lfloor}[x]^{\rceil}_{\lfloor}$ is typed by assigning a union type to (e_1, e_2) and then typing $E^{\lceil}_{\lfloor}[x]^{\rceil}_{\lfloor}$ once for every t_i in the union, while the reduct $\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ (E^{\lceil}_{\lfloor}[x]^{\rceil}_{\lfloor})[(x_1, x_2)/x]$ must be typed by typing e_1 and e_2 with two union types and then typing the substituted expression with every product $t_j \times t_k$. Showing that each $t_j \times t_k$ is a subtype of a t_i ensures that the substituted expression is well typed. The proof consists in recognizing that the union $\bigvee_{i \in I} t_i$ must be a decomposition into a union of some type $t_1 \times t_2$ and that therefore t_1 and t_2 can be decomposed separately into two unions.

All these results rely on the distinction between types that contain $\bot$ and those that do not: they would not hold if we assumed that every type implicitly contains $\bot$.

Despite some technical difficulties, call-by-need seems quite suited to the soundness proof. Hence, it would probably be best to use it for the proof even if we assumed explicitly that the language does not include problematic expressions like $\mathsf{rnd}(t)$. Soundness would then also hold for a call-by-name semantics that is observationally equivalent to call-by-need.

In the following, we develop the proof in detail. In Section 13.4.3, we study the decomposition of product types into unions to derive the result we need for subject reduction for $[\mathrm{R}^{\mathrm{pair}}_{\mathsf{let}}]$ (Lemma 13.19). Then, in Section 13.4.4, we derive the other intermediate results we need, including those needed to deal with $[\mathrm{R}^{v}_{\mathsf{let}}]$ (Lemma 13.26 and Corollary 13.27). Finally, in Section 13.4.5, we prove progress and subject reduction.

13.4.3 Decompositions of product types

A standard result in semantic subtyping – rephrased here from Frisch, Castagna, and Benzaken (2008) – is that we can put types into a disjunctive normal form while preserving their interpretation as sets of values.

13.12 DEFINITION (Atoms and disjunctive normal forms): An *atom* is a type of the form $\bot$, b, $t_1 \times t_2$, or $t_1 \to t_2$.

A *disjunctive normal form* is a finite set of pairs of finite sets of atoms: that is, a set $\{ (P_i, N_i) \mid i \in I \}$ where I is finite and where, for each i, P_i and N_i are finite sets of atoms.

We extend the definition of $[\![\cdot]\!]$ to disjunctive normal forms by defining $[\![\{ (P_i, N_i) \mid i \in I \}]\!] \stackrel{\text{def}}{=} \bigcup_{i \in I} \left(\bigcap_{t \in P_i} [\![t]\!] \setminus \bigcup_{t \in N_i} [\![t]\!] \right)$. □

13.13 DEFINITION: The functions dnf and $\overline{\mathsf{dnf}}$ from types to disjunctive normal forms are defined by mutual induction as follows:

$$\begin{array}{rlrl} \mathsf{dnf}(t) &= \{(\{t\}, \varnothing)\} & \overline{\mathsf{dnf}}(t) &= \{(\varnothing, \{t\})\} \quad \text{if } t \text{ atom} \\ \mathsf{dnf}(t_1 \vee t_2) &= \mathsf{dnf}(t_1) \cup \mathsf{dnf}(t_2) & \overline{\mathsf{dnf}}(t_1 \vee t_2) &= \overline{\mathsf{dnf}}(t_1) \sqcap \overline{\mathsf{dnf}}(t_2) \\ \mathsf{dnf}(\neg t) &= \overline{\mathsf{dnf}}(t) & \overline{\mathsf{dnf}}(\neg t) &= \mathsf{dnf}(t) \\ \mathsf{dnf}(\mathbb{0}) &= \varnothing & \overline{\mathsf{dnf}}(\mathbb{0}) &= \{(\varnothing, \varnothing)\} \end{array}$$

where $\{ (P_i, N_i) \mid i \in I \} \sqcap \{ (P_j, N_j) \mid j \in J \} \stackrel{\text{def}}{=} \{ (P_i \cup P_j, N_i \cup N_j) \mid i \in I, j \in J \}$. □

Induction in this definition is well-founded because it is never applied below type constructors, and contractivity ensures that there are no infinite chains of union and negation in types (as explained in Section 2.2).

13.14 PROPOSITION: For every type t, $[\![t]\!] = [\![\mathsf{dnf}(t)]\!]$. □

Proof: The stronger claim $\forall t.\ [\![t]\!] = [\![\mathsf{dnf}(t)]\!] = \mathsf{Domain} \setminus [\![\overline{\mathsf{dnf}}(t)]\!]$ can be proven easily by induction. □

A further result is that any subtype of $\mathbb{1} \times \mathbb{1}$ (that is, any type whose interpretation only contains pairs) can be expressed as a product decomposition, that is, a finite union of product atoms $(t_1^1 \times t_1^2) \vee \cdots \vee (t_n^1 \times t_n^2)$. To develop the result we need for subject reduction of the rule $[\mathrm{R}^{\mathrm{pair}}_{\mathrm{let}}]$, we study these decompositions of product types. In particular, we introduce a specific form of product decomposition (*fully disjoint* decompositions) that is convenient to derive the result we need.

13.15 DEFINITION (Product decomposition): A *product decomposition* Π is a finite set of *product atoms*, that is, of types of the form $t_1 \times t_2$.

We say that a product decomposition $\Pi = \{ t_i^1 \times t_i^2 \mid i \in I \}$ is *fully disjoint* if $\forall i \in I.\ t_i^1 \times t_i^2 \not\simeq \mathbb{0}$ and if the following conditions hold for all $i_1 \neq i_2 \in I$:

- $(t_{i_1}^1 \wedge t_{i_2}^1 \simeq \mathbb{0}) \vee (t_{i_1}^1 \simeq t_{i_2}^1)$;
- $(t_{i_1}^2 \wedge t_{i_2}^2 \simeq \mathbb{0}) \vee (t_{i_1}^2 \simeq t_{i_2}^2)$. □

13.16 LEMMA: For every type t such that $t \leq \mathbb{1} \times \mathbb{1}$, there exists a product decomposition Π such that $t \simeq \bigvee_{t_1 \times t_2 \in \Pi} t_1 \times t_2$. □

Proof in appendix (p. 293).

13.17 LEMMA: For every product decomposition Π, there exists a product decomposition Π' such that Π' is fully disjoint, that $\bigvee_{t\in\Pi} t \simeq \bigvee_{t'\in\Pi'} t'$, and that $\forall t' \in \Pi.\, \exists t \in \Pi.\, t' \leq t$. □

Proof in appendix (p. 294).

13.18 LEMMA: Let $\Pi = \{ t_i^1 \times t_i^2 \mid i \in I \}$ be a fully disjoint product decomposition and let t^1 and t^2 be two types such that $t^1 \times t^2 \simeq \bigvee_{i\in I} t_i^1 \times t_i^2$. Then, $t^1 \simeq \bigvee_{i\in I} t_i^1$, $t^2 \simeq \bigvee_{i\in I} t_i^2$, and $\forall i_1, i_2 \in I.\, \exists i \in I.\, t_{i_1}^1 \times t_{i_2}^2 \leq t_i^1 \times t_i^2$. □

Proof in appendix (p. 295).

13.19 LEMMA: If $\Gamma \vdash (e_1, e_2) \colon \bigvee_{i\in I} t_i$, then there exist two types $\bigvee_{j\in J} t_j$ and $\bigvee_{k\in K} t_k$ such that

$$\Gamma \vdash e_1 \colon \bigvee_{j\in J} t_j \quad \Gamma \vdash e_2 \colon \bigvee_{k\in K} t_k \quad \forall j \in J.\, \forall k \in K.\, \exists i \in I.\, t_j \times t_k \leq t_i\,. \qquad \square$$

Proof in appendix (p. 296).

13.4.4 Additional results

We derive here the other auxiliary results we need to prove progress and subject reduction. Most are standard results, and they are developed similarly to those in Section 3.3.

13.20 LEMMA (Weakening): Let Γ and Γ' be two type environments such that, whenever $x \in \mathsf{dom}(\Gamma)$, we have $x \in \mathsf{dom}(\Gamma')$ and $\Gamma'(x) \leq \Gamma(x)$.
If $\Gamma \vdash e \colon t$, then $\Gamma' \vdash e \colon t$. □

Proof in appendix (p. 296).

13.21 LEMMA (Admissibility of intersection introduction): If $\Gamma \vdash e \colon t_1$ and $\Gamma \vdash e \colon t_2$, then $\Gamma \vdash e \colon t_1 \wedge t_2$. □

Proof in appendix (p. 297).

13.22 LEMMA (Expression substitution): If $\Gamma, x \colon t' \vdash e \colon t$ and $\Gamma \vdash e' \colon t'$, then $\Gamma \vdash e[e'/x] \colon t$. □

Proof: By induction on the typing derivation for e. □

13.23 LEMMA (Generation): Let Γ be a well-formed type environment and let a be an answer such that $\Gamma \vdash a : t$ holds. Then:

- if $t = \langle t_1 \to t_2 \rangle$, then a is of the form $\mu f : \mathbb{I}.\, \lambda x.\, e$ or $\mathsf{let}\ x = e\ \mathsf{in}\ a'$;
- if $t = \langle t_1 \times t_2 \rangle$, then a is of the form (e_1, e_2) or $\mathsf{let}\ x = e\ \mathsf{in}\ a'$. □

Proof in appendix (p. 299).

13.24 LEMMA: If ε is well typed in an environment Γ (i.e., if $\Gamma \vdash \varepsilon : t$ holds for some t), then $\Gamma \vdash \varepsilon : \mathsf{typeof}(\varepsilon)$. □

Proof: By induction on ε. If it is a variable, a constant, or a function, the result is straightforward (note that $\mathbb{0} \to \mathbb{1}$ is greater than any functional type). If it is a pair, we apply the induction hypothesis and use rule $[\mathrm{T}_{\mathrm{pair}}]$. □

13.25 LEMMA: Let $\bar{\varepsilon}$ be an expression generated by the grammar

$$\bar{\varepsilon} ::= c \mid \mu f : \mathbb{I}.\, \lambda x.\, e \mid (\bar{\varepsilon}, \bar{\varepsilon})$$

(that is, an expression ε without variables). For every $\mathbf{t}$, either $\mathsf{typeof}(\bar{\varepsilon}) \leq \mathbf{t}$ or $\mathsf{typeof}(\bar{\varepsilon}) \leq \neg\mathbf{t}$. □

Proof in appendix (p. 299).

13.26 LEMMA: Let v be a value that is well typed in Γ (i.e., $\Gamma \vdash v : t'$ holds for some t'). Then, for every t, we have either $\Gamma \vdash v : t$ or $\Gamma \vdash v : \neg t$. □

Proof in appendix (p. 300).

13.27 COROLLARY: If $\Gamma \vdash v : \bigvee_{i \in I} t_i$, then, for some $i_0 \in I$, $\Gamma \vdash v : t_{i_0}$. □

Proof in appendix (p. 300).

13.28 LEMMA: Let $\mathbb{I} = \bigwedge_{i \in I} t'_i \to t_i$ (with $|I| > 0$) be a type. There exists a type $\mathbb{I}' = \bigwedge_{k \in K} t'_k \to t_k$ (with $|K| > 0$) such that:

- $\mathbb{I} \simeq \mathbb{I}'$;
- $\forall k_1 \neq k_2 \in K.\, t_{k_1} \wedge t_{k_2} \simeq \mathbb{0}$;
- if $\Gamma \vdash (\mu f : \mathbb{I}.\, \lambda x.\, e) : \mathbb{I}$, then $\forall k \in K.\ \ \Gamma, f : \mathbb{I}, x : t'_k \vdash e : t_k$. □

Proof in appendix (p. 301).

13.4.5 Progress and subject reduction

13.29 THEOREM (Progress): Let Γ be a well-formed type environment. Let e be an expression that is well typed in Γ (that is, $\Gamma \vdash e : t$ holds for some t). Then e is an answer, or e is of the form $E\lceil x \rceil$, or $\exists e'.\ e \rightsquigarrow e'$. □

Proof in appendix (p. 301). By induction on the derivation of $\Gamma \vdash e : t$ and by case analysis on the last typing rule applied. In the cases for $[\mathrm{T}_{\mathrm{app}}]$ and $[\mathrm{T}_{\mathrm{pair}}]$, we use Lemma 13.23. In that for $[\mathrm{T}_{\mathrm{case}}]$, we use Lemma 13.25.

13.30 THEOREM (Subject reduction): Let Γ be a well-formed type environment. If $\Gamma \vdash e : t$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : t$. □

Proof in appendix (p. 303). By induction on the derivation of $\Gamma \vdash e : t$ and by case analysis on the last typing rule applied. In the case for $[\mathrm{T}_{\mathrm{app}}]$, we use Corollaries 13.6 and 13.7 and Lemma 13.28. For $[\mathrm{T}_{\mathrm{proj}}]$ and $[\mathrm{T}_{\mathrm{case}}]$, we use Lemma 13.11; for $[\mathrm{T}_{\mathrm{case}}]$, we also use Lemmas 13.21 and 13.24. For $[\mathrm{T}_{\mathsf{let}}]$, if the reduction occurs by $[\mathrm{R}^{v}_{\mathsf{let}}]$, we use Lemma 13.22 and Corollary 13.27; if it occurs by $[\mathrm{R}^{\mathsf{pair}}_{\mathsf{let}}]$, we use Lemmas 13.19, 13.20 and 13.22; if it occurs by $[\mathrm{R}^{\mathsf{let}}_{\mathsf{let}}]$, we use Lemma 13.20.

We use the following lemma to recover the standard statement of progress for empty type environments.

13.31 LEMMA: If $\Gamma \vdash E\lceil x \rceil : t$, then $x \in \mathsf{dom}(\Gamma)$. □

Proof in appendix (p. 307).

We obtain soundness as a corollary of the previous results.

13.32 COROLLARY (Type soundness): Let e be a well-typed, closed expression (that is, $\varnothing \vdash e : t$ holds for some t). If $e \rightsquigarrow^* e'$ and e' cannot reduce, then e' is an answer and $\varnothing \vdash e' : t$. □

Proof: Corollary of Theorems 13.29 and 13.30 and Lemma 13.31. □

13.33 COROLLARY (Type soundness for the source language): Let $\mathbf{e}$ be a well-typed, closed source language expression (that is, $\varnothing \vdash \mathbf{e} : t$ holds for some t). If $\lceil \mathbf{e} \rceil \rightsquigarrow^* e'$ and e' cannot reduce, then e' is an answer and $\varnothing \vdash e' : t$. □

Proof: Corollary of Proposition 13.10 and Corollary 13.32. □

14 Discussion

We have described how to adapt the framework of semantic subtyping to non-strict languages. We have done so by reusing the subtyping relation of Frisch, Castagna, and Benzaken (2008) unchanged (except for the addition of $\bot$) and reworking the typing rules to avoid the pathological behaviour of semantic subtyping on empty types. Notably, typing rules for constructs like application and projection must handle $\bot$ explicitly. This ensures soundness for call-by-need.

Using our approach, subtyping still behaves set-theoretically: we can still see union, intersection, and negation in types as the corresponding operations on sets. We can still use intersection types to express function overloading since familiar subtyping judgments like

$$(t_1' \to t_1) \wedge (t_2' \to t_2) \leq (t_1' \vee t_2') \to (t_1 \vee t_2)$$

still hold. Moreover, an advantage of this approach is that we can reuse directly the existing results on semantic subtyping (especially as concerns the decision procedure): we have added $\bot$, but it is treated just like a new base type.

The type $\bot$ we introduce has no analogue in well-known type systems like the simply typed λ-calculus or Hindley-Milner typing. However, $\bot$ never appears explicitly in programs (it does not appear in types of the forms $\mathbf{T}$ and $\mathbf{t}$ given at the beginning of Section 13.2). Hence, programmers do not need to use it and to consider the difference between terminating and non-terminating types while writing function interfaces or typecases. Still, sub-expressions of a program can have types with explicit $\bot$ (e.g., the type $\mathsf{Int} \vee \bot$). Such types are not expressible in the grammar of types visible to the programmer. Accordingly, error reporting should be more elaborate to avoid mentioning internal types that are unknown to the programmer.

In the next section, we discuss the interpretation of types and its relationship with the expressions that are actually definable in the language; we explain how we could look for an interpretation that is a better fit for non-strict languages. Then, we present a few directions for future work.

14.1 On the interpretation of types

We have shown that a set-theoretic interpretation of types, adapted to take into account divergence (Definition 13.3), can be the basis for designing a sound type system for a language with non-strict semantics. In this section, we analyze the relation between this interpretation and the expressions that we can define in the language.

Let us first recap some notions of semantic subtyping. The initial intuition which guides semantic subtyping is to see a type as the set of values of that type

in the language we consider. However, we cannot use this intuition directly to define the interpretation, because of a problem of circularity (as discussed in Section 2.1.2). Frisch, Castagna, and Benzaken (2008) solve this by giving an interpretation $[\![\cdot]\!]$ of types as subsets of an interpretation domain where finite relations replace λ-abstractions. Then, they show the result

$$\forall t_1, t_2.\ [\![t_1]\!] \subseteq [\![t_2]\!] \iff [\![t_1]\!]^{\mathcal{V}} \subseteq [\![t_2]\!]^{\mathcal{V}} \qquad \text{where } [\![t]\!]^{\mathcal{V}} \stackrel{\text{def}}{=} \{\, v \mid \varnothing \vdash v \colon t \,\}$$

meaning that a type t_1 is a subtype of a type t_2 if and only if every value v that can be assigned the type t_1 can also be assigned the type t_2. As we have said in Section 2.1.2, this means that we can really reason on subtyping by reasoning on inclusion between sets of values, with both theoretical and practical advantages.

In the following we discuss how an analogous result could hold with a non-strict semantics. First of all, clearly the correspondence cannot be between interpretations of types and sets of values in our case, since then we would identify $\bot$ with $\mathbb{0}$. Hence we should consider, rather than values, sets of "results" of some kind, including (a representation of) divergence. However, whichever notion of result we consider, it is hard to define an interpretation of types such that the desired correspondence holds, that is, such that a type t corresponds to the set of all possible results of expressions of type t.

As one could expect, the key challenge is to provide an interpretation where, as it seems sensible, an arrow type $t_1 \to t_2$ corresponds to the set of λ-abstractions $\{\, (\mu f \colon \mathbb{I}.\, \lambda x.\, e) \mid \varnothing \vdash (\mu f \colon \mathbb{I}.\, \lambda x.\, e) \colon t_1 \to t_2 \,\}$. Our proposed definition of $[\![\cdot]\!]$ (Definition 13.3) is sound with respect to this correspondence, but not complete, that is, not precise enough. We devote the rest of this section to explain why and to discuss the possibility of obtaining a complete definition.

Consider the type $\mathsf{Int} \to \mathbb{0}$. By Definition 13.3, we have

$$\begin{aligned} [\![\mathsf{Int} \to \mathbb{0}]\!] &= \left\{\, \{\, (d^i, d^i_\Omega) \mid i \in I \,\} \;\middle|\; \forall i \in I.\, d^i \in [\![\mathsf{Int}]\!] \implies d^i_\Omega \in [\![\mathbb{0}]\!] \,\right\} \\ &= \left\{\, \{\, (d^i, d^i_\Omega) \mid i \in I \,\} \;\middle|\; \forall i \in I.\, d^i \notin [\![\mathsf{Int}]\!] \,\right\} \end{aligned}$$

(since $[\![\mathbb{0}]\!] = \varnothing$, the implication can only be satisfied if $d \notin [\![\mathsf{Int}]\!]$). This type is not empty, therefore, if a result similar to that of Frisch, Castagna, and Benzaken (2008) held, we would expect to be able to find a function $\mu f \colon \mathbb{I}.\, \lambda x.\, e$ such that $\varnothing \vdash (\mu f \colon \mathbb{I}.\, \lambda x.\, e) \colon \mathsf{Int} \to \mathbb{0}$. Alas, no such function can be defined in our language. This is easy to check: interfaces must include $\bot$ in the codomain of every arrow (since they use the $\twoheadrightarrow$ form), so no interface can be a subtype of $\mathsf{Int} \to \mathbb{0}$. Lifting this syntactic restriction to allow any arrow type in interfaces would not solve the problem: for a function to have type $\mathsf{Int} \to \mathbb{0}$, its body must have type $\mathbb{0}$, which is impossible and indeed *must* be impossible for the system to be sound. It is therefore to be expected that $\mathsf{Int} \to \mathbb{0}$ is uninhabited in the language. This means that our current definition of $[\![\mathsf{Int} \to \mathbb{0}]\!]$ as a non-empty type is imprecise.

Changing $[\![\cdot]\!]$ to make the types of the form $t \to \mathbb{0}$ empty is easy, but it does not solve the problem in general. Using intersection types we can build more challenging examples: for instance, $(\mathsf{Int} \vee \mathsf{Bool} \to \mathsf{Int}) \wedge (\mathsf{Int} \vee \mathsf{String} \to \mathsf{Bool})$. While neither codomain is empty, and neither arrow should be empty, the

whole intersection should: no function, when given an Int as argument, can return a result which is both an Int and a Bool.

In the call-by-value case, it makes sense to have $\mathsf{Int} \to \mathbb{0}$ and the intersection type above be non-empty, because they are both inhabited by functions that diverge on integers. This is because divergence is not represented in the types (or, to put it differently, because it is represented by the type $\mathbb{0}$). A type like $t_1 \to t_2$ is interpreted as a specification of *partial correctness*: a function of this type, when given an argument in t_1, either diverges or returns a result in t_2. In our system, we have introduced a separate non-empty type for divergence. Hence, we should see a type as specifying *total* correctness, where divergence is allowed only for functions whose codomain includes $\bot$.

Let us consider again the current interpretation of arrow types.

$$[\![t_1 \to t_2]\!] = \left\{ \{ (d^i, d^i_\Omega) \mid i \in I \} \;\middle|\; \forall i \in I.\, d^i \in [\![t_1]\!] \implies d^i_\Omega \in [\![t_2]\!] \right\}$$

An arrow type is seen as a set of finite relations: we represent functions extensionally and approximate them with all their finite representations. We use relations instead of functions to account for non-determinism. Within a relation, a pair (d, d') means that the function returns the output d' on the input d; a pair (d, Ω) that the function crashes with a runtime type error on d; by contrast, divergence is represented simply by the absence of a pair. In this way, as said above, a function diverging on some element of $[\![t_1]\!]$ could erroneously belong to the set even if $[\![t_2]\!]$ does not contain $\bot$.

To formalize the requirement of totality on the domain, we could modify the definition in this way:

$$[\![t_1 \to t_2]\!] = \left\{ \{ (d^i, d^i_\Omega) \mid i \in I \} \;\middle|\; [\![t_1]\!] \subseteq \{ d^i \mid i \in I \} \text{ and } \forall i \in I.\, d^i \in [\![t_1]\!] \implies d^i_\Omega \in [\![t_2]\!] \right\}$$

However, if we consider only finite relations as above, the definition makes no sense, since $[\![t_1]\!] \subseteq \{ d^i \mid i \in I \}$ can hold only when $[\![t_1]\!]$ is finite, whereas types can have infinite interpretations. As discussed in Section 2.1.2, the restriction to finite relations is needed because otherwise Domain would have to contain $\mathcal{P}(\mathsf{Domain} \times \mathsf{Domain}_\Omega)$ (writing Domain_Ω for $\mathsf{Domain} \cup \{\Omega\}$), which is impossible by cardinality.

Frisch, Castagna, and Benzaken (2008) point out this problem of cardinality and use finite relations in the domain to avoid it. They motivate this choice with the observation that, while finite relations are not really appropriate to describe functions in a language (since these might have an infinite domain), they are suitable to describe types as far as subtyping is concerned. It can be shown that

$$\forall t_1, t'_1, t_2, t'_2.\ [\![t'_1 \to t_1]\!] \subseteq [\![t'_2 \to t_2]\!] \iff ([\![t'_1]\!] \rightharpoonup [\![t_1]\!]) \subseteq ([\![t'_2]\!] \rightharpoonup [\![t_2]\!])$$

where

$$X \rightharpoonup Y \stackrel{\text{def}}{=} \{ R \in \mathcal{P}(\mathsf{Domain} \times \mathsf{Domain}_\Omega) \mid \forall (d, d') \in R.\, d \in X \implies d' \in Y \}$$

builds the set of possibly infinite relations. This can be generalized to more complex types:

$$\left[\!\!\left[\textstyle\bigwedge_{i \in P} t'_i \to t_i \right]\!\!\right] \subseteq \left[\!\!\left[\textstyle\bigvee_{i \in N} t'_i \to t_i \right]\!\!\right] \iff \bigcap_{i \in P} \left([\![t'_i]\!] \rightharpoonup [\![t_i]\!] \right) \subseteq \bigcup_{i \in N} \left([\![t'_i]\!] \rightharpoonup [\![t_i]\!] \right).$$

The equivalence above is used by Frisch, Castagna, and Benzaken (2008), through the notion of *extensional interpretation*, to argue that the restriction to finite relations does not impair the precision of subtyping.

Let us try to proceed analogously in our case: that is, to find a new interpretation of types that matches the behaviour of possibly infinite relations that are total on their domain, while introducing an approximation to ensure that the domain is definable. The latter point means, notably, that functions must be represented as finite objects. The following definition of a *model* specifies the properties that such an interpretation should satisfy.

14.1 DEFINITION (Model): A set $\mathsf{Domain}^{\mathsf{m}}$ along with a function $[\![\cdot]\!]^{\mathsf{m}} \colon \mathsf{Type} \to \mathcal{P}(\mathsf{Domain}^{\mathsf{m}})$ is a *model* if the following hold:

1. the set $\mathsf{Domain}^{\mathsf{m}}$ satisfies

$$\mathsf{Domain}^{\mathsf{m}} = \{\bot\} \uplus \mathsf{Const} \uplus (\mathsf{Domain}^{\mathsf{m}} \times \mathsf{Domain}^{\mathsf{m}}) \uplus \mathsf{Domain}^{\mathsf{m}}_{\mathsf{fun}}$$

   for some set $\mathsf{Domain}^{\mathsf{m}}_{\mathsf{fun}}$;

2. for all b, t, t_1, and t_2,

$$\begin{aligned} [\![\bot]\!]^{\mathsf{m}} &= \{\bot\} & [\![t_1 \vee t_2]\!]^{\mathsf{m}} &= [\![t_1]\!]^{\mathsf{m}} \cup [\![t_2]\!]^{\mathsf{m}} \\ [\![b]\!]^{\mathsf{m}} &= \mathbb{B}(b) & [\![\neg t]\!]^{\mathsf{m}} &= \mathsf{Domain}^{\mathsf{m}} \setminus [\![t]\!]^{\mathsf{m}} \\ [\![t_1 \times t_2]\!]^{\mathsf{m}} &= [\![t_1]\!]^{\mathsf{m}} \times [\![t_2]\!]^{\mathsf{m}} & [\![\mathbb{0}]\!]^{\mathsf{m}} &= \varnothing \end{aligned}$$

3. for all t_1 and t_2, $[\![t_1 \to t_2]\!]^{\mathsf{m}} \subseteq [\![\mathbb{0} \to \mathbb{1}]\!]^{\mathsf{m}} = \mathsf{Domain}^{\mathsf{m}}_{\mathsf{fun}}$;

4. for every finite, non-empty intersection $\bigwedge_{i \in P} t'_i \to t_i$ and every finite union $\bigvee_{i \in N} t'_i \to t_i$,

$$\begin{aligned} [\![\textstyle\bigwedge_{i \in P} t'_i \to t_i]\!]^{\mathsf{m}} \subseteq [\![\textstyle\bigvee_{i \in N} t'_i \to t_i]\!]^{\mathsf{m}} \iff \\ \textstyle\bigcap_{i \in P} \left([\![t'_i]\!]^{\mathsf{m}} \twoheadrightarrow [\![t_i]\!]^{\mathsf{m}}\right) \subseteq \bigcup_{i \in N} \left([\![t'_i]\!]^{\mathsf{m}} \twoheadrightarrow [\![t_i]\!]^{\mathsf{m}}\right) \end{aligned}$$

   where

$$\begin{aligned} X \twoheadrightarrow Y \stackrel{\text{def}}{=} \big\{ R \in \mathcal{P}(\mathsf{Domain}^{\mathsf{m}} \times \mathsf{Domain}^{\mathsf{m}}) \;\big|\; \\ \mathsf{dom}(R) \supseteq X \text{ and } \forall (d, d') \in R.\, d \in X \implies d' \in Y \big\} \end{aligned}$$

   (with $\mathsf{dom}(R) = \{\, d \mid \exists d'.\, (d, d') \in R \,\}$). □

We set the above conditions for an interpretation $[\![\cdot]\!]^{\mathsf{m}} \colon \mathsf{Type} \to \mathcal{P}(\mathsf{Domain}^{\mathsf{m}})$ to form a model. The first constrains $\mathsf{Domain}^{\mathsf{m}}$ to have the same structure as Domain, except that we do not fix the subset $\mathsf{Domain}^{\mathsf{m}}_{\mathsf{fun}}$ in which arrow types are interpreted. The second and third conditions fix the definition of $[\![\cdot]\!]^{\mathsf{m}}$ completely except for arrow types. The fourth condition ensures that subtyping on arrow types behaves as set containment between the sets of relations that are total on the domains of the arrow types.[1]

1 We do not use the error element Ω in the definition of $X \twoheadrightarrow Y$, because the requirement of totality makes it unnecessary: errors on a given input can be represented in a relation by the absence of a pair.

An interesting result is that, even though we do not know whether an interpretation of types which is a model can actually be found, we can compare such a hypothetical model with the interpretation $[\![\cdot]\!]$ defined in Section 13.1. Indeed $[\![\cdot]\!]$ turns out to be a sound approximation of every model; that is, the subtyping relation $\leq$ defined in Definition 13.4 from $[\![\cdot]\!]$ is contained in every subtyping relation $\leq_{[\![\cdot]\!]^{\mathsf{m}}}$ defined from some interpretation $[\![\cdot]\!]^{\mathsf{m}}$ that is a model. We prove here that this holds for non-recursive types. The proof relies on the following lemma, which is analogous to (one implication of) Lemma 2.16.

14.2 LEMMA: Let $[\![\cdot]\!]^{\mathsf{m}} : \mathsf{Type} \to \mathcal{P}(\mathsf{Domain}^{\mathsf{m}})$ be a model. Let P and N be finite sets of types of the form $t_1 \to t_2$, with $P \neq \varnothing$. Then:

$$\begin{aligned}&\exists t'_1 \to t'_2 \in N.\ [\![t'_1 \setminus \bigvee_{t_1 \to t_2 \in P} t_1]\!]^{\mathsf{m}} = \varnothing \text{ and}\\ &\quad \big(\forall P' \subsetneq P.\ [\![t'_1 \setminus \bigvee_{t_1 \to t_2 \in P'} t_1]\!]^{\mathsf{m}} = \varnothing \text{ or } [\![\bigwedge_{t_1 \to t_2 \in P \setminus P'} t_2 \setminus t'_2]\!]^{\mathsf{m}} = \varnothing\big)\\ &\qquad\qquad \Longrightarrow \bigcap_{t_1 \to t_2 \in P} [\![t_1 \to t_2]\!]^{\mathsf{m}} \subseteq \bigcup_{t_1 \to t_2 \in N} [\![t_1 \to t_2]\!]^{\mathsf{m}}\end{aligned}$$

□

Proof in appendix (p. 307).

14.3 PROPOSITION: Let $[\![\cdot]\!]^{\mathsf{m}} : \mathsf{Type} \to \mathcal{P}(\mathsf{Domain}^{\mathsf{m}})$ be a model. Let t_1 and t_2 be two finite (that is, non-recursive) types. If $[\![t_1]\!] \subseteq [\![t_2]\!]$, then $[\![t_1]\!]^{\mathsf{m}} \subseteq [\![t_2]\!]^{\mathsf{m}}$. □

We conjecture that the result holds for recursive types too, but that proof is left for future work.

Showing that models exist would be important to understand the connection between our types and the semantics. To use a model $[\![\cdot]\!]^{\mathsf{m}}$ to define subtyping for the use of a type checker, though, we would also need to show that the resulting definition is decidable. Otherwise, $[\![\cdot]\!]$ would remain the definition used in a practical implementation since it is sound and decidable, though less precise (that is, incomplete with respect to the correspondence that we have discussed).

14.2 Future work

A natural goal for future work is to search for an alternative interpretation of types that satisfies the conditions of Definition 14.1. Other directions include the following.

IMPLICIT TYPING AND POLYMORPHISM: It would be interesting to recast the work in this chapter in an implicitly typed setting like that of Chapter 3. The difficulty is that the approach described in Section 3.3 to derive negation types for functions cannot be reused here without modification. This is because it allows us to derive negation types only for functions that are closed (without free variables). In Section 3.3, this is not a problem: we need to derive negation

types only for values, and only closed functions are values. Here, instead, we need the rule to be applicable also to functions with free variables; then, the relation $\lambda x.\, e \not\downarrow_n t' \to t$ must be changed to account for the type environment, but this change is problematic.

Giving an implicitly typed presentation should also allow us to extend the system with polymorphism without difficulty. In contrast, adding polymorphism to the explicitly typed language would require us to give a more complex semantics similar to that of Castagna et al. (2014), with explicit tracking of instantiations.

ENSURING SOUNDNESS BY CHANGING SUBTYPING: A different approach to use semantic subtyping with non-strict languages would be to change the interpretation of types (and, as a result, the definition of subtyping) to avoid the pathological behaviour on $\mathbb{0}$, and then to use standard typing rules. This would avoid the need to introduce $\bot$ explicitly in the types.

We have explored this alternative approach, but we have not found it promising. A modified subtyping relation loses important properties – especially results on the decomposition of product types – that we need to prove soundness via subject reduction. The approach we have adopted here is more suited to this technical work. However, a modified relation could yield a different type system for the source language, provided that we can relate it to the current system for the internal language.

TRACK TERMINATION MORE PRECISELY: It would also be interesting to study more expressive typing rules that can track termination with some precision. For example, we could change the application rule so that it does not always introduce $\bot$. In function interfaces, some arrows could include $\bot$ and some could not: then, overloaded function types would express that a function behaves differently on terminating or diverging arguments. For example, $\lambda x.\, x+1$ could have type $(\mathsf{Int} \to \mathsf{Int}) \wedge (\bot \to \bot)$, while $\lambda x.\, 3$ could have type $\mathbb{1} \to \mathsf{Int}$: the former diverges on diverging arguments, the latter always terminates. It would be interesting to explore forms of termination analysis to obtain greater precision. The difficulty is to ensure that the type $\mathbb{0}$ remains uninhabited and that all diverging expressions still have types that include $\bot$. This is trivial in the current system, but it is no longer straightforward with more precise typing rules.

LANGUAGE AND TYPE SYSTEM EXTENSIONS: A further direction for future work is to extend the language and the type system we have considered with more features. The starting inspiration for the work in this chapter was the Nix Expression Language. To type Nix effectively, we would need to study how to add polymorphism, record types, some form of type inference, and gradual typing (since some dynamic programming idioms will surely remain beyond the reach of the type system). The work in the other parts of this thesis can provide a starting point towards this goal.

Conclusion

15 Conclusion

The semantic subtyping approach that we follow was developed initially for XDuce (Hosoya and Pierce, 2003), a domain-specific language for the processing of XML documents. Its extension with higher-order functions (Benzaken, Castagna, and Frisch, 2003; Frisch, Castagna, and Benzaken, 2008) made the approach more viable for general-purpose functional languages. The addition of parametric polymorphism (Castagna and Xu, 2011; Castagna et al., 2014, 2015b) continued along this path. Other work has applied the semantic subtyping approach to different settings, including object-oriented languages (Dardha, Gorla, and Varacca, 2013; Ancona and Corradi, 2016), XML and NoSQL query languages (Benzaken et al., 2013; Castagna et al., 2015a), and process calculi (Castagna, De Nicola, and Varacca, 2008).

This thesis contributes to this path by studying three different settings and showing how to adapt set-theoretic types and semantic subtyping to them.

The first setting is that of implicitly typed languages with type inference. We have shown how to define inference and have soundness and completeness properties. This has also required work to prove type safety: negation types pose challenges that had been considered up to now only for explicitly typed languages.

The second line of work is that on gradual typing. Gradual typing with set-theoretic types had only been studied in a monomorphic setting. We have extended it to a polymorphic setting and defined sound (though not complete) type inference. Moreover, during this work we have realized that a declarative formulation of gradual typing was possible, and indeed useful to integrate polymorphic gradual typing with set-theoretic types while keeping a simple description. We have described this formulation also for systems without subtyping.

Finally, we have considered non-strict languages, that had not been studied up to now in relation to semantic subtyping. We have shown how to give a sound type system for such languages by adding to types an explicit representation for divergence.

A point to be stressed is that, throughout this work, we have been able to preserve the existing results on semantic subtyping. Notably, the subtyping relations defined in the different parts of the thesis can all be decided by the existing algorithms, with at most trivial modifications. Analogously, we can rely on the tallying algorithm for constraint solving. We argue that this illustrates that semantic subtyping is an effective technique to define expressive subtyping relations with set-theoretic types in a wide variety of settings.

We have also met some limits of the approach. For example, in Parts I and II, we have suggested that changes to the tallying algorithm might be desirable. It must be noted also that the original semantic connotation of semantic subtyp-

ing does not always hold in these settings. It is, indeed, already weakened in polymorphic semantic subtyping by having to treat all types as non-singletons (as explained in Section 2.1.3). It fails more noticeably in Part II (where we must have ? \ ? be non-empty) and in Part III (as discussed in Section 14.1). However, union and intersection types can still be thought of in terms of their set-theoretic counterparts (and negation too except in Part II): therefore, the guiding intuition of semantic subtyping is still valid to some extent. We have also proven some results that show that thinking of types as sets of values is partly justified, especially when considering type connectives and ground types (for example, the results in Section 3.3.6). However, the focus has been more on how to obtain expressive subtyping and less on justifying it semantically. We have outlined how we could look for a better-fitting interpretation in Part III. It is less clear whether it would make sense to try to have subtyping on gradual types be set-theoretic too. Currently, it seems to yield an ill-behaved subtyping relation. It is possible that this could be changed by using a different definition of materialization, but the current definition has the advantage of coinciding with a well-known relation from the gradual typing literature.

15.1 Future work

We have discussed directions for future work in Section 7.2, in Section 11.2, and in Sections 14.1 and 14.2. We recall here some of the most significant.

Type inference with annotations: The work on type inference in Chapter 5 should be improved to achieve stronger completeness results and to characterize when type annotations are needed. This could require an adaptation of the tallying algorithm to deal better with explicit polymorphism from type annotations.

Record typing: While the type system in Part I is very expressive, it lacks a way to type record-update operations precisely as permitted by row polymorphism. We should explore whether we can add row polymorphism or other features that can provide similar expressiveness.

Complete type inference for gradual set-theoretic types: Type inference in Section 10.4 is sound but not complete. We can try to achieve completeness by leveraging the techniques in Chapters 4 and 5 and by modifying tallying to deal with materialization constraints.

Intersection types with gradual typing: The type system in Chapter 10 does not include a rule to introduce intersection types. This makes intersection types less useful, in particular to express function overloading. Adding typecases to the language and extending the type system to allow intersection introduction would be a major step forward in expressiveness.

Finding a model of types for non-strict languages: Reusing the set-theoretic interpretation of types from Chapter 2 and adjusting the type system has allowed us to describe sound typing also for non-strict languages.

However, as we have discussed, a different interpretation could provide more precise subtyping with a closer connection to the semantics.

Implicit typing for non-strict languages: The language in Part III is explicitly typed. Considering an implicitly typed language would make an extension with polymorphism simpler, since we would not need to consider types in the semantics. To do so, we need to adapt the techniques used in Chapter 3.

Appendices

A Additional proofs

We report here the proofs that we had omitted or only sketched in the main text. The statements follow the same numbering as in the text.

Implicit typing and type inference

Adding type annotations

5.5 LEMMA: If $P; M; \Delta; \sigma \Vdash \langle\!\langle \mathbf{e} : t \rangle\!\rangle^{\Delta}$ and if $\mathsf{dom}(\sigma) \sharp \Delta$ and $\mathsf{var}(\mathbf{e}) \subseteq \Delta$, then $P; M; \Delta \Vdash \mathbf{e} : t\sigma$. □

Proof: By induction on $\mathbf{e}$ and by case analysis on the shape of $\mathbf{e}$.

Case: $\mathbf{e} = \hat{x}$
We have $P; M; \Delta; \sigma \Vdash \bigwedge_{i \in I}(\hat{x} \mathrel{\dot{\leq}} t_i)$ where $\mathsf{d}(t) = \{ t_i \mid i \in I \}$.
Therefore, $P(\hat{x}) = \langle M_1 \rangle t_1$ and, for every $i \in I$ there is a σ_i such that:

$$t_1\sigma_i \leq t_i\sigma \qquad M \leq M_1\sigma_i \qquad \mathsf{dom}(\sigma_i) \sharp \Delta\,.$$

By Property 5.3, I is not empty and $\bigwedge_{i \in I} t_i \simeq t$; therefore, $\bigwedge_{i \in I} t_i\sigma \simeq t\sigma$.
For every $i \in I$, we have $P; M_1\sigma_i; \Delta \Vdash \hat{x} : t_1\sigma_i$ by $[\mathrm{T}^{\mathrm{ra}}_{\hat{x}}]$.
Then, by $[\mathrm{T}^{\mathrm{ra}}_{\leq}]$, we have $P; M; \Delta \Vdash \hat{x} : t_i\sigma$.
Using $[\mathrm{T}^{\mathrm{ra}}_{\wedge}]$, $P; M; \Delta \Vdash \hat{x} : \bigwedge_{i \in I} t_i\sigma$. By $[\mathrm{T}^{\mathrm{ra}}_{\leq}]$, $P; M; \Delta \Vdash \hat{x} : t\sigma$.

Case: $\mathbf{e} = x$
We have $P; M; \Delta; \sigma \Vdash (x \mathrel{\dot{\leq}} t)$, therefore $M(x) \leq t\sigma$.
We derive $P; M; \Delta \Vdash x : t\sigma$ by $[\mathrm{T}^{\mathrm{ra}}_{x}]$ and $[\mathrm{T}^{\mathrm{ra}}_{\leq}]$.

Case: $\mathbf{e} = c$
We have $P; M; \Delta; \sigma \Vdash (b_c \mathrel{\dot{\leq}} t)$, therefore $b_c\sigma \leq t\sigma$.
We derive $P; M; \Delta \Vdash c : t\sigma$ by $[\mathrm{T}^{\mathrm{ra}}_{c}]$ and $[\mathrm{T}^{\mathrm{ra}}_{\leq}]$.

Case: $\mathbf{e} = \lambda x.\, \mathbf{e}'$

Subcase: $\mathsf{d}^{\Delta}_{\rightarrow}(t) = \{ t'_i \to t_i \mid i \in I \} \neq \emptyset$
We have $P; M; \Delta; \sigma \Vdash \bigwedge_{i \in I}(\mathsf{def}\, x : t'_i\, \mathsf{in}\, \langle\!\langle \mathbf{e}' : t_i \rangle\!\rangle^{\Delta})$.
Therefore, for every $i \in I$, we have $P; (M, x : t'_i\sigma); \Delta; \sigma \Vdash \langle\!\langle \mathbf{e}' : t_i \rangle\!\rangle^{\Delta}$.
By IH and $[\mathrm{T}^{\mathrm{ra}}_{\lambda}]$, we obtain $P; M; \Delta \Vdash \lambda x.\, \mathbf{e}' : (t'_i \to t_i)\sigma$.
Applying $[\mathrm{T}^{\mathrm{ra}}_{\wedge}]$, we have $P; M; \Delta \Vdash \lambda x.\, \mathbf{e}' : \bigwedge_{i \in I}(t'_i \to t_i)\sigma$.
By Property 4.25, we have $\bigwedge_{i \in I}(t'_i \to t_i) \simeq t$. We conclude by $[\mathrm{T}^{\mathrm{ra}}_{\leq}]$.

Subcase: $\mathsf{d}^{\Delta}_{\rightarrow}(t) = \emptyset$
We have:

$$P; M; \Delta; \sigma \Vdash \exists \alpha_1, \alpha_2.\, (\mathsf{def}\, x : \alpha_1\, \mathsf{in}\, \langle\!\langle \mathbf{e}' : \alpha_2 \rangle\!\rangle^{\Delta}) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)$$

(with $\alpha_1, \alpha_2 \sharp t, \mathbf{e}', \Delta$). Therefore there exist t_1 and t_2 such that

$$P; (M, x : t_1); \Delta; \sigma \cup [t_1/\alpha_1, t_2/\alpha_2] \Vdash \langle\!\langle \mathbf{e}' : \alpha_2 \rangle\!\rangle^{\Delta} \quad t_1 \to t_2 \leq t\sigma \ .$$

We apply the IH and conclude by $[\mathrm{T}^{\mathrm{ra}}_{\lambda}]$ and $[\mathrm{T}^{\mathrm{ra}}_{\leq}]$.

Case: $\mathbf{e} = \mathbf{e}_1\ \mathbf{e}_2$
We have:

$$P; M; \Delta; \sigma \Vdash \exists \alpha. \langle\!\langle \mathbf{e}_1 : \alpha \to t \rangle\!\rangle^{\Delta} \wedge \langle\!\langle \mathbf{e}_2 : \alpha \rangle\!\rangle^{\Delta} \qquad \alpha \sharp t, \mathbf{e}_1, \mathbf{e}_2, \Delta \ .$$

Therefore there exists a t' such that

$$P; M; \Delta; \sigma \cup [t'/\alpha] \Vdash \langle\!\langle \mathbf{e}_1 : \alpha \to t \rangle\!\rangle^{\Delta} \quad P; M; \Delta; \sigma \cup [t'/\alpha] \Vdash \langle\!\langle \mathbf{e}_2 : \alpha \rangle\!\rangle^{\Delta} \ .$$

We apply the IH and conclude by $[\mathrm{T}^{\mathrm{ra}}_{\mathrm{app}}]$.

Case: $\mathbf{e} = (\mathbf{e}_1, \mathbf{e}_2)$
Similar to the previous cases.

Case: $\mathbf{e} = \pi_i\ \mathbf{e}'$
We consider the case $i = 1$; the other is symmetrical.
We have $P; M; \Delta; \sigma \Vdash \langle\!\langle e' : t \times \mathbb{1} \rangle\!\rangle^{\Delta}$.
By IH, $P; M; \Delta \Vdash e' : t \times \mathbb{1}$. By $[\mathrm{T}^{\mathrm{ra}}_{\mathrm{proj}}]$, $P; M; \Delta \Vdash \pi_1\ e' : t$.

Case: $\mathbf{e} = (\mathbf{e}_0 \in \mathbf{t}\ ?\ \mathbf{e}_1 : \mathbf{e}_2)$
We have

$$P; M; \Delta; \sigma \Vdash \exists \alpha. \langle\!\langle \mathbf{e}_0 : \alpha \rangle\!\rangle^{\Delta} \wedge \left((\alpha \mathrel{\dot{\leq}} \neg\mathbf{t}) \vee \langle\!\langle \mathbf{e}_1 : t \rangle\!\rangle^{\Delta}\right) \wedge \left((\alpha \mathrel{\dot{\leq}} \mathbf{t}) \vee \langle\!\langle \mathbf{e}_2 : t \rangle\!\rangle^{\Delta}\right)$$

(with $\alpha \sharp t, \mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2, \Delta$), therefore for some t' we have:

$$P; M; \Delta; \sigma \cup [t'/\alpha] \Vdash \langle\!\langle \mathbf{e}_0 : \alpha \rangle\!\rangle^{\Delta}$$
$$t' \leq \neg\mathbf{t} \text{ or } P; M; \Delta; \sigma \cup [t'/\alpha] \Vdash \langle\!\langle \mathbf{e}_1 : t \rangle\!\rangle^{\Delta}$$
$$t' \leq \mathbf{t} \text{ or } P; M; \Delta; \sigma \cup [t'/\alpha] \Vdash \langle\!\langle \mathbf{e}_2 : t \rangle\!\rangle^{\Delta}$$

By IH we obtain

$$P; M; \Delta \Vdash \mathbf{e}_0 : t' \quad t' \leq \neg\mathbf{t} \text{ or } P; M; \Delta \Vdash \mathbf{e}_1 : t\sigma \quad t' \leq \mathbf{t} \text{ or } P; M; \Delta \Vdash \mathbf{e}_2 : t\sigma$$

and we conclude by $[\mathrm{T}^{\mathrm{ra}}_{\mathrm{case}}]$.

Case: $\mathbf{e} = (\mathsf{let}\ \vec{\alpha}\ x = \mathbf{e}_1\ \mathsf{in}\ \mathbf{e}_2)$
We have $P; M; \Delta; \sigma \Vdash \mathsf{let}\ \hat{x} : \forall \vec{\alpha}; \alpha[\langle\!\langle \mathbf{e}_1 : \alpha \rangle\!\rangle^{\Delta \cup \vec{\alpha}}].\ \alpha\ \mathsf{in}\ \langle\!\langle \mathbf{e}_2 : t \rangle\!\rangle^{\Delta}$. Therefore

$$P; M_1; \Delta \cup \vec{\alpha}; \sigma_1 \Vdash \langle\!\langle \mathbf{e}_1 : \alpha \rangle\!\rangle^{\Delta \cup \vec{\alpha}} \qquad (P, \hat{x} : \langle M_1 \rangle \alpha\sigma_1); M; \Delta; \sigma \Vdash \langle\!\langle \mathbf{e}_2 : t \rangle\!\rangle^{\Delta}$$
$$M \leq M_1\sigma_1' \qquad \mathsf{dom}(\sigma_1) \sharp \Delta, \vec{\alpha} \qquad \vec{\alpha} \sharp \Delta, M_1 \ .$$

By IH we have

$$P; M_1; \Delta \cup \vec{\alpha} \Vdash \mathbf{e}_1 : \alpha\sigma_1 \qquad (P, \hat{x} : \langle M_1 \rangle \alpha\sigma_1); M; \Delta \Vdash \mathbf{e}_2 : t\sigma$$

and we conclude by $[\mathrm{T}^{\mathrm{ra}}_{\mathsf{let}}]$.

Case: $\mathbf{e} = (\mathbf{e}' :: t')$
We have $P; M; \Delta; \sigma \Vdash \langle\!\langle e' : t' \rangle\!\rangle^{\Delta} \wedge (t' \mathrel{\dot{\leq}} t)$.
Therefore, $P; M; \Delta; \sigma \Vdash \langle\!\langle e' : t' \rangle\!\rangle^{\Delta}$ and $t'\sigma \leq t\sigma$.
Since $\mathsf{var}(e) \subseteq \Delta$, $\mathsf{var}(t') \subseteq \Delta$. Since $\mathsf{dom}(\sigma) \sharp \Delta$, $t'\sigma = t'$.
By IH we have $P; M; \Delta \Vdash \mathbf{e}' : t'$. By $[\mathrm{T}^{\mathrm{ra}}_{::}]$ and $[\mathrm{T}^{\mathrm{ra}}_{\leq}]$, $P; M; \Delta \Vdash \mathbf{e} : t\sigma$. □

5.6 LEMMA: If $P; M; \varnothing \Vdash e : t\sigma$ can be derived in $\mathcal{T}^{\mathrm{ra}\setminus\wedge}$, then $P; M; \varnothing; \sigma \Vdash \langle\!\langle e : t \rangle\!\rangle^{\varnothing}$. □

Proof: By induction on e and by case analysis on the shape of e.
In each case, we invert the judgment $P; M; \Delta \Vdash e : t\sigma$. The inversion lemma can be derived analogously to how we did for the reformulated system without annotations in Definition 4.15 and Lemma 4.16.

Case: $e = \hat{x}$
We have $P(\hat{x}) = \langle M' \rangle t'$ and, for some σ', $t'\sigma' \leq t\sigma$ and $M \leq M'\sigma'$.
By Property 5.3, we have $\mathsf{d}(t) = \{ t_i \mid i \in I \} \neq \varnothing$ and $\bigwedge_{i\in I} t_i \simeq t$.
Since $t \leq \bigwedge_{i\in I} t_i$, for each $i \in I$ we have $t \leq t_i$. Therefore, $t'\sigma' \leq t_i\sigma$.
Therefore, $P; M; \varnothing; \sigma \Vdash \bigwedge_{i\in I}(\hat{x} \mathrel{\dot{\leq}} t_i)$.

Case: $e = x$
We have $M(x) \leq t\sigma$, therefore $P; M; \Delta; \sigma \Vdash (x \mathrel{\dot{\leq}} t)$.

Case: $e = c$
Straightforward.

Case: $e = \lambda x.\, e'$
We have $P; (M, x : t_1); \varnothing \Vdash e' : t_2$ and $t_1 \to t_2 \leq t\sigma$.

Subcase: $\mathsf{d}^{\Delta}_{\to}(t) = \{ t'_i \to t_i \mid i \in I \} \neq \varnothing$
We have $\langle\!\langle e : t \rangle\!\rangle^{\varnothing} = \bigwedge_{i\in I}(\mathsf{def}\, x : t'_i \,\mathsf{in}\, \langle\!\langle e' : t_i \rangle\!\rangle^{\varnothing})$.
By Property 5.3, we have:

$$\bigwedge_{i\in I} t'_i \to t_i \simeq t \qquad \mathsf{var}(\bigwedge_{i\in I} t'_i \to t_i) = \varnothing$$
$$\forall i \in I.\ t'_i \simeq \mathbb{0} \implies t_i \simeq \mathbb{1}\ .$$

For every $i \in I$, we prove $P; M; \varnothing; \sigma \Vdash \mathsf{def}\, x : t'_i \,\mathsf{in}\, \langle\!\langle e' : t_i \rangle\!\rangle^{\varnothing}$ as follows.
Note that $t'_i\sigma = t'_i$ and $t_i\sigma = t_i$.
Since $t_1 \to t_2 \leq t\sigma$, we have $t_1 \to t_2 \leq t'_i \to t_i$. By definition of subtyping, either we have $t'_i \leq t_1$ and $t_2 \leq t_i$ or $t'_i \leq \mathbb{0}$; but in the latter case, we have $t_i \simeq \mathbb{1}$, which also ensures $t_2 \leq t_i$.
Therefore, we have $P; (M, x : t'_i); \varnothing \Vdash e' : t_i$ by $[\mathrm{T}^{\mathrm{ra}}_{\leq}]$.
By IH, we obtain $P; (M, x : t'_i); \varnothing; \sigma \Vdash \langle\!\langle e' : t_i \rangle\!\rangle^{\varnothing}$.
We conclude by $[\mathrm{C}^{\mathrm{sata}}_{\mathsf{def}}]$.

Subcase: $\mathsf{d}^{\Delta}_{\to}(t) = \varnothing$
Let α_1 and α_2 be such that $\alpha_1, \alpha_2 \sharp t, \sigma$. Let $\hat{\sigma} = \sigma \cup [t_1/\alpha_1, t_2/\alpha_2]$.

Then, $\langle\!\langle e\colon t\rangle\!\rangle^{\varnothing} = \exists\alpha_1,\alpha_2.\,(\mathsf{def}\ x\colon \alpha_1\ \mathsf{in}\ \langle\!\langle e'\colon \alpha_2\rangle\!\rangle^{\varnothing}) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)$,
and we have $P; (M, x\colon \alpha_1\hat{\sigma}); \varnothing \Vdash e'\colon \alpha_2\hat{\sigma}$.
Therefore, by IH, $P; (M, x\colon \alpha_1\hat{\sigma}); \varnothing; \hat{\sigma} \Vdash \langle\!\langle e'\colon \alpha_2\rangle\!\rangle^{\varnothing}$.
Hence, we have $P; M; \Delta; \sigma \Vdash \langle\!\langle e\colon t\rangle\!\rangle^{\varnothing}$.

Case: $e = e_1\ e_2$
We have $P; M; \varnothing \Vdash e_1\colon t' \to t\sigma$ and $P; M; \varnothing \Vdash e_2\colon t'$.
Let α be such that $\alpha \sharp t$. Let $\hat{\sigma} = \sigma \cup [t'/\alpha]$.
Then, $\langle\!\langle e\colon t\rangle\!\rangle^{\varnothing} = \exists\alpha.\,\langle\!\langle e_1\colon \alpha \to t\rangle\!\rangle^{\varnothing} \wedge \langle\!\langle e_2\colon \alpha\rangle\!\rangle^{\varnothing}$.
We have $P; M; \varnothing \Vdash e_1\colon (\alpha \to t)\hat{\sigma}$ and $P; M; \varnothing \Vdash e_2\colon \alpha\hat{\sigma}$.
Therefore, by IH,

$$P; M; \varnothing; \hat{\sigma} \Vdash \langle\!\langle e_1\colon \alpha \to t\rangle\!\rangle^{\varnothing} \qquad P; M; \varnothing; \hat{\sigma} \Vdash \langle\!\langle e_2\colon \alpha\rangle\!\rangle^{\varnothing}\,.$$

Hence, we have $P; M; \varnothing; \sigma \Vdash \langle\!\langle e\colon t\rangle\!\rangle^{\varnothing}$.

Case: $e = (e_1, e_2)$
Analogous to the previous case.

Case: $e = \pi_i\ e'$
We consider the case $i = 1$; the other is symmetrical.
We have $P; M; \varnothing \Vdash e'\colon t\sigma \times \mathbb{1}$.
By IH, we obtain $P; M; \varnothing; \sigma \Vdash \langle\!\langle e'\colon t \times \mathbb{1}\rangle\!\rangle^{\varnothing}$.

Case: $e = (e_0 \in \mathbf{t}\ ?\ e_1 : e_2)$
Analogous to the previous cases.

Case: $e = (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)$
We have:

$$P; M_1; \varnothing \Vdash e_1\colon t_1 \qquad (P, \hat{x}\colon \langle M_1\rangle t_1); M; \varnothing \Vdash e_2\colon t\sigma \qquad M \leq M_1\sigma'$$

We choose a type variable α, and we have $P; M_1; \varnothing \Vdash e_1\colon \alpha[t_1/\alpha]$.
Therefore, by IH,

$$P; M_1; \varnothing; [t_1/\alpha] \Vdash \langle\!\langle e_1\colon \alpha\rangle\!\rangle \qquad (P, \hat{x}\colon \langle M_1\rangle t_1); M; \varnothing; \sigma \Vdash \langle\!\langle e\colon t\rangle\!\rangle$$

and we obtain $P; M; \varnothing; \sigma \Vdash \langle\!\langle e\colon t\rangle\!\rangle$. □

5.7 LEMMA: If $P; \Delta \vdash C \rightsquigarrow D \mid M \mid \vec{\alpha}$ and $\sigma \Vdash_{\Delta} D$, then $P; M\sigma; \Delta; \sigma|_{\setminus\vec{\alpha}} \Vdash C$. □

Proof: By structural induction on C and by case analysis on the shape of C. Most cases are analogous to those in the proof of Lemma 4.28. The interesting cases are those for $(\hat{x} \mathrel{\dot{\leq}} t)$ and let constraints.

Case: $C = (\hat{x} \mathrel{\dot{\leq}} t)$
We have

$$P; \Delta \vdash C \rightsquigarrow \{t_1[\vec{\beta}/\vec{\alpha}] \mathrel{\dot{\leq}} t\} \mid M_1[\vec{\beta}/\vec{\alpha}] \mid \vec{\beta} \qquad t_1[\vec{\beta}/\vec{\alpha}]\sigma \leq t\sigma$$
$$P(\hat{x}) = \langle M_1\rangle t_1 \qquad \vec{\alpha} = \mathsf{var}(\langle M_1\rangle t_1) \setminus \Delta \qquad \vec{\beta} \sharp t, \Delta$$

and we must show $P; M_1[\vec{\beta}/\vec{\alpha}]\sigma; \Delta; \sigma|_{\setminus\vec{\beta}} \Vdash C$, which requires finding a σ_1 such that

$$t_1\sigma_1 \leq t\sigma|_{\setminus\vec{\beta}} \qquad M_1[\vec{\beta}/\vec{\alpha}]\sigma \leq M_1\sigma_1 \qquad \mathsf{dom}(\sigma_1) \sharp \Delta\,.$$

We choose $\sigma_1 = [\vec{\beta}/\vec{\alpha}]\sigma$. We have $t\sigma = t\sigma|_{\setminus\vec{\beta}}$ since $\vec{\beta} \sharp t$.

Case: $C = (\mathsf{let}\ \hat{x} \colon \forall\vec{\alpha}; \alpha[C_1].\ \alpha\ \mathsf{in}\ C_2)$
We have:

$$P; \Delta \vdash C \rightsquigarrow D_2 \mid M_1\sigma_1[\vec{\gamma}/\vec{\beta}] \wedge M_2 \mid \vec{\alpha}_2 \cup \vec{\gamma} \qquad \sigma \Vdash_\Delta D_2$$
$$P; \Delta \cup \vec{\alpha} \vdash C_1 \rightsquigarrow D_1 \mid M_1 \mid \vec{\alpha}_1$$
$$(P, \hat{x} \colon \langle M_1\sigma_1\rangle\alpha\sigma_1); \Delta \vdash C_2 \rightsquigarrow D_2 \mid M_2 \mid \vec{\alpha}_2$$
$$\sigma_1 \in \mathsf{tally}_{\Delta\cup\vec{\alpha}}(D_1) \quad \vec{\alpha} \sharp \Delta, M_1 \quad \vec{\beta} = \mathsf{var}(M_1\sigma_1) \quad \vec{\alpha}_1 \sharp \alpha \quad \vec{\gamma} \sharp C_1, \vec{\alpha}_2, \Delta$$

By Property 4.25, we have $\sigma_1 \Vdash_{\Delta\cup\vec{\alpha}} D_1$.
Analogously to Lemma 4.27, we can prove that, if $P; \Delta \vdash C \rightsquigarrow D \mid M \mid \vec{\alpha}$, then $\mathsf{var}(D) \cup \mathsf{var}(M) \subseteq \mathsf{var}(C) \cup \vec{\alpha} \cup \Delta$.
Since $\vec{\gamma} \sharp C_1, \vec{\alpha}_2, \Delta$, then $\vec{\gamma} \sharp D_2$. Therefore, $\sigma|_{\setminus\vec{\gamma}} \Vdash D_2$.
By IH we obtain:

$$P; M_1\sigma_1; \Delta \cup \vec{\alpha}; \sigma_1|_{\setminus\vec{\alpha}_1} \Vdash C_1$$
$$(P, \hat{x} \colon \langle M_1\sigma_1\rangle\alpha\sigma_1); M_2\sigma|_{\setminus\vec{\gamma}}; \Delta; \sigma_2|_{\setminus(\vec{\alpha}_2\cup\vec{\gamma})} \Vdash C_2$$

We have $\alpha\sigma_1 = \alpha\sigma_1|_{\setminus\vec{\alpha}_1}$ because $\vec{\alpha}_1 \sharp \alpha$.
We have $M_2\sigma|_{\setminus\vec{\gamma}} = M_2\sigma$ because $\vec{\gamma} \sharp M_2$.
Therefore, we have $(P, \hat{x} \colon \langle M_1\sigma_1\rangle\alpha\sigma_1|_{\setminus\vec{\alpha}_1}); M_2\sigma; \sigma_2|_{\setminus(\vec{\alpha}_2\cup\vec{\beta})} \Vdash C_2$.
We have $(M_1\sigma_1[\vec{\beta}/\vec{\alpha}] \wedge M_2)\sigma \leq M_2\sigma$.
Therefore, by the same result as Lemma 4.23,

$$(P, \hat{x} \colon \langle M_1\sigma_1\rangle\alpha\sigma_1|_{\setminus\vec{\alpha}_1}); (M_1\sigma_1[\vec{\beta}/\vec{\alpha}] \wedge M_2)\sigma; \sigma_2|_{\setminus(\vec{\alpha}_2\cup\vec{\gamma})} \Vdash C_2\,.$$

To conclude, we also need to find σ_1' such that

$$(M_1\sigma_1[\vec{\beta}/\vec{\alpha}] \wedge M_2)\sigma \leq M_1\sigma_1\sigma_1' \ :$$

we take $\sigma_1' = [\vec{\beta}/\vec{\alpha}]\sigma$. □

Gradual typing

Gradual typing for Hindley-Milner systems

9.6 LEMMA: If $\Gamma_2 \vdash e \colon \tau$ and $\Gamma_1 \sqsubseteq^\forall \Gamma_2$, then $\Gamma_1 \vdash e \colon \tau$. □

Proof: By induction on the derivation of $\Gamma_2 \vdash e \colon \tau$ and by case analysis on the last rule applied.

Case: $[\mathrm{T}_x]$

We have $e = x$. By inversion of $[\mathrm{T}_x]$, we have:

$$\Gamma_2(x) = \forall \vec{\alpha}_2.\, \tau_2 \qquad \tau = \tau_2[\vec{t}_2/\vec{\alpha}_2]$$

By definition of $\Gamma_1 \sqsubseteq^{\forall} \Gamma_2$, we have $\Gamma_1(x) \sqsubseteq^{\forall} \Gamma_2(x)$. Let $\forall \vec{\alpha}_1.\, \tau_1$ be $\Gamma_1(x)$. Then we can find an instance $\tau_1[\vec{t}_1/\vec{\alpha}_1]$ of $\Gamma_1(x)$ such that $\tau_1[\vec{t}_1/\vec{\alpha}_1] \sqsubseteq \tau$. We have $\Gamma_1 \vdash x \colon \tau_1[\vec{t}_1/\vec{\alpha}_1]$ by $[\mathrm{T}_x]$ and $\Gamma_1 \vdash x \colon \tau$ by $[\mathrm{T}_\sqsubseteq]$.

Case: $[\mathrm{T}_c]$ Straightforward.

Case: $[\mathrm{T}_\lambda]$, $[\mathrm{T}_{\lambda:}]$, $[\mathrm{T}_{\mathrm{app}}]$, $[\mathrm{T}_{\mathrm{pair}}]$, $[\mathrm{T}_{\mathrm{proj}}]$, $[\mathrm{T}_\sqsubseteq]$
By direct application of the IH.
For $[\mathrm{T}_\lambda]$ and $[\mathrm{T}_{\lambda:}]$, for every τ, $\tau \sqsubseteq^{\forall} \tau$: therefore $(\Gamma_1, x \colon \tau) \sqsubseteq^{\forall} (\Gamma_2, x \colon \tau)$.

Case: $[\mathrm{T}_{\mathsf{let}}]$
We have derived $\Gamma_2 \vdash (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2) \colon \tau$ from the premises:

$$\Gamma_2 \vdash e_1 \colon \tau_1 \qquad \Gamma_2, x \colon \forall \vec{\alpha}, \vec{\beta}.\, \tau_1 \vdash e_2 \colon \tau \qquad \vec{\alpha}, \vec{\beta} \sharp \Gamma_2 \text{ and } \vec{\beta} \sharp \Gamma_2$$

By IH, we have Ⓐ $\Gamma_1 \vdash e_1 \colon \tau_1$.
Since $\sqsubseteq^{\forall}$ is reflexive, $\Gamma_1, x \colon \forall \vec{\alpha}, \vec{\beta}.\, \tau_1 \sqsubseteq^{\forall} \Gamma_2, x \colon \forall \vec{\alpha}, \vec{\beta}.\, \tau_1$.
By IH, we have Ⓑ $\Gamma_1, x \colon \forall \vec{\alpha}, \vec{\beta}.\, \tau_1 \vdash e_2 \colon \tau$.
By Lemma 9.5, we have Ⓒ $\mathsf{var}(\Gamma_1) \subseteq \mathsf{var}(\Gamma_2)$.
From Ⓒ we obtain Ⓓ $\vec{\alpha}, \vec{\beta} \sharp \Gamma_1$.
From Ⓐ, Ⓑ, Ⓓ, and $\vec{\beta} \sharp e_1$, we have $\Gamma_1 \vdash (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2) \colon \tau$. □

9.8 PROPOSITION: For every two types τ_1 and τ_2,

$$\tau_1 \sim \tau_2 \iff \exists \tau.\ \tau_1 \sqsubseteq \tau \text{ and } \tau_2 \sqsubseteq \tau\ . \qquad \square$$

Proof: We first prove the implication from left to right.
Note that if $\tau_1 = ?$ then we can take $\tau = \tau_2$ since $? \sqsubseteq \tau_2$ and $\tau_2 \sqsubseteq \tau_2$. Similarly, if $\tau_2 = ?$ then we can take $\tau = \tau_1$. We prove the result by induction on τ_1 for the cases where both τ_1 and τ_2 are not $?$.

Case: $\tau_1 = \alpha$ Then we have $\tau_2 = \alpha$ and we can take $\tau = \tau_1 = \tau_2$.

Case: $\tau_1 = b$ Then we have $\tau_2 = b$ and we can take $\tau = \tau_1 = \tau_2$.

Case: $\tau_1 = \tau_1' \times \tau_1''$
By consistency, we have $\tau_2 = \tau_2' \times \tau_2''$ where $\tau_1' \sim \tau_2'$ and $\tau_1'' \sim \tau_2''$.
By IH, there exist two types τ' and τ'' such that $\tau_i' \sqsubseteq \tau'$ and $\tau_i'' \sqsubseteq \tau''$ for every $i \in \{1, 2\}$.
Then, we have $\tau_i' \times \tau_i'' \sqsubseteq \tau' \times \tau''$ for every $i \in \{1, 2\}$, whence the result.

Case: $\tau_1 = \tau_1' \to \tau_1''$ Analogous to the previous case.

We now prove the other direction. As before, if $\tau_1 = ?$ or $\tau_2 = ?$ then the result is immediate. We reason by induction over τ for the cases where both τ_1 and τ_2 are not $?$.

Case: $\tau = ?$ We have $\tau_1 = \tau_2 = ?$, which is impossible.

Case: $\tau = \alpha$ Then $\tau_1 = \tau_2 = \alpha$, and the result is immediate.

Case: $\tau = b$ Same as before.

Case: $\tau = \tau' \to \tau''$
By materialization, we have $\tau_i = \tau_i' \to \tau_i''$ where $\tau_i' \sqsubseteq \tau'$ and $\tau_i'' \sqsubseteq \tau''$ for every $i \in \{1, 2\}$. By IH, we then have $\tau_1' \sim \tau_2'$ and $\tau_1'' \sim \tau_2''$ and the result follows by definition of consistency.

Case: $\tau = \tau' \times \tau''$ Analogous to the previous case. □

9.9 PROPOSITION: If $\Gamma \vdash_{\mathsf{ST}} e : \tau$, then $\Gamma \vdash_{\mathsf{1}} e : \tau$. Conversely, if $\Gamma \vdash_{\mathsf{1}} e : \tau$, then there exists a type τ' such that $\Gamma \vdash_{\mathsf{ST}} e : \tau'$ and $\tau' \sqsubseteq \tau$. □

Proof: We prove the two results by induction over e and the last rule used in the typing derivation.
To prove that $\Gamma \vdash_{\mathsf{ST}} e : \tau$ implies $\Gamma \vdash_{\mathsf{1}} e : \tau$, the cases are the following.

Case: [GVar]
We have $\Gamma \vdash_{\mathsf{ST}} x : \tau$ and, by hypothesis, $\Gamma(x) = \tau$. We conclude by $[\mathrm{T}_x]$.

Case: [GConst]
We have $\Gamma \vdash_{\mathsf{ST}} c : \tau$ and, by hypothesis, $\Delta c : \tau$, which is equivalent to $b_c = \tau$ in our system. We conclude by $[\mathrm{T}_c]$.

Case: [GLam] This rule is identical to $[\mathrm{T}_{\lambda:}]$.

Case: [GApp1]
We have $\Gamma \vdash_{\mathsf{ST}} e_1\ e_2 : ?$, with $\Gamma \vdash_{\mathsf{ST}} e_1 : ?$ and $\Gamma \vdash_{\mathsf{ST}} e_2 : \tau_2$.
By IH, we have $\Gamma \vdash_{\mathsf{1}} e_1 : ?$ and $\Gamma \vdash_{\mathsf{1}} e_2 : \tau_2$.
Then, by $[\mathrm{T}_{\sqsubseteq}]$ we obtain $\Gamma \vdash_{\mathsf{1}} e_1 : \tau_2 \to ?$ since $? \sqsubseteq \tau_2 \to ?$.
We can then apply rule $[\mathrm{T}_{\mathrm{app}}]$ to deduce that $\Gamma \vdash_{\mathsf{1}} e_1\ e_2 : ?$.

Case: [GApp2]
We have $\Gamma \vdash_{\mathsf{ST}} e_1\ e_2 : \tau'$, with $\Gamma \vdash_{\mathsf{ST}} e_1 : \tau \to \tau'$, $\Gamma \vdash_{\mathsf{ST}} e_2 : \tau_2$ and $\tau \sim \tau_2$.
By IH, we have $\Gamma \vdash_{\mathsf{1}} e_1 : \tau \to \tau'$ and $\Gamma \vdash_{\mathsf{1}} e_2 : \tau_2$.
Moreover, by Proposition 9.8, we know that there exists a type τ such that $\tau \sqsubseteq \tau$ and $\tau_2 \sqsubseteq \tau$.
Therefore, by applying $[\mathrm{T}_{\sqsubseteq}]$ we deduce that $\Gamma \vdash_{\mathsf{1}} e_1 : \tau \to \tau'$ and $\Gamma \vdash_{\mathsf{1}} e_2 : \tau$. We conclude by applying $[\mathrm{T}_{\mathrm{app}}]$ to deduce that $\Gamma \vdash_{\mathsf{ST}} e_1\ e_2 : \tau'$.

For the opposite direction, the cases are the following.

Case: $[\mathrm{T}_x]$ By hypothesis, $\Gamma(x) = \tau$. We conclude by rule [GVar].

Case: $[\mathrm{T}_c]$
We have $b_c = \Delta c$ in the system of Siek and Taha (2006).
We conclude by rule [GConst].

Case: $[\mathrm{T}_{\mathrm{app}}]$

We have $\Gamma \vdash_1 e_1\ e_2 : \tau$, with $\Gamma \vdash_1 e_1 : \tau' \to \tau$ and $\Gamma \vdash_1 e_2 : \tau'$.
By IH, we have $\Gamma \vdash_{\mathsf{ST}} e_1 : \tau_1$ and $\Gamma \vdash_{\mathsf{ST}} e_2 : \tau_2$ where $\tau_1 \sqsubseteq \tau' \to \tau$ and $\tau_2 \sqsubseteq \tau'$. Then, if $\tau_1 = ?$ then we deduce by rule [GApp1] that $\Gamma \vdash_{\mathsf{ST}} e_1\ e_2 : ?$ and $? \sqsubseteq \tau$, hence the result. Otherwise, we have $\tau_1 = \tau_1' \to \tau_1''$ where $\tau_1' \sqsubseteq \tau'$ and $\tau_1'' \sqsubseteq \tau$. Since $\tau_2 \sqsubseteq \tau'$, we deduce by Proposition 9.8 that $\tau_1' \sim \tau_2$. Therefore, we deduce by rule [GApp2] that $\Gamma \vdash_{\mathsf{ST}} e_1\ e_2 : \tau_1''$ and the result follows from the fact that $\tau_1'' \sqsubseteq \tau$.

Case: [$\mathrm{T}_{\lambda:}$]
We have $\Gamma \vdash_1 \lambda x : \tau'.\, e : \tau' \to \tau$, with $\Gamma, x : \tau' \vdash_1 e : \tau$. By IH, $\Gamma, x : \tau' \vdash_{\mathsf{ST}} e : \tau''$ where $\tau'' \sqsubseteq \tau$. Thus, by rule [GLam], we obtain $\Gamma \vdash_{\mathsf{ST}} \lambda x : \tau'.\, e : \tau' \to \tau''$, and the result follows from the fact that $\tau' \to \tau'' \sqsubseteq \tau' \to \tau$.

Case: [$\mathrm{T}_{\sqsubseteq}$]
We have $\Gamma \vdash_1 e : \tau$, with $\Gamma \vdash_1 e : \tau'$ and $\tau' \sqsubseteq \tau$. By IH, we have $\Gamma \vdash_{\mathsf{ST}} e : \tau''$ where $\tau'' \sqsubseteq \tau'$. By transitivity of the materialization, $\tau'' \sqsubseteq \tau$ and the result follows. □

9.14 PROPOSITION (Soundness of solve): If $\sigma \in \mathsf{solve}_\Delta(D)$, then the following hold:

- $\sigma \Vdash_\Delta D$;
- $\mathsf{dom}(\sigma) \subseteq \mathsf{var}(D)$;
- $\mathsf{var}(D)\sigma \subseteq \mathsf{var}_{\dot\sqsubseteq}(D)\sigma \cup \Delta$. □

Proof: Let σ be in $\mathsf{solve}_\Delta(D)$, where $D = \{ (t_i^1 \mathrel{\dot\lesssim} t_i^2) \mid i \in I \} \cup \{ (\tau_j \mathrel{\dot\lesssim} \alpha_j) \mid j \in J \}$. Then, we have:

$$\sigma = (\sigma_0' \circ \sigma_0)^\dagger|_{\mathsf{TVar}} \qquad \sigma_0 = \mathsf{unify}_\Delta(\overline{T^1 \doteq T^2}) \qquad \sigma_0' = [\vec\alpha'/\vec X] \cup [\vec X'/\vec\alpha]$$
$$\overline{T^1 \doteq T^2} = \{ (t_i^1 \doteq t_i^2) \mid i \in I \} \cup \{ (T_j \doteq \alpha_j) \mid j \in J \}$$
$$\vec X = \mathsf{FVar} \cap \mathsf{var}_{\dot\sqsubseteq}(D)\sigma_0 \qquad \vec\alpha = \mathsf{var}(D) \setminus (\Delta \cup \mathsf{dom}(\sigma_0) \cup \mathsf{var}_{\dot\sqsubseteq}(D)\sigma_0)$$
$$\vec\alpha', \vec X' \text{ fresh}$$

We first prove $\sigma \Vdash_\Delta D$. First, we show that, for every $i \in I$, we have $t_i^1\sigma = t_i^2\sigma$. Note that, since $\mathsf{var}(t_i^1) \cup \mathsf{var}(t_i^2) \subseteq \mathsf{TVar}$, we have $t_i^1\sigma = (t_i^1\sigma_0\sigma_0')^\dagger$ and $t_i^2\sigma = (t_i^2\sigma_0\sigma_0')^\dagger$. By the properties of unification, we have $t_i^1\sigma_0 = t_i^2\sigma_0$. Then, we also have $t_i^1\sigma_0\sigma_0' = t_i^2\sigma_0\sigma_0'$ and finally $t_i^1\sigma = t_i^2\sigma$.

Now, we show that, for every $j \in J$, we have $\tau_j\sigma \sqsubseteq \alpha_j\sigma$. We have $\tau_j\sigma = (\tau_j\sigma_0\sigma_0')^\dagger$ and $\alpha_j\sigma = (\alpha_j\sigma_0\sigma_0')^\dagger$. By the properties of unification, we have $T_j\sigma_0 = \alpha_j\sigma_0$ and therefore $(T_j\sigma_0\sigma_0')^\dagger = (\alpha_j\sigma_0\sigma_0')^\dagger$. Therefore, we must show $(\tau_j\sigma_0\sigma_0')^\dagger \sqsubseteq (T_j\sigma_0\sigma_0')^\dagger$, which holds trivially since $\tau_j = T_j^\dagger$.

Now, we show that, for every $j \in J$ and every $\beta \in \mathsf{var}(\tau_j)$, $\beta\sigma$ is a static type. Note that $\beta \in \mathsf{var}_{\dot\sqsubseteq}(D)$. We have $\beta\sigma = (\beta\sigma_0\sigma_0')^\dagger$. If $\beta\sigma$ were not static, there would be an $X \in \mathsf{var}(\beta\sigma_0\sigma_0')$: we show that this cannot happen. If there were an $X \in \mathsf{var}(\beta\sigma_0\sigma_0')$, then there would be an $A \in \mathsf{TVar} \cup \mathsf{FVar}$ such that $A \in \mathsf{var}(\beta\sigma_0)$ and $X \in \mathsf{var}(A\sigma_0')$. We would have $A \in \mathsf{var}_{\dot\sqsubseteq}(D)\sigma_0$. Therefore, if

$A \in \mathsf{FVar}$, then $A \in \vec{X}$ and it would be mapped to a static type variable; if $A \in \mathsf{TVar}$, then it could not be in $\mathsf{dom}(\sigma_0')$, so it could not be mapped to a type containing frame variables.

Finally, we show that $\mathsf{dom}(\sigma) \cap \Delta = \emptyset$. Let $\alpha \in \Delta$. We show $\alpha \notin \mathsf{dom}(\sigma)$, that is, $\alpha\sigma = \alpha$. We have $\alpha\sigma = (\alpha\sigma_0\sigma_0')^\dagger$. By the properties of unification, since $\alpha \in \Delta$, we have $\alpha\sigma_0 = \alpha$. We also have $\alpha\sigma_0' = \alpha$ because $\alpha \notin \vec{\alpha}$.

To prove $\mathsf{dom}(\sigma) \subseteq \mathsf{var}(D)$, consider $\alpha \notin \mathsf{var}(D)$. We prove $\alpha \notin \mathsf{dom}(\sigma)$, that is, $\alpha\sigma = \alpha$. We have $\alpha\sigma = (\alpha\sigma_0\sigma_0')^\dagger$. By the properties of unification, since $\alpha \notin \mathsf{var}(D)$, $\alpha\sigma_0 = \alpha$. Then, since $\alpha \notin \mathsf{var}(D)$, we have $\alpha \notin \vec{\alpha}$; hence, $\alpha\sigma_0' = \alpha$.

To prove $\mathsf{var}(D)\sigma \subseteq \mathsf{var}_{\dot{\sqsubseteq}}(D)\sigma \cup \Delta$, consider an arbitrary $\alpha \in \mathsf{var}(D)\sigma$. We show $\alpha \in \mathsf{var}_{\dot{\sqsubseteq}}(D)\sigma \cup \Delta$. By definition of $\mathsf{var}(D)\sigma$, there must exist a $\beta \in \mathsf{var}(D)$ such that $\alpha \in \mathsf{var}(\beta\sigma)$. We have $\beta\sigma = (\beta\sigma_0\sigma_0')^\dagger$. Either $\alpha \in \mathsf{var}(\beta\sigma_0) \setminus \mathsf{dom}(\sigma_0')$ or $\alpha \in \mathsf{var}(\sigma_0')$.

- If $\alpha \in \mathsf{var}(\beta\sigma_0) \setminus \mathsf{dom}(\sigma_0')$, then $\alpha \in \mathsf{var}(D)$ (because $\beta \in \mathsf{var}(D)$ and because solutions of unification do not introduce new variables). Then, $\alpha \in \Delta \cup \mathsf{dom}(\sigma_0) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)\sigma_0$. The case $\alpha \in \mathsf{dom}(\sigma_0)$ is impossible because σ_0 is idempotent. Therefore, $\alpha \in \Delta \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)\sigma_0$ and (since $\alpha \notin \mathsf{dom}(\sigma_0')$) $\alpha \in \Delta \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)\sigma$.
- If $\alpha \in \mathsf{var}(\sigma_0')$, then $\alpha \in \vec{X}\sigma_0'$. Therefore, there exists an $X \in \mathsf{var}_{\dot{\sqsubseteq}}(D)\sigma_0$ such that $\alpha \in \mathsf{var}(X\sigma_0')$. Hence, $\alpha \in \mathsf{var}_{\dot{\sqsubseteq}}(D)\sigma$. □

9.15 LEMMA: Let $\sigma : \mathsf{TVar} \to \mathsf{GType}$ and $\sigma' : \mathsf{Var} \to \mathsf{TFrame}$ be two type substitutions such that $\forall \alpha \in \mathsf{TVar}. (\alpha\sigma')^\dagger = \alpha\sigma$. For every T, we have $T^\dagger\sigma \sqsubseteq (T\sigma')^\dagger$. □

Proof: We choose $\hat{\sigma} : \mathsf{TVar} \to \mathsf{TFrame}$ such that:

$$\forall \alpha \in \mathsf{TVar}.\ (\alpha\hat{\sigma})^\dagger = \alpha\sigma \qquad \mathsf{fvar}(\hat{\sigma}) \sharp \mathsf{dom}(\sigma'), \mathsf{fvar}(T)\ .$$

We define $\check{\sigma} : \mathsf{FVar} \to \mathsf{GType}$ as

$$\check{\sigma} = [(X\sigma')^\dagger/X]_{X \in \mathsf{dom}(\sigma')} \cup [?/X]_{X \in \mathsf{fvar}(T\hat{\sigma}) \setminus \mathsf{dom}(\sigma')}\ .$$

We have $(T\hat{\sigma})^\dagger = T^\dagger\sigma$ because:

- for every $\alpha \in \mathsf{var}(T)$, we have $(\alpha\hat{\sigma})^\dagger = \alpha\sigma = \alpha^\dagger\sigma$;
- for every $X \in \mathsf{var}(T)$, we have $(X\hat{\sigma})^\dagger = X^\dagger = ? = ?\sigma = X^\dagger\sigma$.

We have $T\hat{\sigma}\check{\sigma} = (T\sigma')^\dagger$ because:

- for every $\alpha \in \mathsf{var}(T) \cap \mathsf{dom}(\hat{\sigma})$, since $\mathsf{fvar}(\hat{\sigma}) \sharp \mathsf{dom}(\check{\sigma})$, we have $\alpha\hat{\sigma}\check{\sigma} = \alpha\hat{\sigma}$ and $\alpha(\hat{\sigma} \cup \check{\sigma}) = \alpha\hat{\sigma}$;
- for every $\alpha \in \mathsf{var}(T) \setminus \mathsf{dom}(\sigma)$, since $\alpha\sigma = \alpha$, also $\alpha\hat{\sigma} = \alpha$ and $\alpha\sigma' = \alpha$: then we have $\alpha\hat{\sigma}\check{\sigma} = \alpha = (\alpha\sigma')^\dagger$;
- for every $X \in \mathsf{var}(T) \cap \mathsf{dom}(\sigma')$, we have $X\hat{\sigma}\check{\sigma} = X\check{\sigma} = (X\sigma')^\dagger$;

- for every $X \in \mathsf{var}(T) \setminus \mathsf{dom}(\sigma')$, we have $X \in \mathsf{var}(T\hat{\sigma}) \setminus \mathsf{dom}(\sigma')$: then, $X\hat{\sigma}\check{\sigma} = X\check{\sigma} = ? = X^\dagger = (X\sigma')^\dagger$.

Therefore, we have $T\hat{\sigma} \in \star(T^\dagger\sigma)$ and $T\hat{\sigma}\check{\sigma} = (T\sigma')^\dagger$ with $\check{\sigma} : \mathsf{FVar} \to \mathsf{GType}$: hence, $T^\dagger\sigma \sqsubseteq (T\sigma')^\dagger$. □

9.16 PROPOSITION (Completeness of solve): If $\sigma \Vdash_\Delta D$, then there exist two type substitutions σ' and σ'' such that:

- $\sigma' \in \mathsf{solve}_\Delta(D)$;
- $\mathsf{dom}(\sigma'') \subseteq \mathsf{var}(\sigma') \setminus \mathsf{var}(D)$;
- for every α, $\alpha\sigma'(\sigma \cup \sigma'') \sqsubseteq \alpha(\sigma \cup \sigma'')$;
- for every α such that $\alpha\sigma'$ is static, $\alpha\sigma'(\sigma \cup \sigma'') = \alpha(\sigma \cup \sigma'')$. □

Proof: Let $D = \{ (t_i^1 \dot{\leq} t_i^2) \mid i \in I \} \cup \{ (\tau_j \dot{\leq} \alpha_j) \mid j \in J \}$ and let $\sigma : \mathsf{TVar} \to \mathsf{GType}$ be such that $\sigma \Vdash_\Delta D$. The first step of computing $\mathsf{solve}_\Delta(D)$ is to construct

$$\overline{T^1 \doteq T^2} = \{ (t_i^1 \doteq t_i^2) \mid i \in I \} \cup \{ (T_j \doteq \alpha_j) \mid j \in J \}$$

with each T_j such that $T_j^\dagger = \tau_j$ and with unique frame variables.

First, we show that from σ we can obtain a substitution $\check{\sigma} : \mathsf{Var} \to \mathsf{TFrame}$ which is a unifier for $\overline{T^1 \doteq T^2}$. For every $j \in J$, we have $\tau_j\sigma \sqsubseteq \alpha_j\sigma$; furthermore, σ is static on all variables of τ_j. By definition of materialization, there exist a type frame $T_j' \in \star(\tau_j\sigma)$ and a substitution $\sigma_j : \mathsf{FVar} \to \mathsf{GType}$ such that $T_j'\sigma_j = \alpha_j\sigma$. In particular, we can choose $T_j' = T_j\sigma$ (because $T_j\sigma \in \star(\tau_j\sigma)$ and because it has unique frame variables) and we can assume $\mathsf{dom}(\sigma_j) = \mathsf{fvar}(T_j)$. Let $\hat{\sigma} = \sigma \cup \bigcup_{j \in J} \sigma_j$: $\hat{\sigma}$ is well defined since the frame variables in every T_j are distinct. We choose an arbitrary frame variable $\check{X}$. Let $\check{\sigma} : \mathsf{Var} \to \mathsf{TFrame}$ be such that $\forall A \in \mathsf{Var}.\ (A\check{\sigma})^\dagger = A\hat{\sigma}$ and that $\mathsf{fvar}(\check{\sigma}) \subseteq \{\check{X}\}$. We have $\mathsf{dom}(\check{\sigma}) \cap \Delta = \emptyset$, since $\mathsf{dom}(\sigma) \cap \Delta = \emptyset$, $\mathsf{dom}(\check{\sigma}) \setminus \mathsf{dom}(\sigma) \subseteq \mathsf{FVar}$, and $\Delta \subseteq \mathsf{TVar}$. Moreover, $\check{\sigma}$ is a unifier for $\overline{T^1 \doteq T^2}$.

By the properties of unification, we have $\mathsf{unify}_\Delta(\overline{T^1 \doteq T^2}) = \sigma_0$ and $\check{\sigma} = \check{\sigma} \circ \sigma_0$.

By definition of solve, we have:

$$\sigma' \in \mathsf{solve}_\Delta(D) \qquad \sigma' = (\sigma_0' \circ \sigma_0)^\dagger|_{\mathsf{TVar}} \qquad \sigma_0' = [\vec{\alpha}'/\vec{X}] \cup [\vec{X}'/\vec{\alpha}]$$
$$\vec{X} = \mathsf{FVar} \cap \mathsf{var}_{\dot{\sqsubseteq}}(D)\sigma_0 \qquad \vec{\alpha} = \mathsf{var}(D) \setminus (\Delta \cup \mathsf{dom}(\sigma_0) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)\sigma_0)$$
$$\vec{\alpha}', \vec{X}' \text{ fresh}$$

Since $\vec{\alpha}'$ and $\vec{X}'$ are fresh, we can assume they are outside $\mathsf{dom}(\check{\sigma})$ and $\mathsf{var}(\check{\sigma})$.

We choose $\sigma'' = [(\vec{X}\check{\sigma})^\dagger/\vec{\alpha}']$. Since $\vec{\alpha}'$ is chosen fresh by solve, it is outside of $\mathsf{var}(D)$: therefore, it is in $\mathsf{var}(\sigma') \setminus \mathsf{var}(D)$.

We must show:

$$\forall \alpha.\ \alpha\sigma'(\sigma \cup \sigma'') \sqsubseteq \alpha(\sigma \cup \sigma'')$$
$$\forall \alpha.\ \alpha\sigma' \implies \alpha\sigma'(\sigma \cup \sigma'') = \alpha(\sigma \cup \sigma'')$$

If $\alpha \notin \mathsf{dom}(\sigma')$, the results hold trivially.
We consider the case $\alpha \in \mathsf{dom}(\sigma')$. Then, we have $\alpha \notin \vec{\alpha}'$.
We have:

$$\alpha(\sigma \cup \sigma'') = \alpha\sigma = (\alpha\check{\sigma})^\dagger = (\alpha\sigma_0\check{\sigma})^\dagger$$

We have:

$$\begin{aligned}
\alpha\sigma'(\sigma \cup \sigma'') &= (\alpha\sigma_0\sigma_0')^\dagger(\sigma \cup \sigma'') \\
&\sqsubseteq (\alpha\sigma_0\sigma_0'(\check{\sigma} \cup [\vec{X}\check{\sigma}/\vec{\alpha}']))^\dagger && \text{by Lemma 9.15} \\
&= (\alpha\sigma_0([\vec{\alpha}'/\vec{X}] \cup [\vec{X}'/\vec{\alpha}])(\check{\sigma} \cup [\vec{X}\check{\sigma}/\vec{\alpha}']))^\dagger \\
&= (\alpha\sigma_0(\check{\sigma}|_{\mathsf{dom}(\check{\sigma})\setminus\vec{\alpha}} \cup [\vec{X}'/\vec{\alpha}]))^\dagger \\
&\sqsubseteq (\alpha\sigma_0\check{\sigma})^\dagger
\end{aligned}$$

If $\alpha\sigma'$ is static, then $\mathsf{fvar}(\alpha\sigma_0\sigma_0') = \varnothing$ and therefore $\mathsf{var}(\alpha\sigma_0) \sharp \vec{\alpha}$ and $\mathsf{fvar}(\alpha\sigma_0) \subseteq \vec{X}$. Then:

$$\begin{aligned}
\alpha\sigma'(\sigma \cup \sigma'') &= (\alpha\sigma_0\sigma_0')^\dagger(\sigma \cup \sigma'') \\
&= \alpha\sigma_0\sigma_0'(\sigma \cup \sigma'') \\
&= \alpha\sigma_0[\vec{\alpha}'/\vec{X}](\sigma \cup [(\vec{X}\check{\sigma})^\dagger/\vec{\alpha}'] \\
&= \alpha\sigma_0(\sigma \cup [(\vec{X}\check{\sigma})^\dagger/\vec{X}] \\
&= (\alpha\sigma_0\check{\sigma})^\dagger
\end{aligned}$$

□

9.17 LEMMA (Stability of typing under type substitution): If $\Gamma \vdash e \rightsquigarrow E : \tau$, then, for every static type substitution σ, we have $\Gamma\sigma \vdash e\sigma \rightsquigarrow E\sigma : \tau\sigma$. □

Proof: By induction on the derivation of $\Gamma \vdash e \rightsquigarrow E : \tau$ and by case analysis on the last rule applied.

Case: $[\mathrm{T}_x]$
We have $\Gamma \vdash x \rightsquigarrow x\,[\vec{t}] : \tau[\vec{t}/\vec{\alpha}]$, with $\Gamma(x) = \forall\vec{\alpha}.\,\tau$.
By α-renaming, $\vec{\alpha} \sharp \sigma$. Therefore, $(\Gamma\sigma)(x) = \forall\vec{\alpha}.\,\tau\sigma$.
Since the $\vec{t}\sigma$ are all static, by $[\mathrm{T}_x]$ we have Ⓐ $\Gamma\sigma \vdash x \rightsquigarrow x[\vec{t}\sigma] : \tau\sigma[\vec{t}\sigma/\vec{\alpha}]$.
Since $\vec{\alpha} \sharp \sigma$, we have Ⓑ $\tau\sigma[\vec{t}\sigma/\vec{\alpha}] = \tau[\vec{t}/\vec{\alpha}]\sigma$.
From Ⓐ and Ⓑ, we have $\Gamma\sigma \vdash x \rightsquigarrow x\,[\vec{t}]\sigma : \tau[\vec{t}/\vec{\alpha}]\sigma$.

Case: $[\mathrm{T}_c]$
Straightforward, since $b_c\sigma = b_c$.

Case: $[\mathrm{T}_\lambda]$, $[\mathrm{T}_{\lambda:}]$, $[\mathrm{T}_{\mathrm{app}}]$, $[\mathrm{T}_{\mathrm{pair}}]$, $[\mathrm{T}_{\mathrm{proj}}]$
Direct application of the IH. For $[\mathrm{T}_\lambda]$, note that $t\sigma$ is always static.

Case: $[\mathrm{T}_\sqsubseteq]$
$\tau' \sqsubseteq \tau$ implies $\tau'\sigma \sqsubseteq \tau\sigma$ for any type substitution σ.

Case: $[\mathrm{T}_{\mathsf{let}}]$
We have $\Gamma \vdash (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2) \rightsquigarrow (\mathsf{let}\ x = \Lambda\vec{\alpha},\vec{\beta}.\,E_1\ \mathsf{in}\ E_2) : \tau$, derived

from

$$\text{Ⓐ}\ \Gamma \vdash e_1 \rightsquigarrow E_1 : \tau_1 \qquad \text{Ⓑ}\ \Gamma, x : \forall \vec{\alpha},\vec{\beta}.\, \tau_1 \vdash e_2 \rightsquigarrow E_2 : \tau$$
$$\text{Ⓒ}\ \vec{\alpha},\vec{\beta} \sharp \Gamma \text{ and } \vec{\beta} \sharp e_1$$

Let $\vec{\alpha}_1$ and $\vec{\beta}_1$ be vectors of distinct variables chosen outside $\mathsf{var}(\Gamma)$, $\mathsf{var}(e_1)$, $\mathsf{dom}(\sigma)$, and $\mathsf{var}(\sigma)$. Let $\rho = [\vec{\alpha}_1/\vec{\alpha}] \cup [\vec{\beta}_1/\vec{\beta}]$.
By IH from Ⓐ we have $\Gamma\rho \vdash e_1\rho \rightsquigarrow E_1\rho : \tau_1\rho$.
By Ⓒ, we have Ⓓ $\Gamma \vdash e_1[\vec{\alpha}_1/\vec{\alpha}] \rightsquigarrow E_1\rho : \tau_1\rho$.
By IH from Ⓓ we have Ⓔ $\Gamma\sigma \vdash e_1[\vec{\alpha}_1/\vec{\alpha}]\sigma \rightsquigarrow E_1\rho\sigma : \tau_1\rho\sigma$.
By IH from Ⓑ we have Ⓕ $\Gamma\sigma, x : (\forall \vec{\alpha},\vec{\beta}.\, \tau_1)\sigma \vdash e_2\sigma \rightsquigarrow E_2\sigma : \tau\sigma$.
By α-renaming from Ⓕ we have Ⓖ $\Gamma\sigma, x : (\forall \vec{\alpha}_1,\vec{\beta}_1.\, \tau_1\rho)\sigma \vdash e_2\sigma \rightsquigarrow E_2\sigma : \tau\sigma$.
From Ⓖ, since $\vec{\alpha}_1,\vec{\beta}_1 \sharp \sigma$, we have Ⓗ $\Gamma\sigma, x : (\forall \vec{\alpha}_1,\vec{\beta}_1.\, \tau_1\rho\sigma) \vdash e_2\sigma \rightsquigarrow E_2\sigma : \tau\sigma$.
By $[\mathsf{T}_{\mathsf{let}}]$ from Ⓔ and Ⓗ, we have $\Gamma\sigma \vdash (\mathsf{let}\ \vec{\alpha}_1\ x = e_1[\vec{\alpha}_1/\vec{\alpha}]\sigma\ \mathsf{in}\ e_2\sigma) \rightsquigarrow (\mathsf{let}\ x = \Lambda\vec{\alpha}_1,\vec{\beta}_1.\, E_1\rho\sigma\ \mathsf{in}\ E_2\sigma) : \tau\sigma$.
This concludes the proof because $\mathsf{let}\ \vec{\alpha}_1\ x = e_1[\vec{\alpha}_1/\vec{\alpha}]\sigma\ \mathsf{in}\ e_2\sigma$ and $(\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2)\sigma$ are equivalent by α-renaming, as are $\mathsf{let} x = \Lambda\vec{\alpha}_1,\vec{\beta}_1.\, E_1\rho\sigma\ \mathsf{in}\ E_2\sigma$ and $(\mathsf{let}\ x = \Lambda\vec{\alpha},\vec{\beta}.\, E_1\ \mathsf{in}\ E_2)\sigma$. □

9.21 LEMMA: If $\Gamma; \Delta \vdash C \rightsquigarrow D$, then $\mathsf{var}(\Gamma) \cap \mathsf{var}(D) \subseteq \mathsf{var}(C) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)$. □

Proof: By induction on C (the form of C determines the derivation).

Case: $C = (t_1 \mathrel{\dot{\leq}} t_2)$ or $C = (\tau \mathrel{\dot{\sqsubseteq}} \alpha)$ We have $\mathsf{var}(D) \subseteq \mathsf{var}(C)$.

Case: $C = (\tau \mathrel{\dot{\sqsubseteq}} \alpha)$ We have $\mathsf{var}(D) \subseteq \mathsf{var}_{\dot{\sqsubseteq}}(D) \cup \{\alpha\}$ and $\alpha \in \mathsf{var}(C)$.

Case: $C = (\mathsf{def}\ x : \tau\ \mathsf{in}\ C')$
By IH, $\mathsf{var}(\Gamma, x : \tau) \cap \mathsf{var}(D) \subseteq \mathsf{var}(C') \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)$. This directly yields the result since $\mathsf{var}(C') \subseteq \mathsf{var}(C)$.

Case: $C = (\exists \vec{\alpha}.\, C')$
By IH, $\mathsf{var}(\Gamma) \cap \mathsf{var}(D) \subseteq \mathsf{var}(C') \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)$. The side condition on the rule imposes $\vec{\alpha} \sharp \Gamma$. Then, $\mathsf{var}(\Gamma) \cap \mathsf{var}(D) \subseteq \mathsf{var}(C) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)$ since $\mathsf{var}(C) = \mathsf{var}(C') \setminus \vec{\alpha}$.

Case: $C = (C_1 \wedge C_2)$
By IH, for both i, $\mathsf{var}(\Gamma) \cap \mathsf{var}(D_i) \subseteq \mathsf{var}(C_i) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D_i)$. This directly implies $\mathsf{var}(\Gamma) \cap \mathsf{var}(D_1 \cup D_2) \subseteq \mathsf{var}(C_1 \wedge C_2) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D_1 \cup D_2)$.

Case: $C = (\mathsf{let}\ x : \forall \vec{\alpha}; \alpha[C_1]^{\vec{\alpha}_1}.\, \alpha\ \mathsf{in}\ C_2)$
By IH,

$$\mathsf{var}(\Gamma) \cap \mathsf{var}(D_1) \subseteq \mathsf{var}(C_1) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D_1)$$
$$\mathsf{var}(\Gamma, x : \forall \vec{\alpha},\vec{\beta}.\, \alpha\sigma_1) \cap \mathsf{var}(D_2) \subseteq \mathsf{var}(C_2) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D_2)$$

We have

$$\begin{aligned} D &= D_2 \cup \mathsf{equiv}(\sigma_1, D_1) \\ \mathsf{var}(D) &= \mathsf{var}(D_2) \cup \mathsf{var}(D_1)\sigma_1 \cup \mathsf{var}_{\dot{\sqsubseteq}}(D_1) \cup S \cup S\sigma_1 \\ \mathsf{var}_{\dot{\sqsubseteq}}(D) &= \mathsf{var}_{\dot{\sqsubseteq}}(D_2) \cup \mathsf{var}(D_1)\sigma_1 \cup \mathsf{var}_{\dot{\sqsubseteq}}(D_1) \\ \mathsf{var}(C) &= (\mathsf{var}(C_1) \setminus (\vec{\alpha} \cup \{\alpha\})) \cup \mathsf{var}(C_2) \end{aligned}$$

where $S = \{\, \alpha \in \mathsf{dom}(\sigma_1) \mid \alpha\sigma_1 \text{ static} \,\}$.
Consider an arbitrary $\beta \in \mathsf{var}(\Gamma) \cap \mathsf{var}(D)$.

Subcase: $\beta \in \mathsf{var}(D_2)$
Then $\beta \in \mathsf{var}(C_2) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D_2)$ and hence $\beta \in \mathsf{var}(C) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)$.

Subcase: $\beta \in \mathsf{var}(D_1)\sigma_1 \cup \mathsf{var}_{\dot{\sqsubseteq}}(D_1)$
Then $\beta \in \mathsf{var}_{\dot{\sqsubseteq}}(D)$.

Subcase: $\beta \in S$
Then $\beta \in \mathsf{dom}(\sigma_1)$. By Proposition 9.14, $\beta \in \mathsf{var}(D_1)$.
Since $\beta \in \mathsf{var}(\Gamma) \cap \mathsf{var}(D_1)$, we have $\beta \in \mathsf{var}(C_1) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D_1)$. Since $\beta \in \mathsf{var}(\Gamma)$, by the side conditions of the rule we know $\beta \neq \alpha$ and $\beta \notin \vec{\alpha}$. Therefore, $\beta \in \mathsf{var}(C) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)$.

Subcase: $\beta \in S\sigma_1$
Then $\beta \in \mathsf{var}(\gamma\sigma_1)$ for some $\gamma \in \mathsf{dom}(\sigma_1)$ such that $\gamma\sigma_1$ is static.
By Proposition 9.14, $\gamma \in \mathsf{var}(D_1)$. Then $\beta \in \mathsf{var}(D_1)\sigma_1 \subseteq \mathsf{var}_{\dot{\sqsubseteq}}(D)$. □

9.22 LEMMA:

$$\forall \Gamma, \Delta, e, \alpha, D, \sigma. \quad \left.\begin{array}{l} \Gamma; \Delta \vdash \langle\!\langle e : \alpha \rangle\!\rangle \rightsquigarrow D \\ \sigma \in \mathsf{solve}_\Delta(D) \\ \mathsf{var}(e) \subseteq \Delta \\ \alpha \notin \mathsf{var}(\Gamma) \end{array}\right\} \implies \mathsf{static}(\sigma, \mathsf{var}(\Gamma))$$

□

Proof: Consider an arbitrary $\beta \in \mathsf{var}(\Gamma)$. We show that $\beta\sigma$ is static.

Case: $\beta \notin \mathsf{dom}(\sigma)$ Then $\beta\sigma = \beta$, which is static.

Case: $\beta \in \mathsf{dom}(\sigma)$
Then $\beta \in \mathsf{var}(D)$ (by Proposition 9.14), and therefore $\beta \in \mathsf{var}(\Gamma) \cap \mathsf{var}(D)$.
By Lemma 9.21, $\beta \in \mathsf{var}(\langle\!\langle e : \alpha \rangle\!\rangle) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)$.

Subcase: $\beta \in \mathsf{var}(\langle\!\langle e : \alpha \rangle\!\rangle)$
This case is impossible because $\mathsf{var}(\langle\!\langle e : \alpha \rangle\!\rangle) = \mathsf{var}(e) \cup \{\alpha\}$, $\mathsf{dom}(\sigma) \not\# \mathsf{var}(e)$ (because $\mathsf{var}(e) \subseteq \Delta$), and $\alpha \notin \mathsf{var}(\Gamma)$.

Subcase: $\beta \in \mathsf{var}_{\dot{\sqsubseteq}}(D)$ Since $\sigma \Vdash_\Delta D$, $\beta\sigma$ must be static. □

9.23 LEMMA:

$$\forall \Gamma, \Delta, D_1, \sigma_1, \rho, \sigma, \sigma'. \quad \left.\begin{array}{l} \sigma \Vdash_\Delta \mathsf{equiv}(\sigma_1, D_1) \\ \mathsf{dom}(\rho) \sharp \Gamma\sigma_1 \\ \mathsf{static}(\sigma', \mathsf{var}(\mathsf{equiv}(\sigma_1, D_1))\sigma) \\ \mathsf{static}(\sigma_1, \mathsf{var}(\Gamma)) \end{array}\right\} \Longrightarrow \Gamma\sigma\sigma' = \Gamma\sigma_1\rho\sigma\sigma'$$

□

Proof: Consider an arbitrary $x \in \mathsf{dom}(\Gamma)$. We have $\Gamma(x) = \forall\vec{\alpha}.\,\tau$. We assume by α-renaming that $\vec{\alpha} \sharp \sigma_1, \rho, \sigma, \sigma'$; then, $(\Gamma\sigma\sigma')(x) = \forall\vec{\alpha}.\,\tau\sigma\sigma'$ and $(\Gamma\sigma_1\rho\sigma\sigma')(x) = \forall\vec{\alpha}.\,\tau\sigma_1\rho\sigma\sigma'$. We must show $\tau\sigma\sigma' = \tau\sigma_1\rho\sigma\sigma'$. We show $\forall\alpha \in \mathsf{var}(\tau).\ \alpha\sigma\sigma' = \alpha\sigma_1\rho\sigma\sigma'$. Consider an arbitrary $\alpha \in \mathsf{var}(\tau)$.

Case: $\alpha \in \vec{\alpha}$
Then (by our choice of naming) $\alpha\sigma\sigma' = \alpha$ and $\alpha\sigma_1\rho\sigma\sigma' = \alpha$.

Case: $\alpha \notin \vec{\alpha}$
Then $\alpha \in \mathsf{var}(\Gamma)$ and hence: $\mathsf{var}(\alpha\sigma_1) \subseteq \mathsf{var}(\Gamma\sigma_1)$, and $\alpha\sigma_1\rho = \alpha\sigma_1$, and $\alpha\sigma_1$ is static.

Subcase: $\alpha \notin \mathsf{dom}(\sigma_1)$
Then $\alpha\sigma_1 = \alpha$, $\alpha\sigma_1\rho = \alpha$, and $\alpha\sigma_1\rho\sigma\sigma' = \alpha\sigma\sigma'$.

Subcase: $\alpha \in \mathsf{dom}(\sigma_1)$
Then $\{(\alpha \dot{\leq} \alpha\sigma_1), (\alpha\sigma_1 \dot{\leq} \alpha)\} \subseteq \mathsf{equiv}(\sigma_1, D_1)$.
Therefore, we have $\alpha\sigma_1\sigma = \alpha\sigma$ and $\alpha\sigma_1\sigma\sigma' = \alpha\sigma\sigma'$. □

9.24 THEOREM (Soundness of type inference): Let $\mathcal{D}$ be a derivation of $\Gamma; \mathsf{var}(e) \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$. Let σ be a type substitution such that $\sigma \Vdash_{\mathsf{var}(e)} D$. Then, we have $\Gamma\sigma \vdash e \rightsquigarrow \{\!|e|\!\}^{\mathcal{D}}_{\sigma} : t\sigma$. □

Proof: We show the following, stronger result (for all $\mathcal{D}$, Γ, Δ, e, t, D, σ, σ'):

$$\left.\begin{array}{l} \mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D \\ \sigma \Vdash_\Delta D \\ \mathsf{static}(\sigma', \mathsf{var}(D)\sigma) \\ \mathsf{var}(e) \subseteq \Delta \end{array}\right\} \Longrightarrow \Gamma\sigma\sigma' \vdash e\sigma' \rightsquigarrow \{\!|e|\!\}^{\mathcal{D}}_{\sigma}\sigma' : t\sigma\sigma'$$

This result implies the statement: we take $\Delta = \mathsf{var}(e)$ and $\sigma' = [\,]$ (the identity substitution).

The proof is by structural induction on e.

Case: $e = x$
We have

Ⓐ $\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle x : t \rangle\!\rangle \rightsquigarrow D$ Ⓑ $\sigma \Vdash_\Delta D$ Ⓒ $\mathsf{static}(\sigma', \mathsf{var}(D)\sigma)$.

By Lemma 9.20 from Ⓐ:

$$\Gamma(x) = \forall\vec{\alpha}.\,\tau \qquad D = \{(\tau[\vec{\beta}/\vec{\alpha}] \dot{\sqsubseteq} \alpha), (\alpha \dot{\lesssim} t)\}\ .$$

Assuming $\vec{\alpha} \sharp \sigma, \sigma'$ by α-renaming, we have $(\Gamma\sigma\sigma')(x) = \forall\vec{\alpha}.\,\tau\sigma\sigma'$.
By Ⓑ and Ⓒ, we know that the types $\vec{\beta}\sigma\sigma'$ are static.
Since $\vec{\alpha} \sharp \sigma, \sigma'$, we have $\forall\alpha \in \mathsf{var}(\tau).\ \alpha\sigma\sigma'[\vec{\beta}\sigma\sigma'/\vec{\alpha}] = \alpha[\vec{\beta}/\vec{\alpha}]\sigma\sigma'$.
Therefore, $\tau\sigma\sigma'[\vec{\beta}\sigma\sigma'/\vec{\alpha}] = \tau[\vec{\beta}/\vec{\alpha}]\sigma\sigma'$.
By Lemma 9.19, $\tau[\vec{\beta}/\vec{\alpha}]\sigma\sigma' \sqsubseteq \alpha\sigma\sigma'$.
By Lemma 9.18, $\alpha\sigma\sigma' = t\sigma\sigma'$.
By $[\mathrm{T}_x]$, $\Gamma\sigma\sigma' \vdash x \rightsquigarrow x\,[\vec{\beta}\sigma\sigma']\colon \tau\sigma\sigma'[\vec{\beta}\sigma\sigma'/\vec{\alpha}]$.
By $[\mathrm{T}_\sqsubseteq]$, $\Gamma\sigma\sigma' \vdash x \rightsquigarrow x\,[\vec{\beta}\sigma\sigma']\langle\tau[\vec{\beta}/\vec{\alpha}]\sigma\sigma' \overset{\ell}{\Rightarrow} \alpha\sigma\sigma'\rangle\colon t\sigma\sigma'$.
This concludes this case since $\{\!|x|\!\}^{\mathcal{D}}_{\sigma}\sigma' = x\,[\vec{\beta}\sigma\sigma']\langle\tau[\vec{\beta}/\vec{\alpha}]\sigma\sigma' \overset{\ell}{\Rightarrow} \alpha\sigma\sigma'\rangle$.

Case: $e = c$
We have $\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle c\colon t \rangle\!\rangle \rightsquigarrow D$.
By Lemma 9.20, $D = \{b_c \dot{\lesssim} t\}$. By Lemma 9.18, $b_c\sigma\sigma' = t\sigma\sigma'$.
By $[\mathrm{T}_c]$, $\Gamma\sigma\sigma' \vdash c\sigma\sigma' \rightsquigarrow c\colon t\sigma\sigma'$. Note that $\{\!|c|\!\}^{\mathcal{D}}_{\sigma}\sigma' = c$.

Case: $e = \lambda x.\,e'$
We have $\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle \lambda x.\,e'\colon t \rangle\!\rangle \rightsquigarrow D$.
By Lemma 9.20:

$$\mathcal{D}' :: (\Gamma, x\colon \alpha_1); \Delta \vdash \langle\!\langle e'\colon \alpha_2 \rangle\!\rangle \rightsquigarrow D'$$
$$D = D' \cup \{(\alpha_1 \dot{\sqsubseteq} \alpha_1), (\alpha_1 \to \alpha_2 \dot{\lesssim} t)\}$$

We know that $\alpha_1\sigma\sigma'$ is static.
By Lemma 9.18, $(\alpha_1 \to \alpha_2)\sigma\sigma' = t\sigma\sigma'$.
By IH, $\Gamma\sigma\sigma', x\colon \alpha_1\sigma\sigma' \vdash e'\sigma\sigma' \rightsquigarrow \{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma'\colon \alpha_2\sigma\sigma'$.
By $[\mathrm{T}_\lambda]$, $\Gamma\sigma\sigma' \vdash (\lambda x.\,e'\sigma\sigma') \rightsquigarrow \lambda^{(\alpha_1\to\alpha_2)\sigma\sigma'}x.\,\{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma'\colon (\alpha_1 \to \alpha_2)\sigma\sigma'$.
Therefore, $\Gamma\sigma\sigma' \vdash (\lambda x.\,e'\sigma\sigma') \rightsquigarrow \lambda^{(\alpha_1\to\alpha_2)\sigma\sigma'}x.\,\{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma'\colon t\sigma\sigma'$.
Note that $\{\!|\lambda x.\,e|\!\}^{\mathcal{D}}_{\sigma}\sigma' = \lambda^{(\alpha_1\to\alpha_2)\sigma\sigma'}x.\,\{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma'$.

Case: $e = \lambda x\colon \tau.\,e'$ We have $\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle \lambda x\colon \tau.\,e'\colon t \rangle\!\rangle \rightsquigarrow D$.
By Lemma 9.20:

$$\mathcal{D}' :: (\Gamma, x\colon \tau); \Delta \vdash \langle\!\langle e'\colon \alpha_2 \rangle\!\rangle \rightsquigarrow D'$$
$$D = D' \cup \{(\tau \dot{\sqsubseteq} \alpha_1), (\alpha_1 \to \alpha_2 \dot{\lesssim} t)\}\ .$$

By Lemma 9.19, $\tau\sigma\sigma' \sqsubseteq \alpha_1\sigma\sigma'$.
By Lemma 9.18, $(\alpha_1 \to \alpha_2)\sigma\sigma' = t\sigma\sigma'$.
By IH, $\Gamma\sigma\sigma', x\colon \tau\sigma\sigma' \vdash e'\sigma\sigma' \rightsquigarrow \{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma'\colon \alpha_2\sigma\sigma'$.
By $[\mathrm{T}_{\lambda:}]$, $\Gamma\sigma\sigma' \vdash (\lambda x\colon \tau.\,e')\sigma\sigma' \rightsquigarrow \lambda^{(\tau\to\alpha_2)\sigma\sigma'}x.\,\{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma'\colon (\tau \to \alpha_2)\sigma\sigma'$.
By $[\mathrm{T}_\sqsubseteq]$,

$$\Gamma\sigma\sigma' \vdash (\lambda x\colon \tau.\,e')\sigma\sigma' \rightsquigarrow$$
$$\left(\lambda^{(\tau\to\alpha_2)\sigma\sigma'}x.\,\{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma'\right)\langle(\tau \to \alpha_2)\sigma\sigma' \overset{\ell}{\Rightarrow} (\alpha_1 \to \alpha_2)\sigma\sigma'\rangle\colon (\alpha_1 \to \alpha_2)\sigma\sigma'.$$

Therefore,

$$\Gamma\sigma\sigma' \vdash (\lambda x\colon \tau.\, e')\sigma\sigma' \rightsquigarrow \\ \left(\lambda^{(\tau\to\alpha_2)\sigma\sigma'} x.\, \{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma'\right)\langle(\tau\to\alpha_2)\sigma\sigma' \overset{\ell}{\Rightarrow}(\alpha_1\to\alpha_2)\sigma\sigma'\rangle\colon t\sigma\sigma' \,.$$

Note that

$$\{\!|\lambda x\colon \tau.\, e|\!\}^{\mathcal{D}}_{\sigma}\sigma' = \\ \left(\lambda^{(\tau\to\alpha_2)\sigma\sigma'} x.\, \{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma'\right)\langle(\tau\to\alpha_2)\sigma\sigma' \overset{\ell}{\Rightarrow}(\alpha_1\to\alpha_2)\sigma\sigma'\rangle \,.$$

Case: $e = e_1\ e_2$

We have $\mathcal{D} :: \Gamma;\Delta \vdash \langle\!\langle e_1\ e_2\colon t\rangle\!\rangle \rightsquigarrow D$.

By Lemma 9.20:

$$\mathcal{D}_1 :: \Gamma;\Delta \vdash \langle\!\langle e_1\colon \alpha\to t\rangle\!\rangle \rightsquigarrow D_1 \qquad \mathcal{D}_2 :: \Gamma;\Delta \vdash \langle\!\langle e_2\colon \alpha\rangle\!\rangle \rightsquigarrow D_2 \\ D = D_1 \cup D_2$$

By IH, $\Gamma\sigma\sigma' \vdash e_1\sigma\sigma' \rightsquigarrow \{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma}\sigma'\colon (\alpha\to t)\sigma\sigma'$.
By IH, $\Gamma\sigma\sigma' \vdash e_2\sigma\sigma' \rightsquigarrow \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}\sigma'\colon \alpha\sigma\sigma'$.
By $[\mathrm{T}_{\mathrm{app}}]$, $\Gamma\sigma\sigma' \vdash (e_1\ e_2)\sigma\sigma' \rightsquigarrow \{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma}\sigma'\ \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}\sigma'\colon t\sigma\sigma'$.
We have $\{\!|e_1\ e_2|\!\}^{\mathcal{D}}_{\sigma}\sigma' = \{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma}\sigma'\ \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}\sigma'$.

Case: $e = (e_1, e_2)$

We have $\mathcal{D} :: \Gamma;\Delta \vdash \langle\!\langle (e_1, e_2)\colon t\rangle\!\rangle \rightsquigarrow D$.

By Lemma 9.20:

$$\mathcal{D}_1 :: \Gamma;\Delta \vdash \langle\!\langle e_1\colon \alpha_1\rangle\!\rangle \rightsquigarrow D_1 \qquad \mathcal{D}_2 :: \Gamma;\Delta \vdash \langle\!\langle e_2\colon \alpha_2\rangle\!\rangle \rightsquigarrow D_2 \\ D = D_1 \cup D_2 \cup \{\alpha_1\times\alpha_2 \mathrel{\dot{\leq}} t\}$$

By Lemma 9.18, $(\alpha_1\times\alpha_2)\sigma\sigma' = t\sigma\sigma'$.
By IH, $\Gamma\sigma\sigma' \vdash e_1\sigma\sigma' \rightsquigarrow \{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma}\sigma'\colon \alpha_1\sigma\sigma'$.
By IH, $\Gamma\sigma\sigma' \vdash e_2\sigma\sigma' \rightsquigarrow \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}\sigma'\colon \alpha_2\sigma\sigma'$.
By $[\mathrm{T}_{\mathrm{pair}}]$, $\Gamma\sigma\sigma' \vdash (e_1, e_2)\sigma\sigma' \rightsquigarrow \left(\{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma}\sigma', \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}\sigma'\right)\colon t\sigma\sigma'$.
We have $\{\!|(e_1, e_2)|\!\}^{\mathcal{D}}_{\sigma}\sigma' = \left(\{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma}\sigma', \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}\sigma'\right)$.

Case: $e = \pi_i\ e'$ We have $\mathcal{D} :: \Gamma;\Delta \vdash \langle\!\langle \pi_i\ e'\colon t\rangle\!\rangle \rightsquigarrow D$.

By Lemma 9.20:

$$\mathcal{D}' :: \Gamma;\Delta \vdash \langle\!\langle e'\colon \alpha_1\times\alpha_2\rangle\!\rangle \rightsquigarrow D' \qquad D = D' \cup \{\alpha_i \mathrel{\dot{\leq}} t\}$$

By Lemma 9.18, $\alpha_i\sigma\sigma' = t\sigma\sigma'$.
By IH, $\Gamma\sigma\sigma' \vdash e'\sigma\sigma' \rightsquigarrow \{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma'\colon (\alpha_1\times\alpha_2)\sigma\sigma'$.
By $[\mathrm{T}_{\mathrm{proj}}]$, $\Gamma\sigma\sigma' \vdash (\pi_i\ e')\sigma\sigma' \rightsquigarrow \pi_i\ (\{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma')\colon t\sigma\sigma'$.
We have $\{\!|\pi_i\ e'|\!\}^{\mathcal{D}}_{\sigma}\sigma' = (\pi_i\ \{\!|e'|\!\}^{\mathcal{D}'}_{\sigma})\sigma'$.

Case: $e = (\mathsf{let}\,\vec{\alpha}\,x = e_1\ \mathsf{in}\ e_2)$ We have $\mathcal{D} :: \Gamma;\Delta \vdash \langle\!\langle \mathsf{let}\,\vec{\alpha}\,x = e_1\ \mathsf{in}\ e_2\colon t\rangle\!\rangle \rightsquigarrow D$.

By Lemma 9.20:

$$\mathcal{D}_1 :: \Gamma; \Delta \cup \vec{\alpha} \vdash \langle\!\langle e_1 : \alpha \rangle\!\rangle \rightsquigarrow D_1$$
$$\mathcal{D}_2 :: (\Gamma, x : \forall \vec{\alpha}, \vec{\beta}.\, \alpha\sigma_1); \Delta \vdash \langle\!\langle e_2 : t \rangle\!\rangle \rightsquigarrow D_2$$
$$D = D_2 \cup \mathsf{equiv}(\sigma_1, D_1) \qquad \sigma_1 \in \mathsf{solve}_{\Delta \cup \vec{\alpha}}(D_1)$$
$$\vec{\alpha} \sharp \mathsf{var}(\Gamma\sigma_1) \qquad \vec{\beta} = \mathsf{var}(\alpha\sigma_1) \setminus (\mathsf{var}(\Gamma\sigma_1) \cup \vec{\alpha} \cup \mathsf{var}(e_1))$$

Let $\vec{\alpha}_1$ and $\vec{\beta}_1$ be vectors of distinct variables chosen outside $\mathsf{var}(e_1)$, $\mathsf{dom}(\sigma)$, $\mathsf{var}(\sigma)$, $\mathsf{dom}(\sigma')$, and $\mathsf{var}(\sigma')$. Let $\rho = [\vec{\alpha}_1/\vec{\alpha}] \cup [\vec{\beta}_1/\vec{\beta}]$.
Since $\vec{\beta} \sharp e_1$ and $\vec{\alpha}_1 \sharp \sigma'$, we have $e\sigma' = (\mathsf{let}\ \vec{\alpha}_1\ x = e_1\rho\sigma'\ \mathsf{in}\ e_2\sigma')$.
We have $\{\!|e|\!\}^{\mathcal{D}}_{\sigma} = \big(\mathsf{let}\ x = (\Lambda\vec{\alpha}_1, \vec{\beta}_1.\ \{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma_1}\rho\sigma)\ \mathsf{in}\ \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}\big)$.
Since $\vec{\alpha}_1, \vec{\beta}_1 \sharp \sigma'$, we have

$$\{\!|e|\!\}^{\mathcal{D}}_{\sigma}\sigma' = \big(\mathsf{let}\ x = (\Lambda\vec{\alpha}_1, \vec{\beta}_1.\ \{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma_1}\rho\sigma\sigma')\ \mathsf{in}\ \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}\sigma'\big)\ .$$

Considering e_1, we have $\sigma_1 \Vdash_{\Delta \cup \vec{\alpha}} D_1$.
We prove $\mathsf{static}(\sigma' \circ \sigma \circ \rho, \mathsf{var}(D_1)\sigma_1)$.
Take an arbitrary $\alpha \in \mathsf{var}(D_1)\sigma_1$.

- If $\alpha \in \mathsf{dom}(\rho)$, then $\alpha\rho$ is a variable in $\vec{\alpha}_1, \vec{\beta}_1$ and $\alpha\rho = \alpha\rho\sigma\sigma'$ (because $\vec{\alpha}_1, \vec{\beta}_1 \sharp \sigma, \sigma'$): hence $\alpha\rho\sigma\sigma'$ is static.
- If $\alpha \notin \mathsf{dom}(\rho)$, then $\alpha\rho\sigma\sigma' = \alpha\sigma\sigma'$.
  We have $(\alpha \stackrel{\cdot}{\sqsubseteq} \alpha) \in \mathsf{equiv}(\sigma_1, D_1)$.
  Since $\mathsf{equiv}(\sigma_1, D_1) \subseteq D$, $\alpha\sigma$ is static. Furthermore, $\mathsf{var}(\alpha\sigma) \subseteq \mathsf{var}(D)\sigma$; hence, $\alpha\sigma\sigma'$ is static too.

We have $\mathsf{var}(e_1) \subseteq \Delta \cup \vec{\alpha}$.
By IH, $\Gamma\sigma_1\rho\sigma\sigma' \vdash e_1\rho\sigma\sigma' \rightsquigarrow \{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma_1}\rho\sigma\sigma' : \alpha\sigma_1\rho\sigma\sigma'$.
Since $\mathsf{dom}(\sigma) \cap \mathsf{var}(e_1\rho) = \emptyset$, we have $e_1\rho\sigma\sigma' = e_1\rho\sigma'$.
By inversion, $\alpha \notin \mathsf{var}(\Gamma)$.
By Lemma 9.22, we have $\mathsf{static}(\sigma_1, \mathsf{var}(\Gamma))$.
By Lemma 9.23, $\Gamma\sigma\sigma' = \Gamma\sigma_1\rho\sigma\sigma'$.
We obtain $\Gamma\sigma\sigma' \vdash e_1\rho\sigma' \rightsquigarrow \{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma_1}\rho\sigma\sigma' : \alpha\sigma_1\rho\sigma\sigma'$.
Considering e_2, we have:

$$\sigma \Vdash_{\Delta} D_2 \qquad \mathsf{static}(\sigma', \mathsf{var}(D_2)\sigma) \qquad \mathsf{var}(e_2) \subseteq \Delta$$

By IH, $\Gamma\sigma\sigma', x : (\forall \vec{\alpha}, \vec{\beta}.\, \alpha\sigma_1)\sigma\sigma' \vdash e_2\sigma' \rightsquigarrow \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}\sigma' : t\sigma\sigma'$.
Since $\vec{\alpha}_1, \vec{\beta}_1 \sharp \sigma, \sigma'$, $(\forall \vec{\alpha}, \vec{\beta}.\, \alpha\sigma_1)\sigma\sigma' = (\forall \vec{\alpha}_1, \vec{\beta}_1.\, \alpha\sigma_1\rho\sigma\sigma')$.
We obtain $\Gamma\sigma\sigma', x : (\forall \vec{\alpha}_1, \vec{\beta}_1.\, \alpha\sigma_1\rho\sigma\sigma') \vdash e_2\sigma' \rightsquigarrow \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}\sigma' : t\sigma\sigma'$.
Moreover, $\vec{\alpha}_1, \vec{\beta}_1 \sharp \Gamma\sigma\sigma'$ and $\vec{\beta}_1 \sharp e_1\rho\sigma'$.
Therefore, by $[\mathrm{T}_{\mathsf{let}}]$, $\Gamma\sigma\sigma' \vdash e\sigma' \rightsquigarrow \{\!|e|\!\}^{\mathcal{D}}_{\sigma}\sigma' : t\sigma\sigma'$. □

9.26 LEMMA: If $\Gamma; \Delta \vdash C \rightsquigarrow D \mid \vec{\alpha}$, then $\mathsf{var}(D) \subseteq \mathsf{var}(\Gamma) \cup \mathsf{var}(C) \cup \vec{\alpha}$. □

Proof: By induction on the derivation of $\Gamma; \Delta \vdash C \rightsquigarrow D \mid \vec{\alpha}$. All cases are straightforward except that of let constraints.
Let $C = \mathsf{let}\, x : \forall \vec{\alpha}; \alpha[C_1]^{\vec{\alpha}'}.\, \alpha\ \mathsf{in}\ C_2$. Assume $\Gamma; \Delta \vdash C \rightsquigarrow D_2 \cup \mathsf{equiv}(\sigma_1, D_1) \mid$

$\vec{\alpha}_3$. Consider an arbitrary $\beta \in \mathsf{var}(D_2) \cup \mathsf{equiv}(\sigma_1, D_1)$. We must show $\beta \in \mathsf{var}(\Gamma) \cup \mathsf{var}(C) \cup \vec{\alpha}_3$.

Case: $\beta \in \mathsf{var}(D_2)$
By IH, we have $\beta \in \mathsf{var}(\Gamma) \cup \mathsf{var}(\forall \vec{\alpha}, \vec{\beta}.\, \alpha\sigma_1) \cup \mathsf{var}(C_2) \cup \vec{\alpha}_2$.
If $\beta \in \mathsf{var}(\Gamma) \cup \mathsf{var}(C_2) \cup \vec{\alpha}_2$, then $\beta \in \mathsf{var}(\Gamma) \cup \mathsf{var}(C) \cup \vec{\alpha}_3$.
If $\beta \in \mathsf{var}(\forall \vec{\alpha}, \vec{\beta}.\, \alpha\sigma_1)$, then either $\beta = \alpha$ or $\beta \in \mathsf{var}(\sigma_1)$.

- If $\beta = \alpha$, then $\beta \in \vec{\alpha}_3$.
- If $\beta \in \mathsf{var}(\sigma_1)$, either $\beta \in \mathsf{var}(D_1)$ or not. In the latter case, $\beta \in \vec{\alpha}_3$. In the former, by IH, we have $\beta \in \mathsf{var}(\Gamma) \cup \mathsf{var}(C_1) \cup \vec{\alpha}_1$. Note that $\mathsf{var}(C_1) \subseteq \mathsf{var}(C) \cup \{\alpha\} \cup \vec{\alpha}$. Then, $\beta \in \mathsf{var}(\Gamma) \cup \mathsf{var}(C) \cup \vec{\alpha}_3$.

Case: $\beta \in \mathsf{var}(\mathsf{equiv}(\sigma_1, D_1))$
By Proposition 9.14, $\mathsf{dom}(\sigma_1) \subseteq \mathsf{var}(D_1)$. Then, $\beta \in \mathsf{var}(D_1) \cup \mathsf{var}(\sigma_1)$. Both cases have already been treated above. □

9.27 LEMMA: If $\Gamma; \Delta \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D \mid \vec{\alpha}$, then $\mathsf{var}(t) \subseteq \mathsf{var}(D)$. □

Proof: We define a function v mapping structured constraints to sets of type variables. We show these two results, which together imply the statement:

- for every t and e, $\mathsf{var}(t) \subseteq v(\langle\!\langle e : t \rangle\!\rangle)$;
- for every Γ, Δ, C, D, and $\vec{\alpha}$, if $\Gamma; \Delta \vdash C \rightsquigarrow D \mid \vec{\alpha}$, then $v(C) \subseteq \mathsf{var}(D)$.

The function v is defined by induction on the structured constraint as follows:

$$v(t_1 \mathrel{\dot{\leq}} t_2) = \mathsf{var}(t_2) \qquad v(\tau \mathrel{\dot{\sqsubseteq}} \alpha) = \varnothing \qquad v(x \mathrel{\dot{\sqsubseteq}} \alpha) = \varnothing$$
$$v(\mathsf{def}\, x : \tau \,\mathsf{in}\, C) = v(C) \qquad v(\exists \vec{\alpha}.\, C) = v(C) \setminus \vec{\alpha}$$
$$v(C_1 \wedge C_2) = v(C_1) \cup v(C_2) \qquad v(\mathsf{let}\, x : \forall \vec{\alpha}; \alpha[C_1]^{\vec{\alpha}'}.\, \alpha \,\mathsf{in}\, C_2) = v(C_2)$$

The two results are proven easily by induction, respectively on e and on the derivation of $\Gamma; \Delta \vdash C \rightsquigarrow D \mid \vec{\alpha}$. □

9.29 THEOREM (Completeness of type inference): If $\Gamma \vdash e : \tau$, then, for every fresh type variable α, there exist D and σ such that $\Gamma; \mathsf{var}(e) \vdash \langle\!\langle e : \alpha \rangle\!\rangle \rightsquigarrow D$ and $[\tau/\alpha] \cup \sigma \Vdash_{\mathsf{var}(e)} D$. □

Proof: We show the following, stronger result (for all Γ, σ, e, t, Δ, and $\mathfrak{U}$):

$$\left.\begin{array}{l} \Gamma\sigma \vdash e : t\sigma \\ \mathsf{static}(\sigma, \Gamma) \\ \mathsf{dom}(\sigma) \sharp \Delta \supseteq \mathsf{var}(e) \\ \mathfrak{U} \sharp \Delta, t, \Gamma, \mathsf{dom}(\sigma) \end{array}\right\} \implies \exists D, \vec{\alpha}, \sigma'.\ \left\{\begin{array}{l} \Gamma; \Delta \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D \mid \vec{\alpha} \\ \sigma \cup \sigma' \Vdash_\Delta D \\ \mathsf{dom}(\sigma') \subseteq \vec{\alpha} \subseteq \mathfrak{U} \end{array}\right.$$

This result implies the statement: take $t = \alpha$ (with $\alpha \sharp \Gamma, \mathsf{var}(e)$), $\sigma = [\tau/\alpha]$,

and $\Delta = \mathsf{var}(e)$.

The proof is by structural induction on e.

Case: $e = x$

We have $\Gamma\sigma \vdash x : t\sigma$. Therefore, $x \in \mathsf{dom}(\Gamma)$.

Let $\Gamma(x)$ be $\forall\vec{\alpha}.\,\tau$ and assume, by α-renaming, $\vec{\alpha} \sharp \sigma$. Then, $(\Gamma\sigma)(x) = \forall\vec{\alpha}.\,\tau\sigma$.

By inversion of the typing rules, there exists an instance $\tau\sigma[\vec{t}/\vec{\alpha}]$ of $(\Gamma\sigma)(x)$ such that $\tau\sigma[\vec{t}/\vec{\alpha}] \sqsubseteq t\sigma$.

We take $\alpha \in \mathfrak{U}$. Then, $\langle\!\langle x : t \rangle\!\rangle = \exists\alpha.\,(x \mathrel{\dot{\sqsubseteq}} \alpha) \wedge (\alpha \mathrel{\dot{\leq}} t)$ (since $\alpha \sharp t$).

We take $\vec{\beta} \in \mathfrak{U}$ (with $\vec{\beta} \sharp \alpha$). We have

$$\Gamma; \Delta \vdash (x \mathrel{\dot{\sqsubseteq}} \alpha) \rightsquigarrow \{\tau[\vec{\beta}/\vec{\alpha}] \mathrel{\dot{\sqsubseteq}} \alpha\} \mid \vec{\beta} \qquad \Gamma; \Delta \vdash (\alpha \mathrel{\dot{\leq}} t) \rightsquigarrow \{\alpha \mathrel{\dot{\leq}} t\} \mid \varnothing$$

and therefore (since $\alpha \sharp \Gamma, \vec{\beta}$)

$$\Gamma; \Delta \vdash \langle\!\langle x : t \rangle\!\rangle \rightsquigarrow \{(\tau[\vec{\beta}/\vec{\alpha}] \mathrel{\dot{\sqsubseteq}} \alpha), (\alpha \mathrel{\dot{\leq}} t)\} \mid \vec{\beta} \cup \{\alpha\}$$

We take $\sigma' = [t\sigma/\alpha] \cup [\vec{t}/\beta]$ and show $\sigma \cup \sigma' \Vdash_\Delta \{(\tau[\vec{\beta}/\vec{\alpha}] \mathrel{\dot{\sqsubseteq}} \alpha), (\alpha \mathrel{\dot{\leq}} t)\}$:

- $\alpha(\sigma \cup \sigma') = t\sigma$ and $t(\sigma \cup \sigma') = t\sigma$;
- $\tau[\vec{\beta}/\vec{\alpha}](\sigma \cup \sigma') = \tau\sigma[\vec{t}/\vec{\alpha}]$ (because $\mathsf{var}(\tau) \setminus \vec{\alpha} \subseteq \mathsf{var}(\Gamma) \sharp \mathsf{dom}(\sigma')$);
- $\sigma \cup \sigma'$ is static on $\mathsf{var}(\tau[\vec{\beta}/\vec{\alpha}])$, because σ is static on $\mathsf{var}(\Gamma)$ and σ' is static on $\vec{\beta}$.

Case: $e = c$

By Lemma 9.25, $t\sigma = b_c$.

Moreover, $\langle\!\langle c : t \rangle\!\rangle = (b_c \mathrel{\dot{\leq}} t)$.

We can derive $\Gamma; \Delta \vdash \langle\!\langle c : t \rangle\!\rangle \rightsquigarrow (b_c \mathrel{\dot{\leq}} t) \mid \varnothing$.

Taking $\sigma' = [\,]$, we have $\sigma \cup \sigma' \Vdash_\Delta (b_c \mathrel{\dot{\leq}} t)$ since $b_c = t\sigma$ and $\mathsf{dom}(\sigma) \sharp \Delta$.

Case: $e = (\lambda x.\, e_1)$

By Lemma 9.25:

$$t\sigma = t_1 \to \tau_1 \qquad \Gamma\sigma, x : t_1 \vdash e_1 : \tau_1$$

We partition the variable pool as $\mathfrak{U} = \{\alpha_1, \alpha_2\} \uplus \mathfrak{U}_1$.

Let $\hat{\sigma} = \sigma \cup [t_1/\alpha_1] \cup [\tau_1/\alpha_2]$.

We have

$$\begin{aligned}&\langle\!\langle (\lambda x.\, e_1) : t \rangle\!\rangle \\ &\qquad = \exists\alpha_1, \alpha_2.\, (\mathsf{def}\, x : \alpha_1\, \mathsf{in}\, \langle\!\langle e_1 : \alpha_2 \rangle\!\rangle) \wedge (\alpha_1 \mathrel{\dot{\sqsubseteq}} \alpha_1) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\end{aligned}$$

since $\alpha_1, \alpha_2 \sharp t, e_1$.

Since $\alpha_1, \alpha_2 \sharp t, \Gamma$, we have $\Gamma\sigma = \Gamma\hat{\sigma}$ and $t\sigma = t\hat{\sigma}$.

We have $\mathsf{static}(\hat{\sigma}, (\Gamma, x : \alpha_1))$ and $(\Gamma, x : \alpha_1)\hat{\sigma} \vdash e_1 : \alpha_2\hat{\sigma}$.

By IH:

$$(\Gamma, x : \alpha_1); \Delta \vdash \langle\!\langle e_1 : \alpha_2 \rangle\!\rangle \rightsquigarrow D_1 \mid \vec{\alpha}_1 \quad \hat{\sigma} \cup \sigma_1' \Vdash_\Delta D_1 \quad \mathsf{dom}(\sigma_1') \subseteq \vec{\alpha}_1 \subseteq \mathfrak{U}_1$$

Then we have

$$\Gamma; \Delta \vdash \langle\!\langle (\lambda x.\, e_1) \colon t \rangle\!\rangle \rightsquigarrow D_1 \cup \{(\alpha_1 \mathrel{\dot{\sqsubseteq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\} \mid \vec{\alpha}_1 \cup \{\alpha_1, \alpha_2\}$$

since $\alpha_1, \alpha_2 \sharp \Gamma, \vec{\alpha}_1$.
We take $\sigma' = [t_1/\alpha_1] \cup [\tau_1/\alpha_2] \cup \sigma_1'$. Note that $\sigma \cup \sigma' = \hat{\sigma} \cup \sigma_1'$.
We have $\sigma \cup \sigma' \Vdash_\Delta D_1 \cup \{(\alpha_1 \mathrel{\dot{\sqsubseteq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\}$ because $\alpha_1(\sigma \cup \sigma') = t_1$ is static and because $(\alpha_1 \to \alpha_2)(\sigma \cup \sigma') = t_1 \to \tau_1 = t\sigma = t(\sigma \cup \sigma')$.

Case: $e = (\lambda x \colon \tau.\, e_1)$
By Lemma 9.25:

$$t\sigma = \tau' \to \tau_1 \qquad \tau \sqsubseteq \tau' \qquad \Gamma\sigma, x \colon \tau \vdash e_1 \colon \tau_1$$

We partition the variable pool as $\mathfrak{U} = \{\alpha_1, \alpha_2\} \uplus \mathfrak{U}_1$. Let $\hat{\sigma} = \sigma \cup [\tau'/\alpha_1] \cup [\tau_1/\alpha_2]$.
We have

$$\langle\!\langle (\lambda x \colon \tau.\, e_1) \colon t \rangle\!\rangle = \exists \alpha_1, \alpha_2.\, (\mathsf{def}\, x \colon \tau\ \mathsf{in}\ \langle\!\langle e_1 \colon \alpha_2 \rangle\!\rangle) \wedge (\tau \mathrel{\dot{\sqsubseteq}} \alpha_1) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)$$

since $\alpha_1, \alpha_2 \sharp t, \tau, e_1$.
Since $\alpha_1, \alpha_2 \sharp t, \Gamma$, we have $\Gamma\sigma = \Gamma\hat{\sigma}$ and $t\sigma = t\hat{\sigma}$.
We have $\tau\hat{\sigma} = \tau\sigma = \tau$ because $\alpha_1, \alpha_2 \sharp \tau$ and $\mathsf{var}(\tau) \subseteq \Delta$.
We have $\mathsf{static}(\hat{\sigma}, (\Gamma, x \colon \tau))$ and $(\Gamma, x \colon \tau)\hat{\sigma} \vdash e_1 \colon \alpha_2\hat{\sigma}$.
By IH:

$$(\Gamma, x \colon \tau); \Delta \vdash \langle\!\langle e_1 \colon \alpha_2 \rangle\!\rangle \rightsquigarrow D_1 \mid \vec{\alpha}_1 \quad \hat{\sigma} \cup \sigma_1' \Vdash_\Delta D_1 \quad \mathsf{dom}(\sigma_1') \subseteq \vec{\alpha}_1 \subseteq \mathfrak{U}_1$$

Then we have

$$\Gamma; \Delta \vdash \langle\!\langle (\lambda x \colon \tau.\, e_1) \colon t \rangle\!\rangle \rightsquigarrow D_1 \cup \{(\tau \mathrel{\dot{\sqsubseteq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\} \mid \vec{\alpha}_1 \cup \{\alpha_1, \alpha_2\}$$

since $\alpha_1, \alpha_2 \sharp \Gamma, \vec{\alpha}_1$.
We take $\sigma' = [\tau'/\alpha_1] \cup [\tau_1/\alpha_2] \cup \sigma_1'$. Note that $\sigma \cup \sigma' = \hat{\sigma} \cup \sigma_1'$.
We have $\sigma \cup \sigma' \Vdash_\Delta D_1 \cup \{(\tau \mathrel{\dot{\sqsubseteq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\}$ because $\tau(\sigma \cup \sigma') = \tau \sqsubseteq \tau' = \alpha_1(\sigma \cup \sigma')$, because $\sigma \cup \sigma'$ is static on τ (since it is the identity), and because $(\alpha_1 \to \alpha_2)(\sigma \cup \sigma') = \tau' \to \tau_1 = t\sigma = t(\sigma \cup \sigma')$.

Case: $e = e_1\ e_2$
By Lemma 9.25, we have $\Gamma\sigma \vdash e_1 \colon \tau \to t\sigma$ and $\Gamma\sigma \vdash e_2 \colon \tau$.
We partition the variable pool as $\mathfrak{U} = \{\alpha\} \uplus \mathfrak{U}_1 \uplus \mathfrak{U}_2$. Let $\hat{\sigma} = \sigma \cup [\tau/\alpha]$.
Since $\alpha \sharp t, e_1, e_2$, we have $\langle\!\langle e_1\ e_2 \colon t \rangle\!\rangle = \exists \alpha.\, \langle\!\langle e_1 \colon \alpha \to t \rangle\!\rangle \wedge \langle\!\langle e_2 \colon \alpha \rangle\!\rangle$.
Since $\alpha \sharp t, \Gamma, \Gamma\sigma = \Gamma\hat{\sigma}$ and $t\sigma = t\hat{\sigma}$.
We have $\mathsf{static}(\hat{\sigma}, \Gamma)$, $\Gamma\hat{\sigma} \vdash e_1 \colon (\alpha \to t)\hat{\sigma}$ and $\Gamma\hat{\sigma} \vdash e_2 \colon \alpha\hat{\sigma}$.
By IH:

$$\Gamma; \Delta \vdash \langle\!\langle e_1 \colon \alpha \to t \rangle\!\rangle \rightsquigarrow D_1 \mid \vec{\alpha}_1 \quad \hat{\sigma} \cup \sigma_1' \Vdash_\Delta D_1 \quad \mathsf{dom}(\sigma_1') \subseteq \vec{\alpha}_1 \subseteq \mathfrak{U}_1$$
$$\Gamma; \Delta \vdash \langle\!\langle e_2 \colon \alpha \rangle\!\rangle \rightsquigarrow D_2 \mid \vec{\alpha}_2 \quad \hat{\sigma} \cup \sigma_2' \Vdash_\Delta D_2 \quad \mathsf{dom}(\sigma_2') \subseteq \vec{\alpha}_2 \subseteq \mathfrak{U}_2$$

Then, since $\vec{\alpha}_1 \sharp \vec{\alpha}_2$ and $\alpha \sharp \Gamma, (\vec{\alpha}_1 \cup \vec{\alpha}_2)$, we have $\Gamma; \Delta \vdash \langle\!\langle e_1\ e_2 : t \rangle\!\rangle \rightsquigarrow D_1 \cup D_2 \mid \vec{\alpha}_1 \cup \vec{\alpha}_2 \cup \{\alpha\}$.
We take $\sigma' = [\tau/\alpha] \cup \sigma_1' \cup \sigma_2'$.
By Lemma 9.26, we have that $\sigma_1' \sharp \mathsf{var}(D_2)$ and $\sigma_2' \sharp \mathsf{var}(D_1)$.
Then, by Lemma 9.28, $\sigma \cup \sigma' \Vdash_\Delta D_1 \cup D_2$.

Case: $e = (e_1, e_2)$ or $e = \pi_i\ e_1$
Analogous to the previous cases.

Case: $e = (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2)$
By Lemma 9.25:

$$\Gamma\sigma \vdash e_1 : \tau_1 \qquad \Gamma\sigma, x : \forall \vec{\alpha}, \vec{\beta}.\ \tau_1 \vdash e_2 : t\sigma \qquad \vec{\alpha}, \vec{\beta} \sharp \Gamma\sigma \qquad \vec{\beta} \sharp e_1$$

By α-renaming, we can assume $\vec{\alpha} \subseteq \mathfrak{U}$. We partition the variable pool as $\mathfrak{U} = \{\alpha\} \uplus \vec{\alpha} \uplus \mathfrak{U}_1 \uplus \mathfrak{U}_2 \uplus \mathfrak{U}_3$. Let $\hat{\sigma} = \sigma \cup [\tau_1/\alpha]$. We have:

$$\langle\!\langle e : t \rangle\!\rangle = \mathsf{let}\ x : \forall \vec{\alpha}; \alpha[\langle\!\langle e_1 : \alpha \rangle\!\rangle]^{\mathsf{var}(e_1) \setminus \vec{\alpha}}.\ \alpha\ \mathsf{in}\ \langle\!\langle e_2 : t \rangle\!\rangle$$

$$\Gamma\sigma = \Gamma\hat{\sigma}\ \text{and}\ t\sigma = t\hat{\sigma} \qquad \mathsf{static}(\hat{\sigma}, \Gamma) \qquad \Gamma\hat{\sigma} \vdash e_1 : \alpha\hat{\sigma}$$

By IH (using $\Delta \cup \vec{\alpha}$ instead of Δ):

$$\Gamma; \Delta \cup \vec{\alpha} \vdash \langle\!\langle e_1 : \alpha \rangle\!\rangle \rightsquigarrow D_1 \mid \vec{\alpha}_1 \qquad \hat{\sigma} \cup \sigma_1' \Vdash_{\Delta \cup \vec{\alpha}} D_1 \qquad \mathsf{dom}(\sigma_1') \subseteq \vec{\alpha}_1 \subseteq \mathfrak{U}_1$$

Since $\hat{\sigma} \cup \sigma_1' \Vdash_{\Delta \cup \vec{\alpha}} D_1$, by Proposition 9.16, there exist two substitutions σ_1 and $\tilde{\sigma}_1$ such that

$$\sigma_1 \in \mathsf{solve}_{\Delta \cup \vec{\alpha}}(D_1) \qquad \mathsf{dom}(\tilde{\sigma}_1) \subseteq \mathsf{var}(\sigma_1) \setminus \mathsf{var}(D_1)$$

$$\forall \alpha.\ \alpha\sigma_1(\hat{\sigma} \cup \sigma_1' \cup \tilde{\sigma}_1) \sqsubseteq \alpha(\hat{\sigma} \cup \sigma_1' \cup \tilde{\sigma}_1)$$

$$\forall \alpha.\ \alpha\sigma_1\ \mathsf{static} \implies \alpha\sigma_1(\hat{\sigma} \cup \sigma_1' \cup \tilde{\sigma}_1) = \alpha(\hat{\sigma} \cup \sigma_1' \cup \tilde{\sigma}_1)$$

We can choose the variables in $\mathsf{var}(\sigma_1) \setminus \mathsf{var}(D_1)$ freely from a set of fresh variables: we take them from $\mathfrak{U}_3$.
Let $\check{\sigma} = \sigma \cup [\tau_1/\alpha] \cup \sigma_1' \cup \tilde{\sigma}_1$.
We have $\Gamma\sigma = \Gamma\check{\sigma}$ and $t\sigma = t\check{\sigma}$.
Let $\vec{\gamma} = \mathsf{var}(\alpha\sigma_1) \setminus (\mathsf{var}(\Gamma\sigma_1) \cup \vec{\alpha} \cup (\mathsf{var}(e_1) \setminus \vec{\alpha})) = \mathsf{var}(\alpha\sigma_1) \setminus (\mathsf{var}(\Gamma\sigma_1) \cup \vec{\alpha} \cup \mathsf{var}(e_1))$. Let $\Gamma' = (\Gamma, x : \forall \vec{\alpha}, \vec{\gamma}.\ \alpha\sigma_1)$.
We show $\mathsf{static}(\sigma_1, \Gamma)$. Take $\beta \in \mathsf{var}(\Gamma)$. If $\beta \notin \mathsf{var}(D_1)$, then $\beta\sigma_1 = \beta$, which is static. Otherwise, by Lemma 9.21, we have $\beta \in \mathsf{var}(\langle\!\langle e_1 : \alpha \rangle\!\rangle) \cap \mathsf{var}_{\sqsubseteq}(D_1)$.
We have $\mathsf{var}(\langle\!\langle e_1 : \alpha \rangle\!\rangle) = \mathsf{var}(e_1) \cap \{\alpha\}$. The case $\beta = \alpha$ is impossible because $\alpha \notin \mathsf{var}(\Gamma)$. If $\beta \in \mathsf{var}(e_1)$, then $\beta\sigma_1 = \beta$. If $\beta \in \mathsf{var}_{\sqsubseteq}(D_1)$, then $\beta\sigma_1$ is static.
Note that $\check{\sigma} = \hat{\sigma} \cup \sigma_1' \cup \tilde{\sigma}_1$. Therefore we have:

$$\forall \alpha.\ \alpha\sigma_1\check{\sigma} \sqsubseteq \alpha\check{\sigma} \qquad \forall \alpha.\ \alpha\sigma_1\ \mathsf{static} \implies \alpha\sigma_1\check{\sigma} = \alpha\check{\sigma}$$

We have $\Gamma\check{\sigma} = \Gamma\sigma_1\check{\sigma}$ because, for every $\alpha \in \mathsf{var}(\Gamma)$, $\alpha\sigma_1$ is static.
We show $\mathfrak{U}_2 \sharp \Gamma'$.

We already have $\mathfrak{U}_2 \sharp \Gamma$. It remains to show that the variables of $\forall\vec{\alpha},\vec{\gamma}.\,\alpha\sigma_1$ are not in $\mathfrak{U}_2$, which is true because all these variables are either α or variables in $\mathsf{var}(\sigma_1)$, and $\mathsf{var}(\sigma_1) \subseteq \mathsf{var}(D_1) \cup \mathfrak{U}_3$.

We show $\mathsf{static}(\check{\sigma}, \Gamma')$.

We have $\mathsf{static}(\check{\sigma}, \Gamma)$ since $\check{\sigma}$ and σ are equal on $\mathsf{var}(\Gamma)$. We must show that, for every variable $\beta \in \mathsf{var}(\forall\vec{\alpha},\vec{\gamma}.\,\alpha\sigma_1)$, $\beta\check{\sigma}$ is a static type. We have $\beta \in \mathsf{var}(\alpha\sigma_1)\setminus(\vec{\alpha}\cup\vec{\gamma})$. By definition of $\vec{\gamma}$, we have $\beta \in \mathsf{var}(\Gamma\sigma_1)\cup\mathsf{var}(e_1)$. If $\beta \in \mathsf{var}(\Gamma\sigma_1)$, then there exists a $\gamma \in \mathsf{var}(\Gamma)$ such that $\beta \in \mathsf{var}(\gamma\sigma_1)$; since $\gamma\check{\sigma}$ is static and $\gamma\check{\sigma} = \gamma\sigma_1\check{\sigma}$, $\gamma\sigma_1\check{\sigma}$ is static; therefore, $\beta\check{\sigma}$ is static as well. If $\beta \in \mathsf{var}(e_1)$, then $\beta \in \Delta$ and $\beta\check{\sigma} = \beta$.

We show $\mathsf{static}(\check{\sigma}, \mathsf{var}(D_1)\sigma_1)$.

Consider $\gamma \in \mathsf{var}(D_1)\sigma_1$. By Proposition 9.14, $\gamma \in \mathsf{var}_{\dot{\sqsubseteq}}(D_1)\sigma_1 \cup \Delta \cup \vec{\alpha}$. If $\gamma \in \Delta \cup \vec{\alpha}$, we have $\gamma\check{\sigma} = \gamma$. If $\gamma \in \mathsf{var}_{\dot{\sqsubseteq}}(D_1)\sigma_1$, there exists $\gamma' \in \mathsf{var}_{\dot{\sqsubseteq}}(D_1)$ such that $\gamma \in \mathsf{var}(\gamma'\sigma_1)$. We know that $\gamma'\check{\sigma}$ is static. Since $\gamma'\sigma_1$ is static too, we have $\gamma'\sigma_1\check{\sigma} = \gamma'\check{\sigma}$. This implies that $\gamma\check{\sigma}$ must be static.

Now we show $\Gamma'\check{\sigma} \vdash e_2 : t\check{\sigma}$.

We apply Lemma 9.6 by showing $\Gamma'\check{\sigma} \sqsubseteq^{\forall} (\Gamma\sigma, x : \forall\vec{\alpha},\vec{\beta}.\,\tau_1)$. Since $\Gamma\check{\sigma} = \Gamma\sigma$, we must only show $(\forall\vec{\alpha},\vec{\gamma}.\,\alpha\sigma_1)\check{\sigma} \sqsubseteq^{\forall} \forall\vec{\alpha},\vec{\beta}.\,\tau_1$. Note that $\alpha\check{\sigma} = \tau_1$ and $\alpha\sigma_1\check{\sigma} \sqsubseteq \alpha\check{\sigma}$. Hence, we have $\forall\vec{\alpha},\vec{\beta}.\,\alpha\sigma_1\check{\sigma} \sqsubseteq^{\forall} \forall\vec{\alpha},\vec{\beta}.\,\tau_1$. Since $\sqsubseteq^{\forall}$ is transitive, we conclude by showing

$$(\forall\vec{\alpha},\vec{\gamma}.\,\alpha\sigma_1)\check{\sigma} \sqsubseteq^{\forall} \forall\vec{\alpha},\vec{\beta}.\,\alpha\sigma_1\check{\sigma}\ .$$

We choose fresh variables $\vec{\alpha}_1,\vec{\gamma}_1$ (ensuring $\vec{\alpha}_1,\vec{\gamma}_1 \sharp \check{\sigma}$) and let $\rho = [\vec{\alpha}_1/\vec{\alpha}] \cup [\vec{\gamma}_1/\vec{\gamma}]$; then $(\forall\vec{\alpha},\vec{\gamma}.\,\alpha\sigma_1)\check{\sigma} = \forall\vec{\alpha}_1,\vec{\gamma}_1.\,\alpha\sigma_1\rho\check{\sigma}$.

To show $\forall\vec{\alpha}_1,\vec{\gamma}_1.\,\alpha\sigma_1\rho\check{\sigma} \sqsubseteq^{\forall} \forall\vec{\alpha},\vec{\beta}.\,\alpha\sigma_1\check{\sigma}$, we consider an arbitrary instance $\alpha\sigma_1\check{\sigma}\tilde{\sigma}$ of $\forall\vec{\alpha},\vec{\beta}.\,\alpha\sigma_1\check{\sigma}$, with $\tilde{\sigma} : \vec{\alpha},\vec{\beta} \to \mathsf{SType}$. We choose the instance $\alpha\sigma_1\rho\check{\sigma}\tilde{\sigma}'$ of $\forall\vec{\alpha}_1,\vec{\gamma}_1.\,\alpha\sigma_1\rho\check{\sigma}$, with $\tilde{\sigma}' = [\vec{\alpha}\tilde{\sigma}/\vec{\alpha}_1] \cup [\vec{\gamma}\check{\sigma}\tilde{\sigma}/\vec{\gamma}_1]$. We must show that $\tilde{\sigma}'$ is a valid instantiation. It has the correct domain, but it remains to show that $\vec{\alpha}\tilde{\sigma}$ and $\vec{\gamma}\check{\sigma}\tilde{\sigma}$ are static. For $\vec{\alpha}\tilde{\sigma}$, the result is immediate. If $\gamma \in \vec{\gamma}$, instead, we must show that $\gamma\check{\sigma}\tilde{\sigma}$ is static. We have $\gamma \in \mathsf{var}(\alpha\sigma_1)$. By Lemma 9.27, we have $\alpha \in \mathsf{var}(D_1)$. Hence, $\gamma \in \mathsf{var}(D_1)\sigma_1$. We have already shown $\mathsf{static}(\check{\sigma}, \mathsf{var}(D_1)\sigma_1)$. Hence, $\gamma\check{\sigma}$ is static; since $\tilde{\sigma}$ is static, $\gamma\check{\sigma}\tilde{\sigma}$ is static too. Now, we must show $\alpha\sigma_1\rho\check{\sigma}\tilde{\sigma}' \sqsubseteq \alpha\sigma_1\check{\sigma}\tilde{\sigma}$; actually, we show that the two types are equal. Consider $\beta \in \mathsf{var}(\alpha\sigma_1)$: we must show $\beta\rho\check{\sigma}\tilde{\sigma}' = \beta\check{\sigma}\tilde{\sigma}$.

- If $\beta \in \mathsf{dom}(\rho)$, then $\beta\rho\check{\sigma}\tilde{\sigma}' = \beta\rho\tilde{\sigma}'$. In particular, if $\beta \in \vec{\alpha}$, then $\beta\rho\check{\sigma}\tilde{\sigma}' = \beta\tilde{\sigma} = \beta\check{\sigma}\tilde{\sigma}$ (because $\beta\check{\sigma} = \beta$ since $\check{\sigma}$ is not defined on $\vec{\alpha}$). If $\beta \in \vec{\gamma}$, then $\beta\rho\check{\sigma}\tilde{\sigma}' = \beta\check{\sigma}\tilde{\sigma}$.
- If $\beta \notin \mathsf{dom}(\rho)$, then $\beta\rho\check{\sigma}\tilde{\sigma}' = \beta\check{\sigma}$. Since $\beta \in \mathsf{var}(\alpha\sigma_1)$, necessarily $\beta \in \mathsf{var}(\Gamma\sigma_1) \cup \mathsf{var}(e_1)$. If $\beta \in \mathsf{var}(\Gamma\sigma_1)$, then $\mathsf{var}(\beta\check{\sigma}) \subseteq \mathsf{var}(\beta\sigma_1\check{\sigma}) = \mathsf{var}(\Gamma\sigma)$; but then, since $\mathsf{dom}(\tilde{\sigma}) \sharp \Gamma\sigma$, we have $\beta\check{\sigma}\tilde{\sigma} = \beta\check{\sigma}$. If $\beta \in \mathsf{var}(e_1)$, since $\beta \notin \alpha$, we have $\beta \in \Delta$ and therefore $\beta\check{\sigma}\sigma = \beta = \beta\check{\sigma}$.

We apply the IH using the premises:

$$\Gamma'\check{\sigma} \vdash e_2 : t\check{\sigma} \qquad \mathsf{static}(\check{\sigma}, \Gamma')$$
$$\mathsf{dom}(\check{\sigma}) \sharp \Delta \supseteq \mathsf{var}(e_2) \qquad \mathfrak{U}_2 \sharp \Delta, t, \Gamma', \mathsf{dom}(\check{\sigma})$$

We derive:

$$\Gamma'; \Delta \vdash \langle\!\langle e_2 : t \rangle\!\rangle \rightsquigarrow D_2 \mid \vec{\alpha}_2 \qquad \check{\sigma} \cup \sigma_2' \Vdash_\Delta D_2 \qquad \mathsf{dom}(\sigma_2') \subseteq \vec{\alpha}_2 \subseteq \mathfrak{U}_2$$

We show $\vec{\alpha} \sharp \Gamma\sigma_1$ by contradiction. Assume that there exists an $\alpha \in \vec{\alpha}$ such that $\alpha \in \mathsf{var}(\Gamma\sigma_1)$. Then, since $\check{\sigma}$ is not defined on $\vec{\alpha}$, we would have $\alpha \in \mathsf{var}(\Gamma\sigma_1\check{\sigma})$. But $\Gamma\sigma_1\check{\sigma} = \Gamma\check{\sigma} = \Gamma\sigma$. Then, we would have $\alpha \in \mathsf{var}(\Gamma\sigma)$, which is impossible.
From the premises

$$\Gamma; \Delta \cup \vec{\alpha} \vdash \langle\!\langle e_1 : \alpha \rangle\!\rangle \rightsquigarrow D_1 \mid \vec{\alpha}_1$$
$$(\Gamma, x : \forall \vec{\alpha}, \vec{\gamma}.\, \alpha\sigma_1); \Delta \vdash \langle\!\langle e_2 : t \rangle\!\rangle \rightsquigarrow D_2 \mid \vec{\alpha}_2$$
$$\sigma_1 \in \mathsf{solve}_{\Delta \cup \vec{\alpha}}(D_1) \qquad \vec{\alpha} \sharp \Gamma\sigma_1$$
$$\vec{\gamma} = \mathsf{var}(\alpha\sigma_1) \setminus (\mathsf{var}(\Gamma\sigma_1) \cup \vec{\alpha} \cup (\mathsf{var}(e_1) \setminus \vec{\alpha}))$$

we derive

$$\Gamma; \Delta \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D_2 \cup \mathsf{equiv}(\sigma_1, D_1) \mid \vec{\alpha}_3$$

where $\vec{\alpha}_3 = \{\alpha\} \cup \vec{\alpha} \cup \vec{\alpha}_1 \cup \vec{\alpha}_2 \cup (\mathsf{var}(\sigma_1) \setminus \mathsf{var}(D_1)) \subseteq \mathfrak{U}$.
Let $\sigma' = [\tau_1/\alpha] \cup \sigma_1' \cup \tilde{\sigma}_1 \cup \sigma_2'$. We have $\mathsf{dom}(\sigma') \subseteq \vec{\alpha}_3 \subseteq \mathfrak{U}$.
It remains to prove that $\sigma \cup \sigma' \Vdash_\Delta D_2 \cup \mathsf{equiv}(\sigma_1, D_1)$. Note that $\sigma \cup \sigma' = \check{\sigma} \cup \sigma_2'$. Therefore, we have $\sigma \cup \sigma' \Vdash_\Delta D_2$. We show that $\sigma \cup \sigma'$ solves $\mathsf{equiv}(\sigma_1, D_1)$.

- When $\beta \in \mathsf{var}_{\stackrel{\cdot}{\sqsubseteq}}(D_1)$, we must show that $\beta(\sigma \cup \sigma')$ is a static type. Note that, since $\hat{\sigma} \cup \sigma_1' \Vdash_{\Delta \cup \vec{\alpha}} D_1$, we know $\beta(\hat{\sigma} \cup \sigma_1')$ is static. This gives the result we need since $\sigma \cup \sigma' = (\hat{\sigma} \cup \sigma_1') \cup (\tilde{\sigma}_1 \cup \sigma_2')$ and $\mathsf{dom}(\tilde{\sigma}_1 \cup \sigma_2') \sharp \mathsf{var}(D_1)$.
- When $\beta \in \mathsf{var}(D_1)\sigma_1$, we must show that $\beta(\sigma \cup \sigma')$ is a static type. We have shown $\mathsf{static}(\check{\sigma}, \mathsf{var}(D_1)\sigma_1)$. This is sufficient because $\sigma \cup \sigma' = \check{\sigma} \cup \sigma_2'$ and $\mathsf{dom}(\sigma_2') \sharp \mathsf{var}(D_1) \cup \mathsf{var}(\sigma_1)$.
- When $\beta \in \mathsf{dom}(\sigma_1)$ and $\beta\sigma_1$ is static, we must show $\beta(\sigma \cup \sigma') = \beta\sigma_1(\sigma \cup \sigma')$. We have $\beta\check{\sigma} = \beta\sigma_1\check{\sigma}$, which gives the result we need since $\check{\sigma}$ and $\sigma \cup \sigma'$ differ only on variables outside $\mathsf{dom}(\sigma_1)$ and $\mathsf{var}(\sigma_1)$. □

Gradual typing for set-theoretic types

10.5 LEMMA: Let T be a type frame with $\mathsf{var}(T) = \{ A_i \mid i \in I \}$. There exists a type frame T' such that the four sets

$$\mathsf{var}^{+\mathsf{cov}}(T') \subseteq \{ A_i^{+\wedge} \mid i \in I \} \qquad \mathsf{var}^{+\mathsf{cnt}}(T') \subseteq \{ A_i^{+\vee} \mid i \in I \}$$
$$\mathsf{var}^{-\mathsf{cov}}(T') \subseteq \{ A_i^{-\wedge} \mid i \in I \} \qquad \mathsf{var}^{-\mathsf{cnt}}(T') \subseteq \{ A_i^{-\vee} \mid i \in I \}$$

are pairwise disjoint and that

$$T = T'\big([A_i/A_i^{+\wedge}]_{i\in I} \cup [A_i/A_i^{+\vee}]_{i\in I} \cup [A_i/A_i^{-\wedge}]_{i\in I} \cup [A_i/A_i^{-\vee}]_{i\in I}\big)\,. \qquad \square$$

Proof: Clearly, T' is definable as a tree: it is the tree that coincides with T except on variables, and that, where T has a variable A_i, has one of $A_i^{+\wedge}$, $A_i^{+\vee}$, $A_i^{-\wedge}$, or $A_i^{-\vee}$ depending on the position of that occurrence of A_i. The tree T' is also clearly contractive and the sets of variables in different positions are disjoint.

For T' to be a type frame, it must also be regular. Since T is regular, it can be described by a finite system of equations

$$\begin{cases} x_1 = \bar{T}_1 \\ \quad\vdots \\ x_n = \bar{T}_n \end{cases}$$

such that every $\bar{T}_i$ is an inductively generated term of the grammar

$$\bar{T} ::= x \mid X \mid \alpha \mid b \mid \bar{T} \times \bar{T} \mid \bar{T} \to \bar{T} \mid \bar{T} \vee \bar{T} \mid \neg\bar{T} \mid \mathbb{0}$$

(x serves as a recursion variable) and that (reading the equations as a tree) $T = x_1$.

Then, T' can be defined as $x_1^{+\wedge}$ where

$$\begin{cases} x_1^{+\wedge} = f^{+\wedge}(\bar{T}_1) \\ x_1^{+\vee} = f^{+\vee}(\bar{T}_1) \\ x_1^{-\wedge} = f^{-\wedge}(\bar{T}_1) \\ x_1^{-\vee} = f^{-\vee}(\bar{T}_1) \\ \quad\vdots \\ x_n^{-\vee} = f^{-\vee}(\bar{T}_n) \end{cases}$$

and where (defining $\overline{+} = -$, $\overline{=} = +$, $\overline{\wedge} = \vee$, and $\overline{\vee} = \wedge$) $f^{pv}(\bar{T})$ is defined inductively as:

$$f^{pv}(x) = x^{pv} \qquad f^{pv}(X) = X^{pv} \qquad f^{pv}(\alpha) = \alpha^{pv} \qquad f^{pv}(b) = b$$
$$f^{pv}(\bar{T}_1 \times \bar{T}_2) = f^{pv}(\bar{T}_1) \times f^{pv}(\bar{T}_2) \qquad f^{pv}(\bar{T}_1 \to \bar{T}_2) = f^{p\overline{v}}(\bar{T}_1) \to f^{pv}(\bar{T}_2)$$
$$f^{pv}(\bar{T}_1 \vee \bar{T}_2) = f^{pv}(\bar{T}_1) \vee f^{pv}(\bar{T}_2) \qquad f^{pv}(\neg\bar{T}') = \neg f^{\overline{p}\overline{v}}(T') \qquad f^{pv}(\mathbb{0}) = \mathbb{0}$$

At most $4n$ equations are needed to define T' (they could be less, since some x_i^{pv} could be unreachable from $x_1^{+\wedge}$). Therefore, T' is regular. $\square$

10.9 LEMMA:

$$\left.\begin{array}{l} T \not\leq \mathbb{0} \\ \text{either } \{X, Y\} \sharp \mathsf{fvar}^-(T) \text{ or } \{X, Y\} \sharp \mathsf{fvar}^+(T) \end{array}\right\} \implies T[X/Y] \not\leq \mathbb{0}$$

$\square$

Proof: We first give some auxiliary definitions.

Let s range over the two symbols $\boxplus$ and $\boxminus$. We define $\overline{s}$ as follows: $\overline{\boxplus} \stackrel{\text{def}}{=} \boxminus$ and $\overline{\boxminus} \stackrel{\text{def}}{=} \boxplus$.

Given a type frame T', we write $T' \vDash \boxplus$ if $\{X, Y\} \sharp \mathsf{fvar}^-(T)$ and $T' \vDash \boxminus$ if $\{X, Y\} \sharp \mathsf{fvar}^+(T)$.

Note that, for all T', T_1, and T_2, we have:

$$\begin{aligned}
(\neg T' \vDash s) &\implies (T' \vDash \overline{s}) \\
(T_1 \vee T_2 \vDash s) &\implies (T_1 \vDash s) \wedge (T_2 \vDash s) \\
(T_1 \times T_2 \vDash s) &\implies (T_1 \vDash s) \wedge (T_2 \vDash s) \\
(T_1 \to T_2 \vDash s) &\implies (T_1 \vDash s) \wedge (T_2 \vDash s)
\end{aligned}$$

We define a function F^s on domain element tags (finite sets of variables):

$$F^{\boxplus}(L) = \begin{cases} L \cup \{X, Y\} & \text{if } X \in L \text{ or } Y \in L \\ L & \text{otherwise} \end{cases}$$

$$F^{\boxminus}(L) = \begin{cases} L \setminus \{X, Y\} & \text{if } X \notin L \text{ or } Y \notin L \\ L & \text{otherwise} \end{cases}$$

We also define F on domain elements as follows:

$$\begin{aligned}
F^s(c^L) &= c^{F^s(L)} \\
F^s((d_1, d_2)^L) &= (F^s(d_1), F^s(d_2))^{F^s(L)} \\
F^s(\{(d_1, d_1'), \ldots, (d_n, d_n')\}^L) &= \{(F^{\overline{s}}(d_1), F^s(d_1')), \ldots, (F^{\overline{s}}(d_n), F^s(d_n'))\}^{F^s(L)} \\
F^s(\Omega) &= \Omega
\end{aligned}$$

We must show:

$$\left.\begin{array}{l} T \not\leq \mathbb{0} \\ \text{either } \{X, Y\} \sharp \mathsf{fvar}^-(T) \text{ or } \{X, Y\} \sharp \mathsf{fvar}^+(T) \end{array}\right\} \implies T[X/Y] \not\leq \mathbb{0}$$

This can be restated as:

$$\left.\begin{array}{l} \exists d \in \mathsf{Domain}.\ (d : T) \\ \exists s.\ T \vDash s \end{array}\right\} \implies \exists d' \in \mathsf{Domain}.\ (d' : T[X/Y])$$

We prove the following, stronger claim:

$$\forall d, T, s. \quad T \vDash s \implies \begin{cases} (d : T) \implies (F^s(d) : T[X/Y]) \\ \neg(d : T) \implies \neg(F^{\overline{s}}(d) : T[X/Y]) \end{cases}$$

by induction on the pair (d, T), ordered lexicographically. For a given d, T, and s, we assume $T \vDash s$ and proceed by case analysis on T and d.

Let $\sigma = [X/Y]$.

Case: $T = \alpha$

Since $\alpha\sigma = \alpha$, we must show

$$(d : \alpha) \implies (F^s(d) : \alpha) \qquad \neg(d : \alpha) \implies \neg(F^{\overline{s}}(d) : \alpha)\,.$$

If $(d : \alpha)$, then $\alpha \in \mathsf{tags}(d)$ and also $\alpha \in \mathsf{tags}(F^s(d))$.
Likewise, if $d \notin [\![\alpha]\!]$, then $\alpha \notin \mathsf{tags}(d)$ and also $\alpha \notin \mathsf{tags}(F^{\overline{s}}(d))$.

Case: $T = Z$, with $Z \neq X$ and $Z \neq Y$

Like the previous case.

Case: $T = X$

Since $X \in \mathsf{fvar}^+(X)$, we have $s = \boxplus$.
We must show

$$(d : X) \implies (F^{\boxplus}(d) : X) \qquad \neg(d : X) \implies \neg(F^{\boxminus}(d) : X)\,.$$

If $(d : X)$, then $X \in \mathsf{tags}(d)$ and $X \in \mathsf{tags}(F^{\boxplus}(d))$. If $\neg(d : X)$, then $X \notin \mathsf{tags}(d)$ and $X \notin \mathsf{tags}(F^{\boxminus}(d))$.

Case: $T = Y$

Since $Y \in \mathsf{fvar}^+(Y)$, we have $s = \boxplus$.
We must show

$$(d : Y) \implies (F^{\boxplus}(d) : X) \qquad \neg(d : Y) \implies \neg(F^{\boxminus}(d) : X)\,.$$

If $(d : Y)$, then $Y \in \mathsf{tags}(d)$ and $X \in \mathsf{tags}(F^{\boxplus}(d))$.
If $\neg(d : Y)$, then $Y \notin \mathsf{tags}(d)$ and then $X \notin \mathsf{tags}(F^{\boxminus}(d))$.

Case: $T = b$

Since $b\sigma = b$, we must show

$$(d : b) \implies (F^s(d) : b) \qquad \neg(d : b) \implies (F^{\overline{s}}(d) : b)\,.$$

If $(d : b)$, then $d = c^L$ with $c \in \mathbb{B}(b)$. Then, $F^s(d) = c^{F^s(L)}$ and $(F^s(d) : b)$.
If $\neg(d : b)$ and d is of the form c^L, then $c \notin \mathbb{B}(b)$: then, $F^{\overline{s}}(d) \notin [\![b]\!]$. If d is not of the form c^L, then $F^{\overline{s}}(d)$ is not either and we have $F^{\overline{s}}(d) \notin [\![b]\!]$.

Case: $T = T_1 \times T_2$

Since $T \vDash s$, we have $T_1 \vDash s$ and $T_2 \vDash s$.
We must show

$$(d : T_1 \times T_2) \implies (F^s(d) : T_1\sigma \times T_2\sigma)$$
$$\neg(d : T_1 \times T_2) \implies \neg(F^{\overline{s}}(d) : T_1\sigma \times T_2\sigma)\,.$$

If $(d : T_1 \times T_2)$, then d is of the form $(d_1, d_2)^L$ and, for both i, $(d_i : T_1)$. We have $F^s(d) = (F^s(d_1), F^s(d_2))^{F^s(L)}$. By IH, $(d_1 : T_1)$ implies $(F^s(d_1) : T_1\sigma)$; likewise for d_2. Therefore, $(F^s(d) : T_1\sigma \times T_2\sigma)$.
If $\neg(d : T_1 \times T_2)$ and $d = (d_1, d_2)^L$, then either $\neg(d_1 : T_1)$ or $\neg(d_2 : T_2)$. Then, by IH, either $\neg(F^{\overline{s}}(d_1) : T_1\sigma)$ or $\neg(F^{\overline{s}}(d_2) : T_2\sigma)$. Therefore, $\neg(F^{\overline{s}}(d) : T_1\sigma \times T_2\sigma)$. If d is of another form, then the result is immediate.

Case: $T = T_1 \to T_2$

Since $T \vDash s$, we have $T_1 \vDash s$ and $T_2 \vDash s$.

We must show

$$(d : T_1 \to T_2) \implies (F^s(d) : T_1\sigma \to T_2\sigma)$$
$$\neg(d : T_1 \to T_2) \implies \neg(F^{\overline{s}}(d) : T_1\sigma \to T_2\sigma)\,.$$

If $(d : T_1 \to T_2)$, then d is of the form $\{\,(d_j, d'_j) \mid j \in J\,\}^L$ and, for all $j \in J$, we have:

$$(d_j : T_1) \implies (d'_j : T_2)\,.$$

We have $F^s(d) = \{\,(F^{\overline{s}}(d_j), F^s(d'_j)) \mid j \in J\,\}^{F^s(L)}$.

For every j, by the induction hypothesis applied to T_1 and d_j, and to T_2 and d'_j, we get

$$(d_j : T_1) \implies (F^s(d_j) : T_1\sigma) \qquad \neg(d_j : T_1) \implies \neg(F^{\overline{s}}(d_j) : T_1\sigma)$$
$$(d'_j : T_2) \implies (F^s(d'_j) : T_2\sigma) \qquad \neg(d'_j : T_2) \implies \neg(F^{\overline{s}}(d'_j) : T_2\sigma)\,.$$

We must show, for all $j \in J$:

$$(F^{\overline{s}}(d_j) : T_1\sigma) \implies (F^s(d'_j) : T_2\sigma)$$

which we prove using the induction hypothesis (in particular, using the contrapositive of the second implication derived by induction).

If $\neg(d : T_1 \to T_2)$ and d is of the form $\{\,(d_j, d'_j) \mid j \in J\,\}^L$, then there exists a $j_0 \in J$ such that

$$(d_{j_0} : T_1) \qquad \neg(d'_{j_0} \in T_2)\,.$$

We have $F^{\overline{s}}(d) = \{\,(F^s(d_j), F^{\overline{s}}(d'_j)) \mid j \in J\,\}^{F^{\overline{s}}(L)}$. By IH, we show

$$(F^s(d_{j_0}) : T_1\sigma) \qquad \neg(F^{\overline{s}}(d_{j_0}) : T_2\sigma)\,.$$

If d is of another form, we have the result directly. then we get the result directly.

Case: $T = T_1 \vee T_2$

Since $T \vDash s$, we have $T_1 \vDash s$ and $T_2 \vDash s$.

By the induction hypothesis applied to d and T_i, we get

$$(d : T_i) \implies (F^s(d) : T_i\sigma) \qquad \neg(d : T_i) \implies \neg(F^{\overline{s}}(d) : T_i\sigma)\,.$$

We must show

$$(d : T_1 \vee T_2) \implies (F^s(d) : T_1\sigma \vee T_2\sigma)$$
$$\neg(d : T_1 \vee T_2) \implies (F^{\overline{s}}(d) : T_1\sigma \vee T_2\sigma)\,.$$

To show the first implication, assume $(d : T_1 \vee T_2)$: then either $(d : T_1)$ or $(d : T_2)$; then either $(F^s(d) : T_1\sigma)$ or $(F^s(d) : T_2\sigma)$; then $(F^s(d) : T_1\sigma \vee T_2\sigma)$.
To show the second, assume $\neg(d : T_1 \vee T_2)$: then $\neg(d : T_1)$ and $\neg(d : T_2)$; then $\neg(F^{\overline{s}}(d) : T_1)$ and $\neg(F^{\overline{s}}(d) : T_2)$; then $\neg(F^{\overline{s}}(d) : T_1 \vee T_2)$.

Case: $T = \neg T'$

Since $T \vDash s$, $T' \vDash \overline{s}$.

By applying the induction hypothesis to d and T', we get

$$(d : T') \implies (F^{\overline{s}}(d) : T'\sigma) \qquad \neg(d : T') \implies \neg(F^{s}(d) : T'\sigma)\,.$$

We must show

$$(d : \neg T') \implies (F^{s}(d) : \neg(T'\sigma)) \qquad \neg(d : \neg T') \implies \neg(F^{\overline{s}}(d) : \neg(T'\sigma))\,.$$

For the first implication, assume $(d : \neg T')$: then $\neg(d : T')$, $\neg(F^s(d) : T'\sigma)$, and $(F^s(d) : \neg(T'\sigma))$. For the second, assume $\neg(d : \neg T')$: then $\neg\neg(d : T')$, that is, $(d : T')$; hence $(F^{\overline{s}}(d) : T'\sigma)$, and $\neg(F^{\overline{s}}(d) : \neg(T'\sigma))$.

Case: $T = \mathbb{0}$

Both implications are trivial. □

10.10 LEMMA:

$$\left.\begin{array}{l} T_1 \leq T_2 \\ X \in \mathsf{fvar}^+(T_1) \implies X \notin \mathsf{fvar}^+(T_2) \\ X \in \mathsf{fvar}^-(T_1) \implies X \notin \mathsf{fvar}^-(T_2) \\ Y \sharp T_1, T_2, X \end{array}\right\} \implies T_1[Y/X] \leq T_2$$

□

Proof: If $X \notin \mathsf{fvar}(T_1)$, the result is immediate because $T_1[Y/X] = T_1$. If $X \notin \mathsf{fvar}(T_2)$, then we have $T_2 = T_2[Y/X]$ and the result can be derived by Proposition 10.2. We consider the case $X \in \mathsf{fvar}(T_1) \cap \mathsf{fvar}(T_2)$. In this case, we have $X \notin \mathsf{fvar}^+(T_1) \cap \mathsf{fvar}^-(T_1)$: otherwise, X could not occur in T_2. Therefore, X occurs only positively or only negatively in T_1.

Given T_1, T_2, X, and Y satisfying

$$X \in \mathsf{fvar}^+(T_1) \implies X \notin \mathsf{fvar}^+(T_2) \qquad X \in \mathsf{fvar}^-(T_1) \implies X \notin \mathsf{fvar}^-(T_2)$$
$$Y \sharp T_1, T_2, X\,,$$

we must show $T_1 \leq T_2 \implies T_1[Y/X] \leq T_2$.

We show the contrapositive: $T_1[Y/X] \not\leq T_2 \implies T_1 \not\leq T_2$. Assume $T_1[Y/X] \not\leq T_2$.

We have $T_1 = T_1[Y/X][X/Y]$ and $T_2 = T_2[X/Y]$. Let $T = T_1[Y/X] \setminus T_2$. We have $T \not\leq \mathbb{0}$ by definition of subtyping.

We show that either $\{X, Y\} \sharp \mathsf{fvar}^-(T)$ or $\{X, Y\} \sharp \mathsf{fvar}^+(T)$ holds. Note that

$$\mathsf{fvar}^+(T) = \mathsf{fvar}^+(T_1[Y/X]) \cup \mathsf{fvar}^-(T_2)$$
$$\mathsf{fvar}^-(T) = \mathsf{fvar}^-(T_1[Y/X]) \cup \mathsf{fvar}^+(T_2)\,.$$

If $X \in \mathsf{fvar}^+(T_1)$, then $X \notin \mathsf{fvar}^-(T_1)$ and $X \notin \mathsf{fvar}^+(T_2)$: therefore, $\{X, Y\} \sharp \mathsf{fvar}^-(T)$. If $X \in \mathsf{fvar}^-(T_1)$, then $X \notin \mathsf{fvar}^+(T_1)$ and $X \notin \mathsf{fvar}^-(T_2)$: therefore, $\{X, Y\} \sharp \mathsf{fvar}^+(T)$.

By Lemma 10.9, we have $T[X/Y] \not\leq \mathbb{0}$: that is, $(T_1[Y/X] \setminus T_2)[X/Y] \not\leq \mathbb{0}$; that is, $T_1[Y/X][X/Y] \not\leq T_2[X/Y]$, which is $T_1 \not\leq T_2$. □

10.12 LEMMA:

$$\left.\begin{array}{l} T \not\leq \mathbb{0} \\ X \notin \mathsf{fvar}^{\mathsf{even}}(T) \\ Y \notin \mathsf{fvar}^{\mathsf{odd}}(T) \end{array}\right\} \implies T[X/Y] \not\leq \mathbb{0}$$

□

Proof: We first give some auxiliary definitions.

Let s range over the two symbols $\triangle$ and $\triangledown$. We define $\overline{s}$ as follows: $\overline{\triangle} \stackrel{\text{def}}{=} \triangledown$ and $\overline{\triangledown} \stackrel{\text{def}}{=} \triangle$.

Given a type frame T', we write $T' \vDash \triangle$ if $X \notin \mathsf{fvar}^{\mathsf{odd}}(T')$ and $Y \notin \mathsf{fvar}^{\mathsf{even}}(T')$; we write $T' \vDash \triangledown$ if $X \notin \mathsf{fvar}^{\mathsf{even}}(T')$ and $Y \notin \mathsf{fvar}^{\mathsf{odd}}(T')$.

Note that, for all T', T_1, and T_2, we have:

$$\begin{aligned} (\neg T' \vDash s) &\implies (T' \vDash s) \\ (T_1 \vee T_2 \vDash s) &\implies (T_1 \vDash s) \wedge (T_2 \vDash s) \\ (T_1 \times T_2 \vDash s) &\implies (T_1 \vDash s) \wedge (T_2 \vDash s) \\ (T_1 \to T_2 \vDash s) &\implies (T_1 \vDash \overline{s}) \wedge (T_2 \vDash s) \end{aligned}$$

We define a function F^s on domain element tags (finite sets of variables) as:

$$F^{\triangle}(L) = L \qquad F^{\triangledown}(L) = \begin{cases} L \cup \{X\} & \text{if } Y \in L \\ L \setminus \{X\} & \text{if } Y \notin L \end{cases}$$

We also define F on domain elements as follows:

$$\begin{aligned} F^s(c^L) &= c^{F^s(L)} \\ F^s((d_1, d_2)^L) &= (F^s(d_1), F^s(d_2))^{F^s(L)} \\ F^s(\{(d_1, d_1'), \ldots, (d_n, d_n')\}^L) &= \{(F^{\overline{s}}(d_1), F^s(d_1')), \ldots, (F^{\overline{s}}(d_n), F^s(d_n'))\}^{F^s(L)} \\ F^s(\Omega) &= \Omega \end{aligned}$$

We must show:

$$\left.\begin{array}{l} T \not\leq \mathbb{0} \\ X \notin \mathsf{fvar}^{\mathsf{even}}(T) \\ Y \notin \mathsf{fvar}^{\mathsf{odd}}(T) \end{array}\right\} \implies T[X/Y] \not\leq \mathbb{0}$$

This can be restated as:

$$\left.\begin{array}{l} \exists d \in \mathsf{Domain}.\ (d : T) \\ T \vDash \triangledown \end{array}\right\} \implies \exists d' \in \mathsf{Domain}.\ (d' : T[X/Y])$$

We prove the following, stronger claim:

$$\forall d, T, s. \quad T \vDash s \implies \big((d : T) \iff (F^s(d) : T[X/Y])\big)$$

by induction on the pair (d, T), ordered lexicographically. For a given d, T, and s, we assume $T \vDash s$ and proceed by case analysis on T and d.

Let $\sigma = [X/Y]$.

Case: $T = \alpha$

Note that $\alpha\sigma = \alpha$.

$$\begin{aligned}(d : \alpha) &\iff \alpha \in \mathsf{tags}(d)\\ &\iff \alpha \in \mathsf{tags}(F^s(d))\end{aligned}$$

(neither $F^\triangle$ nor $F^\triangledown$ affect variables other than X)

$$\iff (F^s(d) : \alpha)$$

Case: $T = Z$, with $Z \neq X$ and $Z \neq Y$

Like the previous case.

Case: $T = X$

Note that we must have $T \vDash \triangle$ because $X \in \mathsf{fvar}^{\mathsf{even}}(X)$ and $X \notin \mathsf{fvar}^{\mathsf{odd}}(X)$.

Note that $X\sigma = X$.

$$\begin{aligned}(d : X) &\iff X \in \mathsf{tags}(d)\\ &\iff X \in \mathsf{tags}(F^\triangle(d))\\ &\iff (F^\triangle(d) : X)\end{aligned}$$

Case: $T = Y$

Note that we must have $T \vDash \triangledown$ because $Y \in \mathsf{fvar}^{\mathsf{even}}(Y)$ and $Y \notin \mathsf{fvar}^{\mathsf{odd}}(Y)$.

Note that $Y\sigma = X$.

$$\begin{aligned}(d : Y) &\iff Y \in \mathsf{tags}(d)\\ &\iff X \in \mathsf{tags}(F^\triangledown(d))\\ &\iff (F^\triangledown(d) : X)\end{aligned}$$

Case: $T = b$

Note that $b\sigma = b$.

If $(d : b)$, then d must be of the form c^L with $c \in \mathbb{B}(b)$. Then, $F^s(d) = c^{F^s(L)}$ and $(F^s(d) : b)$.

If $(F^s(d) : b)$, then $F^s(d)$ must be of the form c^L with $c \in \mathbb{B}(b)$. Then, $d = c^{L'}$ and $(d : b)$.

Case: $T = T_1 \times T_2$

If $(d : T_1 \times T_2)$, then $d = (d_1, d_2)^L$, $(d_1 : T_1)$, and $(d_2 : T_2)$. We have $F^s(d) = (F^s(d_1), F^s(d_2))^{F^s(L)}$. By IH we have, for $i \in \{1, 2\}$, $(d_i : T_i) \iff (F^s(d_i) : T_i\sigma)$; hence, $(F^s(d) : T_1\sigma \times T_2\sigma)$.

If $(F^s(d) : T_1\sigma \times T_2\sigma)$, then $F^s(d) = (d_1, d_2)^L$, $(d_1 : T_1\sigma)$, and $(d_2 : T_2\sigma)$. Then, we have $d = (d_1', d_2')^{L'}$, with $d_1 = F^s(d_1')$ and $d_2 = F^s(d_2')$. By IH we have, for $i \in \{1, 2\}$, $(d_i' : T_i) \iff (d_i : T_i\sigma)$; hence, $(d : T_1 \times T_2)$.

Case: $T = T_1 \to T_2$

Note that, since $T \vDash s$, we have $T_1 \vDash \overline{s}$ and $T_2 \vDash s$.
If $(d : T_1 \to T_2)$, then $d = \{\, (d_j, d'_j) \mid j \in J \,\}^L$ and

$$\forall j \in J.\ (d_j : T_1) \implies (d'_j : T_2)\,.$$

Then, $F^s(d) = \{\, (F^{\overline{s}}(d_j), F^s(d'_j)) \mid j \in J \,\}^{F^s(L)}$. By IH, for every $j \in J$,

$$(d_j : T_1) \iff (F^{\overline{s}}(d_j) : T_1\sigma) \qquad (d'_j : T_2) \iff (F^s(d'_j) : T_2\sigma)\,.$$

Therefore, we have

$$\forall j \in J.\ (F^{\overline{s}}(d_j) : T_1\sigma) \implies (F^s(d'_j) : T_2\sigma)$$

and hence $(F^s(d) : T_1\sigma \to T_2\sigma)$.
If $(F^s(d) : T_1\sigma \to T_2\sigma)$, then $F^s(d) = \{\, (d_j, d'_j) \mid j \in J \,\}^L$ and

$$\forall j \in J.\ (d_j : T_1\sigma) \implies (d'_j : T_2\sigma)\,.$$

Then, $d = \{\, (\bar{d}_j, \bar{d}'_j) \mid j \in J \,\}^{L'}$, with, for every $j \in J$, $F^{\overline{s}}(\bar{d}_j) = d_j$ and $F^s(\bar{d}'_j) = d'_j$. By IH, for every $j \in J$,

$$(\bar{d}_j : T_1) \iff (d_j : T_1\sigma) \qquad (\bar{d}'_j : T_2) \iff (d'_j : T_2\sigma)\,.$$

Therefore, we have

$$\forall j \in J.\ (\bar{d}_j : T_1) \implies (\bar{d}'_j : T_2)$$

and hence $(d : T_1 \to T_2)$.

Case: $T = T_1 \vee T_2$

$$\begin{aligned}
(d : T_1 \vee T_2) &\iff (d : T_1) \vee (d : T_2) \\
&\iff (F^s(d) : T_1\sigma) \vee (F^s(d) : T_2\sigma) && \text{by IH} \\
&\iff (F^s(d) : T_1\sigma \vee T_2\sigma)
\end{aligned}$$

Case: $T = \neg T'$

$$\begin{aligned}
(d : \neg T') &\iff \neg(d : T') \\
&\iff \neg(F^s(d) : T'\sigma) && \text{by IH} \\
&\iff (F^s(d) : \neg(T'\sigma))
\end{aligned}$$

Case: $T = \mathbb{0}$

Trivial, since $(d : \mathbb{0})$ never holds for any d and since $\mathbb{0}\sigma = \mathbb{0}$. □

10.14 LEMMA:

$$T \le \mathbb{0} \implies \exists T', \vec{X}, \vec{Y}.\ \begin{cases} T' \le \mathbb{0} \\ T = T'[\vec{X}/\vec{Y}] \\ \mathsf{fvar}^{\mathsf{even}}(T') \sharp \mathsf{fvar}^{\mathsf{odd}}(T') \end{cases}$$

□

Proof: Assume that $\mathsf{fvar}(T) = \{X_1, \ldots, X_n\}$.

By Corollary 10.7, we can find T' such that $\mathsf{fvar}^{\mathsf{even}}(T') \subseteq \{X_1, \ldots, X_n\}$ is disjoint from $\mathsf{fvar}^{\mathsf{odd}}(T') \subseteq \{X'_1, \ldots, X'_n\}$ and that $T = T'[X_i/X'_i]_{i=1}^n$.

We must prove $T' \leq \mathbb{0}$. We have $T \leq \mathbb{0}$, which is $T'[X_i/X'_i]_{i=1}^n \leq \mathbb{0}$. Therefore, we also have $T'[X_i/X'_i]_{i=1}^n[X'_i/X_i]_{i=1}^n \leq \mathbb{0}$ (by Proposition 10.2), which is $T'[X'_i/X_i]_{i=1}^n \leq \mathbb{0}$.

Let $\vec{X}$ be the vector $X_1 \ldots X_n$ and $\vec{X}'$ be the vector $X'_1 \ldots X'_n$. We have $\vec{X} \sharp \mathsf{fvar}^{\mathsf{odd}}(T')$ and $\vec{X}' \sharp \mathsf{fvar}^{\mathsf{even}}(T')$. We also have $\vec{X} \sharp \vec{Y}$.

By Lemma 10.13, we have

$$T' \not\leq \mathbb{0} \implies T'[\vec{X}'/\vec{X}] \not\leq \mathbb{0}$$

and, by contrapositive,

$$T'[\vec{X}'/\vec{X}] \leq \mathbb{0} \implies T' \leq \mathbb{0}$$

which yields $T' \leq \mathbb{0}$. □

10.17 LEMMA: If $\tau_1 \leq^? \tau_2$, then $\tau_1^\bullet \leq \tau_2^\bullet$. □

Proof: By definition of $\tau_1 \leq^? \tau_2$, there exist T_1 and T_2 such that:

$$T_1^\dagger = \tau_1 \qquad T_2^\dagger = \tau_2 \qquad T_1 \leq T_2$$
$$\mathsf{fvar}^+(T_1) \sharp \mathsf{fvar}^-(T_1) \qquad \mathsf{fvar}^+(T_2) \sharp \mathsf{fvar}^-(T_2)\,.$$

Let $\vec{X} = \big(\mathsf{fvar}^+(T_1) \cap \mathsf{fvar}^-(T_2)\big) \cup \big(\mathsf{fvar}^-(T_1) \cap \mathsf{fvar}^+(T_2)\big)$ and let $\vec{Y}$ be a vector of variables outside T_1 and T_2. Since T_1 and T_2 are polarized, we have

$$\forall X \in \vec{X}.\ \begin{cases} X \in \mathsf{fvar}^+(T_1) \implies X \notin \mathsf{fvar}^+(T_2) \\ X \in \mathsf{fvar}^-(T_1) \implies X \notin \mathsf{fvar}^-(T_2) \end{cases}$$

and we can apply Lemma 10.11 to derive $T_1[\vec{Y}/\vec{X}] \leq T_2$.

We have

$$\mathsf{fvar}^+(T_1[\vec{Y}/\vec{X}], T_2) \sharp \mathsf{fvar}^-(T_1[\vec{Y}/\vec{X}], T_2)\,.$$

We apply Lemma 10.15 to $T_1[\vec{Y}/\vec{X}]$ and T_2 to find T'_1, T'_2, $\vec{X}'$, and $\vec{Y}'$ such that:

$$T'_1 \leq T'_2 \qquad T_1[\vec{Y}/\vec{X}] = T'_1[\vec{X}'/\vec{Y}'] \qquad T_2 = T'_2[\vec{X}'/\vec{Y}']$$
$$\mathsf{fvar}^{\mathsf{even}}(T'_1, T'_2) \sharp \mathsf{fvar}^{\mathsf{odd}}(T'_1, T'_2)\,.$$

We have

$$\tau_1 = T_1^\dagger = (T_1[\vec{Y}/\vec{X}])^\dagger = (T'_1[\vec{X}'/\vec{Y}'])^\dagger = (T'_1)^\dagger$$
$$\tau_2 = T_2^\dagger = (T'_2[\vec{X}'/\vec{Y}'])^\dagger = (T'_2)^\dagger\,.$$

We also have

$$\mathsf{fvar}^+(T'_1, T'_2) \sharp \mathsf{fvar}^-(T'_1, T'_2) \qquad \mathsf{fvar}^{\mathsf{even}}(T'_1, T'_2) \sharp \mathsf{fvar}^{\mathsf{odd}}(T'_1, T'_2)$$

and therefore the following four sets are disjoint

$$\mathsf{fvar}^{+\mathsf{cov}}(T_1', T_2') \qquad \mathsf{fvar}^{+\mathsf{cnt}}(T_1', T_2') \qquad \mathsf{fvar}^{-\mathsf{cov}}(T_1', T_2') \qquad \mathsf{fvar}^{-\mathsf{cnt}}(T_1', T_2') \,.$$

Then, by Lemma 10.16, we have $\tau_1^\bullet \leq \tau_2^\bullet$. □

10.18 LEMMA: Let τ_1 and τ_2 be two gradual types. Let $T_1 \in \star^{\mathsf{var}}(\tau_1)$ and $T_2 \in \star^{\mathsf{var}}(\tau_2)$ be such that $T_1 \leq T_2$. Then, $\tau_1^\bullet \leq \tau_2^\bullet$. □

Proof: We have

$$T_1^\dagger = \tau_1 \qquad T_2^\dagger = \tau_2$$
$$\mathsf{fvar}^{\mathsf{cov}}(T_1) \sharp \mathsf{fvar}^{\mathsf{cnt}}(T_1) \qquad \mathsf{fvar}^{\mathsf{cov}}(T_2) \sharp \mathsf{fvar}^{\mathsf{cnt}}(T_2) \qquad T_1 \leq T_2 \,.$$

We apply Lemma 10.15 to T_1 and T_2 to find T_1', T_2', $\vec{X}$, and $\vec{Y}$ such that:

$$T_1' \leq T_2' \qquad T_1 = T_1'[\vec{X}/\vec{Y}] \qquad T_2 = T_2'[\vec{X}/\vec{Y}]$$
$$\mathsf{fvar}^{\mathsf{even}}(T_1', T_2') \sharp \mathsf{fvar}^{\mathsf{odd}}(T_1', T_2') \,.$$

Since we have

$$\mathsf{fvar}^{\mathsf{cov}}(T_1') \sharp \mathsf{fvar}^{\mathsf{cnt}}(T_1') \qquad \mathsf{fvar}^{\mathsf{cov}}(T_2') \sharp \mathsf{fvar}^{\mathsf{cnt}}(T_2')$$
$$\mathsf{fvar}^{\mathsf{even}}(T_1', T_2') \sharp \mathsf{fvar}^{\mathsf{odd}}(T_1', T_2') \,,$$

we also have

$$\mathsf{fvar}^{+}(T_1') \sharp \mathsf{fvar}^{-}(T_1') \text{ and } \mathsf{fvar}^{+}(T_2') \sharp \mathsf{fvar}^{-}(T_2') \,.$$

Let $\vec{X}' = \left(\mathsf{fvar}^{+}(T_1') \cap \mathsf{fvar}^{-}(T_2')\right) \cup \left(\mathsf{fvar}^{-}(T_1') \cap \mathsf{fvar}^{+}(T_2')\right)$ and let $\vec{Y}'$ be a vector of variables outside T_1' and T_2'. We have

$$\forall X \in \vec{X}'. \begin{cases} X \in \mathsf{fvar}^{+}(T_1') \implies X \notin \mathsf{fvar}^{+}(T_2') \\ X \in \mathsf{fvar}^{-}(T_1') \implies X \notin \mathsf{fvar}^{-}(T_2') \end{cases}$$

and we can apply Lemma 10.11 to derive $T_1'[\vec{Y}'/\vec{X}'] \leq T_2'$.

We have

$$\tau_1 = T_1^\dagger = (T_1'[\vec{X}/\vec{Y}])^\dagger = (T_1')^\dagger = (T_1'[\vec{Y}'/\vec{X}'])^\dagger$$
$$\tau_2 = T_2^\dagger = (T_2'[\vec{X}/\vec{Y}])^\dagger = (T_2')^\dagger \,.$$

Let $T_1'' = T_1'[\vec{Y}'/\vec{X}']$.

We also have

$$\mathsf{fvar}^{+}(T_1'', T_2') \sharp \mathsf{fvar}^{-}(T_1'', T_2') \qquad \mathsf{fvar}^{\mathsf{even}}(T_1'', T_2') \sharp \mathsf{fvar}^{\mathsf{odd}}(T_1'', T_2')$$

and therefore the following four sets are disjoint

$$\mathsf{fvar}^{+\mathsf{cov}}(T_1'', T_2') \quad \mathsf{fvar}^{+\mathsf{cnt}}(T_1'', T_2') \quad \mathsf{fvar}^{-\mathsf{cov}}(T_1'', T_2') \quad \mathsf{fvar}^{-\mathsf{cnt}}(T_1'', T_2') \,.$$

Then, by Lemma 10.16, we have $\tau_1^\bullet \leq \tau_2^\bullet$. □

10.22 PROPOSITION:

$$\forall T, \sigma_1, \sigma_2. \quad \left.\begin{array}{l} \sigma_1|_{\mathsf{var}^{\mathsf{cov}}(T)} \leq \sigma_2|_{\mathsf{var}^{\mathsf{cov}}(T)} \\ \sigma_2|_{\mathsf{var}^{\mathsf{cnt}}(T)} \leq \sigma_1|_{\mathsf{var}^{\mathsf{cnt}}(T)} \end{array}\right\} \implies T\sigma_1 \leq T\sigma_2$$

□

Proof: We define

$$P(T, \sigma_1, \sigma_2) \overset{\text{def}}{\iff} \left(\sigma_1|_{\mathsf{var}^{\mathsf{cov}}(T)} \leq \sigma_2|_{\mathsf{var}^{\mathsf{cov}}(T)}\right) \text{ and } \left(\sigma_2|_{\mathsf{var}^{\mathsf{cnt}}(T)} \leq \sigma_1|_{\mathsf{var}^{\mathsf{cnt}}(T)}\right)$$

and note that the following hold

$$\begin{aligned} P(A, \sigma_1, \sigma_2) &\implies A\sigma_1 \leq A\sigma_2 \\ P(T_1 \times T_2, \sigma_1, \sigma_2) &\implies P(T_1, \sigma_1, \sigma_2) \text{ and } P(T_2, \sigma_1, \sigma_2) \\ P(T_1 \to T_2, \sigma_1, \sigma_2) &\implies P(T_1, \sigma_2, \sigma_1) \text{ and } P(T_2, \sigma_1, \sigma_2) \\ P(T_1 \vee T_2, \sigma_1, \sigma_2) &\implies P(T_1, \sigma_1, \sigma_2) \text{ and } P(T_2, \sigma_1, \sigma_2) \\ P(\neg T', \sigma_1, \sigma_2) &\implies P(T', \sigma_2, \sigma_1) \end{aligned}$$

We show the following result (which implies the statement)

$$\forall \sigma_1, \sigma_2, d, T. \quad \left.\begin{array}{l} P(T, \sigma_1, \sigma_2) \\ (d : T\sigma_1) \end{array}\right\} \implies (d : T\sigma_2)$$

by induction on (d, T).

Case: $T = b$ or $T = \mathbb{0}$ Trivial, since $T\sigma_1 = T = T\sigma_2$.

Case: $T = A$
We have $A\sigma_1 \leq A\sigma_2$ and $(d : A\sigma_1)$, which implies $(d : A\sigma_2)$.

Case: $T = T_1 \times T_2$
We have $T\sigma_1 = (T_1\sigma_1) \times (T_2\sigma_1)$ and $T\sigma_2 = (T_1\sigma_2) \times (T_2\sigma_2)$.
Since $(d : T\sigma_1)$, we have $d = (d_1, d_2)$ and $(d_i : T_i\sigma_1)$.
Since $P(T_i, \sigma_1, \sigma_2)$ holds for both i, by IH we have $(d_i : T_i\sigma_2)$.
Then, $(d : T\sigma_2)$.

Case: $T = T_1 \to T_2$
We have $T\sigma_1 = (T_1\sigma_1) \to (T_2\sigma_1)$ and $T\sigma_2 = (T_1\sigma_2) \to (T_2\sigma_2)$.
Since $(d : T\sigma_1)$, we have $d = \{ (d_i, d_i') \mid i \in I \}$ and, for every $i \in I$, $(d_i : T_1\sigma_1) \implies (d_i' : T_2\sigma_1)$.
We have $P(T_1, \sigma_2, \sigma_1)$ and $P(T_2, \sigma_1, \sigma_2)$.
For every d_i such that $(d_i : T_1\sigma_2)$, by IH we have $(d_i : T_1\sigma_1)$, therefore $(d_i' : T_2\sigma_1)$, and, by IH, $(d_i' : T_2\sigma_2)$.
Therefore, $\forall i \in I.\ (d_i : T_1\sigma_2) \implies (d_i' : T_2\sigma_2)$, and hence $(d : T\sigma_2)$.

Case: $T = T_1 \vee T_2$
We have either $(d : T_1\sigma_1)$ or $(d : T_2\sigma_1)$. Therefore, since $P(T_i, \sigma_1, \sigma_2)$ holds for both i, by IH we have either $(d : T_1\sigma_2)$ or $(d : T_2\sigma_2)$, and hence $(d : T\sigma_2)$.

Case: $T = \neg T'$

We have $\neg(d : T'\sigma_1)$. Since $P(T', \sigma_2, \sigma_1)$, by IH $(d : T'\sigma_2) \implies (d : T'\sigma_1)$.
Therefore, by contrapositive, we have $\neg(d : T'\sigma_2)$, hence $(d : \neg T'\sigma_2)$. □

10.23 LEMMA: If $\tau_1 \leq^? \tau_2 \sqsubseteq \tau_3$, then, for some τ_2', we have $\tau_1 \sqsubseteq \tau_2' \leq^? \tau_3$. □

Proof: By Lemma 10.21, since $\tau_2 \sqsubseteq \tau_3$, there exist T_2 and $\sigma : \mathsf{fvar}(T_2) \to \mathsf{GType}$ such that $T_2^\dagger = \tau_2$, that $T_2\sigma = \tau_3$, and that $\mathsf{fvar}^{\mathsf{cov}}(T_2) \cap \mathsf{fvar}^{\mathsf{cnt}}(T_2) = \emptyset$. Assume that $\mathsf{fvar}^{\mathsf{cov}} = \{X_1, \ldots, X_n\}$ and $\mathsf{fvar}^{\mathsf{cnt}} = \{Y_1, \ldots, Y_m\}$.

Let $\bar\sigma = [(X_i\sigma)^{\oslash}/X_i]_{i=1}^n \cup [(Y_i\sigma)^{\otimes}/Y_i]_{i=1}^n$. We have $(T_2\bar\sigma)^\dagger = T_2\sigma = \tau_3$.
Let $\hat\sigma = [\bigwedge_{j=1}^n X_j\bar\sigma/X_i]_{i=1}^n \cup [\bigvee_{j=1}^m Y_j\bar\sigma/Y_i]_{i=1}^m$.
Let $\check\sigma = [\bigwedge_{j=1}^n X_j\bar\sigma/X^1, \bigvee_{j=1}^m Y_j\bar\sigma/X^0]$.
We have:

$$\forall i = 1, \ldots, n.\ X_i\hat\sigma \leq X_i\bar\sigma \qquad \forall i = 1, \ldots, m.\ Y_i\bar\sigma \leq Y_i\sigma$$

We take $\tau_2' = (\tau_1^{\oslash}\check\sigma)^\dagger$. We must show:

$$\tau_1 \sqsubseteq (\tau_1^{\oslash}\check\sigma)^\dagger \qquad (\tau_1^{\oslash}\check\sigma)^\dagger \leq^? \tau_3$$

The former holds because $(\tau_1^{\oslash}\check\sigma)^\dagger = \tau_1^{\oslash}[\bigwedge_{j=1}^n X_j\sigma/X^1, \bigvee_{j=1}^m Y_j\sigma/X^0]$ and $\tau_1^{\oslash} \in \star(\tau_1)$.

To show the latter, we show:

$$(\tau_1^{\oslash}\check\sigma)^\dagger \leq^? (\tau_2^{\oslash}\check\sigma)^\dagger \qquad \tau_2^{\oslash}\check\sigma = T_2\hat\sigma \qquad (T_2\hat\sigma)^\dagger \leq^? (T_2\bar\sigma)^\dagger$$

We show $(\tau_1^{\oslash}\check\sigma)^\dagger \leq^? (\tau_2^{\oslash}\check\sigma)^\dagger$. By Proposition 10.19, $\tau_1 \leq^? \tau_2$ implies $\tau_1^{\oslash} \leq \tau_2^{\oslash}$. By Proposition 10.2, $\tau_1^{\oslash}\check\sigma \leq \tau_2^{\oslash}\check\sigma$. Both $\tau_1^{\oslash}\check\sigma$ and $\tau_2^{\oslash}\check\sigma$ are strongly polarized according to variance; therefore, $(\tau_1^{\oslash}\check\sigma)^{\dagger^{\oslash}} = \tau_1^{\oslash}\check\sigma$ and $(\tau_2^{\oslash}\check\sigma)^{\dagger^{\oslash}} = \tau_2^{\oslash}\check\sigma$. Hence, $(\tau_1^{\oslash}\check\sigma)^\dagger \leq^? (\tau_2^{\oslash}\check\sigma)^\dagger$.

To show $\tau_2^{\oslash}\check\sigma = T_2\hat\sigma$, just note that $\tau_2^{\oslash} = T_2([X^1/X_i]_{i=1}^n \cup [X^0/Y_i]_{i=1}^m)$.

Now we show $(T_2\hat\sigma)^\dagger \leq^? (T_2\bar\sigma)^\dagger$. First, note that $\hat\sigma|_{\mathsf{var}^{\mathsf{cov}}(T_2)} \leq \bar\sigma|_{\mathsf{var}^{\mathsf{cov}}(T_2)}$ and $\bar\sigma|_{\mathsf{var}^{\mathsf{cnt}}(T_2)} \leq \hat\sigma|_{\mathsf{var}^{\mathsf{cnt}}(T_2)}$. Hence, by Proposition 10.22, we have $T_2\hat\sigma \leq T_2\bar\sigma$. Since both $T_2\hat\sigma$ and $T_2\bar\sigma$ are strongly polarized according to variance, we have $T_2\hat\sigma = ((T_2\hat\sigma)^\dagger)^{\oslash}$ and $T_2\bar\sigma = ((T_2\bar\sigma)^\dagger)^{\oslash}$. This yields the result we need. □

10.25 PROPOSITION: Let τ be a gradual type and σ_1 and σ_2 two substitutions such that $\forall\alpha \in \mathsf{var}(\tau).\ \alpha\sigma_1 \simeq^? \alpha\sigma_2$. Then, $\tau\sigma_1 \simeq^? \tau\sigma_2$. □

Proof: Let $\mathsf{var}(\tau) = \{\alpha_1, \ldots, \alpha_n\}$. By Corollary 10.8, we find τ' such that $\mathsf{var}^+(\tau') \subseteq \{\alpha_1, \ldots, \alpha_n\}$ is disjoint from $\mathsf{var}^-(\tau') \subseteq \{\alpha_1', \ldots, \alpha_n'\}$ and that $\tau = \tau'[\alpha_i/\alpha_i']_{i=1}^n$.

Now, we define

$$\hat\sigma_1 = [(\alpha_i\sigma_1)^{\oplus}/\alpha_i]_{i=1}^n \cup [(\alpha_i\sigma_1)^{\ominus}/\alpha_i']_{i=1}^n$$
$$\hat\sigma_2 = [(\alpha_i\sigma_2)^{\oplus}/\alpha_i]_{i=1}^n \cup [(\alpha_i\sigma_2)^{\ominus}/\alpha_i']_{i=1}^n .$$

Let $T = \tau'^{\oplus}$.

We show that, for every A, $A\hat\sigma_1 \simeq A\hat\sigma_2$. Note that, for every $i \in I$, we

have $\alpha_i\sigma_1 \simeq^? \alpha_i\sigma_2$ and therefore, by Proposition 10.19, $(\alpha_i\sigma_1)^\oplus \simeq (\alpha_i\sigma_2)^\oplus$ and $(\alpha_i\sigma_1)^\ominus \simeq (\alpha_i\sigma_2)^\ominus$. If $A \notin \{\, \alpha_i \mid i \in I \,\} \cup \{\, \alpha_i \mid i \in I \,\}$, then $A\hat{\sigma}_1 = A = A\hat{\sigma}_2$. If $A = \alpha_i$ for some $i \in I$, then $A\hat{\sigma}_1 = (\alpha_i\sigma_1)^\oplus \simeq (\alpha_i\sigma_2)^\oplus = A\hat{\sigma}_2$. If $A = \alpha_i'$ for some $i \in I$, then $A\hat{\sigma}_1 = (\alpha_i\sigma_1)^\ominus \simeq (\alpha_i\sigma_2)^\ominus = A\hat{\sigma}_2$.

Since, for every A, $A\hat{\sigma}_1 \simeq A\hat{\sigma}_2$, we have $\hat{\sigma}_1|_{\mathsf{var}^{\mathsf{cov}}(T)} \leq \hat{\sigma}_2|_{\mathsf{var}^{\mathsf{cov}}(T)}$, $\hat{\sigma}_2|_{\mathsf{var}^{\mathsf{cnt}}(T)} \leq \hat{\sigma}_1|_{\mathsf{var}^{\mathsf{cnt}}(T)}$, $\hat{\sigma}_2|_{\mathsf{var}^{\mathsf{cov}}(T)} \leq \hat{\sigma}_1|_{\mathsf{var}^{\mathsf{cov}}(T)}$, and $\hat{\sigma}_1|_{\mathsf{var}^{\mathsf{cnt}}(T)} \leq \hat{\sigma}_2|_{\mathsf{var}^{\mathsf{cnt}}(T)}$. By Proposition 10.22, we have $T\hat{\sigma}_1 \simeq T\hat{\sigma}_2$.

We have:

$$T\hat{\sigma}_1 = \tau'^{\oplus}\hat{\sigma}_1 = (\tau\sigma_1)^\oplus \qquad T\hat{\sigma}_2 = \tau'^{\oplus}\hat{\sigma}_2 = (\tau\sigma_2)^\oplus$$

Therefore, we have $(\tau\sigma_1)^\oplus \simeq (\tau\sigma_2)^\oplus$. Hence, $\tau\sigma_1 \simeq^? \tau\sigma_2$. □

10.26 PROPOSITION (Soundness of $\mathsf{tally}^{\doteq}$):

$$\forall \sigma \in \mathsf{tally}^{\doteq}_{\Delta}\Big(\overline{t^1 \mathrel{\dot{\leq}} t^2} \cup \overline{T \doteq \alpha}\Big).\ \begin{cases} \forall (t^1 \mathrel{\dot{\leq}} t^2) \in \overline{t^1 \mathrel{\dot{\leq}} t^2}.\ t^1\sigma \leq t^2\sigma \\ \forall (T \doteq \alpha) \in \overline{T \doteq \alpha}.\ T\sigma = \alpha\sigma \\ \mathsf{dom}(\sigma) \subseteq \mathsf{var}\big(\overline{t^1 \mathrel{\dot{\leq}} t^2} \cup \overline{T \doteq \alpha}\big) \setminus \Delta \end{cases}$$

□

Proof: Let $\sigma \in \mathsf{tally}^{\doteq}_{\Delta}\Big(\overline{t^1 \mathrel{\dot{\leq}} t^2} \cup \overline{T \doteq \alpha}\Big)$, with

$$\overline{t^1 \mathrel{\dot{\leq}} t^2} = \{\, (t_i^1 \mathrel{\dot{\leq}} t_i^2) \mid i \in I \,\} \qquad \overline{T \doteq \alpha} = \{\, (T_j \doteq \alpha_j) \mid j \in J \,\}$$

By definition of $\mathsf{tally}^{\doteq}$, we have:

$$\sigma_0 \in \mathsf{tally}_{\Delta}\Big(\Big\{\, (t_i^1[T_j/\alpha_j]_{j\in J} \mathrel{\dot{\leq}} t_i^2[T_j/\alpha_j]_{j\in J}) \,\Big|\, i \in I \,\Big\}\Big) \quad \sigma = \sigma_0 \cup [T_j\sigma_0/\alpha_j]_{j\in J}$$

Let $i \in I$. We must show $t_i^1\sigma \leq t_i^2\sigma$.

By the properties of tallying, $t_i^1[T_j/\alpha_j]_{j\in J}\sigma_0 \leq t_i^2[T_j/\alpha_j]_{j\in J}\sigma_0$. We have

$$t_i^1[T_j/\alpha_j]_{j\in J}\sigma_0 = t_i^1\sigma \qquad t_i^2[T_j/\alpha_j]_{j\in J}\sigma_0 = t_i^2\sigma$$

and therefore $t_i^1\sigma \leq t_i^2\sigma$.

Let $j \in J$. We must show $T_j\sigma = \alpha_j\sigma$. We have $\alpha_j\sigma = T_j\sigma_0$. We also have $T_j\sigma = T_j\sigma_0$ because $\mathsf{var}(T_j) \cap \{\, \alpha_j \mid j \in J \,\} = \varnothing$ (this is checked in step (1) of the algorithm).

Finally, by the properties of tallying,

$$\mathsf{dom}(\sigma_0) \subseteq \mathsf{var}\Big(\Big\{\, (t_i^1[T_j/\alpha_j]_{j\in J} \mathrel{\dot{\leq}} t_i^2[T_j/\alpha_j]_{j\in J}) \,\Big|\, i \in I \,\Big\}\Big) \setminus \Delta$$

and, as a consequence,

$$\mathsf{dom}(\sigma) \subseteq \mathsf{dom}(\sigma_0) \cup \{\, \alpha_j \mid j \in J \,\} \subseteq \mathsf{var}\big(\overline{t^1 \mathrel{\dot{\leq}} t^2} \cup \overline{T \doteq \alpha}\big) \setminus \Delta\,. \qquad \square$$

10.27 PROPOSITION: If $\sigma \in \mathsf{solve}_{\Delta}(D)$, then $\sigma \Vdash_{\Delta} D$ and $\mathsf{dom}(\sigma) \subseteq \mathsf{var}(D)$. □

Proof: Let

$$D = \{ (t_i^1 \mathrel{\dot{\leq}} t_i^2) \mid i \in I \} \cup \{ (\tau_j \mathrel{\dot{\sqsubseteq}} \alpha_j) \mid j \in J \} \cup \{ (\alpha_k \mathrel{\dot{\sqsubseteq}} \alpha_k) \mid k \in K \}$$

(where we assume, for all $j \in J$, that $\tau_j \neq \alpha_j$).

Let $\sigma \in \mathsf{solve}_\Delta(D)$. Then, by definition of solve, we have the following:

$$\sigma = (\sigma_0' \circ \sigma_0)^\dagger|_{\mathsf{TVar}} \qquad \sigma_0 \in \mathsf{tally}^{\doteq}_\Delta(\{ (t_i^1 \mathrel{\dot{\leq}} t_i^2) \mid i \in I \} \cup \overline{T \doteq \alpha})$$

$$\sigma_0' = [\vec{\alpha}'/\vec{X}] \cup [\vec{X}/\vec{\alpha}]$$

$$\overline{T \doteq \alpha} = \{ (T_j \doteq \alpha_j) \mid j \in J \} \qquad \forall j \in J.\ T_j^\dagger = \tau_j$$

$$\overline{A} = \mathsf{var}_{\sqsubseteq}(D)\sigma_0 \cup \textstyle\bigcup_{i \in I}(\mathsf{var}^{\pm}(t_i^1\sigma_0) \cup \mathsf{var}^{\pm}(t_i^2\sigma_0))$$

$$\vec{X} = \mathsf{FVar} \cap \overline{A} \qquad \vec{\alpha} = \mathsf{var}(D) \setminus (\Delta \cup \mathsf{dom}(\sigma_0) \cup \overline{A}) \qquad \vec{\alpha}' \text{ and } \vec{X} \text{ fresh}$$

We must show the following results:

$$\forall i \in I.\ t_i^1\sigma \leq^? t_i^2\sigma \qquad \forall j \in J.\ \tau_j\sigma \sqsubseteq \alpha_j\sigma$$

$$\mathsf{static}(\sigma, \textstyle\bigcup_{j \in J} \mathsf{var}(\tau_j) \cup \{ \alpha_j \mid j \in J \}) \qquad \mathsf{dom}(\sigma) \subseteq \mathsf{var}(D) \setminus \Delta$$

To show $\forall i \in I. t_i^1\sigma \leq^? t_i^2\sigma$, consider an arbitrary $i \in I$. By Proposition 10.26, we have $t_i^1\sigma_0 \leq t_i^2\sigma_0$. Then, by Proposition 10.2, we have $t_i^1\sigma_0\sigma_0' \leq t_i^2\sigma_0\sigma_0'$. We show that $t_i^1\sigma_0\sigma_0'$ and $t_i^2\sigma_0\sigma_0'$ are polarized, which implies that $(t_i^1\sigma_0\sigma_0')^\dagger \leq^? (t_i^2\sigma_0\sigma_0')^\dagger$ since every polarized type frame T is such that $T \in \star^{\mathsf{pol}}(T^\dagger)$. Consider an arbitrary $j \in \{1, 2\}$: we must show $\mathsf{fvar}^+(t_i^j\sigma_0\sigma_0') \cap \mathsf{fvar}^-(t_i^j\sigma_0\sigma_0') = \varnothing$. By contradiction, assume $X \in \mathsf{fvar}^+(t_i^j\sigma_0\sigma_0') \cap \mathsf{fvar}^-(t_i^j\sigma_0\sigma_0')$. Since the variables in $\vec{\alpha}'$ and $\vec{X}'$ are all distinct, σ_0' does not map different variables to the same variable. Moreover, note that $\mathsf{var}(\sigma_0') \not\sharp \mathsf{var}(t_i^j)$. Therefore, there are two cases:

- $X \in \mathsf{fvar}^+(t_i^j\sigma_0) \cap \mathsf{fvar}^-(t_i^j\sigma_0)$ and $X \notin \mathsf{dom}(\sigma_0')$;
- there exists an $A \in \mathsf{var}^+(t_i^j\sigma_0) \cap \mathsf{var}^-(t_i^j\sigma_0)$ such that $A\sigma_0' = X$.

In the first case, the first condition implies $X \in \overline{A}$: but then $X \notin \mathsf{dom}(\sigma_0')$ is impossible. In the second case, we would have $A \in \overline{A}$: therefore, $A\sigma_0' = X$ is impossible. Finally, $(t_i^1\sigma_0\sigma_0')^\dagger \leq^? (t_i^2\sigma_0\sigma_0')^\dagger$ implies $t_i^1\sigma \mathrel{\dot{\leq}} t_i^2\sigma$ because $\mathsf{var}(t_i^1) \cup \mathsf{var}(t_i^2) \subseteq \mathsf{TVar}$.

To show $\forall j \in J. \tau_j\sigma \sqsubseteq \alpha_j\sigma$, consider an arbitrary $j \in J$. By Proposition 10.26, we have $T_j\sigma_0 = \alpha_j\sigma_0$. Moreover,

$$\tau_j\sigma = (\tau_j\sigma_0\sigma_0')^\dagger = (T_j^\dagger\sigma_0\sigma_0')^\dagger \qquad \alpha_j\sigma = (\alpha_j\sigma_0\sigma_0')^\dagger = (T_j\sigma_0\sigma_0')^\dagger$$

We have $\sigma_j\sigma \sqsubseteq \alpha_j\sigma$ because, for every $\alpha \in \mathsf{tvar}(T_j)$, $(\alpha^\dagger\sigma_0\sigma_0')^\dagger = (\alpha\sigma_0\sigma_0')^\dagger$.

To show $\mathsf{dom}(\sigma) \subseteq \mathsf{var}(D) \setminus \Delta$, consider $\alpha \notin \mathsf{var}(D) \setminus \Delta$: we show $\alpha\sigma = \alpha$. (Note that, trivially, $X\sigma = X$ for every X.) By Proposition 10.26, we have

$$\mathsf{dom}(\sigma_0) \subseteq \mathsf{var}\big(\{ (t_i^1 \mathrel{\dot{\leq}} t_i^2) \mid i \in I \} \cup \overline{T \doteq \alpha}\big) \setminus \Delta$$

Since $\mathsf{tvar}\big(\{ (t_i^1 \mathrel{\dot{\leq}} t_i^2) \mid i \in I \} \cup \overline{T \doteq \alpha}\big) \subseteq \mathsf{var}(D)$, we have $\alpha\sigma_0 = \alpha$. Then, $\alpha\sigma_0' = \alpha$ since $\mathsf{dom}(\sigma_0') \cap \mathsf{TVar} \subseteq \mathsf{var}(D)$.

Finally, to show $\mathsf{static}(\sigma, \bigcup_{j\in J} \mathsf{var}(\tau_j) \cup \{\, \alpha_j \mid j \in J \,\})$, consider an arbitrary $\alpha \in \bigcup_{j\in J} \mathsf{var}(\tau_j) \cup \{\, \alpha_j \mid j \in J \,\}$: we show that $\alpha\sigma$ is static, that is, that $\mathsf{fvar}(\alpha\sigma_0\sigma_0') = \varnothing$. Note that $\alpha \in \mathsf{var}_{\dot{\sqsubseteq}}(D)$. We have $\mathsf{var}(\alpha\sigma_0) \subseteq \mathsf{var}_{\dot{\sqsubseteq}}(D)\sigma_0$ and $\mathsf{var}(\alpha\sigma_0\sigma_0') = \bigcup_{A\in\mathsf{var}(\alpha\sigma_0)} \mathsf{var}(A\sigma_0')$. Therefore, if there existed $X \in \mathsf{var}(\alpha\sigma_0\sigma_0')$, there should exist $A \in \mathsf{var}(\alpha\sigma_0)$ such that $X \in \mathsf{var}(A\sigma_0')$. By definition of σ_0', we would need $A \in \vec{\alpha}$ or $A \in \mathsf{FVar} \setminus \mathsf{dom}(\sigma_0')$: but $\vec{\alpha}$ is disjoint from $\mathsf{var}_{\dot{\sqsubseteq}}(D)\sigma_0$, and $\mathsf{FVar} \cap \mathsf{var}_{\dot{\sqsubseteq}}(D)\sigma_0 \subseteq \mathsf{dom}(\sigma_0')$. □

10.28 LEMMA (Stability of typing under type substitution): If $\Gamma \vdash e \rightsquigarrow E : \tau$, then, for every static type substitution σ, we have $\Gamma\sigma \vdash e\sigma \rightsquigarrow E\sigma : \tau\sigma$. □

Proof: By induction on the derivation of $\Gamma \vdash e \rightsquigarrow E : \tau$ and by case analysis on the last rule applied.

Case: $[\mathrm{T}_x]$
We have $\Gamma \vdash x \rightsquigarrow x\,[\vec{t}] : \tau[\vec{t}/\vec{\alpha}]$, with $\Gamma(x) = \forall\vec{\alpha}.\,\tau$.
Since, by α-renaming, $\vec{\alpha} \sharp \sigma$, we have $(\Gamma\sigma)(x) = \forall\vec{\alpha}.\,\tau\sigma$.
By $[\mathrm{T}_x]$, since the $\vec{t}\sigma$ are all static, we have Ⓐ $\Gamma\sigma \vdash x \rightsquigarrow x[\vec{t}\sigma] : \tau\sigma[\vec{t}\sigma/\vec{\alpha}]$.
Since $\vec{\alpha} \sharp \sigma$, we have $\forall\alpha \in \mathsf{var}(\tau).\ \alpha\sigma[\vec{t}\sigma/\vec{\alpha}] = \alpha[\vec{t}/\vec{\alpha}]\sigma$ and therefore we have Ⓑ $\tau\sigma[\vec{t}\sigma/\vec{\alpha}] = \tau[\vec{t}/\vec{\alpha}]\sigma$.
From Ⓐ and Ⓑ, we have $\Gamma\sigma \vdash x \rightsquigarrow x\,[\vec{t}]\sigma : \tau[\vec{t}/\vec{\alpha}]\sigma$.

Case: $[\mathrm{T}_c]$
Straightforward, since $b_c\sigma = b_c$.

Case: $[\mathrm{T}_\lambda]$, $[\mathrm{T}_{\lambda:}]$, $[\mathrm{T}_{\mathrm{app}}]$, $[\mathrm{T}_{\mathrm{pair}}]$, $[\mathrm{T}_{\mathrm{proj}}]$
Direct application of the IH. For $[\mathrm{T}_\lambda]$, note that $t\sigma$ is always static.

Case: $[\mathrm{T}_\leq]$
By Proposition 10.20, $\tau' \leq^? \tau$ implies $\tau'\sigma \leq^? \tau\sigma$ because σ is static.

Case: $[\mathrm{T}_\sqsubseteq]$
By Proposition 10.3, $\tau' \sqsubseteq \tau$ implies $\tau'\sigma \sqsubseteq \tau\sigma$.

Case: $[\mathrm{T}_{\mathsf{let}}]$
We have $\Gamma \vdash (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2) \rightsquigarrow (\mathsf{let}\ x = \Lambda\vec{\alpha},\vec{\beta}.\,E_1\ \mathsf{in}\ E_2) : \tau$.
By inversion of $[\mathrm{T}_{\mathsf{let}}]$:

$$\text{Ⓐ}\ \ \Gamma \vdash e_1 \rightsquigarrow E_1 : \tau_1 \qquad \text{Ⓑ}\ \ \Gamma, x : \forall\vec{\alpha},\vec{\beta}.\,\tau_1 \vdash e_2 \rightsquigarrow E_2 : \tau$$
$$\text{Ⓒ}\ \ \vec{\alpha},\vec{\beta} \sharp \Gamma \text{ and } \vec{\beta} \sharp e_1$$

Let $\vec{\alpha}_1$ and $\vec{\beta}_1$ be vectors of distinct variables chosen outside $\mathsf{var}(\Gamma)$, $\mathsf{var}(e_1)$, $\mathsf{dom}(\sigma)$, and $\mathsf{var}(\sigma)$. Let $\rho = [\vec{\alpha}_1/\vec{\alpha}] \cup [\vec{\beta}_1/\vec{\beta}]$.
By IH from Ⓐ, since ρ is static, we have $\Gamma\rho \vdash e_1\rho \rightsquigarrow E_1\rho : \tau_1\rho$.
By Ⓒ, we have Ⓓ $\Gamma \vdash e_1[\vec{\alpha}_1/\vec{\alpha}] \rightsquigarrow E_1\rho : \tau_1\rho$.
By IH from Ⓓ, we have Ⓔ $\Gamma\sigma \vdash e_1[\vec{\alpha}_1/\vec{\alpha}]\sigma \rightsquigarrow E_1\rho\sigma : \tau_1\rho\sigma$.
By IH from Ⓑ, we have Ⓕ $\Gamma\sigma, x : (\forall\vec{\alpha},\vec{\beta}.\,\tau_1)\sigma \vdash e_2\sigma \rightsquigarrow E_2\sigma : \tau\sigma$.
By α-renaming from Ⓕ, Ⓖ $\Gamma\sigma, x : (\forall\vec{\alpha}_1,\vec{\beta}_1.\,\tau_1\rho)\sigma \vdash e_2\sigma \rightsquigarrow E_2\sigma : \tau\sigma$.

From Ⓖ, since $\vec{\alpha}_1,\vec{\beta}_1 \sharp \sigma$, Ⓗ $\Gamma\sigma, x\colon (\forall\vec{\alpha}_1,\vec{\beta}_1.\, \tau_1\rho\sigma) \vdash e_2\sigma \rightsquigarrow E_2\sigma\colon \tau\sigma$.
By $[\mathrm{T}_{\mathsf{let}}]$ from Ⓔ and Ⓗ we have

$$\Gamma\sigma \vdash (\mathsf{let}\vec{\alpha}_1 x = e_1[\vec{\alpha}_1/\vec{\alpha}]\sigma \;\mathsf{in}\; e_2\sigma) \rightsquigarrow (\mathsf{let} x = \Lambda\vec{\alpha}_1,\vec{\beta}_1.\, E_1\rho\sigma \;\mathsf{in}\; E_2\sigma)\colon \tau\sigma.$$

This concludes the proof because $\mathsf{let}\,\vec{\alpha}_1\, x = e_1[\vec{\alpha}_1/\vec{\alpha}]\sigma \;\mathsf{in}\; e_2\sigma$ and $(\mathsf{let}\,\vec{\alpha}\, x = e_1 \;\mathsf{in}\; e_2)\sigma$ are equivalent by α-renaming, as are $\mathsf{let} x = \Lambda\vec{\alpha}_1,\vec{\beta}_1.\, E_1\rho\sigma \;\mathsf{in}\; E_2\sigma$ and $(\mathsf{let}\, x = \Lambda\vec{\alpha},\vec{\beta}.\, E_1 \;\mathsf{in}\; E_2)\sigma$. □

10.29 LEMMA (Weakening): Let Γ_1 and Γ_2 be two type environments such that $\Gamma_1 \leq^? \Gamma_2$. If $\Gamma_2 \vdash e \rightsquigarrow E\colon \tau$, then $\Gamma_1 \vdash e \rightsquigarrow E\colon \tau$. □

Proof: By induction on the derivation of $\Gamma_2 \vdash e \rightsquigarrow E\colon \tau$ and by case analysis on the last rule applied.

Case: $[\mathrm{T}_x]$
We have $\Gamma_2 \vdash x \rightsquigarrow x\,[\vec{t}]\colon \tau[\vec{t}/\vec{\alpha}]$, where $\Gamma_2(x) = \forall\vec{\alpha}.\,\tau$. By definition of $\Gamma_1 \leq^? \Gamma_2$, we have $\Gamma_1(x) \leq^? \Gamma_2(x)$, therefore $\Gamma_1(x) = \forall\vec{\alpha}.\,\tau'$ and $\tau' \leq^? \tau$. By $[\mathrm{T}_x]$ we derive $\Gamma_1 \vdash x \rightsquigarrow x\,[\vec{t}]\colon \tau'[\vec{t}/\vec{\alpha}]$; then by $[\mathrm{T}_\leq]$ we derive $\Gamma_1 \vdash x \rightsquigarrow x\,[\vec{t}]\colon \tau[\vec{t}/\vec{\alpha}]$ since $\tau'[\vec{t}/\vec{\alpha}] \leq^? \tau[\vec{t}/\vec{\alpha}]$ (by Proposition 10.20, subtyping is preserved by static type substitutions).

Case: $[\mathrm{T}_c]$ Straightforward.

Case: $[\mathrm{T}_\lambda]$, $[\mathrm{T}_{\lambda:}]$, $[\mathrm{T}_{\mathrm{app}}]$, $[\mathrm{T}_{\mathrm{pair}}]$, $[\mathrm{T}_{\mathrm{proj}}]$, $[\mathrm{T}_\leq]$, $[\mathrm{T}_\sqsubseteq]$
We conclude by direct application of the induction hypothesis. For $[\mathrm{T}_\lambda]$ and $[\mathrm{T}_{\lambda:}]$, note that $\Gamma_1 \leq^? \Gamma_2$ implies $(\Gamma_1, x\colon \tau) \leq^? (\Gamma_2, x\colon \tau)$ for every τ.

Case: $[\mathrm{T}_{\mathsf{let}}]$
We have derived $\Gamma_2 \vdash (\mathsf{let}\,\vec{\alpha}\, x = e_1 \;\mathsf{in}\; e_2) \rightsquigarrow (\mathsf{let}\, x = \Lambda\vec{\alpha},\vec{\beta}.\, E_1 \;\mathsf{in}\; E_2)\colon \tau$ from the premises

$$\Gamma_2 \vdash e_1 \rightsquigarrow E_1\colon \tau_1 \qquad \Gamma_2, x\colon \forall\vec{\alpha},\vec{\beta}.\,\tau_1 \vdash e_2 \rightsquigarrow E_2\colon \tau$$
$$\vec{\alpha},\vec{\beta} \sharp \Gamma_2 \text{ and } \vec{\beta} \sharp e_1\ .$$

Let $\vec{\alpha}_1$ and $\vec{\beta}_1$ be vectors of variables chosen outside $\mathsf{var}(\Gamma_1)$ and $\mathsf{var}(e_1)$. Let $\rho = [\vec{\alpha}_1/\vec{\alpha}] \cup [\vec{\beta}_1/\vec{\beta}]$. Since ρ is a static type substitution, we can apply Lemma 10.28 to derive $\Gamma_2\rho \vdash e_1\rho \rightsquigarrow E_1\rho\colon \tau_1\rho$, which is $\Gamma_2 \vdash e_1\rho \rightsquigarrow E_1\rho\colon \tau_1\rho$ because the $\vec{\alpha}$ and $\vec{\beta}$ variables do not occur in Γ_2.
By induction, we derive $\Gamma_1 \vdash e_1\rho \rightsquigarrow E_1\rho\colon \tau_1\rho$ and $\Gamma_1, x\colon \forall\vec{\alpha},\vec{\beta}.\,\tau_1 \vdash e_2 \rightsquigarrow E_2\colon \tau$. By α-renaming, $\forall\vec{\alpha},\vec{\beta}.\,\tau_1$ is equivalent to $\forall\vec{\alpha}_1,\vec{\beta}_1.\,\tau_1\rho$. Note that the $\vec{\beta}_1$ variables do not occur in $e_1\rho$, because they do not occur in e_1 and they are introduced by ρ only on variables which themselves do not occur in e_1. Therefore, we have

$$\Gamma_1 \vdash e_1\rho \rightsquigarrow E_1\rho\colon \tau_1\rho \qquad \Gamma_1, x\colon \forall\vec{\alpha}_1,\vec{\beta}_1.\,\tau_1\rho \vdash e_2 \rightsquigarrow E_2\colon \tau$$
$$\vec{\alpha}_1,\vec{\beta}_1 \sharp \Gamma_1 \text{ and } \vec{\beta}_1 \sharp e_1\rho$$

from which we derive $\Gamma_1 \vdash (\mathsf{let}\ \vec{\alpha}_1\ x = e_1\rho\ \mathsf{in}\ e_2) \rightsquigarrow (\mathsf{let}\ x = \Lambda\vec{\alpha}_1,\vec{\beta}_1.\, E_1\rho\ \mathsf{in}\ E_2) \colon \tau$, which is the result we need since, by α-renaming, $\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2$ and $\mathsf{let}\ \vec{\alpha}_1\ x = e_1\rho\ \mathsf{in}\ e_2$ are equivalent, as are $(\mathsf{let}\ x = \Lambda\vec{\alpha},\vec{\beta}.\, E_1\ \mathsf{in}\ E_2)$ and $(\mathsf{let}\ x = \Lambda\vec{\alpha}_1,\vec{\beta}_1.\, E_1\rho\ \mathsf{in}\ E_2)$. □

10.33 LEMMA: If $\Gamma;\Delta \vdash C \rightsquigarrow D$, then $\mathsf{var}(\Gamma) \cap \mathsf{var}(D) \subseteq \mathsf{var}(C) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)$. □

Proof: By induction on C (the form of C determines the derivation).

Case: $C = (t_1 \mathrel{\dot{\lesssim}} t_2)$ or $C = (\tau \mathrel{\dot{\sqsubseteq}} \alpha)$ We have $\mathsf{var}(D) \subseteq \mathsf{var}(C)$.

Case: $C = (\tau \mathrel{\dot{\sqsubseteq}} \alpha)$ We have $\mathsf{var}(D) \subseteq \mathsf{var}_{\dot{\sqsubseteq}}(D) \cup \{\alpha\}$ and $\alpha \in \mathsf{var}(C)$.

Case: $C = (\mathsf{def}\ x \colon \tau\ \mathsf{in}\ C')$
By IH, $\mathsf{var}(\Gamma, x \colon \tau) \cap \mathsf{var}(D) \subseteq \mathsf{var}(C') \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)$. This directly yields the result since $\mathsf{var}(C') \subseteq \mathsf{var}(C)$.

Case: $C = (\exists\vec{\alpha}.\, C')$
By IH, $\mathsf{var}(\Gamma) \cap \mathsf{var}(D) \subseteq \mathsf{var}(C') \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)$. The side condition on the rule imposes $\vec{\alpha} \mathrel{\sharp} \Gamma$. Then, $\mathsf{var}(\Gamma) \cap \mathsf{var}(D) \subseteq \mathsf{var}(C) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)$ since $\mathsf{var}(C) = \mathsf{var}(C') \setminus \vec{\alpha}$.

Case: $C = (C_1 \wedge C_2)$
By IH, for both i, $\mathsf{var}(\Gamma) \cap \mathsf{var}(D_i) \subseteq \mathsf{var}(C_i) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D_i)$. This directly implies $\mathsf{var}(\Gamma) \cap \mathsf{var}(D_1 \cup D_2) \subseteq \mathsf{var}(C_1 \wedge C_2) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D_1 \cup D_2)$.

Case: $C = (\mathsf{let}\ x \colon \forall\vec{\alpha};\alpha[C_1]^{\vec{\alpha}_1}.\, \alpha\ \mathsf{in}\ C_2)$
By IH,

$$\mathsf{var}(\Gamma) \cap \mathsf{var}(D_1) \subseteq \mathsf{var}(C_1) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D_1)$$
$$\mathsf{var}(\Gamma, x \colon \forall\vec{\alpha},\vec{\beta}.\, \alpha\sigma_1) \cap \mathsf{var}(D_2) \subseteq \mathsf{var}(C_2) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D_2)$$

We have

$$\begin{aligned} D &= D_2 \cup \mathsf{equiv}(\sigma_1, D_1) \\ \mathsf{var}(D) &= \mathsf{var}(D_2) \cup \mathsf{var}(D_1)\sigma_1 \cup \mathsf{var}_{\dot{\sqsubseteq}}(D_1) \cup S \cup S\sigma_1 \\ \mathsf{var}_{\dot{\sqsubseteq}}(D) &= \mathsf{var}_{\dot{\sqsubseteq}}(D_2) \cup \mathsf{var}(D_1)\sigma_1 \cup \mathsf{var}_{\dot{\sqsubseteq}}(D_1) \\ \mathsf{var}(C) &= (\mathsf{var}(C_1) \setminus (\vec{\alpha} \cup \{\alpha\})) \cup \mathsf{var}(C_2) \end{aligned}$$

where $S = \{\, \alpha \in \mathsf{dom}(\sigma_1) \mid \alpha\sigma_1 \text{ static} \,\}$.
Consider an arbitrary $\beta \in \mathsf{var}(\Gamma) \cap \mathsf{var}(D)$.

Subcase: $\beta \in \mathsf{var}(D_2)$
Then $\beta \in \mathsf{var}(C_2) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D_2)$ and hence $\beta \in \mathsf{var}(C) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)$.

Subcase: $\beta \in \mathsf{var}(D_1)\sigma_1 \cup \mathsf{var}_{\dot{\sqsubseteq}}(D_1)$ Then $\beta \in \mathsf{var}_{\dot{\sqsubseteq}}(D)$.

Subcase: $\beta \in S$
Then $\beta \in \mathsf{dom}(\sigma_1)$. By Proposition 10.27, $\beta \in \mathsf{var}(D_1)$.

Since $\beta \in \mathsf{var}(\Gamma) \cap \mathsf{var}(D_1)$, we have $\beta \in \mathsf{var}(C_1) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D_1)$. Since $\beta \in \mathsf{var}(\Gamma)$, by the side conditions of the rule we know $\beta \neq \alpha$ and $\beta \notin \vec{\alpha}$. Therefore, $\beta \in \mathsf{var}(C) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)$.

Subcase: $\beta \in S\sigma_1$

Then $\beta \in \mathsf{var}(\gamma\sigma_1)$ for some $\gamma \in \mathsf{dom}(\sigma_1)$ such that $\gamma\sigma_1$ is static.

By Proposition 10.27, $\gamma \in \mathsf{var}(D_1)$. Then $\beta \in \mathsf{var}(D_1)\sigma_1 \subseteq \mathsf{var}_{\dot{\sqsubseteq}}(D)$. □

10.34 LEMMA:

$$\forall \Gamma, \Delta, e, \alpha, D, \sigma. \left.\begin{array}{l} \Gamma; \Delta \vdash \langle\!\langle e : \alpha \rangle\!\rangle \rightsquigarrow D \\ \sigma \in \mathsf{solve}_\Delta(D) \\ \mathsf{var}(e) \subseteq \Delta \\ \alpha \notin \mathsf{var}(\Gamma) \end{array}\right\} \implies \mathsf{static}(\sigma, \mathsf{var}(\Gamma))$$

□

Proof: Consider an arbitrary $\beta \in \mathsf{var}(\Gamma)$. We show that $\beta\sigma$ is static.

Case: $\beta \notin \mathsf{dom}(\sigma)$ Then $\beta\sigma = \beta$, which is static.

Case: $\beta \in \mathsf{dom}(\sigma)$

Then $\beta \in \mathsf{var}(D)$ (by Proposition 10.27), and therefore $\beta \in \mathsf{var}(\Gamma) \cap \mathsf{var}(D)$.

By Lemma 10.33, $\beta \in \mathsf{var}(\langle\!\langle e : \alpha \rangle\!\rangle) \cup \mathsf{var}_{\dot{\sqsubseteq}}(D)$.

Subcase: $\beta \in \mathsf{var}(\langle\!\langle e : \alpha \rangle\!\rangle)$

This case is impossible because $\mathsf{var}(\langle\!\langle e : \alpha \rangle\!\rangle) = \mathsf{var}(e) \cup \{\alpha\}$, $\mathsf{dom}(\sigma) \sharp \mathsf{var}(e)$ (because $\mathsf{var}(e) \subseteq \Delta$), and $\alpha \notin \mathsf{var}(\Gamma)$.

Subcase: $\beta \in \mathsf{var}_{\dot{\sqsubseteq}}(D)$ Since $\sigma \Vdash_\Delta D$, $\beta\sigma$ must be static. □

10.35 LEMMA:

$$\forall \Gamma, \Delta, D_1, \sigma_1, \rho, \sigma, \sigma'. \left.\begin{array}{l} \sigma \Vdash_\Delta \mathsf{equiv}(\sigma_1, D_1) \\ \mathsf{dom}(\rho) \sharp \Gamma\sigma_1 \\ \mathsf{static}(\sigma', \mathsf{var}(\mathsf{equiv}(\sigma_1, D_1))\sigma) \\ \mathsf{static}(\sigma_1, \mathsf{var}(\Gamma)) \end{array}\right\} \implies \Gamma\sigma\sigma' \leq^? \Gamma\sigma_1\rho\sigma\sigma'$$

□

Proof: Consider an arbitrary $x \in \mathsf{dom}(\Gamma)$. We have $\Gamma(x) = \forall\vec{\alpha}.\, \tau$. We assume by α-renaming that $\vec{\alpha} \sharp \sigma_1, \rho, \sigma, \sigma'$; then, $(\Gamma\sigma\sigma')(x) = \forall\vec{\alpha}.\, \tau\sigma\sigma'$ and $(\Gamma\sigma_1\rho\sigma\sigma')(x) = \forall\vec{\alpha}.\, \tau\sigma_1\rho\sigma\sigma'$. We must show $\tau\sigma\sigma' \leq^? \tau\sigma_1\rho\sigma\sigma'$. We show $\forall\alpha \in \mathsf{var}(\tau).\, \alpha\sigma\sigma' \simeq^? \alpha\sigma_1\rho\sigma\sigma'$, which implies $\tau\sigma\sigma' \simeq^? \tau\sigma_1\rho\sigma\sigma'$ by Proposition 10.25.

To show $\forall\alpha \in \mathsf{var}(\tau).\, \alpha\sigma\sigma' \simeq^? \alpha\sigma_1\rho\sigma\sigma'$, consider an arbitrary $\alpha \in \mathsf{var}(\tau)$.

Case: $\alpha \in \vec{\alpha}$

Then (by our choice of naming) $\alpha\sigma\sigma' = \alpha$ and $\alpha\sigma_1\rho\sigma\sigma' = \alpha$.

Case: $\alpha \notin \vec{\alpha}$

Then $\alpha \in \mathsf{var}(\Gamma)$ and hence: $\mathsf{var}(\alpha\sigma_1) \subseteq \mathsf{var}(\Gamma\sigma_1)$, and $\alpha\sigma_1\rho = \alpha\sigma_1$, and $\alpha\sigma_1$ is static.

Subcase: $\alpha \notin \mathsf{dom}(\sigma_1)$
Then $\alpha\sigma_1 = \alpha$, $\alpha\sigma_1\rho = \alpha$, and $\alpha\sigma_1\rho\sigma\sigma' = \alpha\sigma\sigma'$.

Subcase: $\alpha \in \mathsf{dom}(\sigma_1)$
Then $\{(\alpha \mathrel{\dot\leq} \alpha\sigma_1), (\alpha\sigma_1 \mathrel{\dot\leq} \alpha)\} \subseteq \mathsf{equiv}(\sigma_1, D_1)$. Therefore, we have $\alpha\sigma_1\sigma \simeq^? \alpha\sigma$ and $\mathsf{static}(\sigma', \mathsf{var}(\alpha\sigma) \cup \mathsf{var}(\alpha\sigma_1\sigma))$. By Proposition 10.20, $\alpha\sigma_1\sigma\sigma' \simeq^? \alpha\sigma\sigma'$. □

10.36 THEOREM (Soundness of type inference): Let $\mathcal{D}$ be a derivation of $\Gamma; \mathsf{var}(e) \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$. Let σ be a type substitution such that $\sigma \Vdash_{\mathsf{var}(e)} D$. Then, we have $\Gamma\sigma \vdash e \rightsquigarrow \{\!|e|\!\}^{\mathcal{D}}_{\sigma} : t\sigma$. □

Proof: We show the following, stronger result (for all $\mathcal{D}$, Γ, Δ, e, t, D, σ, and σ'):

$$\left.\begin{array}{l} \mathcal{D} \text{ is a derivation of } \Gamma; \Delta \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D \\ \sigma \Vdash_{\Delta} D \\ \mathsf{static}(\sigma', \mathsf{var}(D)\sigma) \\ \mathsf{var}(e) \subseteq \Delta \end{array}\right\} \implies \Gamma\sigma\sigma' \vdash e\sigma' \rightsquigarrow \{\!|e|\!\}^{\mathcal{D}}_{\sigma}\sigma' : t\sigma\sigma'$$

This result implies the statement: we take $\Delta = \mathsf{var}(e)$ and $\sigma' = [\,]$ (the identity substitution).

The proof is by structural induction on e.

Case: $e = x$
We have:

Ⓐ $\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle x : t \rangle\!\rangle \rightsquigarrow D$ Ⓑ $\sigma \Vdash_{\Delta} D$ Ⓒ $\mathsf{static}(\sigma', \mathsf{var}(D)\sigma)$

By Lemma 10.32 from Ⓐ:

$$\Gamma(x) = \forall\vec{\alpha}.\, \tau \qquad D = \{(\tau[\vec{\beta}/\vec{\alpha}] \mathrel{\dot\sqsubseteq} \alpha), (\alpha \mathrel{\dot\leq} t)\}$$

Assuming $\vec{\alpha} \sharp \sigma, \sigma'$ by α-renaming, we have $(\Gamma\sigma\sigma')(x) = \forall\vec{\alpha}.\, \tau\sigma\sigma'$.
From Ⓑ and Ⓒ, we know that the types $\vec{\beta}\sigma\sigma'$ are static.
Since $\vec{\alpha} \sharp \sigma, \sigma'$, we have $\forall\alpha \in \mathsf{var}(\tau).\, \alpha\sigma\sigma'[\vec{\beta}\sigma\sigma'/\vec{\alpha}] = \alpha[\vec{\beta}/\vec{\alpha}]\sigma\sigma'$. Therefore, $\tau\sigma\sigma'[\vec{\beta}\sigma\sigma'/\vec{\alpha}] = \tau[\vec{\beta}/\vec{\alpha}]\sigma\sigma'$.
By Lemma 10.31, we have $\tau[\vec{\beta}/\vec{\alpha}]\sigma\sigma' \sqsubseteq \alpha\sigma\sigma'$.
By Lemma 10.30, we have $\alpha\sigma\sigma' \leq^? t\sigma\sigma'$.
By $[\mathrm{T}_x]$, we have $\Gamma\sigma\sigma' \vdash x \rightsquigarrow x\,[\vec{\beta}\sigma\sigma'] : \tau\sigma\sigma'[\vec{\beta}\sigma\sigma'/\vec{\alpha}]$.
By $[\mathrm{T}_{\sqsubseteq}]$ and $[\mathrm{T}_{\leq}]$, $\Gamma\sigma\sigma' \vdash x \rightsquigarrow x\,[\vec{\beta}\sigma\sigma']\langle\tau[\vec{\beta}/\vec{\alpha}]\sigma\sigma' \overset{\ell}{\Rightarrow} \alpha\sigma\sigma'\rangle : t\sigma\sigma'$.
This concludes this case since $\{\!|x|\!\}^{\mathcal{D}}_{\sigma}\sigma' = x\,[\vec{\beta}\sigma\sigma']\langle\tau[\vec{\beta}/\vec{\alpha}]\sigma\sigma' \overset{\ell}{\Rightarrow} \alpha\sigma\sigma'\rangle$.

Case: $e = c$
We have $\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle c : t \rangle\!\rangle \rightsquigarrow D$.

By Lemma 10.32, $D = \{b_c \mathrel{\dot{\leq}} t\}$. By Lemma 10.30, $b_c\sigma\sigma' \leq^? t\sigma\sigma'$.
By $[\mathrm{T}_c]$ and $[\mathrm{T}_{\leq}]$, $\Gamma\sigma\sigma' \vdash c\sigma\sigma' \rightsquigarrow c : t\sigma\sigma'$. Note that $\{\!|c|\!\}^{\mathcal{D}}_{\sigma}\sigma' = c$.

Case: $e = \lambda x.\, e'$
We have $\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle \lambda x.\, e' : t \rangle\!\rangle \rightsquigarrow D$.
By Lemma 10.32:

$$(\Gamma, x : \alpha_1); \Delta \vdash \langle\!\langle e' : \alpha_2 \rangle\!\rangle \rightsquigarrow D' \qquad D = D' \cup \{(\alpha_1 \mathrel{\dot{\sqsubseteq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\}$$

We know that $\alpha_1\sigma\sigma'$ is static.
By Lemma 10.30, $(\alpha_1 \to \alpha_2)\sigma\sigma' \leq^? t\sigma\sigma'$.
By IH, $\Gamma\sigma\sigma', x : \alpha_1\sigma\sigma' \vdash e'\sigma\sigma' \rightsquigarrow \{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma' : \alpha_2\sigma\sigma'$.
By $[\mathrm{T}_\lambda]$, $\Gamma\sigma\sigma' \vdash (\lambda x.\, e'\sigma\sigma') \rightsquigarrow \lambda^{(\alpha_1\to\alpha_2)\sigma\sigma'} x.\, \{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma' : (\alpha_1 \to \alpha_2)\sigma\sigma'$.
By $[\mathrm{T}_{\leq}]$, $\Gamma\sigma\sigma' \vdash (\lambda x.\, e'\sigma\sigma') \rightsquigarrow \lambda^{(\alpha_1\to\alpha_2)\sigma\sigma'} x.\, \{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma' : t\sigma\sigma'$.
We have $\{\!|\lambda x.\, e|\!\}^{\mathcal{D}}_{\sigma}\sigma' = \lambda^{(\alpha_1\to\alpha_2)\sigma\sigma'} x.\, \{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma'$.

Case: $e = \lambda x : \tau.\, e'$
We have $\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle \lambda x : \tau.\, e' : t \rangle\!\rangle \rightsquigarrow D$.
By Lemma 10.32:

$$\mathcal{D}' :: (\Gamma, x : \tau); \Delta \vdash \langle\!\langle e' : \alpha_2 \rangle\!\rangle \rightsquigarrow D'$$
$$D = D' \cup \{(\tau \mathrel{\dot{\sqsubseteq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\}$$

By Lemma 10.31, $\tau\sigma\sigma' \sqsubseteq \alpha_1\sigma\sigma'$.
By Lemma 10.30, $(\alpha_1 \to \alpha_2)\sigma\sigma' \leq^? t\sigma\sigma'$.
By IH, $\Gamma\sigma\sigma', x : \tau\sigma\sigma' \vdash e'\sigma\sigma' \rightsquigarrow \{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma' : \alpha_2\sigma\sigma'$.
By $[\mathrm{T}_{\lambda:}]$, $\Gamma\sigma\sigma' \vdash (\lambda x : \tau.\, e')\sigma\sigma' \rightsquigarrow \lambda^{(\tau\to\alpha_2)\sigma\sigma'} x.\, \{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma' : (\tau \to \alpha_2)\sigma\sigma'$.
By $[\mathrm{T}_{\sqsubseteq}]$,

$$\Gamma\sigma\sigma' \vdash (\lambda x : \tau.\, e')\sigma\sigma' \rightsquigarrow$$
$$\left(\lambda^{(\tau\to\alpha_2)\sigma\sigma'} x.\, \{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma'\right)\langle(\tau \to \alpha_2)\sigma\sigma' \overset{\ell}{\Rightarrow} (\alpha_1 \to \alpha_2)\sigma\sigma'\rangle : (\alpha_1 \to \alpha_2)\sigma\sigma'.$$

By $[\mathrm{T}_{\leq}]$,

$$\Gamma\sigma\sigma' \vdash (\lambda x : \tau.\, e')\sigma\sigma' \rightsquigarrow$$
$$\left(\lambda^{(\tau\to\alpha_2)\sigma\sigma'} x.\, \{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma'\right)\langle(\tau \to \alpha_2)\sigma\sigma' \overset{\ell}{\Rightarrow} (\alpha_1 \to \alpha_2)\sigma\sigma'\rangle : t\sigma\sigma' .$$

We have

$$\{\!|\lambda x : \tau.\, e|\!\}^{\mathcal{D}}_{\sigma}\sigma' = \left(\lambda^{(\tau\to\alpha_2)\sigma\sigma'} x.\, \{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma'\right)\langle(\tau \to \alpha_2)\sigma\sigma' \overset{\ell}{\Rightarrow} (\alpha_1 \to \alpha_2)\sigma\sigma'\rangle .$$

Case: $e = e_1\ e_2$
We have $\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle e_1\ e_2 : t \rangle\!\rangle \rightsquigarrow D$.
By Lemma 10.32:

$$\mathcal{D}_1 :: \Gamma; \Delta \vdash \langle\!\langle e_1 : \alpha \to t \rangle\!\rangle \rightsquigarrow D_1 \qquad \mathcal{D}_2 :: \Gamma; \Delta \vdash \langle\!\langle e_2 : \alpha \rangle\!\rangle \rightsquigarrow D_2$$
$$D = D_1 \cup D_2$$

By IH:

$$\Gamma\sigma\sigma' \vdash e_1\sigma\sigma' \rightsquigarrow \{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma}\sigma' : (\alpha \to t)\sigma\sigma'$$
$$\Gamma\sigma\sigma' \vdash e_2\sigma\sigma' \rightsquigarrow \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}\sigma' : \alpha\sigma\sigma'$$

By $[\mathrm{T}_{\mathrm{app}}]$, $\Gamma\sigma\sigma' \vdash (e_1\ e_2)\sigma\sigma' \rightsquigarrow \{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma}\sigma'\ \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}\sigma' : t\sigma\sigma'$.
Note that $\{\!|e_1\ e_2|\!\}^{\mathcal{D}}_{\sigma}\sigma' = \{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma}\sigma'\ \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}\sigma'$.

Case: $e = (e_1, e_2)$
We have $\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle (e_1, e_2) : t \rangle\!\rangle \rightsquigarrow D$.
By Lemma 10.32:

$$\mathcal{D}_1 :: \Gamma; \Delta \vdash \langle\!\langle e_1 : \alpha_1 \rangle\!\rangle \rightsquigarrow D_1 \qquad \mathcal{D}_2 :: \Gamma; \Delta \vdash \langle\!\langle e_2 : \alpha_2 \rangle\!\rangle \rightsquigarrow D_2$$
$$D = D_1 \cup D_2 \cup \{\alpha_1 \times \alpha_2 \mathrel{\dot{\leq}} t\}$$

By Lemma 10.30, $(\alpha_1 \times \alpha_2)\sigma\sigma' \leq^{?} t\sigma\sigma'$.
By IH,

$$\Gamma\sigma\sigma' \vdash e_1\sigma\sigma' \rightsquigarrow \{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma}\sigma' : \alpha_1\sigma\sigma'$$
$$\Gamma\sigma\sigma' \vdash e_2\sigma\sigma' \rightsquigarrow \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}\sigma' : \alpha_2\sigma\sigma'$$

By $[\mathrm{T}_{\mathrm{pair}}]$ and $[\mathrm{T}_{\leq}]$, $\Gamma\sigma\sigma' \vdash (e_1, e_2)\sigma\sigma' \rightsquigarrow \left(\{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma}\sigma', \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}\sigma'\right) : t\sigma\sigma'$.
We have $\{\!|(e_1, e_2)|\!\}^{\mathcal{D}}_{\sigma}\sigma' = \left(\{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma}\sigma', \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}\sigma'\right)$.

Case: $e = \pi_i\ e'$
We have $\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle \pi_i\ e' : t \rangle\!\rangle \rightsquigarrow D$.
By Lemma 10.32:

$$\mathcal{D}' :: \Gamma; \Delta \vdash \langle\!\langle e' : \alpha_1 \times \alpha_2 \rangle\!\rangle \rightsquigarrow D' \qquad D = D' \cup \{\alpha_i \mathrel{\dot{\leq}} t\}$$

By Lemma 10.30, $\alpha_i\sigma\sigma' \leq^{?} t\sigma\sigma'$.
By IH, $\Gamma\sigma\sigma' \vdash e'\sigma\sigma' \rightsquigarrow \{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma' : (\alpha_1 \times \alpha_2)\sigma\sigma'$.
By $[\mathrm{T}_{\mathrm{proj}}]$ and $[\mathrm{T}_{\leq}]$, $\Gamma\sigma\sigma' \vdash (\pi_i\ e')\sigma\sigma' \rightsquigarrow \pi_i\ (\{\!|e'|\!\}^{\mathcal{D}'}_{\sigma}\sigma') : t\sigma\sigma'$.
We have $\{\!|\pi_i\ e'|\!\}^{\mathcal{D}}_{\sigma}\sigma' = (\pi_i\ \{\!|e'|\!\}^{\mathcal{D}'}_{\sigma})\sigma'$.

Case: $e = (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2)$
We have $\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle \mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2 : t \rangle\!\rangle \rightsquigarrow D$.
By Lemma 10.32:

$$\mathcal{D}_1 :: \Gamma; \Delta \cup \vec{\alpha} \vdash \langle\!\langle e_1 : \alpha \rangle\!\rangle \rightsquigarrow D_1$$
$$\mathcal{D}_2 :: (\Gamma, x : \forall\vec{\alpha},\vec{\beta}.\ \alpha\sigma_1); \Delta \vdash \langle\!\langle e_2 : t \rangle\!\rangle \rightsquigarrow D_2$$
$$D = D_2 \cup \mathsf{equiv}(\sigma_1, D_1) \qquad \sigma_1 \in \mathsf{solve}_{\Delta\cup\vec{\alpha}}(D_1) \qquad \vec{\alpha} \sharp \mathsf{var}(\Gamma\sigma_1)$$
$$\vec{\beta} = \mathsf{var}(\alpha\sigma_1) \setminus (\mathsf{var}(\Gamma\sigma_1) \cup \vec{\alpha} \cup \mathsf{var}(e_1))$$

Let $\vec{\alpha}_1$ and $\vec{\beta}_1$ be vectors of distinct variables chosen outside $\mathsf{var}(e_1)$, $\mathsf{dom}(\sigma)$, $\mathsf{var}(\sigma)$, $\mathsf{dom}(\sigma')$, and $\mathsf{var}(\sigma')$. Let $\rho = [\vec{\alpha}_1/\vec{\alpha}] \cup [\vec{\beta}_1/\vec{\beta}]$.
Since $\vec{\beta} \sharp e_1$ and $\vec{\alpha}_1 \sharp \sigma'$, we have $e\sigma' = (\mathsf{let}\ \vec{\alpha}_1\ x = e_1\rho\sigma'\ \mathsf{in}\ e_2\sigma')$.
We have $\{\!|e|\!\}^{\mathcal{D}}_{\sigma} = \left(\mathsf{let}\ x = (\Lambda\vec{\alpha}_1,\vec{\beta}_1.\ \{\!|e_1|\!\}^{\mathcal{D}_1}_{\sigma_1}\rho\sigma)\ \mathsf{in}\ \{\!|e_2|\!\}^{\mathcal{D}_2}_{\sigma}\right)$.

Since $\vec{\alpha}_1,\vec{\beta}_1 \sharp \sigma'$, we have $\llbracket e \rrbracket^{\mathcal{D}}_{\sigma}\sigma' = \big(\mathsf{let}\ x = (\Lambda\vec{\alpha}_1,\vec{\beta}_1.\ \llbracket e_1 \rrbracket^{\mathcal{D}_1}_{\sigma_1}\rho\sigma\sigma')\ \mathsf{in}\ \llbracket e_2 \rrbracket^{\mathcal{D}_2}_{\sigma}\sigma'\big)$.
Considering e_1, we have $\sigma_1 \Vdash_{\Delta\cup\vec{\alpha}} D_1$.
We show that $\mathsf{static}(\sigma' \circ \sigma \circ \rho, \mathsf{var}(D_1)\sigma_1)$.

To check $\mathsf{static}(\sigma' \circ \sigma \circ \rho, \mathsf{var}(D_1)\sigma_1)$, take an arbitrary $\alpha \in \mathsf{var}(D_1)\sigma_1$.

- If $\alpha \in \mathsf{dom}(\rho)$, then $\alpha\rho$ is a variable in $\vec{\alpha}_1,\vec{\beta}_1$ and $\alpha\rho = \alpha\rho\sigma\sigma'$ (because $\vec{\alpha}_1,\vec{\beta}_1 \sharp \sigma, \sigma'$): hence $\alpha\rho\sigma\sigma'$ is static.
- If $\alpha \notin \mathsf{dom}(\rho)$, then $\alpha\rho\sigma\sigma' = \alpha\sigma\sigma'$.
  We have $(\alpha \mathrel{\dot{\sqsubseteq}} \alpha) \in \mathsf{equiv}(\sigma_1, D_1)$. Since $\mathsf{equiv}(\sigma_1, D_1) \subseteq D$, $\alpha\sigma$ is static. Furthermore, $\mathsf{var}(\alpha\sigma) \subseteq \mathsf{var}(D)\sigma$; hence, $\alpha\sigma\sigma'$ is static too.

We have $\mathsf{var}(e_1) \subseteq \Delta \cup \vec{\alpha}$.
By IH, $\Gamma\sigma_1\rho\sigma\sigma' \vdash e_1\rho\sigma\sigma' \rightsquigarrow \llbracket e_1 \rrbracket^{\mathcal{D}_1}_{\sigma_1}\rho\sigma\sigma' : \alpha\sigma_1\rho\sigma\sigma'$.
Since $\mathsf{dom}(\sigma) \cap \mathsf{var}(e_1\rho) = \varnothing$, we have $e_1\rho\sigma\sigma' = e_1\rho\sigma'$.
By inversion, $\alpha \notin \mathsf{var}(\Gamma)$.
By Lemma 10.34, $\mathsf{static}(\sigma_1, \mathsf{var}(\Gamma))$.
By Lemma 10.35, $\Gamma\sigma\sigma' \leq^{?} \Gamma\sigma_1\rho\sigma\sigma'$.
By Lemma 10.29, $\Gamma\sigma\sigma' \vdash e_1\rho\sigma' \rightsquigarrow \llbracket e_1 \rrbracket^{\mathcal{D}_1}_{\sigma_1}\rho\sigma\sigma' : \alpha\sigma_1\rho\sigma\sigma'$.
Considering e_2, we have $\sigma \Vdash_{\Delta} D_2$, $\mathsf{static}(\sigma', \mathsf{var}(D_2)\sigma)$, $\mathsf{var}(e_2) \subseteq \Delta$.
By IH, $\Gamma\sigma\sigma', x : (\forall\vec{\alpha},\vec{\beta}.\ \alpha\sigma_1)\sigma\sigma' \vdash e_2\sigma' \rightsquigarrow \llbracket e_2 \rrbracket^{\mathcal{D}_2}_{\sigma}\sigma' : t\sigma\sigma'$.
Since $\vec{\alpha}_1,\vec{\beta}_1 \sharp \sigma, \sigma'$, $(\forall\vec{\alpha},\vec{\beta}.\ \alpha\sigma_1)\sigma\sigma' = (\forall\vec{\alpha}_1,\vec{\beta}_1.\ \alpha\sigma_1\rho\sigma\sigma')$.
We have $\Gamma\sigma\sigma', x : (\forall\vec{\alpha}_1,\vec{\beta}_1.\ \alpha\sigma_1\rho\sigma\sigma') \vdash e_2\sigma' \rightsquigarrow \llbracket e_2 \rrbracket^{\mathcal{D}_2}_{\sigma}\sigma' : t\sigma\sigma'$.
We have $\vec{\alpha}_1,\vec{\beta}_1 \sharp \Gamma\sigma\sigma'$ and $\vec{\beta}_1 \sharp e_1\rho\sigma'$.
Finally, by $[\mathrm{T}_{\mathsf{let}}]$, $\Gamma\sigma\sigma' \vdash e\sigma' \rightsquigarrow \llbracket e \rrbracket^{\mathcal{D}}_{\sigma}\sigma' : t\sigma\sigma'$. □

Non-strict languages

A call-by-need language with set-theoretic types

13.6 COROLLARY: Let $\bigwedge_{i\in I} t'_i \to t_i$ (with $|I| > 0$) be such that, for every $i_1, i_2 \in I$, if $i_1 \neq i_2$ then $t'_{i_1} \wedge t'_{i_2} \simeq \mathbb{0}$. Then:

$$\bigwedge_{i\in I} t'_i \to t_i \leq t' \to t \implies \Big(t' \leq \bigvee_{i\in I} t'_i\Big) \wedge \big(\forall i \in I.\ (t'_i \wedge t' \not\simeq \mathbb{0}) \implies (t_i \leq t)\big)$$

□

Proof: By applying Lemma 13.5, we get

$$\Big(t' \leq \textstyle\bigvee_{i\in I} t'_i\Big) \wedge \big(\forall I' \subsetneq I.\ (t' \leq \textstyle\bigvee_{i\in I'} t'_i) \vee (\bigwedge_{i\in I\setminus I'} t_i \leq t)\big)\ .$$

Now consider an arbitrary $i_0 \in I$ such that $t'_{i_0} \wedge t' \not\simeq \mathbb{0}$; we must show $t_{i_0} \leq t$. Instantiating the quantifier above with $I' = I \setminus \{i_0\}$ we get

$$(t' \leq \textstyle\bigvee_{i\in I\setminus\{i_0\}} t'_i) \vee (\bigwedge_{i\in I\setminus(I\setminus\{i_0\})} t_i \leq t)\ .$$

We show $t' \not\leq \bigvee_{i\in I\setminus\{i_0\}} t'_i$, which concludes the proof since the second term of the union is $t_{i_0} \leq t$.

By contradiction, assume $t' \leq \bigvee_{i\in I\setminus\{i_0\}} t'_i$. Note that $t'_{i_0} \wedge \bigvee_{i\in I\setminus\{i_0\}} t'_i \simeq \mathbb{0}$ (because the t'_i are disjoint); therefore we would also have $t'_{i_0} \wedge t \simeq \mathbb{0}$, which is false by hypothesis. □

13.7 COROLLARY: Let $\bar{t} = (\bigwedge_{i\in I} t'_i \to t_i) \wedge (\bigwedge_{j\in J} \neg(t'_j \to t_j))$. If $\bar{t} \not\simeq \mathbb{0}$ and $\bar{t} \leq t' \to t$, then $(\bigwedge_{i\in I} t'_i \to t_i) \leq t' \to t$. □

Proof: By definition of subtyping, we have

$$\begin{aligned}\bar{t} \not\simeq \mathbb{0} &\iff (\textstyle\bigwedge_{i\in I} t'_i \to t_i) \wedge (\bigwedge_{j\in J} \neg(t'_j \to t_j)) \not\leq \mathbb{0}\\ &\iff \textstyle\bigwedge_{i\in I} t'_i \to t_i \not\leq \bigvee_{j\in J} t'_j \to t_j\end{aligned}$$

and

$$\bar{t} \leq t' \to t \iff \textstyle\bigwedge_{i\in I} t'_i \to t_i \leq (\bigvee_{j\in J} t'_j \to t_j) \vee (t' \to t)$$

Let $\bar{\jmath}$ be such that $\bar{\jmath} \notin J$ and let $t'_{\bar{\jmath}} = t'$ and $t_{\bar{\jmath}} = t$. By Lemma 13.5, we derive

$$\begin{aligned}&\forall j_0 \in J.\ \neg\big(\big(t'_{j_0} \leq \textstyle\bigvee_{i\in I} t'_i\big) \wedge \big(\forall I' \subsetneq I.\ (t'_{j_0} \leq \textstyle\bigvee_{i\in I'} t'_i) \vee (\bigwedge_{i\in I\setminus I'} t_i \leq t_{j_0})\big)\big)\\ &\exists j_0 \in J \cup \{\bar{\jmath}\}.\ \big(t'_{j_0} \leq \textstyle\bigvee_{i\in I} t'_i\big) \wedge \big(\forall I' \subsetneq I.\ (t'_{j_0} \leq \textstyle\bigvee_{i\in I'} t'_i) \vee (\bigwedge_{i\in I\setminus I'} t_i \leq t_{j_0})\big)\end{aligned}$$

where clearly the existentially quantified proposition must be true for $\bar{\jmath}$, which allows us to conclude. □

13.8 LEMMA: For every finite set J and every set $\{\, t_j \mid j \in J \,\}$,

$$\textstyle\bigvee_{J'\subseteq J} \big(\bigwedge_{j\in J'} t_j \wedge \bigwedge_{j\in J\setminus J'} \neg t_j\big) \simeq \mathbb{1}$$

(with the convention that an intersection over an empty set is $\mathbb{1}$). □

Proof: We prove this by induction on $|J|$. If $|J| = 0$, then the only J' is J itself, and the equivalence holds. If $|J| > 0$, consider an arbitrary $j_0 \in J$ and let $\bar{J} = J \setminus \{j_0\}$. We have

$$\begin{aligned}&\textstyle\bigvee_{J'\subseteq J} \big(\bigwedge_{j\in J'} t_j \wedge \bigwedge_{j\in J\setminus J'} \neg t_j\big)\\ \simeq\ &\textstyle\bigvee_{J'\subseteq \bar{J}} \big(\bigwedge_{j\in J'} t_j \wedge \bigwedge_{j\in \bar{J}\setminus J'} \neg t_j \wedge \neg t_{j_0}\big)\\ &\quad \vee \textstyle\bigvee_{J'\subseteq \bar{J}} \big(t_{j_0} \wedge \bigwedge_{j\in J'} t_j \wedge \bigwedge_{j\in \bar{J}\setminus J'} \neg t_j\big)\\ \simeq\ &\big(\neg t_{j_0} \wedge \textstyle\bigvee_{J'\subseteq \bar{J}} (\bigwedge_{j\in J'} t_j \wedge \bigwedge_{j\in \bar{J}\setminus J'} \neg t_j)\big)\\ &\quad \vee \big(t_{j_0} \wedge \textstyle\bigvee_{J'\subseteq \bar{J}} (\bigwedge_{j\in J'} t_j \wedge \bigwedge_{j\in \bar{J}\setminus J'} \neg t_j)\big)\\ \simeq\ &(\neg t_{j_0} \wedge \mathbb{1}) \vee (t_{j_0} \wedge \mathbb{1})\end{aligned}$$

(by the induction hypothesis)

$$\simeq \mathbb{1}\,.$$ □

13.9 LEMMA: Let $\mathbb{I} = \bigwedge_{i\in I} t'_i \to t_i$ (with $|I| > 0$) be a type. Then:

$\mathbb{I} \simeq \bigwedge_{\varnothing\subsetneq I'\subseteq I} s_{I'} \to u_{I'}$ where $s_{I'} \stackrel{\text{def}}{=} \bigwedge_{i\in I'} t'_i \wedge \bigwedge_{i\in I\setminus I'} \neg t'_i$ and $u_{I'} \stackrel{\text{def}}{=} \bigwedge_{i\in I'} t_i$

(with the convention: $\bigwedge_{i\in\varnothing} \neg t'_i = \mathbb{1}$). □

Proof: We first show $\mathbb{I} \leq \bigwedge_{\varnothing\subsetneq I'\subseteq I} s_{I'} \to u_{I'}$. To do this, we show that, for every I' such that $\varnothing \subsetneq I' \subseteq I$, we have $\mathbb{I} \leq s_{I'} \to u_{I'}$, that is,

$$\mathbb{I} \leq \left(\bigwedge_{i\in I'} t'_i \wedge \bigwedge_{i\in I\setminus I'} \neg t'_i\right) \to \left(\bigwedge_{i\in I'} t_i\right).$$

We have

$$\begin{aligned}
\mathbb{I} &= \textstyle\bigwedge_{i\in I} t'_i \to t_i \\
&\leq \textstyle\bigwedge_{i\in I'} t'_i \to t_i \\
&\leq \left(\textstyle\bigwedge_{i\in I'} t'_i\right) \to \left(\textstyle\bigwedge_{i\in I'} t_i\right) \\
&\leq \left(\textstyle\bigwedge_{i\in I'} t'_i \wedge \bigwedge_{i\in I\setminus I'} \neg t'_i\right) \to \left(\textstyle\bigwedge_{i\in I'} t_i\right).
\end{aligned}$$

We now consider the opposite direction. To show $\bigwedge_{\varnothing\subsetneq I'\subseteq I} s_{I'} \to u_{I'} \leq \mathbb{I}$, we show that, for every $i \in I$, we have $\bigwedge_{\varnothing\subsetneq I'\subseteq I} s_{I'} \to u_{I'} \leq t'_i \to t_i$. Consider an arbitrary $i_0 \in I$ and let $\bar{I} = I \setminus \{i_0\}$. We have

$$\begin{aligned}
&\textstyle\bigwedge_{\varnothing\subsetneq I'\subseteq I} s_{I'} \to u_{I'} \\
\leq\ &\textstyle\bigwedge_{\substack{\varnothing\subsetneq I'\subseteq I \\ i_0\in I'}} s_{I'} \to u_{I'} \\
\simeq\ &\textstyle\bigwedge_{I'\subseteq\bar{I}} s_{(I'\cup\{i_0\})} \to u_{(I'\cup\{i_0\})} \\
\simeq\ &\textstyle\bigwedge_{I'\subseteq\bar{I}} \left(\left(t'_{i_0} \wedge \bigwedge_{i\in I'} t'_i \wedge \bigwedge_{i\in\bar{I}\setminus I'} \neg t'_i\right) \to \left(t_{i_0} \wedge \bigwedge_{i\in I'} t_i\right)\right) \\
\leq\ &\left(\textstyle\bigvee_{I'\subseteq\bar{I}} \left(t'_{i_0} \wedge \bigwedge_{i\in I'} t'_i \wedge \bigwedge_{i\in\bar{I}\setminus I'} \neg t'_i\right)\right) \to \left(\textstyle\bigvee_{I'\subseteq\bar{I}} \left(t_{i_0} \wedge \bigwedge_{i\in I'} t_i\right)\right) \\
\simeq\ &\left(t'_{i_0} \wedge \textstyle\bigvee_{I'\subseteq\bar{I}} \left(\bigwedge_{i\in I'} t'_i \wedge \bigwedge_{i\in\bar{I}\setminus I'} \neg t'_i\right)\right) \to \left(t_{i_0} \wedge \textstyle\bigvee_{I'\subseteq\bar{I}} \left(\bigwedge_{i\in I'} t_i\right)\right) \\
\simeq\ &t'_{i_0} \to \left(t_{i_0} \wedge \textstyle\bigvee_{I'\subseteq\bar{I}} \left(\bigwedge_{i\in I'} t_i\right)\right)
\end{aligned}$$

(by Lemma 13.8)

$$\leq t'_{i_0} \to t_{i_0}.$$ □

13.16 LEMMA: For every type t such that $t \leq \mathbb{1} \times \mathbb{1}$, there exists a product decomposition Π such that $t \simeq \bigvee_{t_1\times t_2\in\Pi} t_1 \times t_2$. □

Proof: Let t be such that $t \leq \mathbb{1} \times \mathbb{1}$. Let $\mathsf{dnf}(t) = \{(P_i, N_i) \mid i \in I\}$. By Proposition 13.14, we have $[\![t]\!] = [\![\mathsf{dnf}(t)]\!] = [\![t]\!] = \bigcup_{i\in I} \left(\bigcap_{t'\in P_i} [\![t']\!] \setminus \bigcup_{t'\in N_i} [\![t']\!]\right)$.

We show that, for every $i \in I$, there exists a product decomposition Π_i such that

$$\textstyle\bigcap_{t'\in P_i} [\![t']\!] \setminus \bigcup_{t'\in N_i} [\![t']\!] = \bigcup_{t_1\times t_2\in\Pi_i} [\![t_1 \times t_2]\!].$$

This yields the result we need, taking $\Pi = \bigcup_{i\in I} \Pi_i$.

Consider an arbitrary $i \in I$.

Since $t \leq \mathbb{1} \times \mathbb{1}$, we have $\bigcap_{t'\in P_i} [\![t']\!] \setminus \bigcup_{t'\in N_i} [\![t']\!] \leq [\![\mathbb{1} \times \mathbb{1}]\!]$.

If $\bigcap_{t'\in P_i} [\![t']\!] \setminus \bigcup_{t'\in N_i} [\![t']\!]$, we take $\Pi_i = \varnothing$.

Otherwise, every $t' \in P_i$ must be a product atom. Moreover, we have $\bigcap_{t'\in P_i} [\![t']\!] \setminus \bigcup_{t'\in N_i} [\![t']\!] = \bigcap_{t'\in P_i} [\![t']\!] \setminus \bigcup_{t'\in N'_i} [\![t']\!]$, where N'_i is the intersec-

tion of N_i with the set of all product atoms (i.e., N_i' is N_i minus all atoms that are not product atoms).

Using the two properties (for all sets A_1, A_2, B_1, and B_2)

$$(A_1 \times A_2) \cap (B_1 \times B_2) = (A_1 \cap B_1) \times (A_2 \cap B_2)$$
$$(A_1 \times A_2) \setminus (B_1 \times B_2) = \big((A_1 \setminus B_1) \times A_2\big) \cup \big(A_1 \times (A_2 \setminus B_2)\big)$$

we obtain that

$$\bigcap_{t_1 \times t_2 \in P_i} [\![t']\!] \setminus \bigcup_{t_1 \times t_2 \in N_i'} [\![t']\!]$$
$$= \bigcup_{N'' \subseteq N_i'} \left(\left(\bigcap_{t_1 \times t_2 \in P_i} [\![t_1]\!] \setminus \bigcup_{t_1 \times t_2 \in N''} [\![t_1]\!] \right) \times \left(\bigcap_{t_1 \times t_2 \in P_i} [\![t_2]\!] \setminus \bigcup_{t_1 \times t_2 \in N_i' \setminus N''} [\![t_2]\!] \right) \right)$$

which yields directly a product decomposition. □

13.17 LEMMA: For every product decomposition Π, there exists a product decomposition Π' such that Π' is fully disjoint, that $\bigvee_{t \in \Pi} t \simeq \bigvee_{t' \in \Pi'} t'$, and that $\forall t' \in \Pi.\, \exists t \in \Pi.\, t' \leq t$. □

Proof: Let $\Pi = \{\, t_i^1 \times t_i^2 \mid i \in I \,\}$. When $i \in I$ and $I', I_1, I_2 \subseteq I$, and $k \in \{1, 2\}$, we define

$$\mathbb{T}^k(i, I') = t_i^k \wedge \bigwedge_{j \in I'} t_j^k \wedge \bigwedge_{j \in I \setminus \{i\} \setminus I'} \neg t_j^k$$
$$\mathbb{T}(i, I_1, I_2) = \mathbb{T}^1(i, I_1) \times \mathbb{T}^2(i, I_2)$$

and we consider the product decomposition

$$\Pi' = \bigcup_{i \in I} \{\, \mathbb{T}(i, I_1, I_2) \mid I_1 \subseteq I \setminus \{i\}, I_2 \subseteq I \setminus \{i\}, \mathbb{T}^1(i, I_1) \not\simeq \mathbb{0}, \mathbb{T}^2(i, I_2) \not\simeq \mathbb{0} \,\}\,.$$

We first show that Π' is fully disjoint. First, consider an arbitrary element of Π', $\mathbb{T}(i, I_1, I_2) = \mathbb{T}^1(i, I_1) \times \mathbb{T}^2(i, I_2)$. We must show $\mathbb{T}(i, I_1, I_2) \not\simeq \mathbb{0}$, which holds because we explicitly require both $\mathbb{T}^k(i, I_k)$ to be non-empty. Now, we consider two arbitrary elements of Π':

$$\mathbb{T}(i, I_1, I_2) = \mathbb{T}^1(i, I_1) \times \mathbb{T}^2(i, I_2) \qquad \mathbb{T}(i', I_1', I_2') = \mathbb{T}^1(i', I_1') \times \mathbb{T}^2(i', I_2')$$

and we must prove:

$$\big(\mathbb{T}^1(i, I_1) \wedge \mathbb{T}^1(i', I_1') \simeq \mathbb{0}\big) \vee \big(\mathbb{T}^1(i, I_1) \simeq \mathbb{T}^1(i', I_1')\big)$$
$$\big(\mathbb{T}^2(i, I_2) \wedge \mathbb{T}^2(i', I_2') \simeq \mathbb{0}\big) \vee \big(\mathbb{T}^2(i, I_2) \simeq \mathbb{T}^2(i', I_2')\big)\,.$$

We prove the first (the second is proved identically). Note that if $\{i\} \cup I_1 = \{i'\} \cup I_1'$, then $\mathbb{T}^1(i, I_1)$ and $\mathbb{T}^1(i', I_1')$ are the same up to reordering of the intersections: therefore $\mathbb{T}^1(i, I_1) \simeq \mathbb{T}^1(i', I_1')$ holds. Otherwise, assume without loss of generality that there exists an i_0 such that $i_0 \in \{i\} \cup I_1$ but $i_0 \notin \{i'\} \cup I_1'$. Then, we have $\mathbb{T}^1(i, I_1) \leq t_{i_0}^1$ and $\mathbb{T}^1(i', I_1') \leq \neg t_{i_0}^1$. Then, $\mathbb{T}^1(i, I_1) \wedge \mathbb{T}^1(i', I_1') \leq t_{i_0}^1 \wedge \neg t_{i_0}^1 \leq \mathbb{0}$.

Now we show that, for every $\mathbb{T}(i, I_1, I_2) = \mathbb{T}^1(i, I_1) \times \mathbb{T}^2(i, I_2)$ in Π', there exists a $i' \in I$ such that $\mathbb{T}(i, I_1, I_2) \leq t^1_{i'} \times t^2_{i'}$. We simply take $i' = i$, since both $\mathbb{T}^1(i, I_1) \leq t^1_i$ and $\mathbb{T}^2(i, I_2) \leq t^2_i$ always hold.

Finally, we show that $\bigvee_{i \in I} t^1_i \times t^2_i \simeq \bigvee_{t \in \Pi'} t$. We do so by showing that, for every $i \in I$,

$$t^1_i \times t^2_i \simeq \bigvee_{i \in I, I_1 \subseteq I \setminus \{i\}, I_2 \subseteq I \setminus \{i\}, \mathbb{T}^1(i, I_1) \neq \mathbb{0}, \mathbb{T}^2(i, I_2) \neq \mathbb{0}} \mathbb{T}(i, I_1, I_2)\,.$$

Note that we can show this by showing

$$t^1_i \times t^2_i \simeq \bigvee_{i \in I, I_1 \subseteq I \setminus \{i\}, I_2 \subseteq I \setminus \{i\}} \mathbb{T}(i, I_1, I_2)\,,$$

without the conditions of non-emptiness (we have more summands in the union, but they are empty). We have

$$\begin{aligned}
&\bigvee_{i \in I, I_1 \subseteq I \setminus \{i\}, I_2 \subseteq I \setminus \{i\}} \mathbb{T}(i, I_1, I_2) \\
&= \bigvee_{i \in I, I_1 \subseteq I \setminus \{i\}, I_2 \subseteq I \setminus \{i\}} \mathbb{T}^1(i, I_1) \times \mathbb{T}^2(i, I_2) \\
&\simeq \bigvee_{i \in I, I_1 \subseteq I \setminus \{i\}} \Big(\bigvee_{I_2 \subseteq I \setminus \{i\}} \mathbb{T}^1(i, I_1) \times \mathbb{T}^2(i, I_2) \Big) \\
&\simeq \bigvee_{i \in I, I_1 \subseteq I \setminus \{i\}} \mathbb{T}^1(i, I_1) \times \Big(\bigvee_{I_2 \subseteq I \setminus \{i\}} \mathbb{T}^2(i, I_2) \Big)
\end{aligned}$$

(subtyping of product types satisfies $\bigvee_{i \in I}(t \times t_i) \simeq t \times (\bigvee_{i \in I} t_i)$)

$$\begin{aligned}
&\simeq \bigvee_{i \in I, I_1 \subseteq I \setminus \{i\}} \mathbb{T}^1(i, I_1) \times \Big(\bigvee_{I_2 \subseteq I \setminus \{i\}} \big(t^2_i \wedge \bigwedge_{j \in I_2} t^2_j \wedge \bigwedge_{j \in I \setminus \{i\} \setminus I_2} \neg t^2_j \big) \Big) \\
&\simeq \bigvee_{i \in I, I_1 \subseteq I \setminus \{i\}} \mathbb{T}^1(i, I_1) \times \big(t^2_i \wedge \bigvee_{I_2 \subseteq I \setminus \{i\}} \big(\bigwedge_{j \in I_2} t^2_j \wedge \bigwedge_{j \in I \setminus \{i\} \setminus I_2} \neg t^2_j \big) \big) \\
&\simeq \bigvee_{i \in I, I_1 \subseteq I \setminus \{i\}} \mathbb{T}^1(i, I_1) \times \big(t^2_i \wedge \mathbb{1} \big)
\end{aligned}$$

(by Lemma 13.8)

$$\begin{aligned}
&\simeq \bigvee_{i \in I, I_1 \subseteq I \setminus \{i\}} \mathbb{T}^1(i, I_1) \times t^2_i \\
&\simeq t^1_i \times t^2_i
\end{aligned}$$

(proceeding as above). □

13.18 LEMMA: Let $\Pi = \{\, t^1_i \times t^2_i \mid i \in I \,\}$ be a fully disjoint product decomposition and let t^1 and t^2 be two types such that $t^1 \times t^2 \simeq \bigvee_{i \in I} t^1_i \times t^2_i$. Then, $t^1 \simeq \bigvee_{i \in I} t^1_i$, $t^2 \simeq \bigvee_{i \in I} t^2_i$, and $\forall i_1, i_2 \in I.\ \exists i \in I.\ t^1_{i_1} \times t^2_{i_2} \leq t^1_i \times t^2_i$. □

Proof: We have

$$t^1 \times t^2 \simeq \bigvee_{i \in I} t^1_i \times t^2_i \leq \Big(\bigvee_{i \in I} t^1_i \Big) \times \Big(\bigvee_{i \in I} t^2_i \Big)$$

and therefore $t^1 \leq \bigvee_{i \in I} t^1_i$ and $t^2 \leq \bigvee_{i \in I} t^2_i$, since all t^1_i and t^2_i are non-empty.

Since $\bigvee_{i \in I} t^1_i \times t^2_i \leq t^1 \times t^2$, we have, for all $i \in I$, $t^1_i \times t^2_i \leq t^1 \times t^2$ and hence (by definition of subtyping, since t^1_i and t^2_i are non-empty) $t^1_i \leq t^1$ and $t^2_i \leq t^2$. Hence, we also have $\bigvee_{i \in I} t^1_i \leq t^1$ and $\bigvee_{i \in I} t^2_i \leq t^2$. This yields $t^1 \simeq \bigvee_{i \in I} t^1_i$ and $t^2 \simeq \bigvee_{i \in I} t^2_i$.

To prove $\forall i_1, i_2 \in I.\ \exists i \in I.\ t^1_{i_1} \times t^2_{i_2} \leq t^1_i \times t^2_i$, we consider arbitrary i_1 and i_2 in I; we must show $\exists i \in I.\ t^1_{i_1} \times t^2_{i_2} \leq t^1_i \times t^2_i$. Note that $t^1_{i_1} \times t^2_{i_2} \leq t^1 \times t^2$.

Hence, we have

$$\begin{aligned} t^1_{i_1} \times t^2_{i_2} &\simeq (t^1_{i_1} \times t^2_{i_2}) \wedge (t^1 \times t^2) \\ &\simeq (t^1_{i_1} \times t^2_{i_2}) \wedge (\bigvee_{i \in I} t^1_i \times t^2_i) \\ &\simeq \bigvee_{i \in I} \left((t^1_{i_1} \times t^2_{i_2}) \wedge (t^1_i \times t^2_i) \right) \\ &\simeq \bigvee_{i \in I} \left((t^1_{i_1} \wedge t^1_i) \times (t^2_{i_2} \wedge t^2_i) \right) . \end{aligned}$$

Since $t^1_{i_1} \times t^2_{i_2}$ is not empty, there must exist an $i_0 \in I$ such that $(t^1_{i_1} \wedge t^1_{i_0}) \times (t^2_{i_2} \wedge t^2_{i_0})$ is not empty, that is, an i_0 such that $t^1_{i_1} \wedge t^1_{i_0} \not\simeq \mathbb{0}$ and $t^2_{i_2} \wedge t^2_{i_0} \not\simeq \mathbb{0}$. Since the decomposition is fully disjoint, we have $t^1_{i_1} \simeq t^1_{i_0}$ and $t^2_{i_2} \simeq t^2_{i_0}$: therefore $t^1_{i_1} \times t^2_{i_2} \simeq t^1_{i_0} \times t^2_{i_0}$. □

13.19 LEMMA: If $\Gamma \vdash (e_1, e_2) \colon \bigvee_{i \in I} t_i$, then there exist two types $\bigvee_{j \in J} t_j$ and $\bigvee_{k \in K} t_k$ such that

$$\Gamma \vdash e_1 \colon \bigvee_{j \in J} t_j \quad \Gamma \vdash e_2 \colon \bigvee_{k \in K} t_k \quad \forall j \in J.\, \forall k \in K.\, \exists i \in I.\, t_j \times t_k \leq t_i \,. \quad \square$$

Proof: Since $\Gamma \vdash (e_1, e_2) \colon \bigvee_{i \in I} t_i$, by inversion of the typing derivation, we have $\Gamma \vdash e_1 \colon t^1$, $\Gamma \vdash e_2 \colon t^2$, and $t^1 \times t^2 \leq \bigvee_{i \in I} t_i$.

If $t^1 \simeq \mathbb{0}$ or $t^2 \simeq \mathbb{0}$, then we choose $\bigvee_{j \in J} t_j = t^1$ and $\bigvee_{k \in K} t_k = t^2$ (i.e., $|J| = |K| = 1$), which ensures the result.

Now we assume $t^1 \not\simeq \mathbb{0}$ and $t^2 \not\simeq \mathbb{0}$. We have $t^1 \times t^2 \simeq (\bigvee_{i \in I} t_i) \wedge (t^1 \times t^2) \simeq \bigvee_{i \in I} (t_i \wedge (t^1 \times t^2))$. For every i, we have $t_i \wedge (t^1 \times t^2) \leq \mathbb{1} \times \mathbb{1}$; therefore, by Lemma 13.16, we can find a product decomposition Π_i such that $t_i \wedge (t^1 \times t^2) \simeq \bigvee_{(t_1, t_2) \in \Pi_i} t_1 \times t_2$. Then, $\Pi = \bigcup_{i \in I} \Pi_i$ is itself a product decomposition, such that $\bigvee_{(t_1, t_2) \in \Pi} t_1 \times t_2 \simeq t^1 \times t^2$.

By Lemma 13.17, there exists a fully disjoint product decomposition Π' such that

$$\begin{gathered} \bigvee_{(t_1, t_2) \in \Pi} t_1 \times t_2 \simeq \bigvee_{(t_1, t_2) \in \Pi'} t_1 \times t_2 \\ \forall (t'_1, t'_2) \in \Pi'.\, \exists (t_1, t_2) \in \Pi.\; t'_1 \times t'_2 \leq t_1 \times t_2 \,. \end{gathered}$$

Since $t^1 \times t^2 \simeq \bigvee_{(t_1, t_2) \in \Pi'} t_1 \times t_2$, by Lemma 13.18 we have

$$\begin{gathered} t^1 \simeq \bigvee_{(t_1, t_2) \in \Pi'} t_1 \qquad t^2 \simeq \bigvee_{(t_1, t_2) \in \Pi'} t_2 \\ \forall (t'_1, t'_2), (t''_1, t''_2) \in \Pi' \,. \exists (t_1, t_2) \in \Pi'.\; t'_1 \times t''_2 \leq t_1 \times t_2 \,. \end{gathered}$$

Taking the two decompositions above for t^1 and t^2, we have by subsumption

$$\Gamma \vdash e_1 \colon \bigvee_{(t_1, t_2) \in \Pi'} t_1 \qquad \Gamma \vdash e_2 \colon \bigvee_{(t_1, t_2) \in \Pi'} t_2 \,.$$

It remains to prove that $\forall (t'_1, t'_2), (t''_1, t''_2) \in \Pi' \,. \exists i \in I.\, t'_1 \times t''_2 \leq t_i$. Consider two arbitrary (t'_1, t'_2) and (t''_1, t''_2) in Π'. There exists a $(t_1, t_2) \in \Pi'$ such that $t'_1 \times t''_2 \leq t_1 \times t_2$. Therefore, there exists also a $(t_1, t_2) \in \Pi$ such that $t'_1 \times t''_2 \leq t_1 \times t_2$. This $(t_1, t_2) \in \Pi$ belongs to some Π_i and therefore $t_1 \times t_2 \leq t_i$, implying also $t'_1 \times t''_2 \leq t_i$. □

13.20 LEMMA (Weakening): Let Γ and Γ' be two type environments such that, whenever $x \in \mathsf{dom}(\Gamma)$, we have $x \in \mathsf{dom}(\Gamma')$ and $\Gamma'(x) \leq \Gamma(x)$.
If $\Gamma \vdash e : t$, then $\Gamma' \vdash e : t$. □

Proof: For every Γ and Γ', we define

$$\Gamma \leq \Gamma' \stackrel{\text{def}}{\iff} \forall x \in \mathsf{dom}(\Gamma).\ (x \in \mathsf{dom}(\Gamma')) \wedge (\Gamma'(x) \leq \Gamma(x)) \,.$$

We prove that, if $\Gamma \vdash e : t$ and $\Gamma' \leq \Gamma$, then $\Gamma' \vdash e : t$. We proceed by induction on the derivation of $\Gamma \vdash e : t$ and by cases on the last rule applied.

Case: $[\mathrm{T}_x]$ We conclude by $[\mathrm{T}_x]$ and $[\mathrm{T}_\leq]$.

Case: $[\mathrm{T}_c]$ Straightforward.

Case: $[\mathrm{T}_\lambda]$
We can assume by α-renaming that f and x do not appear in Γ and Γ'; then, we have (for all i) $(\Gamma', f : \mathbb{I}, x : \langle T'_i \rangle) \leq (\Gamma, f : \mathbb{I}, x : \langle T'_i \rangle)$ and we apply the IH to conclude.

Case: $[\mathrm{T}_{\mathrm{app}}]$, $[\mathrm{T}_{\mathrm{pair}}]$, $[\mathrm{T}_{\mathrm{proj}}]$, $[\mathrm{T}_\leq]$ Straightforward by IH.

Case: $[\mathrm{T}_{\mathrm{case}}]$, $[\mathrm{T}_{\mathrm{let}}]$ Similar to the previous case. □

13.21 LEMMA (Admissibility of intersection introduction): If $\Gamma \vdash e : t_1$ and $\Gamma \vdash e : t_2$, then $\Gamma \vdash e : t_1 \wedge t_2$. □

Proof: By induction on the derivations of $\Gamma \vdash e : t_1$ and of $\Gamma \vdash e : t_2$. As a measure we use the sum of the depth of the two derivations.

If the last rule applied in the derivation of $\Gamma \vdash e : t_1$ is $[\mathrm{T}_\leq]$, we have $\Gamma \vdash e : t'_1$ and $t'_1 \leq t_1$. We apply the induction hypothesis to $\Gamma \vdash e : t'_1$ and $\Gamma \vdash e : t_2$ to derive $\Gamma \vdash e : t'_1 \wedge t_2$ and then apply $[\mathrm{T}_\leq]$ since $t'_1 \wedge t_2 \leq t_1 \wedge t_2$. If the last rule applied for e_1 is not $[\mathrm{T}_\leq]$, and that for e_2 is, we do the reverse.

Having dealt with the cases where the last rule applied in one derivation at least is $[\mathrm{T}_\leq]$, we can assume for the remainder that the derivations end with the same rule: every derivation for e must end with the application of the rule corresponding to the form of e, possibly followed by applications of $[\mathrm{T}_\leq]$.

Case: $[\mathrm{T}_x]$, $[\mathrm{T}_c]$
We have $t_1 = t_2$ and we can derive $\Gamma \vdash x : t_1 \wedge t_2$ by subsumption.

Case: $[\mathrm{T}_\lambda]$
We have

$$t_1 = \mathbb{I} \wedge (\textstyle\bigwedge_{j \in J} \neg(t'_j \to t_j)) \qquad t_2 = \mathbb{I} \wedge (\textstyle\bigwedge_{k \in K} \neg(t'_k \to t_k))$$

and we must derive $t_1 \wedge t_2$.

By $[\mathrm{T}_\lambda]$ to derive $\mathbb{I} \wedge (\bigwedge_{j\in J} \neg(t'_j \to t_j)) \wedge (\bigwedge_{k\in K} \neg(t'_k \to t_k))$ (which is non-empty by Corollary 13.7 since t_1 and t_2 are both non-empty).
We conclude by $[\mathrm{T}_\leq]$.

Case: $[\mathrm{T}_{\mathrm{app}}]$
We have $e = e_1\ e_2$ and

$$\Gamma \vdash e_1 : \langle t'_1 \to t''_1 \rangle \qquad \Gamma \vdash e_2 : t'_1 \qquad t_1 = \langle t''_1 \rangle$$
$$\Gamma \vdash e_1 : \langle t'_2 \to t''_2 \rangle \qquad \Gamma \vdash e_2 : t'_2 \qquad t_2 = \langle t''_2 \rangle$$

and, by induction, we derive

$$\Gamma \vdash e_1 : \langle t'_1 \to t''_1 \rangle \wedge \langle t'_2 \to t''_2 \rangle \qquad \Gamma \vdash e_2 : t'_1 \wedge t'_2\,.$$

We have

$$\begin{aligned}\langle t'_1 \to t''_1 \rangle \wedge \langle t'_2 \to t''_2 \rangle &= ((t'_1 \to t''_1) \vee \bot) \wedge ((t'_2 \to t''_2) \vee \bot) \\ &\simeq ((t'_1 \to t''_1) \wedge (t'_2 \to t''_2)) \vee \bot \leq \langle (t'_1 \wedge t'_2) \to (t''_1 \wedge t''_2) \rangle\end{aligned}$$

and conclude by $[\mathrm{T}_\leq]$ and $[\mathrm{T}_{\mathrm{app}}]$.

Case: $[\mathrm{T}_{\mathrm{pair}}]$, $[\mathrm{T}_{\mathrm{proj}}]$
Similar to the previous case.
We use the following properties of subtyping:

$$(t^1_1 \wedge t^1_2) \times (t^2_1 \wedge t^2_2) \simeq (t^1_1 \times t^1_2) \wedge (t^2_1 \times t^2_2)$$
$$\langle t^1_1 \times t^1_1 \rangle \wedge \langle t^1_2 \times t^2_2 \rangle \leq \langle (t^1_1 \wedge t^1_2) \times (t^2_1 \wedge t^2_2) \rangle$$

Case: $[\mathrm{T}_{\mathrm{case}}]$
We have

$$\Gamma \vdash \big((x = \varepsilon) \in \mathbf{t}\ ?\ e_1 : e_2\big) : \langle t_1 \rangle \qquad \Gamma \vdash \varepsilon : \langle t'_1 \rangle$$
$$t'_1 \leq \neg\mathbf{t} \text{ or } \Gamma, x : (t'_1 \wedge \mathbf{t}) \vdash e_1 : t_1 \qquad t'_1 \leq \mathbf{t} \text{ or } \Gamma, x : (t'_1 \setminus \mathbf{t}) \vdash e_2 : t_1$$
$$\Gamma \vdash \big((x = \varepsilon) \in \mathbf{t}\ ?\ e_1 : e_2\big) : \langle t_2 \rangle \qquad \Gamma \vdash \varepsilon : \langle t'_2 \rangle$$
$$t'_2 \leq \neg\mathbf{t} \text{ or } \Gamma, x : (t'_2 \wedge \mathbf{t}) \vdash e_1 : t_2 \qquad t'_2 \leq \mathbf{t} \text{ or } \Gamma, x : (t'_2 \setminus \mathbf{t}) \vdash e_2 : t_2$$

and we derive $\Gamma \vdash \big((x = \varepsilon) \in \mathbf{t}\ ?\ e_1 : e_2\big) : \langle t_1 \wedge t_2 \rangle$ from the premises

$$\Gamma \vdash \varepsilon : \langle t'_1 \wedge t'_2 \rangle$$
$$t'_1 \wedge t'_2 \leq \neg\mathbf{t} \text{ or } \Gamma, x : ((t'_1 \wedge t'_2) \wedge \mathbf{t}) \vdash e_1 : t_1 \wedge t_2$$
$$t'_1 \wedge t'_2 \leq \mathbf{t} \text{ or } \Gamma, x : ((t'_1 \wedge t'_2) \setminus \mathbf{t}) \vdash e_2 : t_1 \wedge t_2$$

The first premise can be derived by applying the induction hypothesis and then subsumption, since $\langle t'_1 \rangle \wedge \langle t'_2 \rangle \simeq \langle t'_1 \wedge t'_2 \rangle$. For the second premise, note that, when $t'_1 \wedge t'_2 \not\leq \neg\mathbf{t}$, we have $t'_1 \not\leq \neg\mathbf{t}$ and $t'_2 \not\leq \neg\mathbf{t}$ and therefore we have

$$\Gamma, x : (t'_1 \wedge \mathbf{t}) \vdash e_1 : t_1 \qquad \Gamma, x : (t'_2 \wedge \mathbf{t}) \vdash e_1 : t_2$$

and, by weakening (Lemma 13.20),

$$\Gamma, x : ((t'_1 \wedge t'_2) \wedge \mathbf{t}) \vdash e_1 : t_1 \qquad \Gamma, x : ((t'_1 \wedge t'_2) \wedge \mathbf{t}) \vdash e_1 : t_2\,.$$

Hence, we derive the premise by the induction hypothesis. The third premise is derived analogously to the second. Finally, we apply subsumption since $\langle t_1 \wedge t_2 \rangle \simeq \langle t_1 \rangle \wedge \langle t_2 \rangle$.

Case: $[\mathrm{T}_{\mathsf{let}}]$
We have

$$\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \colon t_1 \qquad \Gamma \vdash e_1 \colon \bigvee_{i \in I} t_i \qquad \forall i \in I.\ \Gamma, x \colon t_i \vdash e_2 \colon t_1$$
$$\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \colon t_2 \qquad \Gamma \vdash e_1 \colon \bigvee_{j \in J} t_j \qquad \forall j \in J.\ \Gamma, x \colon t_j \vdash e_2 \colon t_2$$

By the induction hypothesis we derive $\Gamma \vdash e_1 \colon (\bigvee_{i \in I} t_i) \wedge (\bigvee_{j \in J} t_j)$; by subsumption we obtain $\Gamma \vdash e_1 \colon \bigvee_{(i,j) \in I \times J} (t_i \wedge t_j)$ since the two types are equivalent. Then, for every $(i, j) \in I \times J$, we want to show $\Gamma, x \colon (t_i \wedge t_j) \vdash e_2 \colon t_1 \wedge t_2$, which we show by applying Lemma 13.20 and the induction hypothesis. □

13.23 LEMMA (Generation): Let Γ be a well-formed type environment and let a be an answer such that $\Gamma \vdash a \colon t$ holds. Then:

- if $t = \langle t_1 \to t_2 \rangle$, then a is of the form $\mu f \colon \mathbb{I}.\, \lambda x.\, e$ or $\mathsf{let}\ x = e\ \mathsf{in}\ a'$;
- if $t = \langle t_1 \times t_2 \rangle$, then a is of the form (e_1, e_2) or $\mathsf{let}\ x = e\ \mathsf{in}\ a'$. □

Proof: The typing derivation $\Gamma \vdash a \colon t$ must end with the application of the rule corresponding to the form of a (for an answer, one of $[\mathrm{T}_c]$, $[\mathrm{T}_\lambda]$, $[\mathrm{T}_{\mathrm{pair}}]$, or $[\mathrm{T}_{\mathsf{let}}]$) possibly followed by applications of $[\mathrm{T}_\leq]$. Therefore, if $a = c$, then we must have $b_c \leq t$: by definition of subtyping, this excludes both $t = \langle t_1 \to t_2 \rangle$ and $t = \langle t_1 \times t_2 \rangle$. Similarly, a non-empty intersection of the form $(\bigwedge_{i \in I} t'_i \to t_i) \wedge (\bigwedge_{j \in J} \neg(t'_j \to t_j))$ cannot be a subtype of $\langle t_1 \times t_2 \rangle$, nor can a type $t_1 \times t_2$ be a subtyping of $\langle t_1 \to t_2 \rangle$, except if it is empty (which is impossible by Lemma 13.11). □

13.25 LEMMA: Let $\bar{\varepsilon}$ be an expression generated by the grammar

$$\bar{\varepsilon} ::= c \mid \mu f \colon \mathbb{I}.\, \lambda x.\, e \mid (\bar{\varepsilon}, \bar{\varepsilon})$$

(that is, an expression ε without variables). For every $\mathbf{t}$, either $\mathsf{typeof}(\bar{\varepsilon}) \leq \mathbf{t}$ or $\mathsf{typeof}(\bar{\varepsilon}) \leq \neg\mathbf{t}$. □

Proof: By induction on the pair $(\bar{\varepsilon}, \mathbf{t})$ and by case analysis on $\bar{\varepsilon}$ and $\mathbf{t}$.

If $\mathbf{t} = \mathbb{0}$, we have $\mathsf{typeof}(\bar{\varepsilon}) \leq \neg\mathbf{t}$.

If $\mathbf{t} = \mathbf{t}_1 \vee \mathbf{t}_2$, we apply the induction hypothesis to both $\mathbf{t}_i$. If $\mathsf{typeof}(\bar{\varepsilon}) \leq \mathbf{t}_1$ or $\mathsf{typeof}(\bar{\varepsilon}) \leq \mathbf{t}_2$, then $\mathsf{typeof}(\bar{\varepsilon}) \leq \mathbf{t}_1 \vee \mathbf{t}_2$. Otherwise, we must have $\mathsf{typeof}(\bar{\varepsilon}) \leq \neg\mathbf{t}_1$ and $\mathsf{typeof}(\bar{\varepsilon}) \leq \neg\mathbf{t}_2$; hence, $\mathsf{typeof}(\bar{\varepsilon}) \leq \neg\mathbf{t}_1 \wedge \neg\mathbf{t}_2 \simeq \neg(\mathbf{t}_1 \vee \mathbf{t}_2)$.

If $\mathbf{t} = \neg\mathbf{t}'$, we apply the induction hypothesis to $\mathbf{t}'$. If $\mathsf{typeof}(\bar{\varepsilon}) \leq \mathbf{t}'$, then $\mathsf{typeof}(\bar{\varepsilon}) \leq \neg\mathbf{t}$. Conversely, if $\mathsf{typeof}(\bar{\varepsilon}) \leq \neg\mathbf{t}'$, then $\mathsf{typeof}(\bar{\varepsilon}) \leq \mathbf{t}$.

If $\mathbf{t} = b$ and $\bar{\varepsilon} = c$, then $\mathsf{typeof}(\bar{\varepsilon}) = b_c$. Since $[\![b_c]\!] = \{c\}$, either $\mathsf{typeof}(\bar{\varepsilon}) \leq b$ or $\mathsf{typeof}(\bar{\varepsilon}) \leq \neg b$ holds. If instead $\bar{\varepsilon}$ is not a constant, then $\mathsf{typeof}(\bar{\varepsilon}) \leq \neg b$.

If $\mathbf{t} = \mathbf{t}_1 \between \mathbf{t}_2$ and $\bar{\varepsilon} = (\bar{\varepsilon}_1, \bar{\varepsilon}_2)$, we apply the induction hypothesis to $(\bar{\varepsilon}_1, \mathbf{t}_1)$ and $(\bar{\varepsilon}_2, \mathbf{t}_2)$. If $\mathsf{typeof}(\bar{\varepsilon}_1) \leq \mathbf{t}_1$ and $\mathsf{typeof}(\bar{\varepsilon}_2) \leq \mathbf{t}_2$, then $\mathsf{typeof}((\bar{\varepsilon}_1, \bar{\varepsilon}_2)) \leq \mathbf{t}_1 \between \mathbf{t}_2$. Otherwise, $\mathsf{typeof}((\bar{\varepsilon}_1, \bar{\varepsilon}_2))$ must be subtype of one of the following types: $\neg\mathbf{t}_1 \times \mathbf{t}_2$, $\mathbf{t}_1 \times \neg\mathbf{t}_2$, or $\neg\mathbf{t}_1 \times \neg\mathbf{t}_2$. We also have $\mathsf{typeof}((\bar{\varepsilon}_1, \bar{\varepsilon}_2)) \leq \neg\bot \times \neg\bot$. The intersection of any of the three types above with $\neg\bot \times \neg\bot$ is a subtype of $\neg(\mathbf{t}_1 \between \mathbf{t}_2)$. Finally, if $\bar{\varepsilon}$ is not a pair, then $\mathsf{typeof}(\bar{\varepsilon}) \leq \neg(\mathbf{t}_1 \between \mathbf{t}_2)$.

If $\mathbf{t} = \mathbb{0} \to \mathbb{1}$, then if $\bar{\varepsilon} = \mu f\colon \mathbb{I}.\, \lambda x.\, e$, we have $\mathsf{typeof}(\bar{\varepsilon}) \leq \mathbf{t}$; otherwise, we have $\mathsf{typeof}(\bar{\varepsilon}) \leq \neg\mathbf{t}$. □

13.26 LEMMA: Let v be a value that is well typed in Γ (i.e., $\Gamma \vdash v\colon t'$ holds for some t'). Then, for every t, we have either $\Gamma \vdash v\colon t$ or $\Gamma \vdash v\colon \neg t$. □

Proof: A value v is either a constant c or an abstraction $\mu f\colon \mathbb{I}.\, \lambda x.\, e$. If $v = c$, then $\Gamma \vdash v\colon b_c$ holds. By subsumption, we have $\Gamma \vdash v\colon t$ whenever $b_c \leq t$. By definition, $b_c \leq t$ is equivalent to $c \in [\![t]\!]$. Since $c \in \mathsf{Domain}$, for every type t, either $c \in [\![t]\!]$ or $c \in [\![\neg t]\!] = \mathsf{Domain} \setminus [\![t]\!]$ must hold. Hence, either $\Gamma \vdash v\colon t$ or $\Gamma \vdash v\colon \neg t$ is derivable.

Consider now $v = \mu f\colon \mathbb{I}.\, \lambda x.\, e$. Note that, since v is well typed, we know by inversion of the typing rules that $\Gamma \vdash v\colon \mathbb{I}$ holds. We prove the result by induction on t.

If $t = \bot$, $t = b$, $t = t_1 \times t_2$, or $t = \mathbb{0}$, we have $\mathbb{I} \leq \neg t$ and hence $\Gamma \vdash v\colon \neg t$.

If $t = t_1 \to t_2$, either $\mathbb{I} \leq t_1 \to t_2$ holds or not. If it holds, we can derive $\Gamma \vdash v\colon t_1 \to t_2$ by subsumption. If it does not hold we have (by definition of subtyping) $\mathbb{I} \wedge \neg(t_1 \to t_2) \not\simeq \mathbb{0}$. We can therefore derive $\Gamma \vdash v\colon \mathbb{I} \wedge \neg(t_1 \to t_2)$ and, by subsumption, $\Gamma \vdash v\colon \neg(t_1 \to t_2)$.

If $t = t_1 \vee t_2$, we apply the induction hypothesis to t_1 and t_2. If either $\Gamma \vdash v\colon t_1$ or $\Gamma \vdash v\colon t_2$ hold, $\Gamma \vdash v\colon t_1 \vee t_2$ holds by subsumption. Otherwise, we must have both $\Gamma \vdash v\colon \neg t_1$ and $\Gamma \vdash v\colon \neg t_2$. Then, by Lemma 13.21, we have $\Gamma \vdash v\colon (\neg t_1) \wedge (\neg t_2)$ and, by subsumption, $\Gamma \vdash v\colon \neg(t_1 \vee t_2)$ since $(\neg t_1) \wedge (\neg t_2) \simeq \neg(t_1 \vee t_2)$.

If $t = \neg t'$, by the induction hypothesis we have either $\Gamma \vdash v\colon t'$ or $\Gamma \vdash v\colon \neg t'$. In the former case, we have $\Gamma \vdash v\colon \neg t$ since $t' \simeq \neg\neg t' = \neg t$; in the latter, we have $\Gamma \vdash v\colon t$. □

13.27 COROLLARY: If $\Gamma \vdash v\colon \bigvee_{i \in I} t_i$, then, for some $i_0 \in I$, $\Gamma \vdash v\colon t_{i_0}$. □

Proof: By induction on $|I|$. If $|I| = 1$, the result is straightforward.

If $|I| = 2$, that is, if $\Gamma \vdash v\colon t_1 \vee t_2$, either $\Gamma \vdash v\colon t_1$ holds or not. In the former case, the result holds. In the latter, by Lemma 13.26, we must have $\Gamma \vdash v\colon \neg t_1$. Hence, by Lemma 13.21, we have $\Gamma \vdash v\colon (t_1 \vee t_2) \wedge \neg t_1$, and $(t_1 \vee t_2) \wedge \neg t_1 \simeq (t_1 \wedge \neg t_1) \vee (t_2 \wedge \neg t_1) \leq t_2$, so we can derive $\Gamma \vdash v\colon t_2$ by subsumption.

If $|I| = n > 2$, we have $\Gamma \vdash v : (t_1 \vee \cdots \vee t_{n-1}) \vee t_n$. We apply the induction hypothesis to conclude. □

13.28 LEMMA: Let $\mathbb{I} = \bigwedge_{i \in I} t'_i \to t_i$ (with $|I| > 0$) be a type. There exists a type $\mathbb{I}' = \bigwedge_{k \in K} t'_k \to t_k$ (with $|K| > 0$) such that:

- $\mathbb{I} \simeq \mathbb{I}'$;
- $\forall k_1 \neq k_2 \in K.\ t_{k_1} \wedge t_{k_2} \simeq \mathbb{0}$;
- if $\Gamma \vdash (\mu f : \mathbb{I}.\, \lambda x.\, e) : \mathbb{I}$, then $\forall k \in K.\ \ \Gamma, f : \mathbb{I}, x : t'_k \vdash e : t_k$. □

Proof: Given $\mathbb{I}$, we take

$$\mathbb{I}' = \bigwedge_{\varnothing \subsetneq I' \subseteq I} s_{I'} \to u_{I'}$$
$$\text{where } s_{I'} \stackrel{\text{def}}{=} \bigwedge_{i \in I'} t'_i \wedge \bigwedge_{i \in I \setminus I'} \neg t'_i \text{ and } u_{I'} \stackrel{\text{def}}{=} \bigwedge_{i \in I'} t_i$$

(defining $\bigwedge_{i \in \varnothing} \neg t'_i$ to be $\mathbb{1}$). By Lemma 13.9, we have $\mathbb{I} \simeq \mathbb{I}'$. To prove that the domains are pairwise disjoint, let I'_1 and I'_2 be two non-empty, arbitrary subsets of I; if $I'_1 \neq I'_2$, then there exists an $i_0 \in I$ which is in one set and not in the other. Assume, without loss of generality, $i_0 \in I'_1$ and $i_0 \notin I'_2$. Then:

$$\begin{aligned}
& s_{I'_1} \wedge s_{I'_2} \\
&= \left(\bigwedge_{i \in I'_1} t'_i \wedge \bigwedge_{i \in I \setminus I'_1} \neg t'_i\right) \wedge \left(\bigwedge_{i \in I'_2} t'_i \wedge \bigwedge_{i \in I \setminus I'_2} \neg t'_i\right) \\
&\simeq \left(t'_{i_0} \wedge \bigwedge_{i \in I'_1 \setminus \{i_0\}} t'_i \wedge \bigwedge_{i \in I \setminus I'_1} \neg t'_i\right) \wedge \left(\neg t'_{i_0} \wedge \bigwedge_{i \in I'_2} t'_i \wedge \bigwedge_{i \in I \setminus \{i_0\} \setminus I'_2} \neg t'_i\right) \\
&\simeq t'_{i_0} \wedge \neg t'_{i_0} \wedge \left(\bigwedge_{i \in I'_1 \setminus \{i_0\}} t'_i \wedge \bigwedge_{i \in I \setminus I'_1} \neg t'_i\right) \wedge \left(\bigwedge_{i \in I'_2} t'_i \wedge \bigwedge_{i \in I \setminus \{i_0\} \setminus I'_2} \neg t'_i\right) \\
&\simeq \mathbb{0}\,.
\end{aligned}$$

To prove the third condition, note that $\Gamma \vdash (\mu f : \mathbb{I}.\, \lambda x.\, e) : \mathbb{I}$ implies that, for every $i \in I$, we can derive $\Gamma, f : \mathbb{I}, x : t'_i \vdash e : t_i$. Now consider an arbitrary I' such that $\varnothing \subsetneq I' \subseteq I$. We must show $\Gamma, f : \mathbb{I}, x : s_{I'} \vdash e : u_{I'}$. Note that $s_{I'} \leq t'_i$ for every $i \in I'$. Hence, by Lemma 13.20, we have (for all $i \in I'$) $\Gamma, f : \mathbb{I}, x : s_{I'} \vdash e : t_i$. By Lemma 13.21, we have $\Gamma, f : \mathbb{I}, x : s_{I'} \vdash e : u_{I'}$ since $u_{I'} = \bigwedge_{i \in I'} t_i$. □

13.29 THEOREM (Progress): Let Γ be a well-formed type environment. Let e be an expression that is well typed in Γ (that is, $\Gamma \vdash e : t$ holds for some t). Then e is an answer, or e is of the form $E[\![x]\!]$, or $\exists e'.\ e \rightsquigarrow e'$. □

Proof: By induction on the derivation of $\Gamma \vdash e : t$ and by case analysis on the last typing rule applied.

Case: $[\mathrm{T}_x]$ In this case $e = x$, and therefore e has the form $E[\![x]\!]$.

Case: $[\mathrm{T}_c]$, $[\mathrm{T}_\lambda]$, $[\mathrm{T}_{\mathrm{pair}}]$ In all cases, e is an answer.

Case: $[\mathrm{T}_{\mathrm{app}}]$
We have

$$e = e_1\, e_2 \qquad \Gamma \vdash e : \langle t \rangle \qquad \Gamma \vdash e_1 : \langle t' \to t \rangle \qquad \Gamma \vdash e_2 : t' \,.$$

We apply the induction hypothesis to e_1. If e_1 reduces, then e reduces by the rule $[\mathrm{R}_{\mathrm{ctx}}]$. If e_1 is of the form $E\llbracket x \rrbracket$, then e is of the form $E'\llbracket x \rrbracket$ with $E' = E\, e_2$.
If e_1 is an answer, then by Lemma 13.23 it is either of the form $\mu f : \mathbb{I}.\, \lambda x.\, e'$ or of the form $\mathsf{let}\ x = e''\ \mathsf{in}\ a$. Therefore, e reduces by $[\mathrm{R}_{\mathrm{app}}]$ or $[\mathrm{R}^{\mathsf{let}}_{\mathrm{app}}]$.

Case: $[\mathrm{T}_{\mathrm{proj}}]$
We have

$$e = \pi_i\, e' \qquad \Gamma \vdash e : \langle t_i \rangle \qquad \Gamma \vdash e' : \langle t_1 \times t_2 \rangle \,.$$

We apply the induction hypothesis to e'. If e' reduces, then e reduces by the rule $[\mathrm{R}_{\mathrm{ctx}}]$. If it is of the form $E\llbracket x \rrbracket$, then e is of the form $E'\llbracket x \rrbracket$ with $E' = \pi_i\, E$.
If e' is an answer, then by Lemma 13.23 it is either of the form (e_1, e_2) or of the form $\mathsf{let}\ x = e''\ \mathsf{in}\ a$. Then e reduces by $[\mathrm{R}_{\mathrm{proj}}]$ or $[\mathrm{R}^{\mathsf{let}}_{\mathrm{proj}}]$.

Case: $[\mathrm{T}_{\mathrm{case}}]$
We have

$$e = ((x = \varepsilon) \in \mathbf{t}\ ?\ e_1 : e_2) \qquad \Gamma \vdash e : \langle t \rangle \qquad \Gamma \vdash \varepsilon : \langle t' \rangle$$
$$t' \leq \neg\mathbf{t} \ \text{ or } \ \Gamma, x : (t' \wedge \mathbf{t}) \vdash e_1 : t \qquad t' \leq \mathbf{t} \ \text{ or } \ \Gamma, x : (t' \setminus \mathbf{t}) \vdash e_2 : t \,.$$

We apply the induction hypothesis to ε. If it reduces, then e reduces by $[\mathrm{R}_{\mathrm{ctx}}]$. If it is of the form $E\llbracket y \rrbracket$, then we must have $\varepsilon = y$ and $E = [\,]$ because all other productions in the grammar for E do not appear in the grammar for ε. Then, we have $\varepsilon = F\llbracket y \rrbracket$ (with $F = [\,]$), and hence e is of the form $E\llbracket y \rrbracket$ with $E = ((x = [\,]) \in \mathbf{t}\ ?\ e_1 : e_2)$.
If ε is an answer, it is either generated by the restricted grammar $\bar{\varepsilon} ::= c \mid \mu f : \mathbb{I}.\, \lambda x.\, e \mid (\bar{\varepsilon}, \bar{\varepsilon})$ (i.e., it does not contain variables except under abstractions) or not. In the latter case, ε is of the form $F\llbracket y \rrbracket$ for some F and y, and hence e is of the form $E\llbracket y \rrbracket$. In the former case, by Lemma 13.25, either $\mathsf{typeof}(\varepsilon) \leq \mathbf{t}$ or $\mathsf{typeof}(\varepsilon) \leq \neg\mathbf{t}$. Then e reduces by $[\mathrm{R}^1_{\mathrm{case}}]$ or $[\mathrm{R}^2_{\mathrm{case}}]$.

Case: $[\mathrm{T}_{\mathsf{let}}]$
We have $e = (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)$ and

$$\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : \bigvee_{i \in I} t_i \qquad \forall i \in I.\ \ \Gamma, x : t_i \vdash e_2 : t \,.$$

Since Γ is well formed, by Lemma 13.11, we know that $\bigvee_{i \in I} t_i$ is not empty. As a consequence, at least one of the t_i is non-empty, and hence at least one of the environments $(\Gamma, x : t_i)$ is well formed, and we can apply the induction hypothesis to it.
We derive that e_2 is an answer, or it has the form $E\llbracket y \rrbracket$, or it reduces. If e_2 is an answer, then e is an answer as well. If e_2 reduces, then e reduces by

$[\mathrm{R}_{\mathrm{ctx}}]$. If e_2 is of the form $E\lceil y \rceil$ for some context E and variable y, then either $x = y$ or not. In the latter case, e is of the form $E'\lceil y \rceil$ too.
If $x = y$, we apply the induction hypothesis to e_1. If e_1 is of the form $E''\lceil z \rceil$ for some context E'' and variable z, then e is of such form as well.
If e_1 reduces, then e reduces by $[\mathrm{R}_{\mathrm{ctx}}]$. If e_1 is an answer, then e reduces by $[\mathrm{R}^{v}_{\mathsf{let}}]$, $[\mathrm{R}^{\mathrm{pair}}_{\mathsf{let}}]$, or $[\mathrm{R}^{\mathsf{let}}_{\mathsf{let}}]$.

Case: $[\mathrm{T}_{\leq}]$
We apply the induction hypothesis to the premise and conclude. □

13.30 THEOREM (Subject reduction): Let Γ be a well-formed type environment. If $\Gamma \vdash e : t$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : t$. □

Proof: By induction on the derivation of $\Gamma \vdash e : t$ and by case analysis on the last typing rule applied.

Case: $[\mathrm{T}_x]$, $[\mathrm{T}_c]$, $[\mathrm{T}_\lambda]$, $[\mathrm{T}_{\mathrm{pair}}]$
These cases do not occur, because $e \rightsquigarrow e'$ cannot hold when e is a variable, a constant, an abstraction, or a pair.

Case: $[\mathrm{T}_{\mathrm{app}}]$
We have

$$e = e_1\, e_2 \qquad \Gamma \vdash e : \langle t \rangle \qquad \Gamma \vdash e_1 : \langle t' \to t \rangle \qquad \Gamma \vdash e_2 : t' \,.$$

If $e_1\, e_2 \rightsquigarrow e'$ occurs by the rule $[\mathrm{R}_{\mathrm{ctx}}]$, then $e' = e'_1\, e_2$ and, by the induction hypothesis, $\Gamma \vdash e'_1 : \langle t' \to t \rangle$: we apply $[\mathrm{T}_{\mathrm{app}}]$ again to type e'.
If the reduction occurs by $[\mathrm{R}_{\mathrm{app}}]$, we have

$$e = (\mu f : \mathbb{I}.\, \lambda x.\, e_3)\, e_2 \qquad e' = \big(\mathsf{let}\ f = (\mu f : \mathbb{I}.\, \lambda x.\, e_3)\ \mathsf{in}\ \mathsf{let}\ x = e_2\ \mathsf{in}\ e_3\big)$$

and we must show $\Gamma \vdash e' : \langle t \rangle$. Let $\mathbb{I} = \bigwedge_{i \in I} \mathbf{T}'_i \mathrel{\not\leftrightarrow} \mathbf{T}_i$. The typing derivation for e is

$$[\mathrm{T}_{\mathrm{app}}]\ \frac{[\mathrm{T}_{\leq}]\ \dfrac{[\mathrm{T}_\lambda]\ \dfrac{\forall i \in I.\ \ \Gamma, f : \mathbb{I}, x : \langle \mathbf{T}'_i \rangle \vdash e_3 : \langle \mathbf{T}_i \rangle}{\Gamma \vdash (\mu f : \mathbb{I}.\, \lambda x.\, e_3) : \mathbb{I} \wedge \bigwedge_{j \in J} \neg(t'_j \to t_j)}}{\Gamma \vdash (\mu f : \mathbb{I}.\, \lambda x.\, e_3) : \langle t' \to t \rangle} \qquad \Gamma \vdash e_2 : t'}{\Gamma \vdash (\mu f : \mathbb{I}.\, \lambda x.\, e_3)\, e_2 : \langle t \rangle}$$

The side conditions of $[\mathrm{T}_\lambda]$ and $[\mathrm{T}_{\leq}]$ ensure

$$\mathbb{I} \wedge \bigwedge_{j \in J} \neg(t'_j \to t_j) \not\simeq \mathbb{0} \qquad \mathbb{I} \wedge \bigwedge_{j \in J} \neg(t'_j \to t_j) \leq \langle t' \to t \rangle$$

from which we have (by definition of subtyping) $\mathbb{I} \wedge \bigwedge_{j \in J} \neg(t'_j \to t_j) \leq t' \to t$ and, by Corollary 13.7 , $\mathbb{I} \leq t' \to t$.
By Lemma 13.28, we find a type $\mathbb{I}' = \bigwedge_{k \in K} t'_k \to t_k$ such that $\mathbb{I} \simeq \mathbb{I}'$, that $t'_{k_1} \wedge t'_{k_2} \simeq \mathbb{0}$ when $k_1 \neq k_2$, and that, for all $k \in K$, $\Gamma, f : \mathbb{I}, x : t'_k \vdash e_3 : t_k$.
Since $\mathbb{I} \leq t' \to t$, we also have $\mathbb{I}' \leq t' \to t$. By Corollary 13.6, we have

$t' \leq \bigvee_{k\in K} t'_k$. Let $\bar{K} = \{\, k \in K \mid t' \wedge t'_k \not\simeq \mathbb{0} \,\}$. We have $t' \leq \bigvee_{k\in\bar{K}} t'_k$. By Corollary 13.6, we also have $\bigvee_{k\in\bar{K}} t_k \leq t$ and therefore $\bigvee_{k\in\bar{K}} t_k \leq \langle t\rangle$. We build the typing derivation for e' as follows:

$$\dfrac{\dfrac{\Gamma, f\colon \mathbb{I} \vdash e_2\colon t'}{\Gamma, f\colon \mathbb{I} \vdash e_2\colon \bigvee_{k\in\bar{K}} t'_k} \quad \forall k \in \bar{K}. \quad \dfrac{\dfrac{\Gamma, f\colon \mathbb{I}, x\colon t'_k \vdash e_3\colon t_k}{\Gamma, f\colon \mathbb{I}, x\colon t'_k \vdash e_3\colon \bigvee_{k\in\bar{K}} t_k}}{\Gamma, f\colon \mathbb{I}, x\colon t'_k \vdash e_3\colon \langle t\rangle}}{\Gamma, f\colon \mathbb{I} \vdash (\mathsf{let}\ x = e_2\ \mathsf{in}\ e_3)\colon \langle t\rangle}$$

$$\dfrac{\Gamma \vdash (\mu f\colon \mathbb{I}.\, \lambda x.\, e_3)\colon \mathbb{I} \qquad \Gamma, f\colon \mathbb{I} \vdash (\mathsf{let}\ x = e_2\ \mathsf{in}\ e_3)\colon \langle t\rangle}{\Gamma \vdash \big(\mathsf{let}\ f = (\mu f\colon \mathbb{I}.\, \lambda x.\, e_3)\ \mathsf{in}\ \mathsf{let}\ x = e_2\ \mathsf{in}\ e_3\big)\colon \langle t\rangle}$$

If the reduction occurs by the rule $[\mathrm{R}^{\mathsf{let}}_{\mathrm{app}}]$, we have

$$e = (\mathsf{let}\ x = e'_1\ \mathsf{in}\ a)\ e_2 \qquad e' = (\mathsf{let}\ x = e'_1\ \mathsf{in}\ a\ e_2)$$

and we must show $\Gamma \vdash e'\colon \langle t\rangle$. The typing derivation for e (collapsing the use of $[\mathrm{T}_{\leq}]$) is

$$\dfrac{\dfrac{\Gamma \vdash e'_1\colon \bigvee_{i\in I} t_i \quad \forall i \in I.\ \Gamma, x\colon t_i \vdash a\colon t''}{\Gamma \vdash (\mathsf{let}\ x = e'_1\ \mathsf{in}\ a)\colon \langle t' \to t\rangle}\ t'' \leq \langle t' \to t\rangle \qquad \Gamma \vdash e_2\colon t'}{\Gamma \vdash (\mathsf{let}\ x = e'_1\ \mathsf{in}\ a)\ e_2\colon \langle t\rangle}$$

from which we build the derivation of $\Gamma \vdash (\mathsf{let}\ x = e'_1\ \mathsf{in}\ a\ e_2)\colon \langle t\rangle$ by deriving, for every $i \in I$,

$$\dfrac{\dfrac{\Gamma, x\colon t_i \vdash a\colon t''}{\Gamma, x\colon t_i \vdash a\colon \langle t' \to t\rangle}\ t'' \leq \langle t' \to t\rangle \qquad \Gamma \vdash e_2\colon t'}{\Gamma, x\colon t_i \vdash a\ e_2\colon \langle t\rangle}$$

Case: $[\mathrm{T}_{\mathrm{proj}}]$
We have

$$e = \pi_i\ e'' \qquad \Gamma \vdash e\colon \langle t_i\rangle \qquad \Gamma \vdash e''\colon \langle t_1 \times t_2\rangle\ .$$

If e reduces by rule $[\mathrm{R}_{\mathrm{ctx}}]$, we obtain the result from the induction hypothesis. Otherwise, the reduction must occur by rule $[\mathrm{R}_{\mathrm{proj}}]$ or rule $[\mathrm{R}^{\mathsf{let}}_{\mathrm{proj}}]$.
If $[\mathrm{R}_{\mathrm{proj}}]$ applies, we have $e = \pi_i\ (e_1, e_2)$ and $e' = e_i$. We must show $\Gamma \vdash e_i\colon \langle t_i\rangle$. Note that we have $\Gamma \vdash (e_1, e_2)\colon \langle t_1 \times t_2\rangle$: by inversion of the typing derivation, we have $\Gamma \vdash e_1\colon t'_1$, $\Gamma \vdash e_2\colon t'_2$, and $t'_1 \times t'_2 \leq \langle t_1 \times t_2\rangle$. By Lemma 13.11, we know $t'_1 \not\simeq \mathbb{0}$ and $t'_2 \not\simeq \mathbb{0}$; hence, by definition of subtyping we have $t'_1 \leq t_1$ and $t'_2 \leq t_2$. Therefore we can derive $\Gamma \vdash e_i\colon \langle t_i\rangle$ by $[\mathrm{T}_{\leq}]$.

If $[\mathrm{R}^{\mathsf{let}}_{\mathrm{proj}}]$ applies, we have $e = \pi_i\ (\mathsf{let}\ x = e'''\ \mathsf{in}\ a)$ and $e' = (\mathsf{let}\ x = e'''\ \mathsf{in}\ \pi_i\ a)$. The typing derivation for e (collapsing the use of $[\mathrm{T}_{\leq}]$) is

$$\dfrac{\dfrac{\Gamma \vdash e''' \colon \bigvee_{i\in I} t_i \qquad \forall i \in I.\ \Gamma, x \colon t_i \vdash a \colon t}{\Gamma \vdash (\mathsf{let}\ x = e'''\ \mathsf{in}\ a) \colon t}}{\Gamma \vdash \pi_i\ (\mathsf{let}\ x = e'''\ \mathsf{in}\ a) \colon \langle t_i \rangle}\ t \leq \langle t_1 \times t_2 \rangle$$

from which we build the derivation for $\Gamma \vdash e' \colon \langle t_i \rangle$ as follows:

$$\dfrac{\Gamma \vdash e''' \colon \bigvee_{i\in I} t_i \qquad \forall i \in I.\ \dfrac{\Gamma, x \colon t_i \vdash a \colon t}{\Gamma, x \colon t_i \vdash \pi_i\ a \colon \langle t_i \rangle}\ t \leq \langle t_1 \times t_2 \rangle}{\Gamma \vdash (\mathsf{let}\ x = e'''\ \mathsf{in}\ \pi_i\ a) \colon \langle t_i \rangle}$$

Case: $[\mathrm{T}_{\mathrm{case}}]$
We have

$$e = ((x = \varepsilon) \in \mathbf{t}\ ?\ e_1 : e_2) \qquad \Gamma \vdash e \colon \langle t \rangle \qquad \Gamma \vdash \varepsilon \colon \langle t' \rangle$$
$$t' \leq \neg\mathbf{t}\ \text{ or }\ \Gamma, x \colon (t' \wedge \mathbf{t}) \vdash e_1 \colon t \qquad t' \leq \mathbf{t}\ \text{ or }\ \Gamma, x \colon (t' \setminus \mathbf{t}) \vdash e_2 \colon t\ .$$

If e reduces by rule $[\mathrm{R}_{\mathrm{ctx}}]$, we obtain the result from the induction hypothesis. Otherwise, it reduces by either $[\mathrm{R}^1_{\mathrm{case}}]$ or $[\mathrm{R}^2_{\mathrm{case}}]$.
If $[\mathrm{R}^1_{\mathrm{case}}]$ applies, we have $e' = (\mathsf{let}\, x = \varepsilon\ \mathsf{in}\ e_1)$ and $\mathsf{typeof}(\varepsilon) \leq \mathbf{t}$. Note that ε cannot be a variable: if it were, we would have $\mathsf{typeof}(\varepsilon) = \mathbb{1}$, but this would require $\mathbf{t} \simeq \mathbb{1}$, which is forbidden by the syntax of typecases. Since ε is not a variable, we have $\mathsf{typeof}(\varepsilon) \leq \neg\bot$. Hence, we also have $\mathsf{typeof}(\varepsilon) \leq \mathbf{t} \wedge \neg\bot$. By Lemma 13.24, we can derive $\Gamma \vdash \varepsilon \colon \mathsf{typeof}(\varepsilon)$; then we can derive $\Gamma \vdash \varepsilon \colon \mathbf{t} \wedge \neg\bot$ by $[\mathrm{T}_{\leq}]$ and $\Gamma \vdash \varepsilon \colon \langle t' \rangle \wedge \mathbf{t} \wedge \neg\bot$ by Lemma 13.21; again by $[\mathrm{T}_{\leq}]$, we derive $\Gamma \vdash \varepsilon \colon t' \wedge \mathbf{t}$ because $\langle t' \rangle \wedge \mathbf{t} \wedge \neg\bot \leq t' \wedge \mathbf{t}$. If $\Gamma, x \colon (t' \wedge \mathbf{t}) \vdash e_1 \colon t$ holds, we can derive $\Gamma \vdash e' \colon \langle t \rangle$ by applying $[\mathrm{T}_{\mathsf{let}}]$ and $[\mathrm{T}_{\leq}]$. If $\Gamma, x \colon (t' \wedge \mathbf{t}) \vdash e_1 \colon t$ does not hold, by hypothesis we would have $t' \leq \neg\mathbf{t}$: we show that this cannot occur. If we had $t' \leq \neg\mathbf{t}$, we could derive $\Gamma \vdash \varepsilon \colon \neg\mathbf{t} \wedge \mathbf{t}$ and $\Gamma \vdash \varepsilon \colon \mathbb{0}$ by subsumption. This is impossible by Lemma 13.11.
If $[\mathrm{R}^2_{\mathrm{case}}]$ applies, we proceed similarly. We have $e' = (\mathsf{let}\ x = \varepsilon\ \mathsf{in}\ e_2)$ and $\mathsf{typeof}(\varepsilon) \leq \neg\mathbf{t}$. We have $\mathsf{typeof}(\varepsilon) \leq \neg\bot$ because ε cannot be a variable (since in that case we would have $\mathbf{t} \simeq \mathbb{0}$, which is forbidden by the syntax). We can derive $\Gamma \vdash \varepsilon \colon t' \wedge \neg\mathbf{t}$, and, if $\Gamma, x \colon (t' \setminus \mathbf{t}) \vdash e_2 \colon t$ holds, $\Gamma \vdash e' \colon \langle t \rangle$. As before, we can show that $\Gamma, x \colon (t' \setminus \mathbf{t}) \vdash e_2 \colon t$ must always hold by showing that the alternative, $t' \leq \mathbf{t}$, cannot occur: if we had $t' \leq \mathbf{t}$, we would have $\Gamma \vdash \varepsilon \colon \mathbf{t} \wedge \neg\mathbf{t}$, which is impossible.

Case: $[\mathrm{T}_{\mathsf{let}}]$
We have $e = (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)$ and

$$\Gamma \vdash e \colon t \qquad \Gamma \vdash e_1 \colon \bigvee_{i\in I} t_i \qquad \forall i \in I.\, \Gamma, x \colon t_i \vdash e_2 \colon t\ .$$

If e reduces by rule $[\mathrm{R}_{\mathrm{ctx}}]$, we obtain the result from the induction hypothesis. Otherwise, e reduces by $[\mathrm{R}^{\mathrm{v}}_{\mathsf{let}}]$, $[\mathrm{R}^{\mathrm{pair}}_{\mathsf{let}}]$, or $[\mathrm{R}^{\mathsf{let}}_{\mathsf{let}}]$.

If $[\mathrm{R}^{v}_{\mathsf{let}}]$ applies, we have $e = (\mathsf{let}\ x = v\ \mathsf{in}\ E[\![x]\!])$ and $e' = (E[\![x]\!])[v/x]$. We must show $\Gamma \vdash e' \colon t$. Since $\Gamma \vdash v \colon \bigvee_{i\in I} t_i$, by Corollary 13.27 we have $\Gamma \vdash v \colon t_{i_0}$ for some $i_0 \in I$. We also have $\Gamma, x \colon t_{i_0} \vdash E[\![x]\!] \colon t$. By Lemma 13.22, we derive $\Gamma \vdash (E[\![x]\!])[v/x] \colon t$.
If $[\mathrm{R}^{\mathsf{pair}}_{\mathsf{let}}]$ applies, we have

$$e = \left(\mathsf{let}\ x = (e_1', e_1'')\ \mathsf{in}\ E[\![x]\!]\right)$$
$$e' = \left(\mathsf{let}\ x' = e_1'\ \mathsf{in}\ \mathsf{let}\ x'' = e_1''\ \mathsf{in}\ (E[\![x]\!])[(x', x'')/x]\right)$$

and must show $\Gamma \vdash e' \colon t$. Since $\Gamma \vdash (e_1', e_1'') \colon \bigvee_{i\in I} t_i$, by Lemma 13.19 we can find two types $\bigvee_{j\in J} t_j$ and $\bigvee_{k\in K} t_k$ such that

$$\Gamma \vdash e_1' \colon \bigvee_{j\in J} t_j \qquad \Gamma \vdash e_1'' \colon \bigvee_{k\in K} t_k$$
$$\forall j \in J.\, \forall k \in K.\, \exists i \in I.\ t_j \times t_k \leq t_i\ .$$

We show $\Gamma \vdash e' \colon t$ by showing

$$\forall j \in J.\, \forall k \in K. \quad \Gamma, x' \colon t_j, x'' \colon t_k \vdash (E[\![x]\!])[(x', x'')/x] \colon t$$

which can be derived by Lemma 13.22 from

$$\forall j \in J.\, \forall k \in K. \quad \begin{cases} \Gamma, x' \colon t_j, x'' \colon t_k \vdash (x', x'') \colon t_j \times t_k \\ \Gamma, x' \colon t_j, x'' \colon t_k, x \colon t_j \times t_k \vdash E[\![x]\!] \colon t \end{cases}$$

For every j and k, the second derivation is obtained from

$$\forall i \in I.\, \Gamma, x \colon t_i \vdash E[\![x]\!] \colon t$$

by weakening (Lemma 13.20), because $t_j \times t_k \leq t_i$ for some $i \in I$.
If $[\mathrm{R}^{\mathsf{let}}_{\mathsf{let}}]$ applies, we have

$$e = \left(\mathsf{let}\ x = (\mathsf{let}\ y = e''\ \mathsf{in}\ a)\ \mathsf{in}\ E[\![x]\!]\right)$$
$$e' = \left(\mathsf{let}\ y = e''\ \mathsf{in}\ \mathsf{let}\ x = a\ \mathsf{in}\ E[\![x]\!]\right)\ .$$

The typing derivation for e (collapsing the use of $[\mathrm{T}_{\leq}]$) is

$$\frac{\Gamma \vdash e'' \colon \bigvee_{j\in J} t_j \qquad \forall j \in J.\ \Gamma, y \colon t_j \vdash a \colon t'}{\Gamma \vdash (\mathsf{let}\ y = e''\ \mathsf{in}\ a) \colon \bigvee_{i\in I} t_i}\ t' \leq \bigvee_{i\in I} t_i$$

$$\frac{\Gamma \vdash (\mathsf{let}\ y = e''\ \mathsf{in}\ a) \colon \bigvee_{i\in I} t_i \qquad \forall i \in I.\ \Gamma, x \colon t_i \vdash E[\![x]\!] \colon t}{\Gamma \vdash \left(\mathsf{let}\ x = (\mathsf{let}\ y = e''\ \mathsf{in}\ a)\ \mathsf{in}\ E[\![x]\!]\right) \colon t}$$

We show $\Gamma \vdash e' \colon t$ as follows:

$$\forall j \in J. \quad \frac{\Gamma, y \colon t_j \vdash a \colon \bigvee_{i\in I} t_i \qquad \forall i \in I.\ \Gamma, y \colon t_j, x \colon t_i \vdash E[\![x]\!] \colon t}{\Gamma, y \colon t_j \vdash (\mathsf{let}\ x = a\ \mathsf{in}\ E[\![x]\!]) \colon t}$$

$$\frac{\Gamma \vdash e'' \colon \bigvee_{j\in J} t_j \qquad \forall j \in J.\ \Gamma, y \colon t_j \vdash (\mathsf{let}\ x = a\ \mathsf{in}\ E[\![x]\!]) \colon t}{\Gamma \vdash \left(\mathsf{let}\ y = e''\ \mathsf{in}\ \mathsf{let}\ x = a\ \mathsf{in}\ E[\![x]\!]\right) \colon t}$$

The premise for the typing of $E[\![x]\!]$ is derived by weakening (Lemma 13.20): we can assume $y \notin \mathsf{dom}(\Gamma)$ by α-renaming.

Case: $[\mathrm{T}_{\leq}]$
We have $\Gamma \vdash e : t'$ for some $t' \leq t$. By the induction hypothesis, we derive $\Gamma \vdash e' : t'$, and we apply $[\mathrm{T}_{\leq}]$ to conclude. □

13.31 LEMMA: If $\Gamma \vdash E\lceil x \rceil : t$, then $x \in \mathsf{dom}(\Gamma)$. □

Proof: By induction on the derivation of $\Gamma \vdash E\lceil x \rceil : t$ and by case analysis on the last rule applied.

Case: $[\mathrm{T}_x]$ We have $E\lceil x \rceil = x$ and $x \in \mathsf{dom}(\Gamma)$.

Case: $[\mathrm{T}_c]$, $[\mathrm{T}_\lambda]$, $[\mathrm{T}_{\mathrm{pair}}]$
impossible, since $E\lceil x \rceil$ cannot be a constant, a function, or a pair.

Case: $[\mathrm{T}_{\mathrm{app}}]$
We have $E\lceil x \rceil = e_1\, e_2$, therefore $E = E'\, e_2$ and $e_1 = E'\lceil x \rceil$; we conclude by applying the induction hypothesis to the derivation of $\Gamma \vdash e_1 : \langle t' \to t \rangle$.

Case: $[\mathrm{T}_{\mathrm{proj}}]$
We have $E\lceil x \rceil = \pi_i\, e$, therefore $E = \pi_i\, E'$ and $e = E'\lceil x \rceil$; we conclude by applying the induction hypothesis to the derivation of $\Gamma \vdash e : \langle t_1 \times t_2 \rangle$.

Case: $[\mathrm{T}_{\mathrm{case}}]$
We have $E\lceil x \rceil = \big((y = \varepsilon) \in \mathbf{t}\ ?\ e_1 : e_2\big)$, therefore $E = \big((y = F) \in \mathbf{t}\ ?\ e_1 : e_2\big)$ and $\varepsilon = F\lceil x \rceil$ (hence, $x \neq y$); we also have that ε is well typed in Γ, so we can conclude by showing, by induction on F, that $\Gamma \vdash F\lceil x \rceil : t$ implies $x \in \mathsf{dom}(\Gamma)$.

Case: $[\mathrm{T}_{\mathrm{let}}]$
Since $E\lceil x \rceil = \big(\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2\big)$ we have either $E = \big(\mathsf{let}\ y = e_1\ \mathsf{in}\ E'\big)$ and $e_2 = E'\lceil x \rceil$ or $E = \big(\mathsf{let}\ y = E'\ \mathsf{in}\ E''\lceil y \rceil\big)$ and $e_1 = E'\lceil x \rceil$; in both cases we have a derivation for $E'\lceil x \rceil$ and, by the induction hypothesis, we derive $x \in \mathsf{dom}(\Gamma)$ (in the first case, the derivation is in an environment $(\Gamma, y : t_i)$, but we have $x \neq y$).

Case: $[\mathrm{T}_{\leq}]$ We conclude directly by IH. □

Discussion

14.2 LEMMA: Let $[\![\cdot]\!]^{\mathsf{m}} : \mathsf{Type} \to \mathcal{P}(\mathsf{Domain}^{\mathsf{m}})$ be a model. Let P and N be finite sets of types of the form $t_1 \to t_2$, with $P \neq \varnothing$. Then:

$$\exists t_1' \to t_2' \in N.\ [\![t_1' \setminus \bigvee_{t_1 \to t_2 \in P} t_1]\!]^{\mathsf{m}} = \varnothing \text{ and}$$
$$\big(\forall P' \subsetneq P.\ [\![t_1' \setminus \bigvee_{t_1 \to t_2 \in P'} t_1]\!]^{\mathsf{m}} = \varnothing \text{ or } [\![\bigwedge_{t_1 \to t_2 \in P \setminus P'} t_2 \setminus t_2']\!]^{\mathsf{m}} = \varnothing\big)$$
$$\implies \bigcap_{t_1 \to t_2 \in P} [\![t_1 \to t_2]\!]^{\mathsf{m}} \subseteq \bigcup_{t_1 \to t_2 \in N} [\![t_1 \to t_2]\!]^{\mathsf{m}}$$

□

Proof: We define

$$\mathsf{Tot}(X) \stackrel{\text{def}}{=} \{\, R \in \mathcal{P}(\mathsf{Domain}^{\mathsf{m}} \times \mathsf{Domain}^{\mathsf{m}}) \mid \mathsf{dom}(R) \supseteq X \,\}$$

$$X \rightharpoonup Y \stackrel{\text{def}}{=} \{\, R \in \mathcal{P}(\mathsf{Domain}^{\mathsf{m}} \times \mathsf{Domain}^{\mathsf{m}}) \mid \forall (d, d') \in R.\ d \in X \implies d' \in \,\}$$

$$X \twoheadrightarrow Y \stackrel{\text{def}}{=} \{\, R \in \mathcal{P}(\mathsf{Domain}^{\mathsf{m}} \times \mathsf{Domain}^{\mathsf{m}}) \mid \mathsf{dom}(R) \supseteq X \text{ and } \forall (d, d') \in R.\ d \in X \implies d' \in Y \,\}$$

and therefore we have $X \twoheadrightarrow Y = \mathsf{Tot}(X) \cap (X \rightharpoonup Y)$. We also have $X \rightharpoonup Y = \mathcal{P}(\overline{X \times \overline{Y}^{D_1}}^{D_2})$, using the notation $\overline{A}^{B}$ for $B \setminus A$ and writing D_1 for $\mathsf{Domain}^{\mathsf{m}}$ and D_2 for $\mathsf{Domain}^{\mathsf{m}} \times \mathsf{Domain}^{\mathsf{m}}$.

To show $\bigcap_{t_1 \to t_2 \in P} [\![t_1 \to t_2]\!]^{\mathsf{m}} \subseteq \bigcup_{t_1 \to t_2 \in N} [\![t_1 \to t_2]\!]^{\mathsf{m}}$, by the definition of model, it suffices to show $\bigcap_{t_1 \to t_2 \in P} [\![t_1]\!]^{\mathsf{m}} \twoheadrightarrow [\![t_2]\!]^{\mathsf{m}} \subseteq \bigcup_{t_1 \to t_2 \in N} [\![t_1]\!]^{\mathsf{m}} \twoheadrightarrow [\![t_2]\!]^{\mathsf{m}}$. We will actually show $\bigcap_{t_1 \to t_2 \in P} [\![t_1]\!]^{\mathsf{m}} \twoheadrightarrow [\![t_2]\!]^{\mathsf{m}} \subseteq [\![t_1']\!]^{\mathsf{m}} \twoheadrightarrow [\![t_2']\!]^{\mathsf{m}}$, which is enough to conclude.

To show it, we first show $\bigcap_{t_1 \to t_2 \in P} [\![t_1]\!]^{\mathsf{m}} \rightharpoonup [\![t_2]\!]^{\mathsf{m}} \subseteq [\![t_1']\!]^{\mathsf{m}} \rightharpoonup [\![t_2']\!]^{\mathsf{m}}$, without the requirement of totality.

The premise

$$[\![t_1' \setminus \bigvee_{t_1 \to t_2 \in P} t_1]\!]^{\mathsf{m}} = \varnothing$$
$$\text{and } \big(\forall P' \subsetneq P.\ [\![t_1' \setminus \bigvee_{t_1 \to t_2 \in P'} t_1]\!]^{\mathsf{m}} = \varnothing \text{ or } [\![\bigwedge_{t_1 \to t_2 \in P \setminus P'} t_2 \setminus t_2']\!]^{\mathsf{m}} = \varnothing\big)$$

can be rewritten as

$$[\![t_1']\!]^{\mathsf{m}} \subseteq [\![\bigvee_{t_1 \to t_2 \in P} t_1]\!]^{\mathsf{m}}$$
$$\text{and } \big(\forall P' \subsetneq P.\ [\![t_1']\!]^{\mathsf{m}} \subseteq [\![\bigvee_{t_1 \to t_2 \in P'} t_1]\!]^{\mathsf{m}} \text{ or } [\![\bigwedge_{t_1 \to t_2 \in P \setminus P'} t_2]\!]^{\mathsf{m}} \subseteq [\![t_2']\!]^{\mathsf{m}}\big)$$

and implies

$$[\![t_1']\!]^{\mathsf{m}} \subseteq \bigcup_{t_1 \to t_2 \in P} [\![t_1]\!]^{\mathsf{m}}$$
$$\text{and } \big(\forall P' \subsetneq P.\ [\![t_1']\!]^{\mathsf{m}} \subseteq \bigcup_{t_1 \to t_2 \in P'} [\![t_1]\!]^{\mathsf{m}} \text{ or } \overline{[\![t_2']\!]^{\mathsf{m}}}^{D_1} \subseteq \bigcup_{t_1 \to t_2 \in P \setminus P'} \overline{[\![t_2]\!]^{\mathsf{m}}}^{D_1}\big)$$

and therefore implies

$$\forall P' \subseteq P.\ [\![t_1']\!]^{\mathsf{m}} \subseteq \bigcup_{t_1 \to t_2 \in P'} [\![t_1]\!]^{\mathsf{m}} \text{ or } \overline{[\![t_2']\!]^{\mathsf{m}}}^{D_1} \subseteq \bigcup_{t_1 \to t_2 \in P \setminus P'} \overline{[\![t_2]\!]^{\mathsf{m}}}^{D_1}$$

as well. We can apply Lemma 6.4 of Frisch, Castagna, and Benzaken (2008) to obtain

$$[\![t_1']\!]^{\mathsf{m}} \times \overline{[\![t_2']\!]^{\mathsf{m}}}^{D_1} \subseteq \bigcup_{t_1 \to t_2 \in P} [\![t_1]\!]^{\mathsf{m}} \times \overline{[\![t_2]\!]^{\mathsf{m}}}^{D_1}$$

whence

$$\overline{\bigcup_{t_1 \to t_2 \in P} [\![t_1]\!]^{\mathsf{m}} \times \overline{[\![t_2]\!]^{\mathsf{m}}}^{D_1}}^{D_2} \subseteq \overline{[\![t_1']\!]^{\mathsf{m}} \times \overline{[\![t_2']\!]^{\mathsf{m}}}^{D_1}}^{D_2}$$

and

$$\bigcap_{t_1 \to t_2 \in P} \overline{[\![t_1]\!]^{\mathsf{m}} \times \overline{[\![t_2]\!]^{\mathsf{m}}}^{D_1}}^{D_2} \subseteq \overline{[\![t_1']\!]^{\mathsf{m}} \times \overline{[\![t_2']\!]^{\mathsf{m}}}^{D_1}}^{D_2}$$

and finally

$$\bigcap_{t_1 \to t_2 \in P} \mathcal{P}\big(\overline{[\![t_1]\!]^{\mathsf{m}} \times \overline{[\![t_2]\!]^{\mathsf{m}}}^{D_1}}^{D_2}\big) \subseteq \mathcal{P}\big(\overline{[\![t_1']\!]^{\mathsf{m}} \times \overline{[\![t_2']\!]^{\mathsf{m}}}^{D_1}}^{D_2}\big)$$

where note that the powerset construction obtained is equivalent to the definition of $\rightharpoonup$.

Now we have

$$\bigcap_{t_1 \to t_2 \in P} [\![t_1]\!]^{\mathsf{m}} \rightharpoonup [\![t_2]\!]^{\mathsf{m}} \subseteq [\![t_1']\!]^{\mathsf{m}} \rightharpoonup [\![t_2']\!]^{\mathsf{m}}$$

and we want

$$\bigcap_{t_1 \to t_2 \in P} [\![t_1]\!]^{\mathsf{m}} \twoheadrightarrow [\![t_2]\!]^{\mathsf{m}} \subseteq [\![t_1']\!]^{\mathsf{m}} \twoheadrightarrow [\![t_2']\!]^{\mathsf{m}} ,$$

that is,

$$\bigcap_{t_1 \to t_2 \in P} \Big(\mathsf{Tot}([\![t_1]\!]^{\mathsf{m}}) \cap ([\![t_1]\!]^{\mathsf{m}} \rightharpoonup [\![t_2]\!]^{\mathsf{m}}) \Big) \subseteq \mathsf{Tot}([\![t_1']\!]^{\mathsf{m}}) \cap ([\![t_1']\!]^{\mathsf{m}} \rightharpoonup [\![t_2']\!]^{\mathsf{m}}) .$$

The latter is further equivalent to

$$\mathsf{Tot}([\![\bigvee_{t_1 \to t_2 \in P} t_1]\!]^{\mathsf{m}}) \cap \bigcap_{t_1 \to t_2 \in P} ([\![t_1]\!]^{\mathsf{m}} \rightharpoonup [\![t_2]\!]^{\mathsf{m}}) \subseteq \mathsf{Tot}([\![t_1']\!]^{\mathsf{m}}) \cap ([\![t_1']\!]^{\mathsf{m}} \rightharpoonup [\![t_2']\!]^{\mathsf{m}}) .$$

Note that $[\![t_1' \setminus \bigvee_{t_1 \to t_2 \in P} t_1]\!]^{\mathsf{m}} = \varnothing$ implies $[\![t_1']\!]^{\mathsf{m}} \subseteq [\![\bigvee_{t_1 \to t_2 \in P} t_1]\!]^{\mathsf{m}}$. Therefore, we have $\mathsf{Tot}([\![t_1']\!]^{\mathsf{m}}) \supseteq \mathsf{Tot}([\![\bigvee_{t_1 \to t_2 \in P} t_1]\!]^{\mathsf{m}})$. This allows us to conclude that the containment above holds. □

14.3 PROPOSITION: Let $[\![\cdot]\!]^{\mathsf{m}} : \mathsf{Type} \to \mathcal{P}(\mathsf{Domain}^{\mathsf{m}})$ be a model. Let t_1 and t_2 be two finite (that is, non-recursive) types. If $[\![t_1]\!] \subseteq [\![t_2]\!]$, then $[\![t_1]\!]^{\mathsf{m}} \subseteq [\![t_2]\!]^{\mathsf{m}}$. □

Proof: First, note that $[\![t_1]\!] \subseteq [\![t_2]\!] \iff [\![t_1 \setminus t_2]\!] = \varnothing$ and that $[\![t_1]\!]^{\mathsf{m}} \subseteq [\![t_2]\!]^{\mathsf{m}} \iff [\![t_1 \setminus t_2]\!]^{\mathsf{m}} = \varnothing$. We therefore show this equivalent proposition: for all finite t, if $[\![t]\!] = \varnothing$, then $[\![t]\!]^{\mathsf{m}} = \varnothing$.

We define the function $h(\cdot)$ on finite types by structural induction as follows:

$$h(\bot) = h(b) = h(\mathbb{0}) = 0 \qquad h(t_1 \times t_2) = h(t_1 \to t_2) = \max(h(t_1), h(t_2)) + 1$$
$$h(t_1 \vee t_2) = \max(h(t_1), h(t_2)) \qquad h(\neg t) = h(t)$$

That is, $h(t)$ is the maximum number of $\times$ and $\to$ constructors found on paths from the root of t to the leaves. We use $h(\cdot)$ as the measure for induction.

Now, let us consider an arbitrary finite type t such that $[\![t]\!] = \varnothing$. We want to show $[\![t]\!]^{\mathsf{m}} = \varnothing$.

Let $\mathsf{dnf}(t) = \{ (P_i, N_i) \mid i \in I \}$. By Proposition 13.14, we have $[\![t]\!] = [\![\mathsf{dnf}(t)]\!]$. We can extend the definition of $[\![\cdot]\!]^{\mathsf{m}}$ to disjunctive normal forms as done for $[\![\cdot]\!]$; we obtain that $[\![t]\!]^{\mathsf{m}} = [\![\mathsf{dnf}(t)]\!]^{\mathsf{m}}$.

Since $[\![t]\!] = \varnothing$, we have

$$\forall i \in I.\ \bigcap_{t \in P_i} [\![t]\!] \setminus \bigcup_{t \in N_i} [\![t]\!] = \varnothing .$$

We want to show

$$\forall i \in I.\ \bigcap_{t \in P_i} [\![t]\!]^{\mathsf{m}} \setminus \bigcup_{t \in N_i} [\![t]\!]^{\mathsf{m}} = \varnothing .$$

which would conclude our proof.

We partition atoms into four kinds, according to their form: $\bot$, b, $t_1 \times t_2$, or $t_1 \to t_2$. If t_1 and t_2 are two atoms of different kind, then $[\![t_1]\!]^{\mathsf{m}} \cap [\![t_2]\!]^{\mathsf{m}} = \varnothing$ (the same holds for $[\![\cdot]\!]$).

Consider an arbitrary $i \in I$. Either P_i is empty or it contains at least one atom. First we show that if P_i contains atoms of at least two different kinds, then $\bigcap_{t\in P_i} [\![t]\!]^{\mathsf{m}} \setminus \bigcup_{t\in N_i} [\![t]\!]^{\mathsf{m}}$ is empty. This holds because the intersection is a subset of $[\![t_1]\!]^{\mathsf{m}} \cap [\![t_2]\!]^{\mathsf{m}}$, where t_1 and t_2 are two atoms of different kind in P_i, and we have remarked that atoms of different kinds have disjoint interpretations.

There remain two cases to consider: $P_i = \varnothing$ or P_i non-empty and composed of atoms of a single kind. We consider the case $P_i = \varnothing$ first. Note that in that case

$$\begin{aligned}
&\bigcap_{t\in P_i} [\![t]\!] \setminus \bigcup_{t\in N_i} [\![t]\!] \\
&= \mathsf{Domain} \setminus \bigcup_{t\in N_i} [\![t]\!] \\
&= (\{\bot\} \cup \mathsf{Const} \cup [\![\mathbb{1} \times \mathbb{1}]\!] \cup [\![\mathbb{0} \to \mathbb{1}]\!]) \setminus \bigcup_{t\in N_i} [\![t]\!]
\end{aligned}$$

because the domain Domain can be decomposed as a union of four sets corresponding to the four kinds of atoms. Since the intersection is empty, we have

$$\begin{aligned}
\{\bot\} \setminus \bigcup_{t\in N_i} [\![t]\!] &= \varnothing & \mathsf{Const} \setminus \bigcup_{t\in N_i} [\![t]\!] &= \varnothing \\
[\![\mathbb{1} \times \mathbb{1}]\!] \setminus \bigcup_{t\in N_i} [\![t]\!] &= \varnothing & [\![\mathbb{0} \to \mathbb{1}]\!] \setminus \bigcup_{t\in N_i} [\![t]\!] &= \varnothing
\end{aligned}$$

Setting aside the intersection with Const for a moment, observe that the others are equivalent to

$$\begin{aligned}
\bigcap_{t\in\{\bot\}} [\![t]\!] \setminus \bigcup_{t\in N_i} [\![t]\!] &= \varnothing \\
\bigcap_{t\in\{\mathbb{1}\times\mathbb{1}\}} [\![t]\!] \setminus \bigcup_{t\in N_i} [\![t]\!] &= \varnothing \\
\bigcap_{t\in\{\mathbb{0}\to\mathbb{1}\}} [\![t]\!] \setminus \bigcup_{t\in N_i} [\![t]\!] &= \varnothing
\end{aligned}$$

so they can be treated together with the case of non-empty P_i.

As for the intersection with Const, if $\mathsf{Const} \setminus \bigcup_{t\in N_i} [\![t]\!] = \varnothing$, then $\mathsf{Const} \subseteq \bigcup_{b\in N_i} \mathbb{B}(b)$ (we can ignore atoms of different kind in N_i). But then $\mathsf{Const} \subseteq \bigcup_{b\in N_i} [\![b]\!]^{\mathsf{m}}$, and therefore $\mathsf{Const} \subseteq \bigcup_{t\in N_i} [\![t]\!]^{\mathsf{m}}$, which shows that $\mathsf{Const} \setminus \bigcup_{t\in N_i} [\![t]\!]^{\mathsf{m}} = \varnothing$.

Now, assuming

$$\begin{aligned}
\{\bot\} \setminus \bigcup_{t\in N_i} [\![t]\!]^{\mathsf{m}} &= \varnothing \\
\mathsf{Const} \setminus \bigcup_{t\in N_i} [\![t]\!]^{\mathsf{m}} &= \varnothing \\
[\![\mathbb{1} \times \mathbb{1}]\!]^{\mathsf{m}} \setminus \bigcup_{t\in N_i} [\![t]\!]^{\mathsf{m}} &= \varnothing \\
[\![\mathbb{0} \to \mathbb{1}]\!]^{\mathsf{m}} \setminus \bigcup_{t\in N_i} [\![t]\!]^{\mathsf{m}} &= \varnothing
\end{aligned}$$

we have

$$(\{\bot\} \cup \mathsf{Const} \cup [\![\mathbb{1} \times \mathbb{1}]\!]^{\mathsf{m}} \cup [\![\mathbb{0} \to \mathbb{1}]\!]^{\mathsf{m}}) \setminus \bigcup_{t\in N_i} [\![t]\!]^{\mathsf{m}} = \varnothing$$

which is

$$\mathsf{Domain}^{\mathsf{m}} \setminus \bigcup_{t\in N_i} [\![t]\!]^{\mathsf{m}} = \varnothing\ .$$

We now consider the remaining case. That is, we assume

$$\bigcap_{t\in P_i} [\![t]\!] \setminus \bigcup_{t\in N_i} [\![t]\!] = \varnothing$$

with P_i non-empty and formed of atoms of a single kind, and we show

$$\bigcap_{t\in P_i} [\![t]\!]^{\mathsf{m}} \setminus \bigcup_{t\in N_i} [\![t]\!]^{\mathsf{m}} = \varnothing\ .$$

Equivalently, we assume $\bigcap_{t\in P_i} [\![t]\!] \subseteq \bigcup_{t\in N_i} [\![t]\!]$ and we show $\bigcap_{t\in P_i} [\![t]\!]^{\mathsf{m}} \subseteq \bigcup_{t\in N_i} [\![t]\!]^{\mathsf{m}}$. In doing so, we can disregard the atoms in N_i that are not of the same kind as those in P_i. Therefore, we consider that N_i only contains atoms of that same kind.

If the atoms of P_i are of the kind of $\bot$ (that is, if $P_i = \{\bot\}$) or if they are base types, the result is immediate because the two interpretations are defined identically on these kinds of atoms.

If the atoms of P_i are all products, then we have (by Lemmas 6.4 and 6.5 of Frisch, Castagna, and Benzaken (2008)):

$$\begin{aligned}\bigcap_{t\in P_i} [\![t]\!] \subseteq \bigcup_{t\in N_i} [\![t]\!] \iff \\ \forall N \subseteq N_i.\ [\![\bigwedge_{t_1\times t_2\in P_i} t_1 \wedge \bigwedge_{t_1\times t_2\in N} \neg t_1]\!] = \varnothing \\ \text{or } [\![\bigwedge_{t_1\times t_2\in P_i} t_2 \wedge \bigwedge_{t_1\times t_2\in N_i\setminus N} \neg t_2]\!] = \varnothing\end{aligned}$$

(with the convention $\bigwedge_{t_1\times t_2\in\varnothing} \neg t_i = \mathsf{Domain}$). Since $[\![\cdot]\!]^{\mathsf{m}}$ also satisfies $[\![t_1 \times t_2]\!]^{\mathsf{m}} = [\![t_1]\!]^{\mathsf{m}} \times [\![t_2]\!]^{\mathsf{m}}$, we also have

$$\begin{aligned}\bigcap_{t\in P_i} [\![t]\!]^{\mathsf{m}} \subseteq \bigcup_{t\in N_i} [\![t]\!]^{\mathsf{m}} \iff \\ \forall N \subseteq N_i.\ [\![\bigwedge_{t_1\times t_2\in P_i} t_1 \wedge \bigwedge_{t_1\times t_2\in N} \neg t_1]\!]^{\mathsf{m}} = \varnothing \\ \text{or } [\![\bigwedge_{t_1\times t_2\in P_i} t_2 \wedge \bigwedge_{t_1\times t_2\in N_i\setminus N} \neg t_2]\!]^{\mathsf{m}} = \varnothing\end{aligned}$$

(with the convention $\bigwedge_{t_1\times t_2\in\varnothing} \neg t_i = \mathsf{Domain}^{\mathsf{m}}$). This allows us to conclude $\bigcap_{t\in P_i} [\![t]\!]^{\mathsf{m}} \subseteq \bigcup_{t\in N_i} [\![t]\!]^{\mathsf{m}}$, because we can apply the induction hypothesis to all the types $\bigwedge_{t_1\times t_2\in P_i} t_1 \wedge \bigwedge_{t_1\times t_2\in N} \neg t_1$ and $\bigwedge_{t_1\times t_2\in P_i} t_2 \wedge \bigwedge_{t_1\times t_2\in N_i\setminus N} \neg t_2$. Indeed, note that $h(\cdot)$ on these types is always strictly less than $\max\{\, h(t_1\times t_2) \mid t_1 \times t_2 \in P_i \cup N_i \,\}$, because the $\times$ constructor has been eliminated. Also, $h(t) \geq \max\{\, h(t_1 \times t_2) \mid t_1 \times t_2 \in P_i \cup N_i \,\}$ because any atom $t_1 \times t_2$ appeared under t.

The last case to examine is that of P_i composed only of arrow types. In that case, by Lemma 13.5, we have

$$\begin{aligned}\bigcap_{t_1\to t_2\in P_i} [\![t_1 \to t_2]\!] \subseteq \bigcup_{t_1\to t_2\in N_i} [\![t_1 \to t_2]\!] \iff \\ \exists t'_1 \to t'_2 \in N_i.\ [\![t'_1 \setminus \bigvee_{t_1\to t_2\in P_i} t_1]\!] = \varnothing \text{ and} \\ \big(\forall P \subsetneq P_i.\ [\![t'_1 \setminus \bigvee_{t_1\to t_2\in P} t_1]\!] = \varnothing \text{ or } [\![\bigwedge_{t_1\to t_2\in P_i\setminus P} t_2 \setminus t'_2]\!] = \varnothing\big)\end{aligned}$$

and, by Lemma 14.2,

$$\exists t_1' \to t_2' \in N_i.\ [\![t_1' \setminus \bigvee_{t_1 \to t_2 \in P_i} t_1]\!]^{\mathsf{m}} = \varnothing \text{ and}$$
$$\left(\forall P \subsetneq P_i.\ [\![t_1' \setminus \bigvee_{t_1 \to t_2 \in P} t_1]\!]^{\mathsf{m}} = \varnothing \text{ or } [\![\bigwedge_{t_1 \to t_2 \in P_i \setminus P} t_2 \setminus t_2']\!]^{\mathsf{m}} = \varnothing\right)$$
$$\implies \bigcap_{t_1 \to t_2 \in P_i} [\![t_1 \to t_2]\!]^{\mathsf{m}} \subseteq \bigcup_{t_1 \to t_2 \in N_i} [\![t_1 \to t_2]\!]^{\mathsf{m}}$$

and we can therefore conclude by applying the induction hypothesis (with the same argument as before to show that $h(\cdot)$ decreases). □

B Semantics of the cast languages

We present here the definition of the operational semantics for the cast languages of Chapters 9 and 10. We give the definitions together with some explanation and state the main results: for the proofs, we refer to the full treatment in the paper (Castagna et al., 2019).

Note that we have changed some of the notation with respect to the cited paper for uniformity with the rest of the thesis. Notably, materialization, subtyping on static types, and subtyping on gradual types are denoted here by $\sqsubseteq$, $\leq$, and $\leq^?$, respectively, while in the paper they are $\preccurlyeq$, $\leq_T$, and $\leq$. We have changed the names of the typing rules, but not those of the reduction rules.

B.1 Semantics of the cast language without subtyping

The cast language has a strict reduction semantics defined by the reduction rules in Figure B.1. The semantics is defined in terms of values (ranged over by V), evaluation contexts (ranged over by $\mathcal{E}$), and ground types (ranged over by ρ). The first two are defined as follows:

$$
\begin{aligned}
V &::= c \mid \lambda^{\tau\to\tau} x.\, E \mid (V, V) \\
&\quad \mid V\langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2' \rangle \mid V\langle \tau_1 \times \tau_2 \overset{p}{\Rightarrow} \tau_1' \times \tau_2' \rangle \mid V\langle \rho \overset{p}{\Rightarrow} ? \rangle \\
\mathcal{E} &::= \square \mid E\,\mathcal{E} \mid \mathcal{E}\,V \mid \mathcal{E}\,[\vec{t}\,] \mid (E, \mathcal{E}) \mid (\mathcal{E}, V) \mid \pi_i\,\mathcal{E} \mid \mathsf{let}\ x = \mathcal{E}\ \mathsf{in}\ E \mid \mathcal{E}\langle \tau \overset{p}{\Rightarrow} \tau \rangle
\end{aligned}
$$

As usual there are three value forms with casts (Siek, Thiemann, and Wadler, 2015).

The notion of *ground type* was introduced by Wadler and Findler (2009) to compare types in casts, with the idea that *incompatibility between ground types is the source of all blame*. We give a definition of ground types equivalent to the one of Wadler and Findler (2009), but which uses a different notation that is more convenient when we extend the system to set-theoretic types.

B.1 DEFINITION (Grounding and ground types): For every type $\tau \in \mathsf{GType}$, we define the *grounding* of τ with respect to ?, written $\tau/?$, as follows:

$$
\begin{aligned}
b/? &= b & \alpha/? &= \alpha & ?/? &= ? \\
\tau_1 \to \tau_2/? &= ? \to ? & \tau_1 \times \tau_2/? &= ? \times ?
\end{aligned}
$$

Types τ such that $\tau \neq ?$ and that satisfy $\tau/? = \tau$ are called *ground types* and are ranged over by ρ. □

The reduction rules of Figure B.1 closely follow the presentation of Siek, Thiemann, and Wadler (2015). They are divided into two groups, the reductions for the application of casts to a value and the reductions corresponding to

Cast reductions

[ExpandL]	$V\langle \tau \overset{p}{\Rightarrow} ? \rangle$	$\hookrightarrow$	$V\langle \tau \overset{p}{\Rightarrow} \tau/? \rangle \langle \tau/? \overset{p}{\Rightarrow} ? \rangle$	if $\tau/? \neq \tau$ and $\tau \neq ?$
[ExpandR]	$V\langle ? \overset{p}{\Rightarrow} \tau \rangle$	$\hookrightarrow$	$V\langle ? \overset{p}{\Rightarrow} \tau/? \rangle \langle \tau/? \overset{p}{\Rightarrow} \tau \rangle$	if $\tau/? \neq \tau$ and $\tau \neq ?$
[CastId]	$V\langle \tau \overset{p}{\Rightarrow} \tau \rangle$	$\hookrightarrow$	V	
[Collapse]	$V\langle \rho \overset{p}{\Rightarrow} ? \rangle \langle ? \overset{q}{\Rightarrow} \rho \rangle$	$\hookrightarrow$	V	
[Blame]	$V\langle \rho \overset{p}{\Rightarrow} ? \rangle \langle ? \overset{q}{\Rightarrow} \rho' \rangle$	$\hookrightarrow$	blame q	if $\rho \neq \rho'$

Standard reductions

[CastApp]	$V\langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2' \rangle V'$	$\hookrightarrow$	$V(V'\langle \tau_1' \overset{\bar{p}}{\Rightarrow} \tau_1 \rangle) \langle \tau_2 \overset{p}{\Rightarrow} \tau_2' \rangle$	
[App]	$(\lambda^{\tau_1 \to \tau_2} x.\, E) V$	$\hookrightarrow$	$E[V/x]$	
[ProjCast]	$\pi_i\, (V\langle \tau_1 \times \tau_2 \overset{p}{\Rightarrow} \tau_1' \times \tau_2' \rangle)$	$\hookrightarrow$	$(\pi_i\, V) \langle \tau_i \overset{p}{\Rightarrow} \tau_i' \rangle$	
[Proj]	$\pi_i\, (V_1, V_2)$	$\hookrightarrow$	V_i	
[TypeApp]	$(\Lambda \vec{\alpha}.\, E)\, [\vec{t}]$	$\hookrightarrow$	$E[\vec{t}/\vec{\alpha}]$	
[Let]	let $x = V$ in E	$\hookrightarrow$	$E[V/x]$	
[Context]	$\mathcal{E}[E]$	$\hookrightarrow$	$\mathcal{E}[E']$	if $E \hookrightarrow E'$
[CtxBlame]	$\mathcal{E}[E]$	$\hookrightarrow$	blame p	if $E \hookrightarrow$ blame p

FIGURE B.1 Reduction rules of the cast language without subtyping

the elimination of type constructors. For the former we use the technique by Wadler and Findler (2009) which consists in checking whether a cast is performed between two types with the same top-level constructor and failing when this is not the case. This amounts to checking whether *grounding* the two types (by the rules [Expand_]) yields the same ground type (rule [Collapse]) or not (rule [Blame]). In regards to an implementation, the [ExpandL] rule corresponds to tagging a value with its type constructor (as done in Lisp implementations) and the [Collapse] rule corresponds to untagging a value. Most of the rules of the standard reductions group are taken from Siek, Thiemann, and Wadler (2015) too: we added the rules for type abstractions and applications, for projections, and for let bindings (all absent in the cited work). As usual, the function $\bar{\cdot}$ is involutory, that is, $\bar{\bar{p}} = p$.

The soundness of the cast language is proved via progress and subject reduction. We do not give a direct proof of these properties. They follow from the corresponding properties of the cast language with set-theoretic types of the next section (Lemmas B.3 and B.4) and the conservativity of the extension (Theorem B.7). The same holds true for the property of *blame safety* (Corollary B.6).

B.1.1 Adding subtyping

If we add subtyping to the declarative type system of the cast language as described in Section 9.4, we should also modify the semantics by changing the two rules that use type equality as follows.

[Collapse]	$V\langle \rho \overset{p}{\Rightarrow} ? \rangle \langle ? \overset{q}{\Rightarrow} \rho' \rangle$	$\hookrightarrow$	V	if $\rho \leq^? \rho'$
[Blame]	$V\langle \rho \overset{p}{\Rightarrow} ? \rangle \langle ? \overset{q}{\Rightarrow} \rho' \rangle$	$\hookrightarrow$	blame q	if $\rho \not\leq^? \rho'$

B.2 Semantics of the cast language with set-theoretic types

To add set-theoretic types to the cast language, the operational semantics must be redefined insofar as it depends on the syntax of types.

The first definition we extend is that of *grounding*. The idea is the same as in Appendix B.1: to compute an intermediate type between two types that are in the materialization relation. However, in Appendix B.1 one of these two types was always ? for non-trivial materializations (so that [Collapse] and [Blame] could then eliminate it); but now, because of type connectives, both endpoints may be different from ?. For example, the cast $\langle(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool}) \overset{p}{\Rightarrow} (\mathsf{Int} \to \mathsf{Int}) \wedge ?\rangle$ makes a transition between $\mathsf{Bool} \to \mathsf{Bool}$ and ?, which can be decomposed by first transitioning to the intermediate type $? \to ?$, as done in Appendix B.1. The intermediate type for this cast would therefore be $(\mathsf{Int} \to \mathsf{Int}) \wedge (? \to ?)$ and the endpoint $(\mathsf{Int} \to \mathsf{Int}) \wedge ?$. The intuition to generalize this idea is to apply the grounding operation of Appendix B.1 recursively under type connectives, as formalized in the following definition.

B.2 Definition (Grounding and relative ground types): For all types $\tau, \tau' \in \mathsf{GType}$ such that $\tau' \sqsubseteq \tau$, we define the *grounding* of τ with respect to τ', noted ${}^{\tau}/_{\tau'}$, as follows:

$$
\begin{array}{rlrl}
(\tau_1 \vee \tau_2)/(\tau_1' \vee \tau_2') &= (\tau_1/\tau_1') \vee (\tau_2/\tau_2') & \neg\tau/\neg\tau' &= \neg(\tau/\tau') \\
(\tau_1 \vee \tau_2)/? &= (\tau_1/?) \vee (\tau_2/?) & \neg\tau/? &= \neg(\tau/?) \\
(\tau_1 \to \tau_2)/? &= ? \to ? & (\tau_1 \times \tau_2)/? &= ? \times ? \\
b/? &= b & \mathbb{0}/? &= \mathbb{0} \\
\alpha/? &= \alpha & \tau/\tau' &= \tau' \quad \text{otherwise}
\end{array}
$$

A type τ is *ground with respect to* τ' if and only if ${}^{\tau}/_{\tau'} = \tau$. □

Note that $\tau' \sqsubseteq \tau$ is a precondition to computing ${}^{\tau}/_{\tau'}$. Therefore to ease the presentation any further reference to ${}^{\tau}/_{\tau'}$ will implicitly imply that $\tau' \sqsubseteq \tau$.

In Appendix B.1, *ground types* are types ρ such that $\rho/? = \rho$. They are "skeletons" of types whose only information is the top-level constructor. The values of the form $V\langle\rho \overset{p}{\Rightarrow} ?\rangle$ record the essence of the loss of information induced by materialization. We extend this definition to match the new definition of grounding by saying that a type τ is *ground* with respect to τ' if ${}^{\tau}/_{\tau'} = \tau$. Then, the expressions of the form $V\langle\tau \overset{p}{\Rightarrow} \tau'\rangle$ are values whenever τ is ground with respect to τ'. Intuitively, casts of this form *lose information* about the top-level constructors of a type: an example is the cast $\langle(\mathsf{Int} \to \mathsf{Int}) \wedge (? \to ?) \overset{p}{\Rightarrow} (\mathsf{Int} \to \mathsf{Int}) \wedge ?\rangle$, where we lose information about the $? \to ?$ part, which becomes ?. Once again, this kind of cast records the essence of this loss.

We have accounted for one kind of cast value, but we also need to update the definition of cast values of the form $V\langle\tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2'\rangle$ (and similarly for pairs), because function types are not necessarily syntactic arrows anymore (they can be unions and/or intersections thereof). This can be done by

considering the opposite case of the previous definition, that is, types such that $\tau/\tau' = \tau'$. Intuitively, a cast $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle$ where $\tau/\tau' = \tau'$ *does not lose or gain* information about the top-level constructors of a type: it only acts *below* the top constructors. That is, both the origin and target of such a cast have the same syntactic structure "above" constructors, the same "skeleton". For example, $\langle (\mathsf{Int} \to \mathsf{Int}) \wedge (? \to ?) \overset{p}{\Rightarrow} (\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool}) \rangle$ is such a cast.

Putting everything together, we obtain the following new definition of values:

$$
\begin{aligned}
V ::= {} & c \mid \lambda^{\tau \to \tau} x.\, E \mid (V, V) \mid \Lambda \vec{\alpha}.\, E \\
& \mid V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \qquad \text{where } \tau_1 \neq \tau_2 \\
& \qquad\qquad \text{and where } \tau_1/\tau_2 = \tau_1 \text{ or } \tau_1/\tau_2 = \tau_2 \text{ or } \tau_2/\tau_1 = \tau_1
\end{aligned}
$$

We say that a value is *unboxed* if it is not of the form $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle$. We next need to define a new operator "type" on values (except type abstractions) to resolve particular casts:

$$
\begin{aligned}
\mathsf{type}(c) &= b_c & \mathsf{type}(\lambda^{\tau_1 \to \tau_2} x.\, E) &= \tau_1 \to \tau_2 \\
\mathsf{type}((V_1, V_2)) &= \mathsf{type}(V_1) \times \mathsf{type}(V_2) & \mathsf{type}(V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle) &= \tau_2
\end{aligned}
$$

The semantics is defined by the reduction rules in Figure B.2.

The rules [ExpandL] and [ExpandR] are the immediate counterparts of the rules of the same name presented in Appendix B.1, adapted for the new grounding operator. The other rules of this group use the information provided by the grounding operator to reduce to types that can be easily compared. For example, consider $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \langle \tau_1' \overset{q}{\Rightarrow} \tau_2' \rangle$. If $\tau_1/\tau_2 = \tau_1$, then τ_1 contains all the information about type constructors which the cast lost by going into τ_2. Likewise, if $\tau_2'/\tau_1' = \tau_2'$, then all the information about type constructors is in τ_2', so the second cast *adds* constructor information. Therefore, to simplify the expressions, it suffices to compare τ_1 and τ_2', which is what is done in the rules [Collapse] and [Blame] (the set-theoretic counterparts of their namesakes in Section 9.2.3). The remaining rules for cast reductions follow the same idea, but handle cases that only arise because of set-theoretic types. For example, we can give a constant a dynamic type by subtyping (e.g., $\mathsf{Int} \leq^? \mathsf{Int} \vee ?$ implies $3 : \mathsf{Int} \vee ?$), and thus we can immediately cast the type of a constant to a more precise type, as in the expression $3\langle \mathsf{Int} \vee ? \overset{p}{\Rightarrow} \mathsf{Int} \vee (? \to ?) \rangle$. The rules [UnboxSimpl] and [UnboxBlame] handle such cases by checking if the cast can be removed. The intuition is that the dynamic part of such casts is useless since it has been introduced by subtyping.

The rules for applications and projections also need to be updated because function and product types can now be unions and intersections of arrows or products. For applications, we define a new operator, written $\circ$, which, given a function cast and the type of the argument, computes an approximation of the cast such that both its origin and target types are arrows, so that the usual rule for cast applications as in Appendix B.1 can be applied. More formally, the operation $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle \circ \tau_v$ computes a cast $\langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2' \rangle$ such that $\tau_v \leq^? \tau_1'$, $\tau_2' = \min\{\tau \mid \tau' \leq^? \tau_v \to \tau\}$, $\tau \leq^? \tau_1 \to \tau_2$, and such that the

Cast reductions

[ExpandL]	$V\langle\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$	$\hookrightarrow$	$V\langle\tau_1 \overset{p}{\Rightarrow} \tau_1/\tau_2\rangle\langle\tau_1/\tau_2 \overset{p}{\Rightarrow} \tau_2\rangle$		if $\tau_1/\tau_2 \neq \tau_1$, $\tau_1/\tau_2 \neq \tau_2$
[ExpandR]	$V\langle\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$	$\hookrightarrow$	$V\langle\tau_1 \overset{p}{\Rightarrow} \tau_2/\tau_1\rangle\langle\tau_2/\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$		if $\tau_2/\tau_1 \neq \tau_1$, $\tau_2/\tau_1 \neq \tau_2$
[CastId]	$V\langle\tau \overset{p}{\Rightarrow} \tau\rangle$	$\hookrightarrow$	V		(∗)
[Collapse]	$V\langle\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle\langle\tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle$	$\hookrightarrow$	V	if $\tau_1 \leq^? \tau_2'$,	$\tau_2'/\tau_1' = \tau_2'$ and $\tau_1/\tau_2 = \tau_1$ or $\tau_2/\tau_1 = \tau_1$
[Blame]	$V\langle\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle\langle\tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle$	$\hookrightarrow$	blame q	if $\tau_1 \not\leq^? \tau_2'$,	$\tau_2'/\tau_1' = \tau_2'$ and $\tau_1/\tau_2 = \tau_1$ or $\tau_2/\tau_1 = \tau_1$
[UpSimpl]	$V\langle\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle\langle\tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle$	$\hookrightarrow$	$V\langle\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$	if $\tau_2 \leq^? \tau_2'$,	$\tau_1/\tau_2 = \tau_2$, $\tau_2'/\tau_1' = \tau_2'$
[UpBlame]	$V\langle\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle\langle\tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle$	$\hookrightarrow$	blame q	if $\tau_2 \not\leq^? \tau_2'$,	$\tau_1/\tau_2 = \tau_2$, $\tau_2'/\tau_1' = \tau_2'$
[UnboxSimpl]	$V\langle\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$	$\hookrightarrow$	V	if $\mathsf{type}(V) \leq^? \tau_2$,	$\tau_2/\tau_1 = \tau_2$, V is unboxed
[UnboxBlame]	$V\langle\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$	$\hookrightarrow$	blame p	if $\mathsf{type}(V) \not\leq^? \tau_2$,	$\tau_2/\tau_1 = \tau_2$, V is unboxed

(∗) To ease the notation and to avoid redundant conditions, the rule [CastId] takes precedence over the following ones. All other casts are therefore considered to be non-identity casts.

Standard reductions

[CastApp]	$V\langle\tau \overset{p}{\Rightarrow} \tau'\rangle V'$	$\hookrightarrow$	$(V\ V'\langle\tau_1' \overset{\bar{p}}{\Rightarrow} \tau_1\rangle)\langle\tau_2 \overset{p}{\Rightarrow} \tau_2'\rangle$ where $\langle\tau \overset{p}{\Rightarrow} \tau'\rangle \circ \mathsf{type}(V') = \langle\tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2'\rangle$	if $\tau'/\tau = \tau$ or $\tau/\tau' = \tau'$
[CastProj]	$\pi_i\,(V\langle\tau \overset{p}{\Rightarrow} \tau'\rangle)$	$\hookrightarrow$	$(\pi_i\ V)\langle\tau_i \overset{p}{\Rightarrow} \tau_i'\rangle$ where $\langle\tau_i \overset{p}{\Rightarrow} \tau_i'\rangle = \pi_i\,(\langle\tau \overset{p}{\Rightarrow} \tau'\rangle)$	if $\tau'/\tau = \tau$ or $\tau/\tau' = \tau'$
[FailApp]	$V\langle\tau \overset{p}{\Rightarrow} \tau'\rangle V'$	$\hookrightarrow$	blame p	if $\langle\tau \overset{p}{\Rightarrow} \tau'\rangle \circ \mathsf{type}(V')$ undef.
[FailProj]	$\pi_i\,(V\langle\tau \overset{p}{\Rightarrow} \tau'\rangle)$	$\hookrightarrow$	blame p	if $\pi_i\,(\langle\tau \overset{p}{\Rightarrow} \tau'\rangle)$ undef.
[SimplApp]	$V\langle\tau \overset{p}{\Rightarrow} \tau'\rangle V'$	$\hookrightarrow$	$V\ V'$	if $\tau/\tau' = \tau$
[SimplProj]	$\pi_i\,(V\langle\tau \overset{p}{\Rightarrow} \tau'\rangle)$	$\hookrightarrow$	$\pi_i\ V$	if $\tau/\tau' = \tau$
[App]	$(\lambda^{\tau_1 \to \tau_2} x.\, E)V$	$\hookrightarrow$	$E[V/x]$	
[Proj]	$\pi_i\,(V_1, V_2)$	$\hookrightarrow$	V_i	
[TypeApp]	$(\Lambda\vec{\alpha}.\, E)\,[\vec{t}]$	$\hookrightarrow$	$E[\vec{t}/\vec{\alpha}]$	
[Let]	$\mathsf{let}\ x = V\ \mathsf{in}\ E$	$\hookrightarrow$	$E[V/x]$	
[Context]	$\mathcal{E}[E]$	$\hookrightarrow$	$\mathcal{E}[E']$	if $E \hookrightarrow E'$
[CtxBlame]	$\mathcal{E}[E]$	$\hookrightarrow$	blame p	if $E \hookrightarrow \mathsf{blame}\ p$

FIGURE B.2 Reduction rules of the cast language with set-theoretic types

materialization relation between the two parts of the cast is preserved. This ensures that the resulting approximation is still well typed. The definition of this operator is quite involved, so we present it in the next section. The most important point of this definition is that it requires both types of the cast to be syntactically identical above their constructors, which explains the presence of the grounding condition in [CastApp]. Moreover, this operator can also be undefined in some cases, such as if the origin type of the cast is not an arrow type or if the second type is empty (e.g. $\langle (? \to ?) \wedge \neg(\mathsf{Int} \to \mathsf{Int}) \overset{p}{\Rightarrow} (\mathsf{Int} \to \mathsf{Int}) \wedge \neg(\mathsf{Int} \to \mathsf{Int}) \rangle$). Such ill-formed casts are handled by [FailApp]. We apply the same idea to projections and define an operator, written π_i, that computes an approximation of the first or second component of a cast between two product types. This yields the rules [CastProj] and [FailProj]. The two remaining rules, [SimplApp] and [SimplProj], handle cases that only appear due to the presence of set-theoretic types. For instance, it is now possible to apply (or project) a value that has a dynamic type: $V\langle (\mathsf{Int} \to \mathsf{Int}) \wedge (? \to ?) \overset{p}{\Rightarrow} (\mathsf{Int} \to \mathsf{Int}) \wedge ? \rangle V'$. Here, by subtyping, the function has both type $\mathsf{Int} \to \mathsf{Int}$ and $?$, so it can be applied but it is also dynamic. We show that such casts are unnecessary and can be harmlessly removed; the rules [SimplApp] and [SimplProj] do just that.

We next state the usual type soundness lemmas and theorems for this cast language.

B.3 Lemma (Progress): For every term E such that $\varnothing \vdash E : \forall \vec{\alpha}.\tau$, either there exists a value V such that $E = V$, or there exists a term E' such that $E \hookrightarrow E'$, or there exists a label p such that $E \hookrightarrow \mathsf{blame}\, p$. □

B.4 Lemma (Subject reduction): For all terms E, E' and every context Γ, if $\Gamma \vdash E : \forall \vec{\alpha}.\tau$ and $E \hookrightarrow E'$, then $\Gamma \vdash E' : \forall \vec{\alpha}.\tau$. □

B.5 Theorem (Soundness): For every term E such that $\varnothing \vdash E : \forall \vec{\alpha}.\tau$, either there exists a value V such that $E \hookrightarrow^* V$, or there exists a label p such that $E \hookrightarrow^* \mathsf{blame}\, p$, or E diverges. □

Another result for our language is *blame safety* (Tobin-Hochstadt and Felleisen, 2006; Wadler and Findler, 2009), which guarantees that the statically typed part of a program cannot be blamed. In our system, recall that the typing rules that we presented in Section 9.2 enforce the correspondence between the polarity of the label of a cast and the direction of materialization. That is, we only have casts of the form $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle$ where $\tau' \sqsubseteq \tau$ (i.e., $\tau <:_n \tau'$) for a negative p and $\tau \sqsubseteq \tau'$ (i.e., $\tau' <:_n \tau$) for a positive p. Since all this information is encoded in the typing rules, blame safety is a corollary of Lemma B.4, and can be stated without resorting to positive and negative subtyping:

B.6 Corollary (Blame safety): For every term E such that $\varnothing \vdash E : \forall \vec{\alpha}.\tau$, and every blame label ℓ, $E \not\hookrightarrow^* \mathsf{blame}\, \bar{\ell}$. □

Lastly, an important aspect of the cast language defined in this section is that it is a conservative extension of the cast language defined in Section 9.4; this justifies the choice of the reduction rules. Denoting by Sub the system defined in Section 9.4 and Appendix B.1 and by Set the system defined in this section, there is a strong bisimulation relation between Set and Sub, as stated by the following result.

B.7 THEOREM (Conservativity): For every term E such that $\varnothing \vdash_{\text{Sub}} E : \tau$:

$$E \hookrightarrow_{\text{Sub}} E' \iff E \hookrightarrow_{\text{Set}} E'$$
$$E \hookrightarrow_{\text{Sub}} \mathsf{blame}\, p \iff E \hookrightarrow_{\text{Set}} \mathsf{blame}\, p$$ □

B.2.1 Defining cast application and projection operators

We refer to a type frame of the form b, $T_1 \times T_2$, or $T_1 \to T_2$ as an *atom*. We write $\mathsf{Atom}_{\mathsf{basic}}$, $\mathsf{Atom}_{\mathsf{prod}}$, and $\mathsf{Atom}_{\mathsf{fun}}$ for the set of type frames of the forms b, $T_1 \times T_2$, and $T_1 \to T_2$, respectively. In the following, we use the metavariable a to range over the set $\mathsf{Atom}_{\mathsf{basic}} \cup \mathsf{Atom}_{\mathsf{prod}} \cup \mathsf{Atom}_{\mathsf{fun}} \cup \mathsf{Var}$.

B.8 DEFINITION (Uniform normal form): A *uniform (disjunctive) normal form* (UDNF) is a type frame T of the form

$$\bigvee_{i \in I} \left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right)$$

such that, for all $i \in I$, one of the following three condition holds:

- $P_i \cap \mathsf{Atom}_{\mathsf{basic}} \neq \varnothing$ and $(P_i \cup N_i) \cap (\mathsf{Atom}_{\mathsf{prod}} \cup \mathsf{Atom}_{\mathsf{fun}}) = \varnothing$;
- $P_i \cap \mathsf{Atom}_{\mathsf{prod}} \neq \varnothing$ and $(P_i \cup N_i) \cap (\mathsf{Atom}_{\mathsf{basic}} \cup \mathsf{Atom}_{\mathsf{fun}}) = \varnothing$;
- $P_i \cap \mathsf{Atom}_{\mathsf{fun}} \neq \varnothing$ and $(P_i \cup N_i) \cap (\mathsf{Atom}_{\mathsf{basic}} \cup \mathsf{Atom}_{\mathsf{prod}}) = \varnothing$. □

We define here a function $\mathsf{UDNF}(T)$ which, given a type frame T, produces a uniform normal form that is equivalent to T.

We first define two mutually recursive functions $\mathcal{N}$ and $\mathcal{N}'$ on type frames. These are inductive definitions as no recursive uses of the functions occur below type constructors.

$$\mathcal{N}(a) = a$$
$$\mathcal{N}(T_1 \vee T_2) = \mathcal{N}(T_1) \vee \mathcal{N}(T_2)$$
$$\mathcal{N}(\neg T) = \mathcal{N}'(T)$$
$$\mathcal{N}(\mathbb{0}) = \mathbb{0}$$

$$\begin{aligned}
\mathcal{N}'(a) &= \neg a \\
\mathcal{N}'(T_1 \vee T_2) &= \bigvee_{i\in I, j\in J} \Big(\bigwedge_{a\in P_i\cup P_j} a \wedge \bigwedge_{a\in N_i\cup N_j} \neg a \Big) \\
&\qquad \text{where } \mathcal{N}'(T_1) = \textstyle\bigvee_{i\in I} \big(\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a \big) \\
&\qquad \text{and } \mathcal{N}'(T_2) = \textstyle\bigvee_{j\in J} \big(\bigwedge_{a\in P_j} a \wedge \bigwedge_{a\in N_j} \neg a \big) \\
\mathcal{N}'(\neg T) &= \mathcal{N}(T) \\
\mathcal{N}'(\mathbb{0}) &= \mathbb{1}
\end{aligned}$$

In the definition above, we see $\mathbb{0}$ as the empty union $\bigvee_{i\in\varnothing} T_i$ and $\mathbb{1}$ as the singleton union of the empty intersection $\bigvee_{i\in\{i_0\}} \bigwedge_{a\in\varnothing} a$.

The first step in the computation of $\mathsf{UDNF}(T)$ is to compute $\mathcal{N}(T)$. Then, assuming

$$\mathcal{N}(T) = \bigvee_{i\in I} \Big(\underbrace{\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a}_{\mathcal{I}_i} \Big)$$

we define

$$\mathsf{UDNF}(T) \stackrel{\text{def}}{=} \bigvee_{i\in I} \mathcal{I}_i^{\mathsf{base}} \vee \bigvee_{i\in I} \mathcal{I}_i^{\mathsf{prod}} \vee \bigvee_{i\in I} \mathcal{I}_i^{\mathsf{fun}}$$

where

$$\begin{aligned}
\mathcal{I}_i^{\mathsf{base}} &\stackrel{\text{def}}{=} \mathbb{1}_{\mathsf{B}} \wedge \bigwedge_{a\in P_i\cap(\mathsf{Atom}_{\mathsf{basic}}\cup\mathsf{Var})} a \wedge \bigwedge_{a\in N_i\cap(\mathsf{Atom}_{\mathsf{basic}}\cup\mathsf{Var})} \neg a \\
\mathcal{I}_i^{\mathsf{prod}} &\stackrel{\text{def}}{=} (\mathbb{1}\times\mathbb{1}) \wedge \bigwedge_{a\in P_i\cap(\mathsf{Atom}_{\mathsf{prod}}\cup\mathsf{Var})} a \wedge \bigwedge_{a\in N_i\cap(\mathsf{Atom}_{\mathsf{prod}}\cup\mathsf{Var})} \neg a \\
\mathcal{I}_i^{\mathsf{fun}} &\stackrel{\text{def}}{=} (\mathbb{0}\to\mathbb{1}) \wedge \bigwedge_{a\in P_i\cap(\mathsf{Atom}_{\mathsf{fun}}\cup\mathsf{Var})} a \wedge \bigwedge_{a\in N_i\cap(\mathsf{Atom}_{\mathsf{fun}}\cup\mathsf{Var})} \neg a
\end{aligned}$$

B.9 DEFINITION (Product decomposition and projections): Given a type frame $T \le \mathbb{1}\times\mathbb{1}$, we define its decomposition $\pi(T)$ as

$$\pi(T) \stackrel{\text{def}}{=} \bigcup_{i\in I,\, \mathcal{I}_i\not\le\mathbb{0}} \Big\{ \Big(\underbrace{\bigwedge_{T_1\times T_2\in\overline{P}_i} T_1 \wedge \bigwedge_{T_1\times T_2\in N'} \neg T_1}_{\overline{T}_1}, \underbrace{\bigwedge_{T_1\times T_2\in\overline{P}_i} T_2 \wedge \bigwedge_{T_1\times T_2\in\overline{N}_i\setminus N'} \neg T_2}_{\overline{T}_2} \Big) \;\Big|\; N' \subseteq \overline{N}_i, \overline{T}_1 \not\le \mathbb{0}, \overline{T}_2 \not\le \mathbb{0} \Big\}$$

and its i-th projection $\pi_i(T)$ as

$$\pi_i(T) \stackrel{\text{def}}{=} \bigvee_{(T_1,T_2)\in\pi(T)} T_i$$

where

$$\mathsf{UDNF}(T) = \bigvee_{i\in I} \underbrace{\Big(\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a \Big)}_{\mathcal{I}_i}$$

and where $\overline{P}_i = P_i \cap \mathsf{Atom}_{\mathsf{prod}}$ and $\overline{N}_i = N_i \cap \mathsf{Atom}_{\mathsf{prod}}$. □

We now extend the previous definition of atoms to gradual types. That is, we refer to a gradual type of the form b, $\tau_1 \times \tau_2$, or $\tau_1 \to \tau_2$ as an atom. We write $\mathsf{Atom}^?_{\mathsf{basic}}$, $\mathsf{Atom}^?_{\mathsf{prod}}$, and $\mathsf{Atom}^?_{\mathsf{fun}}$ for the set of gradual types of the forms b, $\tau_1 \times \tau_2$, and $\tau_1 \to \tau_2$, respectively.

In the following, the metavariable a ranges over the set $\mathsf{Atom}^?_{\mathsf{basic}} \cup \mathsf{Atom}^?_{\mathsf{prod}} \cup \mathsf{Atom}^?_{\mathsf{fun}} \cup \mathsf{Var} \cup \{?\}$.

B.10 DEFINITION (Uniform gradual normal form): A *uniform gradual (disjunctive) normal form* (UGDNF) is a gradual type τ of the form

$$\bigvee_{i \in I} \left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right)$$

such that, for all $i \in I$, one of the following three condition holds:

- $P_i \cap \mathsf{Atom}^?_{\mathsf{basic}} \neq \emptyset$ and $(P_i \cup N_i) \cap (\mathsf{Atom}^?_{\mathsf{prod}} \cup \mathsf{Atom}^?_{\mathsf{fun}}) = \emptyset$;
- $P_i \cap \mathsf{Atom}^?_{\mathsf{prod}} \neq \emptyset$ and $(P_i \cup N_i) \cap (\mathsf{Atom}^?_{\mathsf{basic}} \cup \mathsf{Atom}^?_{\mathsf{fun}}) = \emptyset$;
- $P_i \cap \mathsf{Atom}^?_{\mathsf{fun}} \neq \emptyset$ and $(P_i \cup N_i) \cap (\mathsf{Atom}^?_{\mathsf{basic}} \cup \mathsf{Atom}^?_{\mathsf{prod}}) = \emptyset$. □

For every type τ, we define $\mathsf{UGDNF}(\tau) = (\mathsf{UDNF}(\tau^{\oplus}))^{\dagger}$.

In the following, we use ς as an additional metavariable for gradual types.

B.11 DEFINITION (Function cast approximation): For every pair of types τ, τ' such that $\tau' \leq^? \mathbb{0} \to \mathbb{1}$, and every type ς, if

(1) $\mathsf{UGDNF}(\tau) = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n}_{\mathcal{I}_i}$

(2) $\mathsf{UGDNF}(\tau') = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a'_p \wedge \bigwedge_{n \in N_i} \neg a'_n}_{\mathcal{I}'_i}$

(3) $\forall i \in I.\, \mathcal{I}_i \not\leq^? \mathbb{0} \implies \mathcal{I}'_i \not\leq^? \mathbb{0}$

(4) $\forall i \in I. \forall p \in P_i.\, a_p \in \mathsf{Atom}^?_{\mathsf{fun}} \iff a'_p \in \mathsf{Atom}^?_{\mathsf{fun}}$

then we define the approximation of $\langle \tau \overset{p}{\Rrightarrow} \tau' \rangle$ applied to ς, noted $\langle \tau \overset{p}{\Rrightarrow} \tau' \rangle \circ \varsigma$ as follows.

$$\langle \tau \overset{p}{\Rrightarrow} \tau' \rangle \circ \varsigma = \Bigg\langle \bigwedge_{\substack{i \in I \\ \mathcal{I}'_i \not\leq^? \mathbb{0}}} \bigwedge_{\substack{S \subseteq \bar{P}_i \\ \varsigma \leq^? \bigvee_{p \in S} \varsigma'_p}} \bigvee_{p \in S} \varsigma_p \to \bigvee_{\substack{i \in I \\ \mathcal{I}'_i \not\leq^? \mathbb{0}}} \bigvee_{\substack{S \subseteq \bar{P}_i \\ \varsigma \not\leq^? \bigvee_{p \in S} \varsigma'_p}} \bigwedge_{p \in \bar{P}_i \setminus S} \tau_p \overset{p}{\Rrightarrow} \bigwedge_{\substack{i \in I \\ \mathcal{I}'_i \not\leq^? \mathbb{0}}} \bigwedge_{\substack{S \subseteq \bar{P}_i \\ \varsigma \leq^? \bigvee_{p \in S} \varsigma'_p}} \bigvee_{p \in S} \varsigma'_p \to \bigvee_{\substack{i \in I \\ \mathcal{I}'_i \not\leq^? \mathbb{0}}} \bigvee_{\substack{S \subseteq \bar{P}_i \\ \varsigma \not\leq^? \bigvee_{p \in S} \varsigma'_p}} \bigwedge_{p \in \bar{P}_i \setminus S} \tau'_p \Bigg\rangle$$

where, to ease the notation, we pose

$$\bar{P}_i = \{p \in P_i \mid a_p \in \mathsf{Atom}^?_{\mathsf{fun}}\} = \{p \in P_i \mid a'_p \in \mathsf{Atom}^?_{\mathsf{fun}}\}$$

and for every $p \in \bar{P}_i$, $a_p = \varsigma_p \to \tau_p$ and $a'_p = \varsigma'_p \to \tau'_p$.

Otherwise, $\langle \tau \overset{p}{\Rrightarrow} \tau' \rangle \circ \varsigma$ is *undefined*. □

B.12 DEFINITION (Cast projection): For every pair of types τ, τ' such that $\tau' \leq^? \mathbb{1} \times \mathbb{1}$, if

(1) $\mathsf{UGDNF}(\tau) = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n}_{\mathcal{I}_i}$

(2) $\mathsf{UGDNF}(\tau') = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a'_p \wedge \bigwedge_{n \in N_i} \neg a'_n}_{\mathcal{I}'_i}$

(3) $\forall i \in I.\, \mathcal{I}_i \not\leq^? \mathbb{0} \implies \mathcal{I}'_i \not\leq^? \mathbb{0}$

(4) $\forall j \in I. \forall N \subseteq \bar{N}_j. \forall i \in \{1,2\}.\, \pi_i\,(\tau^j_N) \not\leq^? \mathbb{0} \implies \pi_i\,(\tau'^j_N) \not\leq^? \mathbb{0}$

(5) $\forall i \in I. \forall p \in P_i.\, a_p \in \mathsf{Atom}^?_{\mathsf{prod}} \iff a'_p \in \mathsf{Atom}^?_{\mathsf{prod}}$

(6) $\forall i \in I. \forall n \in N_i.\, a_n \in \mathsf{Atom}^?_{\mathsf{prod}} \iff a'_n \in \mathsf{Atom}^?_{\mathsf{prod}}$

then we define the i-th projection of $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle$, noted $\pi_i\,(\langle \tau \overset{p}{\Rightarrow} \tau' \rangle)$ as follows.

$$\pi_i\,(\langle \tau \overset{p}{\Rightarrow} \tau' \rangle) = \left\langle \bigvee_{\substack{j \in I \\ \mathcal{I}'_j \not\leq^? \mathbb{0}}} \bigvee_{\substack{N \subseteq \bar{N}_j \\ \pi_1(\tau'^j_N) \not\leq^? \mathbb{0} \\ \pi_2(\tau'^j_N) \not\leq^? \mathbb{0}}} \pi_i\,(\tau^j_N) \overset{p}{\Rightarrow} \bigvee_{\substack{j \in I \\ \mathcal{I}'_j \not\leq^? \mathbb{0}}} \bigvee_{\substack{N \subseteq \bar{N}_j \\ \pi_1(\tau'^j_N) \not\leq^? \mathbb{0} \\ \pi_2(\tau'^j_N) \not\leq^? \mathbb{0}}} \pi_i\,(\tau'^j_N) \right\rangle$$

where

$$\bar{P}_i = \{p \in P_i \mid a_p \in \mathsf{Atom}^?_{\mathsf{prod}}\} = \{p \in P_i \mid a'_p \in \mathsf{Atom}^?_{\mathsf{prod}}\}$$

$$\bar{N}_i = \{n \in N_i \mid a_n \in \mathsf{Atom}^?_{\mathsf{prod}}\} = \{n \in N_i \mid a'_n \in \mathsf{Atom}^?_{\mathsf{prod}}\}$$

$$\tau^i_N = \left(\bigwedge_{\substack{p \in \bar{P}_i \\ a_p = \tau_1 \times \tau_2}} \tau_1 \wedge \bigwedge_{\substack{n \in N \\ a_n = \tau_1 \times \tau_2}} \neg \tau_1,\ \bigwedge_{\substack{p \in \bar{P}_i \\ a_p = \tau_1 \times \tau_2}} \tau_2 \wedge \bigwedge_{\substack{n \in N_i \setminus N \\ a_n = \tau_1 \times \tau_2}} \neg \tau_2 \right)$$

$$\tau'^i_N = \left(\bigwedge_{\substack{p \in P_i \\ a'_p = \tau'_1 \times \tau'_2}} \tau'_1 \wedge \bigwedge_{\substack{n \in N \\ a'_n = \tau'_1 \times \tau'_2}} \neg \tau'_1,\ \bigwedge_{\substack{p \in P_i \\ a'_p = \tau'_1 \times \tau'_2}} \tau'_2 \wedge \bigwedge_{\substack{n \in N_i \setminus N \\ a'_n = \tau'_1 \times \tau'_2}} \neg \tau'_2 \right)$$

otherwise, $\pi_i\,(\langle \tau \overset{p}{\Rightarrow} \tau' \rangle)$ is *undefined*. □

Bibliography

Aiken, Alexander and Edward L. Wimmers (1993). Type inclusion constraints and type inference. In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. FPCA '93. ACM, pp. 31–41. DOI: 10.1145/165180.165188. Cited on pp. 139, 177.

Aiken, Alexander, Edward L. Wimmers, and T. K. Lakshman (1994). Soft typing with conditional types. In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '94. ACM, pp. 163–173. DOI: 10.1145/174675.177847. Cited on p. 139.

Ancona, Davide and Andrea Corradi (2016). Semantic subtyping for imperative object-oriented languages. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016. ACM, pp. 568–587. DOI: 10.1145/2983990.2983992. Cited on pp. 31, 241.

Ângelo, Pedro and Mário Florido (2018). Gradual intersection types. In: *Workshop on Intersection Types and Related Systems*. Cited on p. 202.

Ariola, Zena M. and Matthias Felleisen (1997). The call-by-need lambda calculus. In: *Journal of Functional Programming* 7.3, pp. 265–301. Cited on pp. 216, 218, 219, 221.

Ariola, Zena M., John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler (1995). A call-by-need lambda calculus. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. ACM, pp. 233–246. DOI: 10.1145/199448.199507. Cited on pp. 216, 221.

Barbanera, Franco, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro (1995). Intersection and union types: syntax and semantics. In: *Information and Computation* 119.2, pp. 202–230. DOI: 10.1006/inco.1995.1086. Cited on p. 139.

Barendregt, Henk, Mario Coppo, and Mariangiola Dezani-Ciancaglini (1983). A filter lambda model and the completeness of type assignment. In: *Journal of Symbolic Logic* 48.4, pp. 931–940. DOI: 10.2307/2273659. Cited on p. 139.

Benzaken, Véronique, Giuseppe Castagna, and Alain Frisch (2003). ℂDuce: an XML-centric general-purpose language. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*. ICFP '03. ACM, pp. 51–63. DOI: 10.1145/944705.944711. Cited on pp. 31, 241.

Benzaken, Véronique, Giuseppe Castagna, Kim Nguyễn, and Jérôme Siméon (2013). Static and dynamic semantics of NoSQL languages. In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '13. ACM, pp. 101–114. DOI: 10.1145/2429069.2429083. Cited on pp. 31, 241.

Bierman, Gavin M., Martín Abadi, and Mads Torgersen (2014). Understanding TypeScript. In: *ECOOP 2014 – Object-Oriented Programming*. Springer Berlin Heidelberg, pp. 257–281. Cited on pp. 140, 202.

Bierman, Gavin M., Erik Meijer, and Mads Torgersen (2007). Lost in translation: formalizing proposed extensions to C♯. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. OOPSLA '07. ACM, pp. 479–498. DOI: 10.1145/1297027.1297063. Cited on p. 140.

Blume, Matthias, Umut A. Acar, and Wonseok Chae (2006). Extensible programming with first-class cases. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*. ICFP '06. ACM, pp. 239–250. DOI: 10.1145/1159803.1159836. Cited on pp. 134, 135.

Campora, John Peter, Sheng Chen, Martin Erwig, and Eric Walkingshaw (2017). Migrating gradual types. In: *Proceedings of the ACM on Programming Languages* 2.POPL, 15:1–15:29. DOI: 10.1145/3158103. Cited on p. 202.

Capretta, Venanzio (2005). General recursion via coinductive types. In: *Logical Methods in Computer Science* Volume 1, Issue 2. DOI: 10.2168/LMCS-1(2:1)2005. Cited on p. 212.

Cartwright, Robert and Mike Fagan (1991). Soft typing. In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI '91. ACM, pp. 278–292. DOI: 10.1145/113445.113469. Cited on p. 139.

Castagna, Giuseppe, Rocco De Nicola, and Daniele Varacca (2008). Semantic subtyping for the pi-calculus. In: *Theoretical Computer Science* 398.1-3, pp. 217–242. DOI: 10.1016/j.tcs.2008.01.049. Cited on pp. 31, 241.

Castagna, Giuseppe and Alain Frisch (2005). A gentle introduction to semantic subtyping. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '05. ACM, pp. 198–199. DOI: 10.1145/1069774.1069793. Cited on p. 211.

Castagna, Giuseppe, Hyeonseung Im, Kim Nguyễn, and Véronique Benzaken (2015a). A core calculus for XQuery 3.0. In: *Programming Languages and Systems*. Springer Berlin Heidelberg, pp. 232–256. Cited on pp. 31, 241.

Castagna, Giuseppe, Kim Nguyễn, Zhiwu Xu, and Pietro Abate (2015b). Polymorphic functions with set-theoretic types. Part 2: local type inference and type reconstruction. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. ACM, pp. 289–302. DOI: 10.1145/2676726.2676991. Cited on pp. 9, 10, 28, 31, 32, 61, 87, 90, 108, 109, 132, 137, 141, 179, 194, 241.

Castagna, Giuseppe, Kim Nguyễn, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani (2014). Polymorphic functions with set-theoretic types. Part 1: syntax, semantics, and evaluation. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. ACM, pp. 5–17. DOI: 10.1145/2535838.2535840. Cited on pp. 9, 31, 32, 61, 131, 137, 238, 241.

Castagna, Giuseppe and Victor Lanvin (2017). Gradual typing with union and intersection types. In: *Proceedings of the ACM on Programming Languages* 1.ICFP, 41:1–41:28. DOI: 10.1145/3110285. Cited on pp. 145, 146, 148, 202–204.

Castagna, Giuseppe, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek (2019). Gradual typing: a new perspective. In: *Proceedings of the ACM on Programming Languages* 3.POPL, 16:1–16:32. DOI: 10.1145/3290329. Cited on pp. 147, 149, 158, 313.

Castagna, Giuseppe, Tommaso Petrucciani, and Kim Nguyễn (2016). Set-theoretic types for polymorphic variants. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. ACM, pp. 378–391. DOI: 10.1145/2951913.2951928. Cited on pp. 34, 90, 132, 134, 163.

Castagna, Giuseppe and Zhiwu Xu (2011). Set-theoretic foundation of parametric polymorphism and subtyping. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. ACM, pp. 94–106. DOI: 10.1145/2034773.2034788. Cited on pp. 31, 39, 44–46, 50, 53, 241.

Chaudhuri, Avik, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi (2017). Fast and precise type checking for JavaScript. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA, 48:1–48:30. DOI: 10.1145/3133872. Cited on pp. 27, 131.

Chugh, Ravi, Patrick M. Rondon, and Ranjit Jhala (2012). Nested refinements: a logic for duck typing. In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. ACM, pp. 231–244. DOI: 10.1145/2103656.2103686. Cited on p. 70.

Cimini, Matteo and Jeremy G. Siek (2016). The Gradualizer: a methodology and algorithm for generating gradual type systems. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. ACM, pp. 443–455. DOI: 10.1145/2837614.2837632. Cited on p. 147.

Constable, Robert L. and Scott Fraser Smith (1987). Partial objects in constructive type theory. In: *IEEE Symposium on Logic in Computer Science (LICS)*, pp. 183–193. Cited on p. 212.

Coppo, Mario and Mariangiola Dezani-Ciancaglini (1980). An extension of the basic functionality theory for the λ-calculus. In: *Notre Dame Journal of Formal Logic* 21.4, pp. 685–693. DOI: 10.1305/ndjfl/1093883253. Cited on pp. 87, 139.

Dardha, Ornela, Daniele Gorla, and Daniele Varacca (2013). Semantic subtyping for objects and classes. In: *Formal Techniques for Distributed Systems*. Springer Berlin Heidelberg, pp. 66–82. Cited on pp. 31, 241.

Davies, Rowan (2005). *Practical refinement-type checking*. PhD thesis. Carnegie Mellon University. Cited on pp. 130, 139.

Dolan, Stephen (2016). *Algebraic subtyping*. PhD thesis. University of Cambridge. Cited on pp. 89, 92, 93, 138.

Dolan, Stephen and Alan Mycroft (2017). Polymorphism, subtyping, and type inference in MLsub. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. ACM, pp. 60–72. DOI: 10.1145/3009837.3009882. Cited on pp. 10, 32, 87, 89, 91–93, 100, 138–140, 177.

Dolstra, Eelco and Andres Löh (2008). NixOS: a purely functional Linux distribution. In: *Proceedings of the 13th ACM SIGPLAN International Conference on*

Functional Programming. ICFP '08. ACM, pp. 367–378. DOI: 10.1145/1411204.1411255. Cited on p. 207.

Dunfield, Joshua (2007). *A unified system of type refinements*. PhD thesis. Carnegie Mellon University. Cited on pp. 130, 139, 212.

Dunfield, Joshua and Neelakantan R. Krishnaswami (2013). Complete and easy bidirectional typechecking for higher-rank polymorphism. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. ACM, pp. 429–442. DOI: 10.1145/2500365.2500582. Cited on p. 140.

Dunfield, Joshua and Frank Pfenning (2003). Type assignment for intersections and unions in call-by-value languages. In: *Foundations of Software Science and Computation Structures*. Springer Berlin Heidelberg, pp. 250–266. Cited on p. 212.

Facebook (2018). *Flow documentation*. Available at https://flow.org/en/docs/. Cited on p. 26.

Freeman, Tim and Frank Pfenning (1991). Refinement types for ML. In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI '91. ACM, pp. 268–277. DOI: 10.1145/113445.113468. Cited on p. 139.

Frisch, Alain (2004). *Théorie, conception et réalisation d'un langage de programmation adapté à XML*. PhD thesis. Université Paris 7 – Denis Diderot. Cited on pp. 55, 132, 135.

Frisch, Alain, Giuseppe Castagna, and Véronique Benzaken (2008). Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. In: *Journal of the ACM* 55.4, 19:1–19:64. DOI: 10.1145/1391289.1391293. Cited on pp. 9, 10, 12, 25, 30–32, 34, 39, 41–43, 50, 55, 61, 68, 131, 137, 207, 210, 216, 222, 223, 227, 228, 233–236, 241, 308, 311.

Fuh, You-Chin and Prateek Mishra (1988). Type inference with subtypes. In: *ESOP '88*. Springer Berlin Heidelberg, pp. 94–114. Cited on p. 139.

Garcia, Ronald (2013). Calculating threesomes, with blame. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. ACM, pp. 417–428. DOI: 10.1145/2500365.2500603. Cited on pp. 147, 154.

Garcia, Ronald and Matteo Cimini (2015). Principal type schemes for gradual programs. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. ACM, pp. 303–315. DOI: 10.1145/2676726.2676992. Cited on pp. 152, 158, 163, 202.

Garcia, Ronald, Alison M. Clark, and Éric Tanter (2016). Abstracting Gradual Typing. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. ACM, pp. 429–442. DOI: 10.1145/2837614.2837670. Cited on pp. 147, 148, 202, 203.

Garrigue, Jacques (2002). Simple type inference for structural polymorphism. In: *International Workshop on Foundations of Object-Oriented Languages (FOOL)*. Informal proceedings. Cited on p. 134.

Garrigue, Jacques (2015). A certified implementation of ML with structural polymorphism and recursive types. In: *Mathematical Structures in Computer Science* 25.4, pp. 867–891. DOI: 10.1017/S0960129513000066. Cited on p. 134.

Gesbert, Nils, Pierre Genevès, and Nabil Layaïda (2011). Parametric polymorphism and semantic subtyping: the logical connection. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. ACM, pp. 107–116. DOI: 10.1145/2034773.2034789. Cited on pp. 31, 45.

Gesbert, Nils, Pierre Genevès, and Nabil Layaïda (2015). A logical approach to deciding semantic subtyping. In: *ACM Transactions on Programming Languages and Systems* 38.1, p. 3. DOI: 10.1145/2812805. Cited on pp. 39, 45, 46, 50, 51.

Henglein, Fritz (1994). Dynamic typing: syntax and proof theory. In: *Science of Computer Programming* 22.3, pp. 197–230. DOI: 10.1016/0167-6423(94)00004-2. Cited on p. 154.

Hosoya, Haruo, Alain Frisch, and Giuseppe Castagna (2009). Parametric polymorphism for XML. In: *ACM Transactions on Programming Languages and Systems* 32.1, 2:1–2:56. DOI: 10.1145/1596527.1596529. Cited on p. 44.

Hosoya, Haruo and Benjamin C. Pierce (2003). XDuce: a statically typed XML processing language. In: *ACM Trans. Internet Technol.* 3.2, pp. 117–148. DOI: 10.1145/767193.767195. Cited on pp. 31, 241.

Hosoya, Haruo, Jérôme Vouillon, and Benjamin C. Pierce (2005). Regular expression types for XML. In: *ACM Transactions on Programming Languages and Systems* 27.1, pp. 46–90. DOI: 10.1145/1053468.1053470. Cited on p. 41.

Ina, Lintaro and Atsushi Igarashi (2011). Gradual typing for generics. In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. ACM, pp. 609–624. DOI: 10.1145/2048066.2048114. Cited on p. 202.

Jafery, Khurram A. and Joshua Dunfield (2017). Sums of uncertainty: refinements go gradual. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. ACM, pp. 804–817. DOI: 10.1145/3009837.3009865. Cited on p. 202.

JetBrains (2018). *Kotlin documentation*. Available at http://kotlinlang.org/docs/reference. Cited on p. 27.

Kfoury, A. J. and J. B. Wells (2004). Principality and type inference for intersection types using expansion variables. In: *Theoretical Computer Science* 311.1-3, pp. 1–70. DOI: 10.1016/j.tcs.2003.10.032. Cited on p. 139.

King, Gavin (2017). *The Ceylon language specification, version 1.3*. Available at https://ceylon-lang.org/documentation/1.3/spec. Cited on p. 27.

Lehmann, Nico and Éric Tanter (2017). Gradual refinement types. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. ACM, pp. 775–788. DOI: 10.1145/3009837.3009856. Cited on p. 202.

Maidl, André Murbach, Fabio Mascarenhas, and Roberto Ierusalimschy (2014). Typed Lua: an optional type system for Lua. In: *Proceedings of the Workshop*

on Dynamic Languages and Applications. Dyla'14. ACM, 3:1–3:10. DOI: 10.1145/2617548.2617553. Cited on p. 202.

Maraist, John, Martin Odersky, and Philip Wadler (1998). The call-by-need lambda calculus. In: *Journal of Functional Programming* 8.3, pp. 275–317. Cited on pp. 216, 219, 221.

Martelli, Alberto and Ugo Montanari (1982). An efficient unification algorithm. In: *ACM Transactions on Programming Languages and Systems* 4.2, pp. 258–282. DOI: 10.1145/357162.357169. Cited on p. 164.

Microsoft (2018). *The TypeScript handbook*. Available at https://www.typescriptlang.org/docs/handbook/basic-types.html. Cited on p. 26.

Miller, Dale (1992). Unification under a mixed prefix. In: *Journal of Symbolic Computation* 14.4, pp. 321–358. DOI: https://doi.org/10.1016/0747-7171(92)90011-R. Cited on pp. 129, 139.

Mitchell, John C. (1991). Type inference with simple subtypes. In: *Journal of Functional Programming* 1.3, pp. 245–285. DOI: 10.1017/S0956796800000113. Cited on p. 139.

Muehlboeck, Fabian and Ross Tate (2018). Empowering union and intersection types with integrated subtyping. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA, 112:1–112:29. DOI: 10.1145/3276482. Cited on pp. 27, 139.

Odersky, Martin and Konstantin Läufer (1996). Putting Type Annotations to Work. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. ACM, pp. 54–67. DOI: 10.1145/237721.237729. Cited on p. 139.

Odersky, Martin, Christoph Zenger, and Matthias Zenger (2001). Colored local type inference. In: *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '01. ACM, pp. 41–53. DOI: 10.1145/360204.360207. Cited on p. 140.

Ohori, Atsushi (1995). A polymorphic record calculus and its compilation. In: *ACM Transactions on Programming Languages and Systems* 17.6, pp. 844–895. DOI: 10.1145/218570.218572. Cited on p. 134.

Okasaki, Chris (1998). *Purely Functional Data Structures*. Cambridge University Press. DOI: 10.1017/CBO9780511530104. Cited on p. 29.

Ortin, Francisco and Miguel García (2011). Union and intersection types to support both dynamic and static typing. In: *Information Processing Letters* 111.6, pp. 278–286. DOI: 10.1016/j.ipl.2010.12.006. Cited on p. 202.

Pearce, David J. (2013). Sound and complete flow typing with unions, intersections and negations. In: *Verification, Model Checking, and Abstract Interpretation*. Springer, pp. 335–354. Cited on pp. 27, 131.

Pearce, David J. and Lindsay Groves (2013). Whiley: a platform for research in software verification. In: *Software Language Engineering*. Springer International Publishing, pp. 238–248. Cited on p. 27.

Peyton Jones, Simon, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields (2007). Practical type inference for arbitrary-rank types. In: *Journal of Functional Programming* 17.1, pp. 1–82. DOI: 10.1017/S0956796806006034. Cited on pp. 139, 140.

Pierce, Benjamin C. (1991). *Programming with intersection types and bounded polymorphism*. PhD thesis. Carnegie Mellon University. Cited on p. 130.

Pierce, Benjamin C. (2002). *Types and programming languages*. MIT Press. Cited on pp. 64, 87.

Pierce, Benjamin C. and David N. Turner (2000). Local type inference. In: *ACM Transactions on Programming Languages and Systems* 22.1, pp. 1–44. DOI: 10.1145/345099.345100. Cited on pp. 139, 141.

Pottier, François (1998). *Type inference in the presence of subtyping: from theory to practice*. Research Report 3483. INRIA. Cited on pp. 89, 93.

Pottier, François (2001). Simplifying subtyping constraints: a theory. In: *Information and Computation* 170.2, pp. 153–183. Cited on pp. 139, 177.

Pottier, François and Didier Rémy (2003). The essence of ML type inference. Unpublished draft of an extended version. Available at http://cristal.inria.fr/attapl/emlti-long.pdf. Cited on p. 129.

Pottier, François and Didier Rémy (2005). The essence of ML type inference. In: *Advanced topics in types and programming languages*. MIT Press. Chapter 10, pp. 389–489. Cited on pp. 10, 32, 102, 104, 163.

Rastogi, Aseem, Avik Chaudhuri, and Basil Hosmer (2012). The ins and outs of gradual type inference. In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. ACM, pp. 481–494. DOI: 10.1145/2103656.2103714. Cited on p. 202.

Rémy, Didier (1989). Type checking records and variants in a natural extension of ML. In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Austin, Texas, USA*, pp. 77–88. Cited on pp. 134, 135.

Rémy, Didier (1993). Type inference for records in a natural extension of ML. In: *Theoretical Aspects of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press. Cited on p. 135.

Reynolds, John C. (1997). Design of the programming language Forsythe. In: *Algol-like languages*. Birkhäuser, pp. 173–233. Cited on pp. 130, 139.

Ronchi Della Rocca, Simona (1988). Principal type scheme and unification for intersection type discipline. In: *Theoretical Computer Science* 59.1-2, pp. 181–209. Cited on p. 139.

Siek, Jeremy G. and Walid Taha (2006). Gradual typing for functional languages. In: *Proceedings of Scheme and Functional Programming Workshop*. ACM, pp. 81–92. Cited on pp. 10, 32, 145, 157, 253.

Siek, Jeremy G. and Walid Taha (2007). Gradual typing for objects. In: *Proceedings of the 21st European Conference on Object-Oriented Programming*. ECOOP'07. Springer-Verlag, pp. 2–27. Cited on pp. 148, 176, 202.

Siek, Jeremy G., Peter Thiemann, and Philip Wadler (2015). Blame and coercion: together again for the first time. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. ACM, pp. 425–435. DOI: 10.1145/2737924.2737968. Cited on pp. 160, 313, 314.

Siek, Jeremy G. and Manish Vachharajani (2008). Gradual typing with unification-based inference. In: *Proceedings of the 2008 Symposium on Dynamic*

Languages. DLS '08. ACM, 7:1–7:12. DOI: 10.1145/1408681.1408688. Cited on pp. 145, 147, 154, 156, 202.

Siek, Jeremy G., Michael M. Vitousek, Matteo Cimini, and John Tang Boyland (2015). Refined criteria for gradual typing. In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Vol. 32. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 274–293. DOI: 10.4230/LIPIcs.SNAPL.2015.274. Cited on pp. 152, 155.

Swamy, Nikhil, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman (2014). Gradual typing embedded securely in JavaScript. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. ACM, pp. 425–437. DOI: 10.1145/2535838.2535889. Cited on p. 202.

Tobin-Hochstadt, Sam and Matthias Felleisen (2006). Interlanguage migration: from scripts to programs. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA '06. ACM, pp. 964–974. DOI: 10.1145/1176617.1176755. Cited on pp. 159, 318.

Tobin-Hochstadt, Sam and Matthias Felleisen (2008). The design and implementation of Typed Scheme. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. ACM, pp. 395–406. DOI: 10.1145/1328438.1328486. Cited on p. 26.

Tobin-Hochstadt, Sam and Matthias Felleisen (2010). Logical types for untyped languages. In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. ACM, pp. 117–128. DOI: 10.1145/1863543.1863561. Cited on pp. 27, 131.

Toro, Matías and Éric Tanter (2017). A gradual interpretation of union types. In: *Proceedings of the 24th Static Analysis Symposium*. SAS '17. Springer International Publishing, pp. 382–404. Cited on pp. 146, 202.

Trifonov, Valery and Scott Smith (1996). Subtyping constrained types. In: *Static Analysis*. Springer Berlin Heidelberg, pp. 349–365. Cited on pp. 89, 93.

Vazou, Niki, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones (2014). Refinement types for Haskell. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. ACM, pp. 269–282. DOI: 10.1145/2628136.2628161. Cited on p. 211.

Wadler, Philip and Robert Bruce Findler (2009). Well-typed programs can't be blamed. In: *Proceedings of the 18th European Symposium on Programming*. ESOP '09. Springer-Verlag, pp. 1–16. DOI: 10.1007/978-3-642-00590-9_1. Cited on pp. 154, 159–161, 313, 314, 318.

Wand, Mitchell (1987). A simple algorithm and proof for type inference. In: *Fundamenta Informaticae* 10, pp. 115–122. Cited on p. 102.

Wright, Andrew K. and Matthias Felleisen (1994). A syntactic approach to type soundness. In: *Information and Computation* 115.1, pp. 38–94. DOI: 10.1006/inco.1994.1093. Cited on p. 66.

Xie, Ningning, Xuan Bi, and Bruno C. d. S. Oliveira (2018). Consistent subtyping for all. In: *Programming Languages and Systems*. Springer International Publishing, pp. 3–30. Cited on p. 202.