HAL open science

Type-based static analysis of structural properties in programming languages

Francisco Alberti

▶ **To cite this version:**

Francisco Alberti. Type-based static analysis of structural properties in programming languages. Software Engineering [cs.SE]. Université Paris-Diderot - Paris VII, 2005. English. NNT : . tel-00010369

HAL Id: tel-00010369

https://theses.hal.science/tel-00010369

Submitted on 3 Oct 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Paris 7 — Denis Diderot
UFR d'Informatique

Doctorat
Programmation : Sémantique, Preuves et Langages

Analyse Statique Typée des Propriétés Structurelles des Programmes

Francisco ALBERTI

Thèse dirigée par
Pierre-Louis CURIEN

Soutenue le
27 mai 2005

Jury

Gavin BIERMAN	Rapporteur
Charles CONSEL	
Guy COUSINEAU	Président
Vincent DANOS	
Flemming NIELSON	Rapporteur

Résumé

Des optimisations pratiques, comme l'*inlining* ou l'évaluation stricte, peuvent être justifiées lorsque l'on découvre comment une valeur est 'utilisée' dans un contexte donné. Cette idée semble maintenant relativement acceptée.

Dans cette thèse, on présente un cadre théorique général d'analyse statique pour l'inférence de propriétés d''usage', que nous préfèrons appeler propriétés structurelles, des programmes fonctionnels. Le terme 'structurel', qui est emprunté à la théorie de la démonstration, est utilisé ici pour suggérer un rapport étroit avec la logique linéaire, où les règles structurelles de contraction et affaiblissement jouent un rôle important.

Ce cadre théorique est formulé sous la forme d'un système de typage à la Church pour un langage intermédiaire, ce dernier étant une version légerement modifiée d'un langage fonctionnel source, dans le style de PCF, mais comportant des annotations structurelles. Le problème de l'analyse statique consiste alors à trouver une traduction du langage source vers le langage intermédiaire. Etant donné qu'il peut y avoir plus d'une seule traduction, on montre que l'on peut voir toutes les traductions possibles comme les solutions d'un ensemble d'inéquations appropriées. De cet ensemble d'inéquations, on s'intéresse en particulier à la plus petite solution, qui correspond à la traduction la plus précise ou optimale. Comme le prouve le prototype que nous avons implémenté, l'inférence des propriétés structurelles pour un langage réel est rélativement simple et efficace à mettre en œuvre.

La plus grande partie de ce manuscrit de thèse est dédiée à un seul cas d'étude, l'analyse linéaire, dont l'objectif est de déterminer les valeurs qui sont utilisées une seule fois. Les raisons de cette démarche sont que l'analyse linéaire a une base théorique très solide (la logique linéaire elle même) et est simple à comprendre. Pour commencer, on décrit une version de l'analyse linéaire très simplifiée, cependant intéressante parce qu'elle aborde d'un nouveau point de vue, celui de l'analyse statique, le problème de trouver la meilleure décoration linéaire pour une preuve intuitioniste.

Des analyses plus puissantes sont ensuite introduites en tant qu'extensions de cette analyse simplifiée. C'est le cas du sous-typage et du polymorphisme d'annotations. Ce dernier, qui est un mécanisme d'abstraction sur des annotations, est une extension clé dans la pratique, car il permet à l'analyse de garder son pouvoir expressif en présence de modules compilés séparement.

On montre finalement comment généraliser l'analyse linéaire à un cadre plus abstrait permettant d'exprimer d'autres types d'analyse structurelle, comme l'analyse affine, relevante, ou bien non-relevante.

On prouve plusieurs propriétés standards pour l'ensemble des systèmes de typage, ainsi que leur correction sémantique par rapport à la sémantique opérationnelle du langage source.

Abstract

It is relatively well-known that many useful optimisations, including inlining and strict evaluation, can be validated if we can determine how a value is 'used' in a given evaluation context.

In this thesis, we introduce a general static analysis framework for inferring 'usage' or, as we prefer to call them, structural properties of functional programs. The term 'structural' is borrowed from proof theory, and is intended to suggest a strong connection with linear logic, for which the structural rules of weakening and contraction play an important role.

The framework is formulated as a Church-style type system for an intermediate language, which is a slightly modified version of a PCF-like source functional language, but with structural annotations in it. We present the problem of static analysis in this context as that of finding a translation from the source into the intermediate language. As there may be more than one possible translation, we show how the set of all possible translations can be compactly characterised as a set of inequations over a suitable algebraic ordered set of annotations. In particular, we are interested in the least solution of this set of inequations, corresponding to the most accurate, or optimal, translation. As our prototype implementation showed us, inferring structural properties for a realistic language is not only simple to put into practice, but also computationally cheap.

Most of this thesis dissertation is concerned with the detailed presentation of a case study, linearity analysis, aimed at determining when values are used exactly once. The reason for such a choice is that linearity analysis has a solid theoretical background, linear logic itself, and is simple to understand. We begin by describing a very simplistic version of linearity analysis, which is interesting in itself as it embodies a new characterisation of the problem of finding the best linear decoration for an intuitionistic proof.

More practically useful analyses are then introduced as extensions to this simpler analysis. These include a notion of subtyping and a mechanism for abstracting over annotation values, known as annotation polymorphism. Annotation polymorphism turns out to be a key feature in practice, as it also allows the analysis to retain its expressive power across separately compiled modules.

We finally show how the framework for linearity analysis can be modified to cope with other interesting types of structural analysis, including affine, relevance (neededness) and non-relevance (dead-code or absence) analysis.

We prove a number of standard type-theoretic properties for the type systems presented, and show their semantic correctness with respect to the operational semantics of the source language.

à *Annick et Jacques*

Remerciements

Tout d'abord, je voudrais remercier mon directeur, Pierre-Louis Curien, qui a fait preuve d'une patience presque illimitée, et surtout, qui m'a empêché d'abandonner ce projet en m'apportant son aide, non seulement sur un plan financier, mais aussi sur un plan humain. Ses encouragements perpétuels, ainsi que ses nombreux et précieux conseils, m'ont sans doute permis d'aboutir.

Je remercie Valeria de Paiva et Eike Ritter, avec qui j'ai fait mes premiers pas en théorie des langages et, en particulier, en logique linéaire. Deux ans de projets de collaboration avec eux m'ont permis de mettre au point les idées de base qui font la matière primordiale de cette thèse. Je remercie Martin Hyland pour m'avoir accueilli pendant trois mois au Département de Mathématiques de l'Université de Cambridge et pour les quelques discussions que nous avons eues au sujet de la logique de Belnap.

Merci à tous ceux avec qui j'ai eu l'opportunité de discuter. En particulier, je voudrais remercier Gavin Bierman, Vincent Danos, Hugo Herbelin, Achim Jung, Matthias Kegelmann, Ian Mackie et Paul-André Melliès, ainsi que mes anciens compagnons de bureau à l'Ecole Normale Supérieure, Jean-Vincent Loddo et Vincent Balat. Je remercie également Emmanuel Chailloux pour ses bons conseils et ses encouragements.

Je remercie Jerôme Kodjabachian et Olivier Trullier, mes chefs chez Mathématiques Appliquées S.A. (MASA), qui ont été très compréhensifs pendant les phases de correction et de mise au point du manuscrit.

Je dédie cette thèse à Annick et Jacques Novak, qui m'ont apporté leur soutien inconditionnel durant de longues années, en particulier pendant les années les plus difficiles. Je remercie tout spécialement Grégori Novak, ainsi que Giselle et Jacques Bouchegnies, qui ont été aussi d'un énorme soutien.

Mes amis Isabel Pons, Marc Parant, Chris Linney ont été à mes côtés, surtout pendant la phase finale de rédaction. Anne-Gwenn Bosser m'a donné le coup de pouce dont j'avais besoin, et elle m'a honoré de son amitié.

Enfin, je remercie ma famille qui veille sur moi de loin, de très loin. Je n'en serais pas arrivé là sans eux.

Contents

French summary: L'analyse structurelle linéaire **1**
1 Introduction 1
2 L'analyse linéaire générale 2
2.1 Le langage source 2
2.2 Le langage intermédiaire 2
2.3 Le sous-typage d'annotations 7
2.4 Le polymorphisme d'annotations 8
3 Propriétés de l'analyse linéaire 9
3.1 Propriétés élémentaires 9
3.2 La correction de l'analyse línéaire 10
3.3 La décoration optimale 11
4 L'*inlining* comme application 12
5 Inférence des annotations 13
5.1 Inférence des contraintes 13
5.2 Correction de l'inférence des contraintes 16
5.3 Solution optimale d'un système de contraintes 16
5.4 Inférence des annotations pour l'analyse contextuelle 18
6 Analyse structurelle abstraite 19
6.1 La notion de structure d'annotations 19
6.2 Quelques exemples familiers 21
7 Conclusion 21

1 Introduction **25**
1.1 Motivations 25
1.1.1 Structural properties 26
1.1.2 Applications 26
1.2 Annotated type systems 27
1.3 Linearity analysis 27
1.4 Annotation polymorphism 28
1.4.1 The poisoning problem 29
1.4.2 Contextual analysis 29
1.4.3 Modular static analysis 30
1.5 Contributions 30
1.6 Plan of the thesis 31
1.7 Prerequisites 31

2 Preliminaries **33**
2.1 The source language . . . 33
2.1.1 Syntax . . . 33
2.1.2 Static semantics . . . 34
2.1.3 Operational semantics . . . 36
2.2 Partial orders . . . 37

3 Linearity analysis **41**
3.0.1 An intermediate linear language . . . 41
3.0.2 An application to inlining . . . 42
3.0.3 Organisation . . . 42
3.1 A brief review of DILL . . . 43
3.1.1 Syntax and typing rules . . . 43
3.1.2 Reduction . . . 45
3.1.3 Substitution . . . 45
3.1.4 Girard's translation . . . 45
3.2 The type system NLL . . . 46
3.2.1 Annotation set . . . 47
3.2.2 Annotated types . . . 47
3.2.3 Annotated preterms . . . 48
3.2.4 Typing contexts . . . 48
3.2.5 Typing rules . . . 49
3.2.6 A remark on primitive operators . . . 51
3.2.7 Examples . . . 51
3.2.8 Reduction . . . 53
3.3 Decorations . . . 53
3.3.1 The problem of static analysis . . . 54
3.4 Towards syntax-directedness . . . 55
3.4.1 Contraction revisited . . . 55
3.4.2 A syntax-directed version of NLL . . . 56
3.5 Type-theoretic properties . . . 59
3.5.1 Some elementary properties . . . 59
3.5.2 Embedding FPL into NLL . . . 60
3.5.3 Substitution . . . 61
3.5.4 Semantic correctness . . . 62
3.5.5 Considering η-reduction . . . 64
3.6 Optimal typings . . . 64
3.7 Applications . . . 67
3.7.1 Inlining . . . 67
3.7.2 Limitations . . . 69
3.7.3 Sharing and single-threading . . . 70

4 Annotation subtyping **71**
4.0.4 Organisation . . . 71
4.1 The Subsumption rule . . . 72
4.1.1 Inlining revisited . . . 73
4.1.2 An illustrative example . . . 74

4.1.3 Digression: context narrowing . . . 75
4.2 Soundness . . . 77
4.3 Minimum typing . . . 78
4.4 Semantic correctness . . . 81
4.4.1 Subject reduction for η-reduction . . . 82

5 Annotation polymorphism 83
5.0.2 Organisation . . . 83
5.1 Separate compilation and optimality . . . 84
5.2 The type system . . . 86
5.2.1 Types . . . 86
5.2.2 Preterms . . . 88
5.2.3 Set of free annotation parameters . . . 88
5.2.4 Annotation substitution . . . 89
5.2.5 Constraint set satisfaction . . . 91
5.2.6 Constraint implication . . . 92
5.2.7 The typing rules . . . 93
5.2.8 Introducing and eliminating generalised types . . . 93
5.2.9 A 'most general' example decoration . . . 96
5.2.10 Reduction . . . 96
5.3 Subtyping annotation polymorphism . . . 98
5.3.1 Soundness . . . 98
5.4 Type-theoretic properties . . . 100
5.4.1 Minimum typings . . . 102
5.4.2 Semantic correctness . . . 103
5.4.3 A word on contextual analysis . . . 104
5.4.4 Inlining revisited again . . . 105
5.5 Towards modular linearity analysis . . . 106
5.5.1 Let-based annotation polymorphism . . . 106
5.5.2 Retricted quantification rules . . . 106
5.6 Emulating the Subsumption rule . . . 108
5.7 Adding type-parametric polymorphism . . . 113
5.7.1 Syntax and typing rules . . . 113
5.7.2 Correctness . . . 114

6 Annotation inference 115
6.0.3 A two-stage process . . . 115
6.0.4 Organisation . . . 116
6.1 Simple annotation inference . . . 116
6.1.1 Relaxing the conditional rule . . . 121
6.1.2 Correctness . . . 122
6.1.3 Avoiding splitting contexts . . . 125
6.2 Solving constraint inequations . . . 128
6.2.1 Characterising the least solution . . . 128
6.2.2 Digression: decorations as closures . . . 129
6.2.3 A graph-based algorithm for computing the least solution . . . 129
6.2.4 Putting it all together . . . 131

6.3 Let-based annotation inference 131
6.3.1 Preliminary remarks 131
6.3.2 Extending the simple inference algorithm 132
6.3.3 Correctness 133
6.3.4 Growing constraint sets 134
6.4 Modular linearity analysis 135

7 Abstract structural analysis 137
7.0.1 Organisation 137
7.1 Structural analysis 138
7.1.1 Basic definitions 138
7.2 Type-theoretic properties 141
7.2.1 A non-distributive counter-example 142
7.2.2 Correctness 143
7.2.3 Annotation inference 144
7.3 Some interesting examples 146
7.3.1 Affine analysis 146
7.3.2 Relevance analysis 148
7.3.3 Combined analyses 149
7.4 Dead-code elimination 149
7.4.1 A simple dead-code elimination transformation 150
7.5 Strictness analysis 151
7.5.1 Approximating strictness properties 152
7.5.2 Some remarks on lazy evaluation 152
7.5.3 Related work 153

8 Conclusions 155
8.1 Summary 155
8.2 Further directions 156
8.2.1 A generic toolkit 156
8.2.2 Computational structural analysis 157
8.2.3 Expressivity and comparison 157

A An alternative presentation 159
A.1 The simple case 159
A.2 The annotation polymorphic case 161

List of Figures

1 La syntaxe de FPL . . . 3
2 La sémantique opérationnelle de FPL . . . 3
3 Les règles de typage de FPL . . . 4
4 La syntaxe de $\mathrm{NLL}^{\forall\leq}$ . . . 5
5 La sémantique opérationnelle de $\mathrm{NLL}^{\forall\leq}$ . . . 5
6 Les règles de typage de $\mathrm{NLL}^{\forall\leq}$ . . . 6
7 Définition de la relation $\leq$ de sous-typage . . . 8
8 Les règles de tranformation de l'*inlining* . . . 12
9 Algorithme d'inférence d'inéquations de contrainte . . . 14
10 Algorithme d'inférence d'inéquations de contrainte (suite) . . . 15
11 Définition de la fonction auxiliaire $(- \leq -)$ . . . 16
12 Définition de la fonction auxiliaire $\mathit{split}(-,-,-)$ . . . 17
13 Règles de typage modifiées de $\mathrm{NLL}^{\forall\nu\leq}$ . . . 17
14 Trois exemples familiers des analyses structurelles . . . 22

2.1 Inductive definition of preterm substitution . . . 34
2.2 The typing rules of FPL . . . 36

3.1 DILL typing rules . . . 44
3.2 Girard's translation . . . 46
3.3 NLL structural rules . . . 49
3.4 NLL typing rules . . . 50
3.5 The 'functional programming' fragment of DILL . . . 50
3.6 Example NLL type derivation . . . 52
3.7 Typing examples of some familiar terms . . . 53
3.8 Modified syntax-directed typing rules for $\mathrm{NLL}^{\uplus}$ . . . 57
3.9 Decoration space for the apply function . . . 67
3.10 The inlining optimisation relation . . . 68

4.1 Subtyping relation on types . . . 72
4.2 The revised inlining relation . . . 74
4.3 Optimal decoration for (fst p) . . . 76
4.4 Modified rules for $\mathrm{NLL}^{\mu\leq}$ . . . 78
4.5 The typing rules of $\mathrm{NLL}^{\mu\leq\uplus}$ . . . 81

5.1 Annotation substitution . . . 90
5.2 The type system $\mathrm{NLL}^{\forall}$ . . . 94

5.3 An example $\mathrm{NLL}^{\forall}$ type derivation 97
5.4 Subtyping relation for $\mathrm{NLL}^{\forall\leq}$ 98
5.5 Modified rules for $\mathrm{NLL}^{\forall\mu\leq}$ 102
5.6 Final version of the inlining relation 105
5.7 Restricted quantification rules for $\mathrm{NLL}^{\forall\mathrm{let}\leq}$ 107
5.8 Definition of $\sigma^\sharp$ and $\sigma^\flat$ 109
5.9 Definition of $(-^\dagger)$ translation 111
5.10 Definition of $(-^\dagger)$ translation (continued) 112

6.1 Generating subtyping constraints 118
6.2 A general definition of $split(-,-,-)$ 118
6.3 Inferring constraint inequations for simple linearity analysis 119
6.4 Inferring constraint inequations for simple linearity analysis (continued) 120
6.5 Inferring constraint inequations for simple linearity analysis without context splitting 126
6.6 Inferring constraint inequations for simple linearity analysis without context splitting (continued) 127
6.7 Annotation inference algorithm for linearity analysis 131
6.8 Extra rules for let-based annotation inference 132

7.1 The abstract typing rules of structural analysis 139
7.2 Modified rules for inferring constraint inequations in structural analysis. . . . 144
7.3 Example critical step in the proof of the substitution property 145
7.4 Definition of the $occurs(-,-)$ function 147
7.5 An annotation structure for sharing and absence analysis 149
7.6 The simple dead-code optimisation relation 151

A.1 $\mathrm{NLL}^{\sqcup}$ typing rules 160
A.2 $\mathrm{NLL}^{\forall\sqcup}$ typing rules 162

French summary: L'analyse structurelle linéaire

1 Introduction

Dans ce résumé, nous présentons l'*analyse linéaire*, une théorie d'analyse statique dont l'objectif est de déterminer, pour un programme donné, l'ensemble des valeurs qui sont utilisées une seule fois. L'analyse statique linéaire s'inscrit dans la tradition des analyses statiques des propriétés d'*usage*, dont nous pouvons distinguer deux grandes familles : celles basées sur une description dénotationnelle du langage source, et celles basées sur la théorie de la démonstration de la logique linéaire.

L'analyse linéaire que nous présentons ici peut se voir comme une instance d'un cadre théorique d'analyse statique de propriétés *structurelles* abstrait, permettant d'exprimer d'autres types d'analyse structurelle, comme l'analyse affine, relevante (*relevance analysis*), ou bien non-relevante (*non-relevance* ou *absence analysis*) [4, 21, 69]. Nous employons le terme 'structurel', emprunté à la théorie de la démonstration, pour suggérer un rapport étroit avec la logique linéaire de Girard [30], où les règles structurelles de contraction et d'affaiblissement jouent un rôle important, et ainsi nous différencier d'autres analyses d'usage introduites par d'autres auteurs, comme l'analyse affine de Wansborough et Peyton-Jones [68, 67], où le lien avec la logique linéaire (ou même la logique affine) est plus vague.

La théorie de l'analyse linéaire est formulée sous la forme d'un système de typage *annotaté*, c'est-à-dire, un système de typage à la Church pour un langage intermédiaire, ce dernier étant une version légerement modifiée d'un langage fonctionnel source, dans le style du langage fonctionnel PCF de Plotkin [55], mais comportant des annotations structurelles. Le problème de l'analyse statique linéaire consiste alors à trouver une traduction du langage source vers le langage intermédiaire. Etant donné qu'il peut y avoir plus d'une seule traduction, nous allons voir qu'il est possible de caractériser toutes les traductions possibles comme des solutions d'un ensemble d'inéquations appropriées. De cet ensemble d'inéquations, nous nous intéressons en particulier à la plus petite solution, qui correspond à la traduction la plus précise ou optimale. Du point de vue de la théorie de la démonstration, cette traduction optimale est en correspondance avec la meilleure *décoration* linéaire pour une preuve intuitionniste, étudiée par Danos, Joinet et Schellinx [26].

En présence de modules compilés séparément, l'analyse linéaire consistant à trouver la traduction optimale s'avère insuffisante. En effet, nous ne pouvons pas typer les définitions des modules avec des types correspondants aux traductions optimales, car ces définitions pourraient être utilisées dans des contextes qui ont besoin de types moins précis pour être

typable. La version de l'analyse linéaire que nous présentons correspond alors à l'analyse linéaire que nous appelons *générale*, car il est possible de typer une définition avec un type annoté *polyvariant* qui, d'une certaine manière, correspond à une définition compacte de l'espace de toutes les traductions, ou décorations, dont la décoration optimale n'est qu'un élément parmi d'autres. Le mécanisme qui nous permet d'augmenter ainsi l'expressivité de l'analyse linéaire est connu sous le nom de *polymorphisme d'annotations*. (Nous avons aussi enrichi l'analyse avec une notion de sous-typage sur les annotations, notion dont nous verrons qu'elle est clairement 'latente' dans la formulation annoté de l'analyse linéaire.) Nous allons montrer qu'il est *toujours* possible de caractériser l'espace des décorations d'un terme du langage source avec un type annoté polymorphe approprié du langage intermédiaire, de façon constructive, à travers la formulation d'un algorithme d'inféréncе des annotations.

Nous présentons l'*inlining*, c'est-à-dire la transformation qui consiste à substituer l'utilisation d'une définition par la définition elle-même *in situ*, comme exemple trivial d'application didactique de l'analyse linéaire.

2 L'analyse linéaire générale

2.1 Le langage source

Le langage source que nous adoptons ici est une variante du langage fonctionnel PCF de Plotkin [55]. La syntaxe et la sémantique opérationnelle du langage, que nous appelerons FPL, sont définies dans les Figures 1 et 2[1].

La méta-variable $\mathbb{G}$ dénote un type de base, tel que les entiers et les booléens, notés int et bool respectivement. Dans les règles, $\Sigma(\pi)$ fait référence au type associé à la constante ou opérateur π dans la théorie, dont on suppose un certain nombre. (En particulier, nous avons au moins $\Sigma(\mathsf{false}) = \Sigma(\mathsf{true}) = \mathsf{bool}$.) Nous écrivons $M[x/N]$ pour la substitution d'un terme N par une variable x dans un terme M.

Les assertions de typage ou séquents bien formés de FPL sont ceux qui peuvent être dérivés en utilisant les règles de typage de la Figure 3.

2.2 Le langage intermédiaire

Le but de l'analyse statique linéaire consiste à trouver, pour un séquent d'un langage source, un séquent du langage intermédiaire, comportant des annotations structurelles.

Le langage intermédiaire correspondant à l'analyse linéaire générale, que nous appelons $\mathrm{NLL}^{\forall\leq}$, est un système de types avec des annotations (*annotated type system*)[2]. Le premier pas dans sa définition consiste à spécifier un ensemble d'annotations $\mathbb{A}$ comportant les propriétés suivantes, nous permettant de classer les occurrences des variables en deux sortes :

1 Linéaire
$\top$ Intuitionniste

[1] FPL est un acronyme de *Functional Programming Language*. Pour être consistant avec la nomenclature adoptée dans la thèse, nous avons décidé de conserver les noms en anglais des règles.

[2] NLL est un acronyme de *Non-linear Linear Language*. La thèse présente l'analyse linéaire de façon progressive, en introduisant d'abord l'analyse linéaire simple (ou monomorphe), en passant par une analyse étendue avec une notion de sous-typage sur les annotations, pour en finir ensuite avec l'analyse linéaire générale (ou polymorphe) qui inclut une notion supplémentaire de polymorphisme sur les annotations. La thèse présente aussi des formulations 'dirigées par la syntaxe' des différentes théories, permettant de prouver plus facilement certains des résultats.

Types	σ	::=	$\mathbb{G}$	Type de base
		|	$\sigma \to \sigma$	Espace des fonctions
		|	$\sigma \times \sigma$	Produit cartesian
Termes	M	::=	π	Fonction primitive
		|	x	Variable
		|	$\lambda x{:}\sigma.M$	Abstraction fonctionnelle
		|	MM	Application
		|	$\langle M, M\rangle$	Paire
		|	$\mathsf{let}\ \langle x, x\rangle = M\ \mathsf{in}\ M$	Projection
		|	$\mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M$	Conditionnel
		|	$\mathsf{fix}\ x{:}\sigma.M$	Récursion
Contextes	Γ	::=	$x_1 : \sigma_1, \ldots, x_n : \sigma_n$	
Séquents	J	::=	$\Gamma \vdash M : \sigma$	

Figure 1: La syntaxe de FPL

$$
\begin{aligned}
(\lambda x{:}\sigma.M)N &\to M[N/x] \\
\mathsf{let}\ \langle x_1, x_2\rangle = \langle M_1, M_2\rangle\ \mathsf{in}\ N &\to N[M_1/x_1, M_2/x_2] \\
\mathsf{if\ true\ then}\ N_1\ \mathsf{else}\ N_2 &\to N_1 \\
\mathsf{if\ false\ then}\ N_1\ \mathsf{else}\ N_2 &\to N_2 \\
\mathsf{fix}\ x{:}\sigma.M &\to M[\mathsf{fix}\ x{:}\sigma.M/x]
\end{aligned}
$$

Figure 2: La sémantique opérationnelle de FPL

$$\frac{\Sigma(\pi) = \sigma}{- \vdash \pi : \sigma}\ \text{Primitive} \qquad \frac{}{x : \sigma \vdash x : \sigma}\ \text{Identity}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x{:}\sigma.M : \sigma \to \tau}\ {\to}_{\mathcal{I}} \qquad \frac{\Gamma_1 \vdash M : \sigma \to \tau \quad \Gamma_2 \vdash N : \sigma}{\Gamma_1, \Gamma_2 \vdash MN : \tau}\ {\to}_{\mathcal{E}}$$

$$\frac{\Gamma_1 \vdash M_1 : \sigma_1 \quad \Gamma_2 \vdash M_2 : \sigma_2}{\Gamma_1, \Gamma_2 \vdash \langle M_1, M_2 \rangle : \sigma_1 \times \sigma_2}\ {\times}_{\mathcal{I}} \qquad \frac{\Gamma_1 \vdash M : \sigma_1 \times \sigma_2 \quad \Gamma_2, x_1 : \sigma_1, x_2 : \sigma_2 \vdash N : \tau}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ \langle x_1, x_2 \rangle = M\ \mathsf{in}\ N : \tau}\ {\times}_{\mathcal{E}}$$

$$\frac{\Gamma_1 \vdash M : \mathsf{bool} \quad \Gamma_2 \vdash N_1 : \sigma \quad \Gamma_2 \vdash N_2 : \sigma}{\Gamma_1, \Gamma_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \sigma}\ \text{Conditional} \qquad \frac{\Gamma, x : \sigma \vdash M : \sigma}{\Gamma \vdash \mathsf{fix}\ x{:}\sigma.M : \sigma}\ \text{Fixpoint}$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma, x : \sigma \vdash M : \tau}\ \text{Weakening} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma, x_1 : \sigma, x_2 : \sigma \vdash M[x_1/x, x_2/x] : \tau}\ \text{Contraction}$$

Figure 3: Les règles de typage de FPL

Du point de vue de l'analyse statique, l'assertion $1 \sqsubseteq \top$ exprime le fait que 1 est une propriété plus précise en terme d'information exploitable que $\top$, et donc préférable dans nos analyses. Du point vue de la sémantique du langage, cette assertion exprime la relation d'inclusion d'un contexte linéaire dans un contexte intuitionniste (du même type). Une fois de plus, du point de vue de la logique, cette assertion est à la base de la définition de la rélation de sous-décoration de Girard proposé par Danos, Joinet et Schellinx, nécessaire dans la définition de décoration linéaire optimale[3].

La syntaxe et la sémantique opérationnelle du langage sont résumés dans les Figures 4 et 5, et les règles de typage dans la Figure 6. Nous écrivons $\phi^t \multimap \psi$ pour le type d'une fonction prenant un argument de type ϕ et renvoyant un résultat de type ψ. L'annotation structurelle t nous donne l'usage de l'argument dans le corps de la fonction : si $t = \top$, il s'agit d'une fonction intuitionniste ; si $t = 1$, il s'agit d'une fonction linéaire. Pour les paires, les annotations structurelles nous renseignent sur l'usage que l'on impose à chaque composante.

Il faut remarquer que t pourrait contenir des paramètres d'annotation. Ainsi, $\phi^{\mathsf{p}} \multimap \psi$ dénote le type d'une fonction qui peut être considérée ou bien comme une fonction linéaire ou bien comme une fonction intuitionniste. Plus généralement, un terme de type *généralisé* $\forall \overline{\mathsf{p}_i} \,|\, \Theta.\phi$, où Θ est un ensemble de contraintes de la forme $\overline{\mathsf{p}_i \sqsupseteq t_i}$, est un terme qui peut être considéré comme ayant plusieurs types de la forme $\phi[\vartheta]$, où chaque type est obtenu en substituant les paramètres p_i par d'autres termes d'annotation t'_i. Les substitutions sont notés $\vartheta : \mathbb{P} \rightharpoonup \mathbb{T} ::= \langle t'_1/\mathsf{p}_1, \ldots, t'_n/\mathsf{p}_n \rangle$. Nous parlerons de substitution *close* lorsque les termes t'_i sont des simples constantes d'annotation a_i, c'est-à-dire lorsque $\mathit{FA}(t_i) = \emptyset$. Dans ce cas-là nous écrirons toujours θ à la place de ϑ. La notation $\mathit{FA}(-)$ est utilisée pour dénoter

[3] La décoration linéaire optimale est une traduction d'une preuve de la logique intuitionniste en une preuve de la logique linéaire intuitionniste, très similaire en structure á la preuve de départ, et tel que chaque occurrence d'un exponentiel '!' dans la preuve est inévitable, car l'hypothèse affectée est ou bien affaiblie ou bien contractée directement ou indirectement. En fait, le fragment monomorphe de $\mathrm{NLL}^{\forall\leq}$ n'est ni plus ni moins que le langage des dćorations auquel Danos, Joinet et Schellinx font référence [26].

Annotations	$\mathbb{A}$	$\equiv$	$\langle\{1,\top\},\sqsubseteq\rangle$	
Relation d'ordre			$a \sqsubseteq a \qquad 1 \sqsubseteq \top$	
Types	ϕ	$::=$	$\mathbb{G}$	Type de base
		$\mid$	$\phi^t \multimap \phi$	Espace des fonctions
		$\mid$	$\phi^t \otimes \phi^t$	Produit tensoriel
		$\mid$	$\forall\overline{\mathsf{p}_i} \mid \Theta.\phi$	Type généralisé
Termes d'annotation	$t \in \mathbb{T}$	$::=$	$a \in \mathbb{A}$	Constante d'annotation
		$\mid$	$\mathsf{p} \in \mathbb{P}$	Paramètre d'annotation
		$\mid$	$t + t$	Contraction des annotations
Termes	M	$::=$	π	Fonction primitive
		$\mid$	x	Variable
		$\mid$	$\lambda x{:}\phi^t.M$	Abstraction fonctionnelle
		$\mid$	MM	Application
		$\mid$	$\langle M, M\rangle^{t,t}$	Paire
		$\mid$	$\mathsf{let}\ \langle x, x\rangle^{t,t} = M\ \mathsf{in}\ M$	Projection
		$\mid$	$\mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M$	Conditionnel
		$\mid$	$\mathsf{fix}\ x{:}\phi.M$	Récursion
		$\mid$	$\Lambda\overline{\mathsf{p}_i} \mid \Theta.M$	Terme généralisé
		$\mid$	$M\ \vartheta$	Terme spécialisé
Contraintes	Θ	$::=$	$t_1 \sqsupseteq t'_1, \ldots, t_n \sqsupseteq t'_n$	
Contextes	Γ	$::=$	$x_1 : {\phi_1}^{t_1}, \ldots, x_n : {\phi_n}^{t_1}$	
Séquents	J	$::=$	$\Theta\,;\Gamma \vdash M : \phi$	

Figure 4: La syntaxe de $\mathrm{NLL}^{\forall\leq}$

$$(\lambda x{:}\phi^t.M)N \to M[N/x]$$
$$\mathsf{let}\ \langle x_1, x_2\rangle^{t_1,t_2} = \langle M_1, M_2\rangle^{t'_1,t'_2}\ \mathsf{in}\ N \to N[M_1/x_1, M_2/x_2]$$
$$\mathsf{if\ true\ then}\ N_1\ \mathsf{else}\ N_2 \to N_1$$
$$\mathsf{if\ false\ then}\ N_1\ \mathsf{else}\ N_2 \to N_2$$
$$\mathsf{fix}\ x{:}\phi.M \to M[\mathsf{fix}\ x{:}\phi.M/x]$$
$$(\Lambda\overline{\mathsf{p}_i} \mid \Theta.M)\ \vartheta \to M[\vartheta]$$

Figure 5: La sémantique opérationnelle de $\mathrm{NLL}^{\forall\leq}$

$$\frac{}{\Theta\,;x:\phi^t \vdash x:\phi}\,\text{Identity} \qquad \frac{\Sigma(\pi)=\sigma}{\Theta\,;-\vdash \pi:\sigma}\,\text{Primitive}$$

$$\frac{\Theta\,;\Gamma, x:\phi^t \vdash M:\psi}{\Theta\,;\Gamma \vdash \lambda x{:}\phi^t.M:\phi^t \multimap \psi}\multimap_{\mathcal{I}}$$

$$\frac{\Theta\,;\Gamma_1 \vdash M:\phi^t \multimap \psi \quad \Theta\,;\Gamma_2 \vdash N:\phi \quad \Theta \rhd |\Gamma_2| \sqsupseteq t}{\Theta\,;\Gamma_1,\Gamma_2 \vdash MN:\psi}\multimap_{\mathcal{E}}$$

$$\frac{\Theta\,;\Gamma_1 \vdash M_1:\phi_1 \quad \Theta\,;\Gamma_2 \vdash M_2:\phi_2 \quad \Theta \rhd |\Gamma_1| \sqsupseteq t_1 \quad \Theta \rhd |\Gamma_2| \sqsupseteq t_2}{\Theta\,;\Gamma_1,\Gamma_2 \vdash \langle M_1, M_2\rangle^{t_1,t_2}:{\phi_1}^{t_1}\otimes{\phi_2}^{t_2}}\otimes_{\mathcal{I}}$$

$$\frac{\Theta\,;\Gamma_1 \vdash M:{\phi_1}^{t_1}\otimes{\phi_2}^{t_2} \quad \Theta\,;\Gamma_2, x_1:{\phi_1}^{t_1}, x_2:{\phi_2}^{t_2} \vdash N:\psi}{\Theta\,;\Gamma_1,\Gamma_2 \vdash \mathsf{let}\ \langle x_1,x_2\rangle^{t_1,t_2} = M\ \mathsf{in}\ N:\psi}\otimes_{\mathcal{E}}$$

$$\frac{\Theta\,;\Gamma_1 \vdash M:\mathsf{bool} \quad \Theta\,;\Gamma_2 \vdash N_1:\phi \quad \Theta\,;\Gamma_2 \vdash N_2:\phi}{\Theta\,;\Gamma_1,\Gamma_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2:\phi}\,\text{Conditional}$$

$$\frac{\Theta\,;\Gamma, x:\phi^t \vdash M:\phi \quad \Theta \rhd |\Gamma, x:\phi^t| \sqsupseteq \top}{\Theta\,;\Gamma \vdash \mathsf{fix}\ x{:}\phi.M:\phi}\,\text{Fixpoint}$$

$$\frac{\Theta,\Theta'\,;\Gamma \vdash M:\phi \quad \overline{\mathsf{p}_i} \not\subseteq FA(\Theta\,;\Gamma) \quad \Theta'\backslash\overline{\mathsf{p}_i} = \emptyset}{\Theta\,;\Gamma \vdash \Lambda\overline{\mathsf{p}_i}\,|\,\Theta'.M:\forall\overline{\mathsf{p}_i}\,|\,\Theta'.\phi}\forall_{\mathcal{I}}$$

$$\frac{\Theta\,;\Gamma \vdash M:\forall\overline{\mathsf{p}_i}\,|\,\Theta'.\phi \quad \Theta \rhd \Theta'[\vartheta] \quad dom(\vartheta) = \overline{\mathsf{p}_i}}{\Theta\,;\Gamma \vdash M\ \vartheta:\phi[\vartheta]}\forall_{\mathcal{E}}$$

$$\frac{\Theta\,;\Gamma \vdash M:\phi \quad \Theta \vdash \phi \le \psi}{\Theta\,;\Gamma \vdash M:\psi}\,\text{Subsumption}$$

$$\frac{\Theta\,;\Gamma \vdash M:\psi \quad \Theta \rhd t \sqsupseteq \top}{\Theta\,;\Gamma, x:\phi^t \vdash M:\psi}\,\text{Weakening}$$

$$\frac{\Theta\,;\Gamma, x_1:\phi^{t_1}, x_2:\phi^{t_2} \vdash M:\psi \quad \Theta \rhd t \sqsupseteq t_1 + t_2}{\Theta\,;\Gamma, x:\phi^t \vdash M[x/x_1, x/x_2]:\psi}\,\text{Contraction}$$

Figure 6: Les règles de typage de $\mathrm{NLL}^{\forall\le}$

l'ensemble de paramètres d'annotation d'un élément syntaxique de la théorie.

En présence de plusieurs paramètres d'annotation, l'ensemble de contraintes Θ d'un type généralisé nous renseigne sur des éventuels *dépendences structurelles* entre paramètres. Pour que toute substitution soit considérée comme valide, elle doit respecter ces dépendences. Par exemple, pour le type $\forall \mathsf{p}, \mathsf{q} \mid \mathsf{p} \sqsupseteq \mathsf{q}.\phi^{\mathsf{p}} \multimap \psi^{\mathsf{q}} \multimap \phi$ nous avons que $\theta \equiv \langle 1 + \top/\mathsf{p}, 1/\mathsf{q}\rangle$ est une substitution valide, car $\theta(\mathsf{p}) \sqsupseteq \theta(\mathsf{q}) \equiv \top \sqsupseteq 1$ est consistent avec l'ordre choisi pour les annotations de $\mathrm{NLL}^{\forall\leq}$.

En général, nous écrivons $\theta \models \Theta$ lorsque nous voulons exprimer le fait qu'une substitution θ *satisfaît* un ensemble de contraintes Θ, c'est-à-dire lorsque $\theta(\Theta) \equiv \overline{\theta(\mathsf{p}_i) \sqsupseteq \theta(t_i)}$ est vrai. D'une certaine manière, si nous interprétons Θ comme un prédicat logique, θ nous donne le moyen de transformer ce prédicat en proposition, et ainsi pouvoir affirmer sa consistance ou inconsistance. Parfois, et de façon équivalente, lorsque nous affirmerons que θ est une *solution* de Θ, nous serons en train d'exprimer le fait que $\theta \in [\Theta]$, où $[\Theta] = \{\theta \mid \theta \models \Theta\}$ dénote l'espaces des solutions de Θ.

Nous écrirons $\Theta \triangleright P$ pour exprimer le fait que si $\theta(\Theta)$ est vrai comme proposition, alors $\theta(P)$ est vrai aussi, pour toute substitution θ appropriée. (Lorsque P est un ensemble de contraintes, cette forme d'implication logique reçoit le nom d'*implication de contraintes*.) Pour que la substitution $\theta(\Theta)$ ait un sens, θ doit recouvrir Θ ou, en d'autres termes, $FA(\Theta) \subseteq dom(\theta)$. Il est aussi nécessaire que θ soit plus qu'une simple substitution, c'est-à-dire qu'elle évalue les contractions, en remplaçant $\theta(t' + t'')$ par $\top$. Lorsque nous voulons obtenir le résultat d'une simple substitution, nous écrirons $\Theta[\theta]$. Ainsi, $(t' + t'')[\theta] \equiv t'[\theta] + t''[\theta]$, ce dernier étant un terme d'annotation très différent de $\theta(t' + t'') = \top$.

La correction de la logique linéaire tient à la contrainte structurelle éxigeant qu'aucun terme contenant des variables linéaires ne puisse être utilisé dans un contexte intuitionniste. Dans $\mathrm{NLL}^{\forall\leq}$, la condition $\Theta \triangleright |\Gamma| \sqsupseteq t$ joue ce rôle bien spéficique. Par exemple, la règle $\multimap_{\mathcal{E}}$ éxige que pour que toute application d'une fonction de type $\phi^t \multimap \psi$ soit valide, les annotations des variables libres de son argument, disons t'_i (declarées dans Γ_2), doivent vérifier la contrainte $|\Gamma_2| = t'_i \sqsupseteq t$. Pour la récursion, nous éxigeons carrément que toutes les annotations soient $\top$, car la réduction d'un terme récursif dépend de la création d'une copie du terme.

2.3 Le sous-typage d'annotations

Même si l'ordre des annotations suggère l'inclusion entre contextes, cette information ne devient exploitable concrètement qu'à partir du moment où l'inclusion est explicitement introduite en tant que relation de sous-typage sur les annotations. La règle de Subsumption nous assure qu'il est toujours possible de substituer un terme de type ϕ par un type ψ qui l'inclut, c'est-à-dire par un type ψ tel que

$$\Theta \vdash \phi \leq \psi$$

est dérivable en utilisant les règles de la Figure 7.

Il faut noter que ϕ et ψ peuvent contenir des paramètres d'annotations, donc Θ spécifie l'ensemble de valeurs des annotations pour lesquelles il est possible d'affirmer $\phi \leq \psi$. (Nous éviterons de l'écrire lorsque la validité de l'inclusion ne dépend pas de Θ.) Par exemple,

$$\phi^1 \multimap \psi \leq \phi^\top \multimap \psi$$

manifeste directement de l'inclusion d'un contexte linéaire dans son contexte intuitionniste correspondant, et donc, par le biais de Subsumption, de la possibilité d'utiliser une fonc-

$$\frac{}{\mathbb{G} \leq \mathbb{G}}$$

$$\frac{\sigma_2 \leq \sigma_1 \quad \tau_1 \leq \tau_2 \quad a_1 \sqsubseteq a_2}{{\sigma_1}^{a_1} \multimap \tau_1 \leq {\sigma_2}^{a_2} \multimap \tau_2}$$

$$\frac{\sigma_1 \leq \sigma_2 \quad \tau_1 \leq \tau_2 \quad a_2 \sqsubseteq a_1 \quad b_2 \sqsubseteq b_1}{{\sigma_1}^{a_1} \otimes {\tau_1}^{b_1} \leq {\sigma_2}^{a_2} \otimes {\tau_2}^{b_2}}$$

Figure 7: Définition de la relation $\leq$ de sous-typage

tion linéaire à la place d'une fonction non-linéaire. L'analyse linéaire devient alors moins dépendente du contexte, donc plus expressive, car un terme peut maintenant avoir plusieurs types (celui suggéré directement par ses annotations ainsi que tous ses super-types). Grâce au sous-typage, l'argument de correction de $\mathrm{NLL}^{\forall\leq}$ peut être étendu aussi à la η-réduction (Proposition 3.13).

Le sous-typage ne suffit malheureusement pas pour rendre l'analyse exploitable en présence des modules compilés séparément. Pour celà, nous avons besoin du polymorphisme d'annotations[4].

2.4 Le polymorphisme d'annotations

Le polymorphisme d'annotations rend l'analyse linéaire *indépendente* du contexte, car il permet d'isoler l'analyse d'un terme de l'analyse des contextes qui l'utilisent. Cette propriété de *modularité* est importante, car elle permet de donner une solution satisfaisante au problème de l'analyse structurelle en présence de modules compilés séparément. Si un terme M est utilisé dans plusieurs contextes, l'idée de base consiste à typer M avec un type généralisé de la forme $\forall\overline{\mathsf{p}_i} \,|\, \Theta'.\phi$, ayant comme instances les types requis par les différents contextes qui l'utilisent. Nous verrons plus tard qu'il n'est pas difficile de trouver le type généralisé qui rend compte de tout l'espace des décorations de M. Dans la théorie, celà nécessite deux règles, $\forall_{\mathcal{I}}$ et $\forall_{\mathcal{E}}$. Elles permettent, respectivement, d'introduire un type généralisé et de l'éliminer ; et dans ce dernier cas de le remplacer par une spécialisation adaptée à un contexte donné.

La règle $\forall_{\mathcal{I}}$ détermine qu'un terme généralisé de la forme $\Lambda\overline{\mathsf{p}_i} \,|\, \Theta'.M$ est de type $\forall\overline{\mathsf{p}_i} \,|\, \Theta'.\phi$ si M est de type ϕ, en tenant compte des contraintes structurelles dans Θ'. La condition $\overline{\mathsf{p}_i} \not\subseteq FA(\Theta\,;\Gamma)$ est standard en logique, interdisant les paramètres d'annotation $\overline{\mathsf{p}_i}$ d'avoir une incidence à l'extérieur du terme. La condition $\Theta' \backslash \overline{\mathsf{p}_i} = \emptyset$ est là pour nous assurer que, dans la construction de $\Lambda\overline{\mathsf{p}_i} \,|\, \Theta'.M$, nous n'allons pas former Θ' à partir des contraintes qui ne relèvent pas des paramètres $\overline{\mathsf{p}_i}$. Ces deux conditions permettent une lecture déterministe de la règle, car le choix de Θ' est déterminé par le choix de $\overline{\mathsf{p}_i}$.

La règle $\forall_{\mathcal{E}}$ détermine que si M est de type généralisé $\forall\overline{\mathsf{p}_i} \,|\, \Theta'.\phi$, pour toute substitution ϑ ayant pour domaine $\overline{\mathsf{p}_i}$, la spécialisation $M\ \vartheta$ est bien formée, et de type $\phi[\vartheta]$, à la condition toutefois que les contraintes spécialisées $\Theta'[\vartheta]$ ne rentrent pas en contradition avec les contraintes structurelles Θ que nous avions déjà.

[4] Au moins que les types des définitions dans les interfaces des modules soient de la forme $\mathbb{G}_1 \times \cdots \times \mathbb{G}_n \to \mathbb{G}$.

3 Propriétés de l'analyse linéaire

Dans cette section nous énumérons quelques propriétés fondamentales de l'analyse linéaire générale.

3.1 Propriétés élémentaires

D'abord, commençons par déceler ce que nous voulons dire par 'linéaire', en terme d'occurrence syntaxique des variables.

Proposition 3.1
Si $\Theta\,;\Gamma, x : \phi^1 \vdash M : \psi$, *alors* x *a une seule occurrence dans* M[5]. □

Une propriété syntaxique importante et celle qui suggère que les séquents de l'analyse linéaire peuvent être vus comme des séquents du langage source 'décorés' avec des annotations. En d'autres termes, si nous oublions les annotations, avec l'aide d'un foncteur d'effacement $(^\circ)$, nous retrouvons les séquents du langage source. La définition de $(^\circ)$ est celle attendue. (En particulier, nous avons $(\forall\overline{\mathsf{p}_i}\,|\,\Theta'.\phi)^\circ = \phi^\circ$.)

Proposition 3.2
Si $\Theta\,;\Gamma \underset{\mathrm{NLL}^{\forall\leq}}{\vdash} M : \phi$, *alors* $\Gamma^\circ \underset{\mathrm{FPL}}{\vdash} M^\circ : \phi^\circ$. □

La proposition suivante établit que toute transformation d'un terme du langage intermédiaire, qui implique une réduction, est aussi, après effacement, une transformation valide du langage source. D'un point de vue théorique, celà veut dire que pour prouver la correction d'une transformation définie au niveau du langage intermédiaire, comme l'*inlining*, il suffit de prouver qu'elle préserve les types de l'analyse linéaire.

Proposition 3.3
Si $M \to N$, *alors* $M^\circ \to N^\circ$. □

La correction du sous-typage d'annotations, tel que nous l'avons présentée, tient à l'existence d'une propriété de 'transférence' des annotations, propriété plus élémentaire qui incarne déjà une forme rudimentaire de sous-typage[6].

Proposition 3.4
La règle suivante est prouvable dans $\mathrm{NLL}^{\forall\leq}$.

$$\frac{\Theta\,;\Gamma, x : \phi^1 \vdash M : \psi}{\Theta\,;\Gamma, x : \phi^\top \vdash M : \psi}\ \text{Transfer}$$

□

[5] La Figure7.4 de la page 147 définit précisement ce que nous voulons dire par 'occurrences' d'une variable dans un terme.

[6] Le nom de Transfer est tiré de DILL, la logique linéaire intuitionniste duale de Barber et Plotkin [5]. Nous retrouvons cette propriété dans la logique linéaire, car $\phi \multimap\, !\phi$. Dans l'analyse statique cette propriété reçoit le nom de propriété de *sub-effecting*.

Aucune théorie de types annotés peut être considérée comme une théorie d'analyse statique si, pour tout terme du langage source, elle n'est pas capable de fournir une analyse correcte, quoique pratiquement inutile. Pour l'analyse linéaire, l'analyse la plus 'modeste', notée ci-dessous ($^\bullet$), et consistant à décorer un terme donné du langage source avec $\top$, est toujours une analyse valide de $\mathrm{NLL}^{\forall\leq}$. D'un point de vue logique, ce fait n'est pas surprenant, car cette analyse correspond à une des traductions données par Girard, la plus célèbre, faisant plonger la logique intuitionniste dans le fragment intuitionniste de la logique linéaire.

Proposition 3.5
Si $\Gamma \underset{\mathrm{FPL}}{\vdash} M : \phi$, *alors* $- ; \Gamma^\bullet \underset{\mathrm{NLL}^{\forall\leq}}{\vdash} M^\bullet : \phi^\bullet$. □

Soit J un séquent du langage source. Alors, nous pouvons exprimer le lien étroit existant entre le langage source et le langage intermédiare par l'affirmation $J \equiv (J^\bullet)^\circ$[7].

Proposition 3.6
Si $\Theta ; \Gamma \underset{\mathrm{NLL}^{\forall\leq}}{\vdash} M : \phi$, *alors il existe* ψ *tel que* $\Theta ; \Gamma \underset{\mathrm{NLL}^{\forall\leq}}{\vdash} M : \psi$ *et* $\psi \leq \psi'$ *pour pour tout autre* ψ' *pour lequel* $\Theta ; \Gamma \underset{\mathrm{NLL}^{\forall\leq}}{\vdash} M : \psi'$.

Démonstration. Voir le Théorème 5.4.14. □

Les propriétés suivantes concernent le polymorphisme d'annotations.

Proposition 3.7
Si $\Theta ; \Gamma \vdash M : \phi$, *alors* $\Theta[\vartheta] ; \Gamma[\vartheta] \vdash M[\vartheta] : \phi[\vartheta]$.

Proposition 3.8
Si $\Theta ; \Gamma \vdash M : \phi$ *and* $\Theta' \triangleright \Theta$, *alors* $\Theta' ; \Gamma \vdash M : \phi$.

Démonstration. La validité de cette assertion dépend du fait que si $\Theta \triangleright P$ est vrai pour un prédicat P et $\Theta' \triangleright \Theta$, alors $\Theta' \triangleright P$. □

Proposition 3.9
Si $\Theta ; \Gamma \vdash M : \phi$, *alors* $\Theta, \Theta' ; \Gamma \vdash M : \phi$.

Démonstration. Immédiat à partir du Lemme 3.8 et du fait que $\Theta, \Theta' \triangleright \Theta$. □

3.2 La correction de l'analyse línéaire

Notre argument sur la correction de l'analyse linéaire prend la forme d'un théorème de *subject reduction* pour la théorie générale $\mathrm{NLL}^{\forall\leq}$. Pour prouver la correction de l'analyse linéaire, nous avons besoin de deux lemmes importants. Le premier est au cœur de la correction du polymorphisme d'annotations, tandis que le deuxième montre que la substitution des termes est bien typée, sous certaines conditions de bon sens 'structurelle' (c'est-à-dire, tant que nous n'essayons pas de substituer un terme qui contient des variables libres linéaires dans un contexte qui pourrait effacer ou dupliquer son argument.).

[7]Cette affirmation restera valide pour toute traduction que nous pourrions définir dans le cadre de l'analyse linéaire, à part celle de Girard.

Lemme 3.10 (Substitution des annotations)
La règle suivante est prouvable dans $\mathrm{NLL}^{\forall\leq}$.

$$\frac{\Theta, \Theta' \,; \Gamma \vdash M : \phi \quad \Theta \triangleright \Theta'[\vartheta] \quad dom(\vartheta) = FA(\Theta') \backslash FA(\Theta)}{\Theta \,; \Gamma[\vartheta] \vdash M[\vartheta] : \phi[\vartheta]} \;\vartheta\text{-Substitution}$$

Démonstration. Voir la démonstration du Lemme 5.4.8. □

Lemme 3.11 (Substitution des termes)
La règle suivante est prouvable dans $\mathrm{NLL}^{\forall\leq}$.

$$\frac{\Theta \,; \Gamma_1, x : {\phi_1}^t \vdash M : \psi \quad \Theta \,; \Gamma_2 \vdash N : \phi_2 \quad |\Gamma_2| \sqsupseteq t \quad \phi_2 \leq \phi_1}{\Theta \,; \Gamma_1, \Gamma_2 \vdash M[N/x] : \psi} \;\text{Substitution}$$

Démonstration. Voir les démonstrations des Lemmes 5.4.16 et 3.5.6, ce dernier étant une extension du premier à l'analyse polymorphe. □

Théorème 3.12 (Correction)
Si $\Theta \,; \Gamma \underset{\mathrm{NLL}^{\forall\leq}}{\vdash} M : \phi$ *et* $M \to N$, *alors* $\Theta \,; \Gamma \underset{\mathrm{NLL}^{\forall\leq}}{\vdash} N : \phi$.

Démonstration. Voir les démonstrations des Théorèmes 5.4.17 et 3.5.7, ce dernier étant une extension du premier à l'analyse polymorphe. □

La correction de $\mathrm{NLL}^{\forall\leq}$ peut-être étendue, grace au sous-type des annotations, pour tenir compte de la η-reduction.

$$\lambda x{:}\phi^t.M\ x \to M \quad \text{if } x \notin FV(M) \tag{η}$$

Proposition 3.13 (Correction pour η)
Si $\Theta \,; \Gamma \underset{\mathrm{NLL}^{\forall\leq}}{\vdash} \lambda x{:}\phi^t.M\ x : \phi^t \multimap \psi$ *et* $x \notin FV(M)$, *alors* $\Gamma \underset{\mathrm{NLL}^{\forall\leq}}{\vdash} M : \phi^t \multimap \psi$.

Démonstration. Voir la démonstration de la Proposition 4.4.4. □

3.3 La décoration optimale

Nous ne pouvons pas parler d'analyse statique linéaire, sans parler d'analyse optimale. L'analyse (ou décoration) optimale n'est ni plus ni moins que l'analyse la plus précise en terme d'information structurelle. De tous les séquents annotés J^*, correspondants à un séquent du langage source donné J, nous nous intéressons aux séquents annotés ne contenant que des occurrences de $\top$ qui sont *inévitables*. Une remarque importante concernant les décorations s'impose. Lorsque nous parlons de décorations, nous faisons référence aux séquents J^* *simples*, où toutes les annotations sont des constantes et les types intervenants sont des types monovariants[8].

Dans le cadre de $\mathrm{NLL}^{\forall\leq}$, l'analyse optimale peut être caracterisée de manière très élégante, comme l'élément le plus petit de l'espace de toutes les décorations d'un séquent du langage source J.

$$\mathfrak{D}_{\mathrm{NLL}}(J) \stackrel{\text{def}}{=} \{J^* \mid (J^*)^\circ = J \text{ et } J^* \text{ est simple}\}.$$

[8]Dans la thèse, le fragment monovariant de l'analyse linéaire générale correspond au système appellé $\mathrm{NLL}^{\leq}$, qui n'est ni plus ni moins que le langage de types standards de Wadler [65] étendu avec une notion de sous-typage.

$$(\lambda x{:}\phi^1.M)N \overset{\text{inl}}{\leadsto} M[N/x]$$
$$\mathsf{let}\ \langle x_1, x_2\rangle^{1,1} = \langle M_1, M_2\rangle^{t_1,t_2}\ \mathsf{in}\ N \overset{\text{inl}}{\leadsto} N[M_1/x_1][M_2/x_2]$$
$$\mathsf{let}\ \langle x_1, x_2\rangle^{1,t} = \langle M_1, M_2\rangle^{t_1,t_2}\ \mathsf{in}\ N \overset{\text{inl}}{\leadsto} \mathsf{let}\ x_2 = M_2\ \mathsf{in}\ N[M_1/x_1]$$
$$\mathsf{let}\ \langle x_1, x_2\rangle^{t,1} = \langle M_1, M_2\rangle^{t_1,t_2}\ \mathsf{in}\ N \overset{\text{inl}}{\leadsto} \mathsf{let}\ x_1 = M_1\ \mathsf{in}\ N[M_2/x_2]$$
$$\mathsf{let}\ x{:}\phi^1 = M\ \mathsf{in}\ N \overset{\text{inl}}{\leadsto} N[M/x]$$
$$(\Lambda\overline{\mathsf{p}_i} \mid \Theta.M)\ \vartheta \overset{\text{inl}}{\leadsto} M[\vartheta]$$

Figure 8: Les règles de tranformation de l'*inlining*

En effet, si nous considérons que pour deux décorations $J_1{}^*$ et $J_2{}^*$, $J_1{}^* \sqsubseteq J_2{}^*$ précisement quand toute annotation dans $J_1{}^*$ est plus petite que son annotation correspondante dans $J_1{}^*$, alors il n'est pas difficile de démontrer que l'espace des décorations de J forme un treillis complet.

Théorème 3.14 (Treillis complet de décorations)
$\langle \mathfrak{D}_{\text{NLL}}(J); \sqsubseteq\rangle$ *est un treillis complet non-vide.*

Démonstration. Voir la démonstration du Théorème 3.6.4. □

La décoration optimale est l'élément le plus petit de $\mathfrak{D}_{\text{NLL}}(J)$. En d'autres termes,

$$J^{\text{opt}} = \sqcap \mathfrak{D}_{\text{NLL}}(J).$$

4 L'*inlining* comme application

Un exemple didactique de l'analyse linéaire générale, que nous pouvons formaliser très simplement, c'est l'*inlining*, une technique d'optimisation assez répandue qui consiste à substituer *in situ* les références à une définition par le corps de la définition elle-même. L'analyse linéaire offre un critère infaillible permettant de savoir si une définition donnée est utilisée une seule fois. Dans ce cas-là, le remplacement de la référence par le corps de la définition est une transformation toujours profitable, car elle ne risque pas d'entraîner ni une augmentation de la taille du programme (car elle a lieu une seule fois), ni une perte de temps de calcul.

Nous définissons la relation de transformation d'*inlining*, s'applicant à des termes annotés du langage intermédiaire, comme la clôture contextuelle des règles de réécriture de base énumérées dans la Figure 8.

Il est intéressant d'observer que $\overset{\text{inl}}{\leadsto} \subseteq \rightarrow$, ainsi la correction de la transformation d'*inlining* est un simple corollaire du théorème de correction de $\text{NLL}^{\forall\leq}$.

Proposition 4.1 (Correction de $\overset{\text{inl}}{\leadsto}$)
Si $\Theta\,;\Gamma \underset{\text{NLL}^{\forall\leq}}{\vdash} M : \phi$ *et* $M \overset{\text{inl}}{\leadsto} N$*, alors* $\Theta\,;\Gamma \underset{\text{NLL}^{\forall\leq}}{\vdash} N : \phi$. □

5 Inférence des annotations

Une fois posées les bases théoriques de l'analyse linéaire, nous pouvons maintenant nous concentrer sur le problème pratique de l'inférence des annotations, c'est-à-dire sur le calcul effectif de la décoration optimale d'un séquent du langage source.

Nous allons procéder, comme c'est le cas généralement, en deux étapes. Soit $\Gamma \vdash M : \sigma$ un séquent de FPL. La première étape, celle de l'inférence des contraintes, consiste à calculer l'ensemble d'inéquations Θ permettant de caractériser toutes les décorations de $\mathrm{NLL}^{\leq}$, c'est-à-dire que l'on demande que

$$\{\Delta[\theta] \vdash X[\theta] : \phi[\theta] \mid \theta \models \Theta \text{ où } \mathit{FA}(X) \cup \mathit{FA}(\Delta) \subseteq \mathit{dom}(\theta)\}$$

soit précisement

$$\mathfrak{D}_{\mathrm{NLL}^{\leq}}(\Gamma \vdash M : \sigma).$$

Celui-ci constitue le critère de correction de cette première étape. Ici, Δ, X et ϕ ne contiennent que des paramètres d'annotation.

La deuxième étape consistera à trouver la solution optimale θ^{opt} qui donne lieu à la décoration optimale $\Delta[\theta^{\mathrm{opt}}] \vdash X[\theta^{\mathrm{opt}}] : \phi[\theta^{\mathrm{opt}}]$.

5.1 Inférence des contraintes

L'algorithme d'inférence des contraintes prend comme entrée une paire $\langle \Gamma, M \rangle$, composée d'un contexte Γ et d'un terme M du langage source, et fourni en sortie un triplet $\langle \Theta, X, \phi \rangle$, formé d'un ensemble de contraintes Θ, un terme X et un type ϕ contenant uniquement des paramètres d'annotation. La notation qu'on utilise pour les configurations de l'algorithme (à chaque étape de son éxécution) est

$$\Theta \,;\Delta \vdash M \Rightarrow X : \phi,$$

où Δ correspond au contexte d'entrèe Γ, mais contenant uniquement des paramètres d'annotation.

Les Figures 9 et 10 présentent une définition inductive, sur la structure des termes du langage source, de l'algorithme d'inférence des contraintes.

L'algorithme fait référence à plusieurs fonctions auxiliaires. La notation $\phi = \mathit{fresh}(\sigma)$ définit ϕ en étant une version décorée de σ uniquement avec des paramètres d'annotation n'ayant aucune occurrence ailleurs dans la règle. L'appel à la fonction $\mathit{split}(\Delta, M, N)$, définie dans la Figure 12, renvoit un triplet $(\Delta_1, \Delta_2, \Theta_1)$, où Δ_1 et Δ_2 constituent une séparation des déclarations de Δ, tel que Δ_1 et Δ_2 contiendront les déclarations des variables libres dans M et N, respectivement. Si $x{:}\phi^{\mathsf{p}}$ est une déclaration partagée, l'ensemble Θ_1 contiendra en plus l'inéquation $\mathsf{p} \sqsupseteq \mathsf{q}_1 + \mathsf{q}_2$, où $x{:}\phi^{\mathsf{q}_1}$ et $x{:}\phi^{\mathsf{q}_2}$ sont les déclarations à employer pour Δ_1 et Δ_2, respectivement. L'appel $(\phi \leq \psi) = \Theta$ renvoit, pour deux types ϕ et ψ, l'ensemble des contraintes Θ pour lequel il se vérifie que $\Theta \vdash \phi \leq \psi$. Nous avons inclu sa définition dans la Figure 11.

$$\frac{\Delta \equiv \overline{x_i : {\phi_i}^{\mathsf{p}_i}}}{\overline{\mathsf{p}_i \sqsupseteq \top}\,;\Delta, x : \phi^{\mathsf{p}} \vdash x \Rightarrow x : \phi}$$

$$\frac{\Sigma(\pi) = \phi \quad \Delta \equiv \overline{x_i : {\phi_i}^{\mathsf{p}_i}}}{\overline{\mathsf{p}_i \sqsupseteq \top}\,;\Delta \vdash \pi \Rightarrow \pi : \phi}$$

$$\frac{\Theta\,;\Delta, x : \phi^{\mathsf{p}} \vdash M \Rightarrow X : \psi \quad \phi = \mathit{fresh}(\sigma) \quad \mathsf{p}\ \text{fresh}}{\Theta\,;\Delta \vdash \lambda x{:}\sigma.M \Rightarrow \lambda x{:}\phi^{\mathsf{p}}.X : \phi^{\mathsf{p}} \multimap \psi}$$

$$\frac{\Theta_2\,;\Delta_1 \vdash M \Rightarrow X : {\phi_1}^{\mathsf{p}} \multimap \psi \quad \Theta_3\,;\Delta_2 \vdash N \Rightarrow Y : \phi_2 \quad \mathit{split}(\Delta, M, N) = (\Delta_1, \Delta_2, \Theta_1) \quad \begin{array}{c}\Delta_2 \equiv \overline{x_i : {\phi_i}^{\mathsf{q}_i}} \\ (\phi_2 \leq \phi_1) = \Theta_4\end{array}}{\Theta_1, \Theta_2, \Theta_3, \Theta_4, \overline{\mathsf{q}_i \sqsupseteq \mathsf{p}}\,;\Delta \vdash MN \Rightarrow XY : \psi}$$

$$\frac{\Theta_2\,;\Delta_1 \vdash M_1 \Rightarrow X_1 : \phi_1 \quad \Theta_3\,;\Delta_2 \vdash M_2 \Rightarrow X_2 : \phi_2 \quad \mathit{split}(\Delta, M, N) = (\Delta_1, \Delta_2, \Theta_1) \quad \begin{array}{c}\Delta_1 \equiv \overline{x_{1,i} : {\phi_{1,i}}^{\mathsf{q}_{1,i}}} \\ \Delta_2 \equiv \overline{x_{2,i} : {\phi_{2,i}}^{\mathsf{q}_{2,i}}}\end{array}}{\Theta_1, \Theta_2, \Theta_3, \overline{\mathsf{q}_{1,i} \sqsupseteq \mathsf{p}_1}, \overline{\mathsf{q}_{2,i} \sqsupseteq \mathsf{p}_2}\,;\Delta \vdash \langle M_1, M_2 \rangle \Rightarrow \langle X_1, X_2 \rangle^{\mathsf{p}_1,\mathsf{p}_2} : {\phi_1}^{\mathsf{p}_1} \otimes {\phi_2}^{\mathsf{p}_2}}$$

Figure 9: Algorithme d'inférence d'inéquations de contrainte

$$\frac{\Theta_2\,;\Delta_1 \vdash M \Rightarrow X : \mathsf{bool} \quad \Theta_3\,;\Delta_2 \vdash N_1 \Rightarrow Y_1 : \phi_1 \quad \Theta_4\,;\Delta_2 \vdash N_2 \Rightarrow Y_2 : \phi_2 \quad \begin{array}{l}\phi = \mathit{fresh}(\phi_1{}^{\circ})\\ (\phi_1 \leq \phi) = \Theta_5\\ (\phi_2 \leq \phi) = \Theta_6\\ \mathit{split}(M, \langle N_1, N_2\rangle) = (\Delta_1, \Delta_2, \Theta_1)\end{array}}{\Theta_1, \Theta_2, \Theta_3, \Theta_4, \Theta_5, \Theta_6\,;\Delta \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 \Rightarrow \mathsf{if}\ X\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2 : \phi}$$

$$\frac{\Theta_1\,;\Delta, x : \phi_1{}^{\mathsf{p}} \vdash M \Rightarrow X : \phi_2 \quad (\phi_1 \leq \phi_2) = \Theta_2 \quad \Delta \equiv \overline{x_i : \psi_i^{\mathsf{q}_i}} \quad \phi_1 = \mathit{fresh}(\sigma) \quad \mathsf{p}\ \text{fresh}}{\Theta_1, \Theta_2, \overline{\mathsf{q}_i \sqsupseteq \top}, \mathsf{p} \sqsupseteq \top\,;\Delta \vdash \mathsf{fix}\ x{:}\sigma.M \Rightarrow \mathsf{fix}\ x{:}\phi_1.X : \phi_2}$$

$$\frac{\Theta_2\,;\Delta_1 \vdash M \Rightarrow X : \phi_1{}^{\mathsf{p}_1} \otimes \phi_2{}^{\mathsf{p}_2} \quad \Theta_3\,;\Delta_2, x_1 : \phi_3{}^{\mathsf{p}_3}, x_2 : \phi_4{}^{\mathsf{p}_4} \vdash N \Rightarrow Y : \psi \quad \begin{array}{l}\mathsf{p}_3, \mathsf{p}_4\ \text{fresh}\\ (\phi_1{}^{\mathsf{p}_1} \otimes \phi_2{}^{\mathsf{p}_2} \leq \phi_3{}^{\mathsf{p}_3} \otimes \phi_4{}^{\mathsf{p}_4}) = \Theta_4\\ \mathit{split}(\Delta, M, N) = (\Delta_1, \Delta_2, \Theta_1)\end{array}}{\Theta_1, \Theta_2, \Theta_3, \Theta_4\,;\Delta \vdash \mathsf{let}\ \langle x_1, x_2\rangle = M\ \mathsf{in}\ N \Rightarrow \mathsf{let}\ \langle x_1, x_2\rangle^{\mathsf{p}_3,\mathsf{p}_4} = X\ \mathsf{in}\ Y : \psi}$$

Figure 10: Algorithme d'inférence d'inéquations de contrainte (suite)

$$\overline{(\mathbb{G} \leq \mathbb{G}) = \emptyset}$$

$$\frac{(\phi_2 \leq \phi_1) = \Theta_1 \quad (\psi_1 \leq \psi_2) = \Theta_2}{(\phi_1{}^{\mathsf{p}_1} \multimap \psi_1 \leq \phi_2{}^{\mathsf{p}_2} \multimap \psi_2) = \Theta_1, \Theta_2, \mathsf{p}_2 \sqsupseteq \mathsf{p}_1}$$

$$\frac{(\phi_1 \leq \phi_2) = \Theta_1 \quad (\psi_1 \leq \psi_2) = \Theta_2}{(\phi_1{}^{\mathsf{p}_1} \otimes \psi_1{}^{\mathsf{q}_1} \leq \phi_2{}^{\mathsf{p}_2} \otimes \psi_2{}^{\mathsf{q}_2}) = \Theta_1, \Theta_2, \mathsf{p}_1 \sqsupseteq \mathsf{p}_2, \mathsf{q}_1, \sqsupseteq \mathsf{q}_2}$$

Figure 11: Définition de la fonction auxiliaire $(- \leq -)$

5.2 Correction de l'inférence des contraintes

La clé de la preuve de correction de l'algorithme d'inférence des contraintes se trouve dans la relation entre les configurations correspondantes à chaque étape de l'algorithme et les séquents d'un fragment de $\mathrm{NLL}^{\forall\leq}$, le système $\mathrm{NLL}^{\forall\nu\leq}$ de types minimums. Les règles de $\mathrm{NLL}^{\forall\nu\leq}$ qui doivent être modifiées, par rapport aux celles de $\mathrm{NLL}^{\forall\leq}$, sont réunies dans la Figure 13.

Lemme 5.1 (Correction pour $\mathrm{NLL}^{\forall\nu\leq}$)
Si $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$, alors $\Theta\,;\Delta \underset{\mathrm{NLL}^{\forall\nu\leq}}{\vdash} X : \phi$.

Démonstration. Voir la démonstration du Théorème 6.1.9. □

L'inclusion dans l'espace des décorations du séquent source $\Delta^\circ \vdash X^\circ : \phi^\circ$ suit en tant que corollaire du lemme ci-dessus et des propriétés de $\mathrm{NLL}^{\forall\leq}$.

Théorème 5.2 (Correction)
Si $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$, alors $\Delta[\theta] \underset{\mathrm{NLL}^{\nu\leq}}{\vdash} X[\theta] : \phi[\theta]$, pour tout $\theta \models \Theta$.

Démonstration. Voir la démonstration du Théorème 6.1.10. □

La preuve complémentaire de complétude, c'est-à-dire d'inclusion de l'espace des décorations dans l'ensemble de substitutions obtenue à partir de Θ, constitue une preuve *constructive* sur l'expressivité du polymorphisme général d'annotations, notamment sur le fait que pour chaque séquent du language source il existe un séquent du langage intermédiare (celui trouvé par notre algorithme) qui peut être interpreté comme une description compacte de l'espace des décorations du séquent du language source sous-jacent.

Théorème 5.3 (Complétude)
Si $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$ et $-\,;\Gamma \vdash N : \psi$ et une décoration dans $\mathrm{NLL}^{\nu\leq}$ de $\Delta^\circ \vdash X^\circ : \phi^\circ$, alors il existe une solution $\theta \models \Theta$, tel que $\Gamma \equiv \Delta[\theta]$, $N \equiv X[\theta]$ et $\psi \equiv \phi[\theta]$.

Démonstration. Voir la démonstration du Théorème 6.1.11. □

5.3 Solution optimale d'un système de contraintes

Une fois que nous avons complété la première étape avec succès, nous pouvons nous concentrer sur le calcul de la solution optimale, la plus petite solution θ qui vérifie l'ensemble

$$split(-, M_1, M_2) = (-, -, \emptyset)$$

$$split((\Delta, x{:}\phi^{\mathsf{p}}), M_1, M_2) = \begin{cases} ((\Delta'_1, x{:}\phi^{\mathsf{p}}), \Delta'_2, \Theta), \\ \quad \text{si } x \in FV(M_1), \text{ mais } x \notin FV(M_2); \\ (\Delta'_1, (\Delta'_2, x{:}\phi^{\mathsf{p}}), \Theta), \\ \quad \text{si } x \in FV(M_2), \text{ mais } x \notin FV(M_1); \\ ((\Delta'_1, x{:}\phi^{\mathsf{p}_1}), (\Delta'_2, x{:}\phi^{\mathsf{p}_2}), (\Theta, \mathsf{p} \sqsupseteq \mathsf{p}_1 + \mathsf{p}_2)), \\ \quad \text{sinon;} \end{cases}$$

$$\text{et où } split(\Delta, M_1, M_2) = (\Delta'_1, \Delta'_2, \Theta).$$

Figure 12: Définition de la fonction auxiliaire $split(-,-,-)$

$$\frac{\Theta\,;\Gamma_1 \vdash M : {\phi_1}^t \multimap \psi \quad \Theta\,;\Gamma_2 \vdash N : \phi_2 \quad \Theta \triangleright |\Gamma_2| \sqsupseteq t \quad \Theta \vdash \phi_2 \leq \phi_1}{\Theta\,;\Gamma_1, \Gamma_2 \vdash MN : \psi} \multimap_{\mathcal{E}}$$

$$\frac{\Theta\,;\Gamma_1 \vdash M : {\phi_1}^{t_1} \otimes {\phi_2}^{t_2} \quad \Theta\,;\Gamma_2, x_1 : {\psi_1}^{t'_1}, x_2 : {\psi_2}^{t'_2} \vdash N : \psi \quad \begin{array}{c} \Theta \triangleright t_i \sqsupseteq t'_i \\ \Theta \vdash \phi_i \leq \psi_i \\ (i = 1, 2) \end{array}}{\Theta\,;\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ \langle x_1, x_2 \rangle^{t'_1, t'_2} = M\ \mathsf{in}\ N : \psi} \otimes_{\mathcal{E}}$$

$$\frac{\Gamma_1 \vdash M : \mathsf{bool} \quad \Gamma_2 \vdash N_1 : \sigma_1 \quad \Gamma_2 \vdash N_2 : \sigma_2 \quad \begin{array}{c} \sigma_1 \leq \sigma \\ \sigma_2 \leq \sigma \end{array}}{\Gamma_1, \Gamma_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \sigma} \text{Conditional}$$

Figure 13: Règles de typage modifiées de $\text{NLL}^{\forall\nu\leq}$

d'inéquations Θ que nous avons obtenu en sortie. Lorsque nous parlons de plus 'petite' solution, nous faisons référence a la substitution close θ^{opt} de l'espace des solutions de Θ,

$$[\Theta] \stackrel{\text{def}}{=} \{\theta \mid \theta \models \Theta\},$$

tel que $\theta^{\text{opt}} \sqsubseteq \theta$ pour tout autre $\theta \in [\Theta]$, et où

$$\theta_1 \sqsubseteq \theta_2 \stackrel{\text{def}}{=} \theta_1(\mathsf{p}) \sqsubseteq \theta_2(\mathsf{p}), \text{ pour tout } \mathsf{p} \in dom(\theta_1).$$

Il faut noter que la relation d'ordre que nous avons choisie est compatible avec l'ordre entre séquents décorés, c'ést-à-dire que si $\theta_1 \sqsubseteq \theta_2$, nous pouvons aussi affirmer $X[\theta_1] \sqsubseteq X[\theta_2]$ (en considérant uniquement les solutions θ_1 et θ_2 qui recouvrent X).

L'algorithme d'inférence des annotations n'opère qu'avec des ensembles des contraintes de la forme $\mathsf{p} \sqsupseteq t$ où t est bien un paramètre d'annotation ou bien $\top$. Il n'est pas difficile de voir que tout ensemble de contraintes de cette forme-là forme un treillis complet.

Proposition 5.4
Pour tout $\Theta \equiv \overline{\mathsf{p}_i \sqsupseteq t_i}$, $\langle[\Theta];\sqsubseteq\rangle$ forme un treillis complet non-vide.

Démonstration. Voir la démonstration de la Proposition 6.2.1 □

La solution optimale θ^{opt} est le plus petit élément du treillis $\sqcap[\Theta]$. Une façon standard, mais aussi très élégante, de suggérer un algorithme général de calcul effectif de $\sqcap[\Theta]$, consiste à montrer qu'il est possible de caracteriser la solution optimale comme le plus petit point fixe d'une fonction continue

$$\mathcal{F}_\Theta : (P \to \mathbb{A}) \to (P \to \mathbb{A}) \quad \text{où} \quad P = FA(\Theta),$$

définie sur l'espace élargi de toutes les substitutions closes ayant comme domaine $FA(\Theta)$. En effet, les points fixes de la fonction

$$\mathcal{F}_\Theta(\theta)(\mathsf{p}) \stackrel{\text{def}}{=} \bigsqcup\{\theta(t) \mid \mathsf{p} \sqsupseteq t \text{ est dans } \Theta\}$$

forment l'ensemble de toutes les substitutions closes qui vérifient Θ. Etant donné que $\mathcal{F}_\Theta$ est une fonction continue, nous pouvons calculer le plus petit point fixe $\mu(\mathcal{F}_\Theta)$ par approximation, en utilisant un résultat bien connu de la théorie d'ordres. Ainsi, si $\overline{\mathsf{p}_i} = FA(\Theta)$,

$$\mu(\mathcal{F}_\Theta) = \bigsqcup_{i \geq 0} \mathcal{F}_\Theta{}^i(\overline{\langle\bot/\mathsf{p}_i\rangle}).$$

5.4 Inférence des annotations pour l'analyse contextuelle

En utilisant la stratégie qui consiste à trouver la décoration optimale, nous pouvons, très brièvement, enoncer une règle de typage pour les définitions, adaptée à la compilation séparée. Une possibilité, consistante avec $\forall_{\mathcal{I}}$, est la règle suivante.

$$\frac{\Theta_1 \,;\Delta \vdash M \Rightarrow X : \phi \quad \overline{\mathsf{p}_i} = FA(\phi) \quad \Theta_2 = \Theta_1 \restriction \overline{\mathsf{p}_i} \quad \Theta_3 = \Theta_1 \backslash \Theta_2}{\Theta_3 \,;\Delta \vdash \mathsf{let}\ x = M \Rightarrow \mathsf{let}\ x = \Lambda\overline{\mathsf{p}_i} \,|\, \Theta_2.X : \forall\overline{\mathsf{p}_i} \,|\, \Theta_2.\phi}$$

Etant donnée une définition $\mathsf{let}\ x = M$ dans un module, nous allons calculer X, la traduction de M dans le langage intermédiaire, et l'ensemble de contraintes Θ. Le type que nous allons sauvegarder dans l'interface du module est $\forall \overline{\mathsf{p}_i} \mid \Theta_2[\theta^{\mathrm{opt}}].\phi[\theta^{\mathrm{opt}}]$, où Θ_2 contient les inéquations de Θ_1 qui ont des paramètres d'annotation libres dans ϕ, car ce sont les seules qui peuvent rentrer en interaction avec les utilisations de la définition. La substitution optimale θ^{opt} est calculée à partir de Θ_3, le reste des inéquations de Θ_1 qui ne font pas référence à des paramètres de ϕ. Celà nous permet d'affirmer que le type dans l'interface ne contiendra que des paramètres dans $\overline{\mathsf{p}_i}$. Pour optimiser la définition elle-même nous pouvons utiliser l'analyse fournie par la décoration partielle $M[\theta^{\mathrm{opt}}]$.

Nous supposons que Δ contient les déclarations externes qui nous permettent de typer M correctement et qui sont donc de la forme

$$\Delta ::= x_1 : (\forall \overline{\mathsf{p}_{1,i}} \mid \Theta_1.\phi_1)^{\mathsf{q}_1}, \ldots, x_n : (\forall \overline{\mathsf{p}_{n,i}} \mid \Theta_n.\phi_n)^{\mathsf{q}_n}.$$

Nous ne pouvons pas toutefois utiliser l'algorithme d'inférence des contraintes tel quel, car, maintenant, les déclarations externes font référence à des types généralisés. Nous allons donc remplacer la première règle de la Figure 9 par la règle suivante :

$$\frac{\vartheta \equiv \overline{\langle \mathsf{p}'_i/\mathsf{p}_i \rangle} \quad \mathsf{p}'_i \text{ fresh} \quad \Delta \equiv \overline{x_i : \phi_i{}^{\mathsf{q}_i}}}{\Theta[\vartheta], \overline{\mathsf{q}_i \sqsupseteq \top}\,; \Delta, x : (\forall \overline{\mathsf{p}_i} \mid \Theta.\phi)^{\mathsf{p}} \vdash x \Rightarrow x\,\vartheta : \phi[\vartheta]}$$

Il est évident que la nouvelle règle génère tout simplement une instance $\phi[\vartheta]$ (un type monovariant) en remplaçant chaque paramètre p_i par un nouveau paramètre p'_i. L'ensemble d'inéquations $\Theta[\vartheta]$ est ensuite rajouté à l'ensemble existant pour préserver, tout au long de l'inférence, les contraintes que les annotations de $\phi[\vartheta]$ doivent respecter à fin que l'analyse finale soit consistante avec les analyses des définitions dans d'autres modules.

6 Analyse structurelle abstraite

Dans cette section, nous présentons un cadre d'analyse statique plus général, un cadre donc abstrait, dans lequel l'analyse linéaire n'est qu'un cas particulier. Notre proposition de cadre abstrait permet de formuler d'autres types d'analyse, dites d'usage ou 'structurelles', en introduisant des ensembles d'annotations différents, où chaque annotation fait référence à un schéma d'usage de ressources différent. Notre motivation a été de montrer qu'il est possible d'exprimer des analyses statiques d'usage, tel que l'analyse affine, d'utilisation effective (*neededness*) ou bien de partage et d'absence (*sharing and absence*), dans un même cadre théorique, et d'en dégager quelques principes de base. D'un point de vue purement logique, les systèmes issues des instances du cadre abstrait sont toutes des logiques dites à de *modalités multiples* [39, 43, 14].

6.1 La notion de structure d'annotations

Le cadre abstrait repose sur la notion de structure d'annotations, définie comme un quintuple

$$\mathbb{A} \equiv \langle A, \sqsubseteq, \mathbf{0}, \mathbf{1}, + \rangle,$$

où

- $\langle A, \sqsubseteq \rangle$ dénote un ensemble ordonné avec un élément maximum $\top$ et où $a \sqcup b$ doit exister pour toute paire d'éléments a et b.

- Les annotations abstraites $\mathbf{0}, \mathbf{1} \in A$ sont deux éléments de l'ensemble, que nous utiliserons pour annoter les annotations des règles d'affaiblissement et d'identité, respectivement.

- L'operateur binaire de contraction $+ : A \times A \to A$, utilisé alors pour annoter la règle homonyme, doit satisfaire les propriétés de commutativité, d'associativité et de distributivité énnoncées ci-dessous.

$$
\begin{aligned}
a + b &= b + a \\
(a + b) + c &= a + (b + c) \\
a \sqcup (b + c) &= (a \sqcup b) + (a \sqcup c)
\end{aligned}
$$

Une analyse structurelle donnée est ainsi déterminée par une structure d'annotations $\mathbb{A}$ adéquate en plus des règles de la théorie de types de l'analyse linéaire, à l'exception des règles structurelles, que nous devons remplacer par un jeu de règles comportant les annotations abstraites[9] :

$$\frac{\Theta \triangleright t \sqsupseteq \mathbf{1}}{\Theta\,;x : \phi^t \vdash x : \phi}\ \text{Identity}$$

$$\frac{\Theta\,;\Gamma \vdash M : \psi \quad \Theta \triangleright t \sqsupseteq \mathbf{0}}{\Theta\,;\Gamma, x : \phi^t \vdash M : \psi}\ \text{Weakening}$$

$$\frac{\Theta\,;\Gamma, x_1 : \phi^{t_1}, x_2 : \phi^{t_2} \vdash M : \psi \quad \Theta \triangleright t \sqsupseteq t_1 + t_2}{\Theta\,;\Gamma, x : \phi^t \vdash M[x/x_1, x/x_2] : \psi}\ \text{Contraction}$$

Les règles structurelles ci-dessus nous permettent d'avoir une idée sur la rélation structurelle que chaque élément abstrait est sensé exprimer, et que nous pourrions résumer, de façon informelle, comme ceci (où $x{:}\phi^a$ est une déclaration arbitraire) :

si	alors
$a \sqsupseteq \mathbf{0}$	$x{:}\phi^a$ peut être effacée
$a \sqsupseteq \mathbf{1}$	$x{:}\phi^a$ peut être utilisée au moins une fois
$a \sqsupseteq b_1 + b_2$	$x{:}\phi^a$ peut être dupliquée

Les propriétés qu'une structure d'annotations $\mathbb{A}$ doit observer sont là pour nous assurer que les propriétés étudiées dans les sections précédentes restent toujours valides dans le cadre abstrait, notamment le lemme de la substitution, qui est au cœur de la correction de l'analyse statique linéaire. Aussi, nous avons voulu conserver d'autres propriétés, comme la propriété de passage (incarné par la règle Transfer), que nous aurions pu rajouter en tant que règle supplémentaire *de facto*.

L'éxistence d'un élément maximum $\top$ est nécessaire pour pouvoir annoter correctement un terme récursif et pour nous assurer de l'existence d'au moins une solution à l'analyse

[9] Voir la Figure 7.1.

(correspondante au fragment intuitionniste de la logique sous-jacente)[10]. La commutativité et associativité de $+$ ne font qu'exprimer le fait que l'ordre d'application de la règle de contraction à des variables au sein d'un contexte ne doit pas être important. La propriété de distributivité joue un rôle fondamental dans la preuve de l'admissibilité de la règle Transfer (de passage) et dans la preuve du lemme de la substitution.

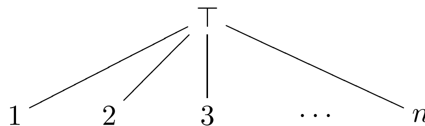
6.2 Quelques exemples familiers

Pour motiver les observations des paragraphes précédants, la Figure 14 réunit trois exemples connus : l'analyse affine, d'utilisation effective, et de partage et d'absence. L'analyse d'utilisation effective n'est ni plus ni moins que la version structurelle de l'analyse de fonctions strictes (*strictness analysis*), traditionnellement formulée dans le cadre de l'interprétation abstraite.

Il faut noter que beaucoup d'ensembles d'annotations, comme ceux qui visent à des analyses plus fines, sont basés sur le nombre de fois qu'une variable est utilisée dans un contexte (c'est-à-dire, qui *comptent* le nombre d'occurrences d'une variable). En général ces analyses sont non-distibutifs et, par la suite, ne vérifient pas le lemme de la substitution. Un exemple trivial est celui qui consiste à prendre comme ensemble d'annotations

$$\mathbb{A} \equiv \langle \mathbb{N} \cup \{\top\}, \sqsubseteq, 0, 1, +\rangle$$

où $n \in \mathbb{N}$ dénote la propriété "utilisé n fois", c'est-à-dire que $+$ dénoterait la somme des naturels (plus un élément $\top$), ordonnés comme suit :



En effet, $\mathbb{A}$ n'est pas distributif, car $n \not\sqsubseteq n + n$ pour tout $n \neq 0$.

7 Conclusion

Nous venons de présenter l'analyse linéaire générale, une théorie d'analyse statique consacrée à la détéction des valeurs qui sont utilisées une seule fois dans un contexte donné. La notion d'usage de l'analyse linéaire est celle héritée de la logique linéaire de Girard : une définition annotée linéairement ne sera jamais ni dupliquée ni ignorée par aucune stratégie d'évaluation. Cela donne à l'analyse linéaire une portée plus importante que celle d'autres analyses de compléxité comparable [67, 35], mais, en même temps, elle est plus conservative et, par conséquent, moins utile dans certains cas pratiques. Nous avons également montré le lien entre l'analyse linéaire optimale et la notion de décoration optimale, issue de la théorie de la démonstration de la logique linéaire intuitionniste [26].

La notion de décoration semble idéale pour donner une idée sur l'expressivité de l'analyse linéaire générale et, en particulier, du polymorphisme d'annotations. En effet, il nous a été

[10]En effet, nous avons $\mathbf{0} \sqsubseteq \top$ et $\mathbf{1} \sqsubseteq \top$ par définition, et $\top + \top = \top$ par distributivité. Il faut noter que nous n'exigeons pas que $\mathbb{A}$ ait dans tous les cas un élément plus petit. Si celui-ci n'existe pas, il faudra le rajouter artificiellement (en prenant comme structure des annotations $\mathbb{A}_\perp$) si nous voulons calculer la décoration optimale tel que nous l'avions décrite précédemment.

L'analyse affine

$$\mathbb{A} \stackrel{\text{def}}{=} \langle \{\top, \leq\!1\}, \sqsubseteq, \leq\!1, \leq\!1, + \rangle$$

où

$$\leq\!1 \sqsubseteq \top$$

et

$$\begin{array}{ccccc} \leq\!1 & + & \leq\!1 & = & \top \\ \leq\!1 & + & \top & = & \top \\ \top & + & \leq\!1 & = & \top \\ \top & + & \top & = & \top \end{array}$$

L'analyse d'utilisation effective

$$\mathbb{A} \stackrel{\text{def}}{=} \langle \{\top, \geq\!1\}, \sqsubseteq, \top, \geq\!1, + \rangle,$$

où

$$\geq\!1 \sqsubseteq \top$$

et

$$\begin{array}{ccccc} \geq\!1 & + & \geq\!1 & = & \geq\!1 \\ \geq\!1 & + & \top & = & \top \\ \top & + & \geq\!1 & = & \top \\ \top & + & \top & = & \top \end{array}$$

L'analyse de partage et d'absence

$$\mathbb{A} \stackrel{\text{def}}{=} \langle \{\top, 0, \geq\!1\}, \sqsubseteq, 0, \geq\!1, + \rangle,$$

où

$$0 \sqsubseteq \top \quad \geq\!1 \sqsubseteq \top$$

et

$$\begin{array}{ccccc} 0 & + & 0 & = & 0 \\ 0 & + & \geq\!1 & = & \top \\ 0 & + & \top & = & \top \\ \geq\!1 & + & 0 & = & \top \\ \geq\!1 & + & \geq\!1 & = & \geq\!1 \\ \geq\!1 & + & \top & = & \top \\ \top & + & 0 & = & \top \\ \top & + & \geq\!1 & = & \top \\ \top & + & \top & = & \top \end{array}$$

Figure 14: Trois exemples familiers des analyses structurelles

possible de construire un type généralisé très particulier, pouvant être interprété comme une description compacte de l'espace de toutes les décorations d'un terme donné du langage source. Nous avons également suggéré une stratégie d'analyse statique qui utilise le polymorphisme d'annotations pour donner une solution satisfaisante au problème de l'analyse statique en présence de modules compilés séparément.

Chapter 1

Introduction

1.1 Motivations

As compiler technology has grown in maturity and sophistication, the need for non-trivial static analysis techniques has become more pressing. This is especially true for those languages not based upon the 'classical' von Neumann evaluation model of current computers. Functional and logic-based languages are both good examples of such languages. The information obtained through static analysis allows modern optimising compilers for these languages to perform more aggressive optimisations, approaching in some cases the overall performance profiles of their imperative counterparts. The ever-growing acceptance of functional and logic-based languages is due not only to the availability of more computational power, but also to the fact that, thanks to modern compiling technology, they can now be regarded as serious alternatives to the more popular imperative and object-oriented languages.

Intuitively, given an input program, the goal of static analysis is to determine at compile-time various properties about the program's run-time behaviour that the compiler may later use to validate the application of particular optimisations. Many properties of interest, especially those about the dynamic behaviour of programs, are undecidable, so the properties computed by static analysers are usually conservative approximations. The literature on static analysis has grown huge over the years, so it would be impossible to provide the reader with a fair survey. The interested reader is referred to Nielson and Hankin's book for an introduction [48].

The properties we shall be studying in this thesis belong to the family of properties known in the literature under the name of *usage properties*. There seems to be as many notions of usage in existence as there are *usage static analyses*, or almost. We may classify usage analyses into two broad families: those based on a denotational description of the source language, and the more recent analyses, themselves based on ideas inpired by Girard's linear logic [30], which include the usage type systems we study here. The latter began their existence first as *usage logics*, of which there are many interesting examples in the literature [65, 13, 14, 39, 70, 4, 20, 43][1].

[1] Johnsson's early system of 'sharing and absence' analysis [41] provides an example of analysis for which the notion of usage adopts a more denotational flavour.

1.1.1 Structural properties

Linear logic divides values into two sorts: *linear* and *non-linear* (or *intuitionistic*). Non-linear values may be used any number of times, whereas linear values may only be used exactly once. All values are, unless explicitly stated, linear by default, which means that functions are not allowed to use their arguments more than once, nor to completely ignore them. The type of such functions is conventionally written $\sigma \multimap \tau$, where σ is the type of the argument and τ is the type of the result. Functions that are permitted to use their arguments any number of times, or none at all, have type $!\sigma \multimap \tau$, where $!\sigma$ is the notation for the type of non-linear values. The linear restriction on function arguments is traditionally formulated by introducing explicit rules, called *structural rules*, that allow only variables of non-linear type to be either duplicated or discarded. The familiar logical formulation of the structural rules is shown below, where the restrictions on variables, according to the Curry-Howard correspondance, take the form of restrictions on context formulae.

$$\frac{\Gamma \vdash \tau}{\Gamma, !\sigma \vdash \tau}\text{ Weakening} \qquad \frac{\Gamma, !\sigma, !\sigma \vdash \tau}{\Gamma, !\sigma \vdash \tau}\text{ Contraction}$$

The theories of static analysis we study here use the structural rules in a fundamental way to distinguish between properties, hence our preference to call them *structural usage analysis*, and the properties they capture, *structural properties*.

Many useful well-known usage static analyses take inspiration from linear logic, but are not formally based on linear logic. There is a good reason for not being too close to linear logic, especially if one is interested in notions of usage that would be useful, for instance, to model sharing. It is simply not true that 'linear' can be taken to mean 'not shared' for any given implementation [18]. These two notions can be seen to coincide, or at least to be compatible, in restricted contexts [65, 18, 61], so their practical usefulness has been seriously compromised[2]. As our usage type systems are well-behaved with respect to any reduction strategy, they must necessarily be less expressive than static analyses especially designed for a particular reduction strategy, so this is another good reason for sometimes not trying to be too close to the notion of usage suggested by linear logic.

1.1.2 Applications

A practical example of a usage analysis that has been applied with a reasonable success is a variation on the theme of *affine analysis*, which is applied in the Glasgow Haskell Compiler to avoid updating environment closures that are accessed at most once [62, 68, 67, 66]. Wright, among others, realised that *relevance analysis* (which is aimed at detecting values that are used once or more than once) could be used to approximate strictness properties [69, 4, 20].

As a way of illustration, we shall show how linearity analysis can be used to justify a simple *inlining* transformation. Informally, inlining consists in replacing a reference to a definition by the definition body itself. We slightly generalise this situation to bindings in general. Transformations like this one are extremely important in compilers (of any language), especially functional language compilers. An interesting case is when a definition is used exactly once, in which case we can safely inline the definition, without risking any undesirable recomputations (or bloating the code with duplicated definitions). Some compilers already

[2] This explains the subsequent lack of interest in the subject. We discuss this problem further in Subsection 3.7.3

have some sort of 'occurrence analyser' which uses variable occurrence information to help in the detection of some trivial cases of inlining, but a more accurate analysis would be preferable [51]. Another interesting case is when bindings are not used at all, a situation that can be easily detected by applying a simple *non-occurrence analysis*.

Perhaps, the most interesting feature of structural analysis is that many properties capturing different usage patterns can be uniformly described in the same logical framework [70, 14].

1.2 Annotated type systems

The different theories of structural analysis we introduce in this work are formulated as *annotated type systems*, which themselves belong to the (ever larger) class of *type-based static analysis* techniques. Type-based static analyses are formulated in terms of a typing relation assigning properties (types) to terms; the static analysis method itself, therefore, takes the form of a type inference algorithm, which generally consists of an extension of Hindley-Milner's type inference techniques [24]. Kuo and Mishra's type system of strictness types seems to have been the first such system [60].

The properties in a type-based analysis need not have any relation to the types of the source language; in fact, it is not even necessary that the source language be typed at all. Annotated type systems, on the other hand, are formulated in terms of an existing typed source language. The types inferred are commonly called *annotated types*, because they correspond to types of the source language annotated with static information. It is in this sense that the annotated type system is understood as a *refinement* of the base type system, as it is also capable of inferring base type information.

Annotated type systems share many of the advantages and the disadvantages of the type-based approach. Types and terms are ideal places for saving the result of the analysis. The annotations in terms are useful to guide and enable compiler optimisations, whereas the annotations in types are useful to convey static information, both internally and externally (to client modules, for instance). Type inference can usually be implemented more efficiently (than traditional semantics-based methods like abstract interpretation [22]) and can save much work when combined with the underlying source language type inference algorithm. However, annotated type systems have been known to be less expressive than their semantics-based counterparts, like abstract interpretation, and recovering some of the expressive power is not only non-trivial, but may sometimes result in algorithms that are as inefficient as other competing semantics-based methods.

We have chosen a Church-style formulation, so our type system infers annotated types for a slightly modified version of the source language whose *annotated terms* also carry static information, and that we call the *intermediate* or *target* language[3]. The type systems of both the source and target languages are related in the sense that typings in the target language correspond to typings of the source language, provided that we erase all static information.

1.3 Linearity analysis

Most of this thesis is concerned with the detailed study of *linearity analysis*, which is the simplest of all structural analyses. The reason for such a choice is that linearity analysis

[3] A Curry-style version can be easily obtained by erasing all type information from the terms.

has a solid theoretical background, linear logic itself, is simple to understand, and can be implemented efficiently. Linearity analysis seemed therefore ideal to give a, hopefully gentle, introduction to the theory and techniques behind structural analysis.

Linearity analysis distinguishes between linear and non-linear values, as linear logic does, except that these are encoded using annotations, which play the role of syntactic markers, indicating the presence or absence of the exponential '!' in types and terms [65]. The choice of having an annotated language was a natural one in this case, since the terms of the annotated target language of linearity analysis correspond, through the Curry-Howard correspondance, to the family of intuitionistic linear logic proofs that allow exponentials only in those places where Girard's classical translation of intuitionistic logic formulae into intuitionistic linear logic formulae would allow them (see Subsection 3.1.4). In other words, the terms of the target linear language encode the 'sub-girardian' proofs of Danos, Joinet and Schellinx [26]. Linearity analysis provides the static analysis view of finding the *optimal decoration*, or optimal translation, for intuitionistic proofs. In our case, we are interested in finding the best annotation conveying the most accurate information.

A standard way of finding the optimal or best linearity analysis for a given source language term consists in using constraint sets to register the *dependencies* existing between annotations in the target term. Roughly speaking, the inequations in the constraint set not only specify how the exponentials should propagate in the target term, but also point at those places where they are unavoidable. Solving the constraint set to find the smallest solution amounts to propagating the exponentials from where they are unavoidable to the required annotations in the target term. There is an elegant way to formalise this process as a fixpoint iteration in a space of solutions, which is an ordered structure of some sort. In fact, the annotated type systems of structural analysis are fundamentally constructed around the notion of annotation structure, which orders annotations in terms of their information content. For the case of linearity analysis, we use a 2-point annotation lattice specifying the absence of an exponential to be preferred over its presence. This order is not artificial; it is directly suggested by linear logic, as it corresponds to the inclusion of linear contexts into intuitionistic contexts.

An important property of all our annotated languages is that structural information is not corrupted by transformations that preserve the operational semantics of the source language. The operational semantics of our source language is formalised in terms of the usual notion of $\beta\eta$-reduction, which means that structural analysis may apply uniformly to many flavours of functional languages (although, as we have previously remarked, this fact may also be taken as one of its main limitations.) There is an exception to this observation. In fact, for our simple linearity analysis, typing information is not preserved across η-reductions of intuitionistic functions. This has motivated the extension of linearity analysis with a notion of *annotation subtyping*. Annotation subtyping is also useful in other contexts, as we shall later see.

1.4 Annotation polymorphism

For a given source term, finding the optimal annotated term for a given source term works reasonably well for self-contained programs, but cannot be easily adapted to realistic programs consisting of several separately compiled modules. The term 'self-contained' is applied here to those programs that do not use any definitions other than the ones provided in the program itself. Most clearly, even the simplest programs that programmers write contain

free variables that refer to definitions existing in exported modules (libraries), so assuming self-containedness is rather unrealistic.

The problem is that the annotations of a definition generally depend upon the annotations of the *use site* (i.e., the context that uses the definition), and the other way round. Assigning the optimal type to a definition would be wrong in some cases, as some use sites may require a weaker type to be typeable. Optimal types for definitions are therefore too restrictive. When compiling a library definition, the compiler has, as one usually expects, no information about the definition's use sites, so the only reasonable solution is to assume the worst, and assign the weakest possible type that would be compatible with all imaginable use sites. (This weakest type always exist and corresponds to Girard's translation.)

1.4.1 The poisoning problem

This observation also points at a weakeness of the analysis that has been identified by Wansbrough and Peyton-Jones as the 'poisoning problem' [68]. Different use sites for a definition may require distinct annotated types, but since a definition can only be assigned a *single* type, this type must necessarily be the weakest type compatible with all the use sites. But now the annotations of the different use sites must also be weakened, to level up with the weakened annotations of their corresponding definition, and so on. This information loss *en avalanche* is precisely a consequence of the fact that, in simple type systems, variables must assume a unique type inside typing contexts. Kuo and Mishra's simple type system of strictness types was also weak due to this same restriction.

For the case of strictness analysis, the solution to this lack of 'information locality' consisted in adding intersection types [19] of a particular sort, known as conjunctive types. A definition was then allowed to have different (but compatible) types for different use sites (contexts), thus augmenting significantly the accuracy of the analysis.

1.4.2 Contextual analysis

The solution we propose here is similar in spirit, although it adopts a slightly different form. It adds to our original structural analysis the possibility of assigning *polymorphic annotated types* to terms. We refer to the improved analysis allowing definitions to be assigned a set of annotated types as *contextual analysis*. In the context of usage type systems, annotation polymorphism is relatively new, and has been implemented only recently, but only in a restricted form [67]. What we call here contextual analysis is known under the name of 'polyvariant' analysis (as opposed to 'monovariant' analysis) in the flow-based static analysis community. Our approach is close in spirit to Gustavsson and Svenningsson's *bounded usage polymorphism* [35]. We sometimes use the term *constrained annotation polymorphism*, because of the fundamental use of constraint sets to restrict the values of bound annotation parameters. Both terms refer to the same entity, for exactly the same application in mind, that of describing families of annotated types. These should not be confused with 'bounded type polymorphism' and 'constrained type polymorphism', which denote distinct type disciplines[4]. The notation we use of typing judgments coupled with constraint sets is closer to constrained type polymorphism, though [1, 23, 49].

[4]After hesitating some time, the author preferred to employ the term *general annotation polymorphism* to refer to the type assignment scheme that allows for complete decoration sets to be compactly characterised using a single annotation-polymorphic term. The fact that we can *always* describe the set of all possible decorations of a given source language term using an annotation-polymorphic term of the extended theory is

1.4.3 Modular static analysis

Using annotation polymorphism, a term can be assigned different types for different use types, thus augmenting the accuracy of the analysis dramatically. We use constrained polymorphic types to give a satisfactory solution to the problem of analysing programs composed of several separately compiled modules. The idea is to assign polymorphic annotated types to definitions in modules. In particular, we are interested in the most general annotated type that has as instances all the annotated types that arise as annotation-monomorphic (monovariant) translations of the definition, and which necessarily constitute all the possible annotation-monomorphic typings. We refer to this set as the *decoration space* of the definition. This strategy is similar to the one used to infer principal polymorphic types for languages like ML (among others) [44]. We shall be concentrating on a restricted form of annotation polymorphism, called *let-based annotation polymorphism*, that will provide the foundations necessary to prove the correctness of the more accurate annotation inference algorithm. The static analysis strategy that treats definitions in modules in this special way will be referred to as *modular static analysis*, and can be understood as an extension of our simple optimal analysis strategy for stand-alone programs.

1.5 Contributions

The main contribution of this thesis comprises the detailed study, from first principles, of a general framework for the static analysis of structural properties for a realistic language[5], including both annotation subtyping and polymorphism. We can summarise our contributions as follows:

- We prove a number of standard type-theoretic properties for various versions of linearity analysis, with and without annotation polymorphism. In particular, we prove the analysis correct with respect to the operational semantics of the source language and motivate the existence of optimal solutions.
- We introduce constrained annotation polymorphism in a general way and prove its correctness, and consider a restricted form of annotation polymorphism we shall use to derive a strategy of static analysis for modular programs.
- We derive two type inference algorithms, for which we prove syntactic soundness and completeness results.
- We introduce structural analysis in the form of an abstract framework that generalises all our previous results and apply it to a few case studies, including non-occurrence, affine and neededness analysis.

Although the approach is not new, we are not aware of the existence of any detailed presentation, from first principles, of a theory for the static analysis of structural properties. At the time of writing, however, two doctoral dissertations have been written on usage type

a corollary of the proof of soundness and completeness of the annotation inference algorithm we introduce in Section 6.1, and our intended meaning of the word 'general' in this context.

[5] We have not addressed explicitly the annotation of data type structures constructed from sums and products, but these are hardly problematic in our framework. An annotation rule for sums can be easily generalised from the rule given for annotating the conditional construct.

systems for affine analysis, involving similar ideas and developments [66, 33]. The difference is that the affine analyses proposed are specifically designed to provide best results for call-by-need languages only, compared to our version of affine analysis, which has a wider range of applicability, but which has been known to provide poor results.

The aim of the author was to bring some results on early work in the study of linear logic proof theory (most notably, on linear decorations) and mutiple modalities logics, into the realm of static analysis, extending the analyses on the way with annotation subtyping and polymorphism to augment their usefulness without invalidating their strong theoretical foundations.

1.6 Plan of the thesis

The thesis is logically organised into two parts. The first part, composed of Chapters 3 to 6, is concerned with linearity analysis, in all its flavours. The second part, comprising only Chapter 7, concerns the analysis of structural properties in a more abstract framework.

The following is a brief summary of the contents of each chapter:

Chapter 2 introduces the source language FPL and recalls some basic standard definitions from order theory that we shall be needing in later chapters.

Chapter 3 presents NLL, a simple annotated type system for the analysis of linearity properties, and shows how it relates to the more standard work in linear type theory. We illustrate how the result of the analysis could be exploited by formalising a simple inlining transformation. Also, we discuss the existence of optimal analyses.

Chapter 4 presents $\mathrm{NLL}^{\leq}$, an extended linearity analysis with a notion of annotation subtyping, which we prove correct.

Chapter 5 presents $\mathrm{NLL}^{\forall}$ and $\mathrm{NLL}^{\forall\leq}$, which extend the previous type systems of linearity analysis with a notion of annotation polymorphism. We consider a subset of $\mathrm{NLL}^{\forall\leq}$, called $\mathrm{NLL}^{\forall\mathrm{let}\leq}$, that will play an important role in the derivation of a type inference algorithm for constrained polymorphic definitions.

Chapter 6 discusses annotation inference, and describes two annotation inference algorithms for suitable fragments of $\mathrm{NLL}^{\leq}$ and $\mathrm{NLL}^{\forall\mathrm{let}\leq}$, respectively.

Chapter 7 presents a type system for the static analysis of structural properties as a generalisation of the type systems studied in the previous chapters. We discuss relevance analysis as an application to the analysis strictness properties.

Chapter 8 concludes and discusses further work.

The appendices provide the following complementary information:

Appendix A presents an alternative presentation of NLL, with an without annotation polymorphism, which we only briefly sketch.

1.7 Prerequisites

We assume basic knowledge on functional programming, type theory and logic, especially linear logic. We use very little knowledge from order theory, so the definitions given in the

preliminaries chapter should be enough.

A good knowledge on linear logic is recommended, but is otherwise not compulsory. In fact, even if linear logic is behind every single bit of type theory shown here (or almost), many key ideas can be grasped with no difficulty through NLL directly.

Chapter 2

Preliminaries

The main purpose of this short chapter is to introduce some of the notation we shall be using throughout, and to recall some standard definitions and results before setting straight into the matter of this thesis.

2.1 The source language

The prototypical simply-typed functional programming language we shall be using as our *source language* is a variant of Plotkin's PCF [55], comprising terms of different types (integer, boolean, function, and pair types). We coin this language FPL, an acronym for 'Functional Programming Language'.

2.1.1 Syntax

The notation is mostly standard. The set Λ^{FPL} of FPL *preterms*, ranged over by M and N, is inductively defined by the following grammar rules:

$$
\begin{array}{rcll}
M & ::= & \pi & \text{Primitive constant or operator} \\
 & \mid & x & \text{Variable} \\
 & \mid & \lambda x{:}\sigma.M & \text{Function abstraction} \\
 & \mid & MM & \text{Function application} \\
 & \mid & \langle M, M \rangle & \text{Pairing} \\
 & \mid & \mathsf{let}\ \langle x, x \rangle = M\ \mathsf{in}\ M & \text{Unpairing} \\
 & \mid & \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M & \text{Conditional} \\
 & \mid & \mathsf{fix}\ x{:}\sigma.M & \text{Fixpoint}
\end{array}
$$

In general, if $\mathcal{L}$ is any language, we shall write $\Lambda^{\mathcal{L}}$ for the set of its preterms. We assume similar conventions for other notations that involve explicit language names.

We assume our source language comes equipped with a predefined set of primitive contants and operators, collectively ranged over by π, which must contain the integers, the booleans, as well as a few standard arithmetic and relational operators:

$$
\begin{array}{rcll}
\pi & ::= & n \in \mathbb{N} & \text{Integer} \\
 & \mid & \mathsf{true} \mid \mathsf{false} & \text{Booleans} \\
 & \mid & + \mid - \mid < \mid = \cdots & \text{Primitive operators}
\end{array}
$$

$$
\begin{aligned}
\pi[\rho] &= \pi \\
x[\rho] &= \rho(x) \\
y[\rho] &= y \quad \text{if } y \notin dom(\rho)) \\
(\lambda x{:}\sigma.M)[\rho] &= \lambda x{:}\sigma.M[\rho \backslash \{x\}] \\
(MN)[\rho] &= M[\rho]N[\rho] \\
(\langle M_1, M_2 \rangle)[\rho] &= \langle M_1[\rho], M_2[\rho] \rangle \\
(\mathsf{let}\ \langle x_1, x_2 \rangle = M\ \mathsf{in}\ N)[\rho] &= \mathsf{let}\ \langle x_1, x_2 \rangle = M[\rho]\ \mathsf{in}\ N[\rho \backslash \{x_1, x_2\}] \\
(\mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2)[\rho] &= \mathsf{if}\ M[\rho]\ \mathsf{then}\ N_1[\rho]\ \mathsf{else}\ N_2[\rho] \\
(\mathsf{fix}\ x{:}\sigma.M)[\rho] &= \mathsf{fix}\ x{:}\sigma.M[\rho \backslash \{x\}]
\end{aligned}
$$

($\rho \backslash \{x_1, \ldots, x_n\}$ is the restriction of ρ to the domain $dom(\rho) \backslash \{x_1, \ldots, x_n\}$.)

Figure 2.1: Inductive definition of preterm substitution

Function abstraction, fixpoint and unpairing are variable-binding constructs. Any occurrences of x in $\lambda x{:}\sigma.M$ and $\mathsf{fix}\ x{:}\sigma.M$ are considered bound; any occurrences of x_1 and x_2 in N are bound in $\mathsf{let}\ \langle x_1, x_2 \rangle = M\ \mathsf{in}\ N$. Any other occurrences of variables are, conversely, free. We shall write $FV(M)$ for the set of free variables in M.

As usual, two preterms M and N that differ only on the names of their bound variables will be considered as syntactically equivalent. When this is necessary, we shall explicitly note this fact $M \equiv_\alpha N$.

If ρ is a finite function mapping variables into preterms, the notation $M[\rho]$ will be used to stand for the 'simultaneous' substitution of $\rho(x_i)$ for the free occurrences of x_i in M, where $x_i \in dom(\rho)$. Substitution is inductively defined on the structure of preterms in Figure 2.1.

We use the term *renaming substitution* for the special case of substitutions mapping variables into variables, and $M[N/x]$ as abbreviation for $M[\rho]$, where $dom(\rho) = \{x\}$ and $\rho(x) = N$. As usual, we must be careful to avoid the capture of any of the free variables in the image of ρ, so we assume that, whenever this problem may arise, we shall use instead a suitable α-equivalent representative of M.

2.1.2 Static semantics

Our source language is a typed language, in the sense that we shall only be interested in those preterms that are *well-typed*, in the sense that they can be assigned a type (for a given type-assignment of its free variables).

The set of FPL *types*, ranged over by σ and τ, is defined inductively by the following grammar rules:

$$
\begin{array}{lcll}
\sigma & ::= & \mathbb{G} & \text{Ground type} \\
 & | & \sigma \to \sigma & \text{Function space} \\
 & | & \sigma \times \sigma & \text{Cartesian product}
\end{array}
$$

The metavariable $\mathbb{G}$ ranges over the predefined *ground-type constants*

$$\mathbb{G} ::= \mathsf{int} \mid \mathsf{bool},$$

standing for the type of integer and boolean values, respectively. We assume that each primitive π has an associated predefined type in the type theory, called its *signature*, and written $\Sigma(\pi)$. These are summarised by the following table:

Primitive	$\Sigma(\pi)$
false, true	bool
n	int
$+, -$	int $\to$ int $\to$ int
$<, =$	int $\to$ int $\to$ bool
and so on...	

In order to give a static semantics to FPL, we shall consider *typing judgments*, which are typing assertions of the form

$$\Gamma \vdash M : \sigma,$$

stating that preterm M has type σ in the typing context Γ.

A *typing context* is a finite partial function mapping variables to types, and that we shall write (following Prawitz notation) as a sequence of variable *typing declarations*:

$$\Gamma ::= x_1 : \sigma_1, \ldots, x_n : \sigma_n.$$

We shall write Γ_1, Γ_2 for the union of the two partial maps Γ_1 and Γ_2. We assume the union to be well-formed, in the sense that their domains are required to be disjoint. Thus, in writing $\Gamma, x : \sigma$, we are implicitly assuming $x \notin dom(\Gamma)$.

A *type system* is a collection of rules comprising an inductive definition of the set of *valid* typing judgments. The type system of FPL is shown in Figure 2.2[1].

If J stands for a valid typing judgment, we shall write $\Pi(J)$ to refer to a *type derivation* or *proof* of J; that is, a type derivation Π with J as its conclusion.

The typing assertion $M : \sigma$ should be understood as an abbreviation for $- \vdash M : \sigma$, where '$-$' stands for the empty typing context. In this case, we must have that M is closed (Proposition 2.1.1a).

A preterm M is *typeable* if there exists Γ and σ for which $\Gamma \vdash M : \sigma$ is valid according to the typing rules of FPL. A preterm M is a *term* if it is typeable.

Proposition 2.1.1
Typings satisfy the following list of basic properties.

- *If* $\Gamma \vdash M : \sigma$, *then* $FV(M) \subseteq dom(\Gamma)$.
- *If* $\Gamma, x : \sigma \vdash M : \tau$ *and* $x \notin FV(M)$, *then* $\Gamma \vdash M : \tau$.
- *If* $\Gamma \vdash M : \sigma$ *and* $\Gamma \vdash M : \tau$, *then* $\sigma \equiv \tau$.

Proof. Easy induction on the derivation of $\Gamma \vdash M : \sigma$ for the first and last properties, and of $\Gamma, x : \sigma \vdash M : \tau$ for the second one. □

Part of the correctness of our static semantics is given by the following important Substitution Lemma, which states that substitution is well-behaved under certain reasonable typing conditions.

[1] We have preferred to formulate the type system of our source language using explicit structural rules, so that the rules visually match up with those of linearity analysis.

$$\frac{\Sigma(\pi) = \sigma}{- \vdash \pi : \sigma}\text{ Primitive} \qquad \frac{}{x : \sigma \vdash x : \sigma}\text{ Identity}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x{:}\sigma.M : \sigma \to \tau} \to_{\mathcal{I}} \qquad \frac{\Gamma_1 \vdash M : \sigma \to \tau \quad \Gamma_2 \vdash N : \sigma}{\Gamma_1, \Gamma_2 \vdash MN : \tau} \to_{\mathcal{E}}$$

$$\frac{\Gamma_1 \vdash M_1 : \sigma_1 \quad \Gamma_2 \vdash M_2 : \sigma_2}{\Gamma_1, \Gamma_2 \vdash \langle M_1, M_2 \rangle : \sigma_1 \times \sigma_2} \times_{\mathcal{I}} \qquad \frac{\Gamma_1 \vdash M : \sigma_1 \times \sigma_2 \quad \Gamma_2, x_1 : \sigma_1, x_2 : \sigma_2 \vdash N : \tau}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ \langle x_1, x_2 \rangle = M\ \mathsf{in}\ N : \tau} \times_{\mathcal{E}}$$

$$\frac{\Gamma_1 \vdash M : \mathsf{bool} \quad \Gamma_2 \vdash N_1 : \sigma \quad \Gamma_2 \vdash N_2 : \sigma}{\Gamma_1, \Gamma_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \sigma}\text{ Conditional}$$

$$\frac{\Gamma, x : \sigma \vdash M : \sigma}{\Gamma \vdash \mathsf{fix}\ x{:}\sigma.M : \sigma}\text{ Fixpoint}$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma, x : \sigma \vdash M : \tau}\text{ Weakening} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma, x_1 : \sigma, x_2 : \sigma \vdash M[x_1/x, x_2/x] : \tau}\text{ Contraction}$$

Figure 2.2: The typing rules of FPL

Lemma 2.1.2 (Substitution)
The following typing rule is admissible.

$$\frac{\Gamma_1 \vdash M : \sigma \quad \Gamma_2, x : \sigma \vdash N : \tau}{\Gamma_1, \Gamma_2 \vdash M[N/x] : \tau}\text{ Substitution}$$

Proof. Easy induction on the derivation of $\Gamma_1 \vdash M : \sigma$. □

2.1.3 Operational semantics

We formalise the operational behaviour of our simple source language by giving a notion of β-reduction. Let $\to \subseteq \Lambda^{\mathrm{FPL}} \times \Lambda^{\mathrm{FPL}}$ be the *reduction relation* obtained by taking the contextual closure of the following axioms:

$$\begin{aligned}
(\lambda x{:}\sigma.M)N &\to M[N/x] \\
\mathsf{let}\ \langle x_1, x_2 \rangle = \langle M_1, M_2 \rangle\ \mathsf{in}\ N &\to N[M_1/x_1, M_2/x_2] \\
\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 &\to N_1 \\
\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 &\to N_2 \\
\mathsf{fix}\ x{:}\sigma.M &\to M[\mathsf{fix}\ x{:}\sigma.M/x]
\end{aligned}$$

We also assume the existence of a number of δ-rules, specifying the behaviour of primitive operators; they all have the following general form:

$$\pi_{op}\ \pi_1, \ldots, \pi_n \to \pi, \quad \text{where} \quad [\![\pi]\!]([\![\pi_1]\!], \ldots, [\![\pi_n]\!]) = [\![\pi]\!],$$

where π_{op} denotes a primitive operator of arity n, and $\pi_1, \ldots, \pi_n$ are primitive constants. The right-hand side of the reduction rule must be interpreted as the application of the semantic function $[\![\pi]\!]$ to the arguments $[\![\pi_1]\!], \ldots, [\![\pi_n]\!]$, which correspond to the semantic elements of some predefined domain. The result of such an application is a constant π. All the intermediate languages we shall be studying are assumed to inherit these reduction rules from the source language, so we shall omit them in the future.

By *contextual closure*, we mean the reduction relation obtained by closing the above axioms with respect to the rule

$$\frac{M \to N}{\mathcal{C}[M] \to \mathcal{C}[N]}$$

where $\mathcal{C}$ is an *evaluation context*. Roughly speaking, an evaluation context is a preterm with a single distinguished hole '$-$' in it. (For instance, the evaluation context $\mathcal{C}[-] \equiv$ $\mathsf{if}\ -\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2$ allows the test of the conditional to be reduced.)

As expected, we shall write $\twoheadrightarrow$ for the reflexive and transitive closure of our reduction relation $\to$.

The operational and static semantics of our source language are related by the following Subject reduction result, stating that typing information is preserved across reductions.

Theorem 2.1.3 (Subject reduction)
If $\Gamma \vdash M : \sigma$ *and* $M \to N$*, then* $\Gamma \vdash N : \sigma$

Proof. By induction on $\to$-derivations and the Substitution Lemma 2.1.2. □

2.2 Partial orders

In this section, we review the necessary basic notions of order theory we shall be needing throughout. The reader is referred to Davey and Priestley's book [27] for a comprehensive introduction to the topic.

Definition 2.2.1 (Partial ordered set)
Let $\sqsubseteq \subseteq A \times A$ be a binary relation on a given set A. We say that $\sqsubseteq$ is a *partial order* if it reflexive, antisymmetric and transitive; that is, if for all $a, b, c \in A$:

(a) $a \sqsubseteq a$

(b) $a \sqsubseteq b$ and $b \sqsubseteq a$ imply $a = b$

(c) $a \sqsubseteq b$ and $b \sqsubseteq c$ imply $a \sqsubseteq c$

A *partially ordered set* (or simply a *poset*) is a set A with an associated partial order relation $\sqsubseteq$. When it is necessary to explicit this association, we shall write $\langle A; \sqsubseteq \rangle$. □

A common example of a partially ordered set is $\wp(A)$ ordered by set inclusion $\subseteq$. As another example, if A_1, A_2,..., A_n is a family of ordered sets, the cartesian product $A_1 \times A_2 \times \cdots \times A_n$ forms an ordered set under the pointwise order, defined by

$$(a_1, \ldots, a_n) \sqsubseteq (b_1, \ldots, b_n) \quad \text{if and only if} \quad a_i \sqsubseteq b_i \text{ for all } 1 \leq i \leq n.$$

Quite predictably, we shall freely write $a \sqsupseteq b$ as an alternative to $a \sqsubseteq b$ and $a \not\sqsubseteq b$ to mean that $a \sqsubseteq b$ does not hold.

Definition 2.2.2 (Induced order)
If $B \subseteq A$ and A is partially ordered, there is a natural order that B inherits from A, called the *induced order* from A, setting $a \sqsubseteq b$, for all $a, b \in B$, if and only if $a \sqsubseteq b$ in A. □

Definition 2.2.3 (Special elements)
Let A be a partially ordered set and let $B \subseteq A$. An element $a \in B$ is a *maximal* element of B if $b \sqsupseteq a$ implies $b = a$, for any $b \in B$. If $a \sqsupseteq b$, for every $b \in B$, we say that a is the *maximum* or *greatest* element of B, and write $a = \max B$. The dual notions of *minimal* and *minimum* or *smallest* elements are defined likewise, but with the order reversed.

The greatest element of A, if it exists, is called *top* and written $\top$. Likewise, the smallest element, in case it exists, is called *bottom* of written $\bot$. □

For the case of $\wp(A)$, for instance, we naturally have $\top = A$ and $\bot = \emptyset$.

We shall be considering partially ordered sets that have a top element, but not necessarily a bottom element. When a bottom element is needed, we shall generally add it artificially by 'lifting' the ordered set we started with.

Definition 2.2.4 (Lifting)
Given a partially ordered set A, the *lift* of A, written $A_\bot$ has elements taken from $A \cup \{\bot\}$, where $\bot \notin A$, and ordered as follows:

$$a \sqsubseteq b \text{ in } A_\bot \quad \text{if and only if} \quad a = \bot \text{ or } a \sqsubseteq b \text{ in } A.$$

□

Definition 2.2.5 (Lower and upper bounds)
Let A be a partially ordered set and let $B \subseteq A$. An element $a \in A$ is an *upper bound* of B if $a \sqsupseteq b$ for all $b \in B$. Dually, an element $a \in A$ is a *lower bound* of B if $a \sqsubseteq b$ for all $b \in B$. The *least upper bound* or *join* of B (if it exists), and written $\sqcup B$, is the smallest of all the upper bounds of B:

$$\sqcup B = \min \{a \in A \mid a \sqsupseteq b \text{ for all } b \in B\}.$$

Likewise, we define the *greatest lower bound* or *meet* dually:

$$\sqcap B = \max \{a \in A \mid a \sqsubseteq b \text{ for all } b \in B\}.$$

□

Notice that if $B = \emptyset$, then $\sqcup B = \bot$ if A has a bottom element; and, dually, $\sqcap B = \top$, if B has a top element. If, on the other hand, $B = A$, we have that $\sqcup B = \top$, in case B has a top element, and $\sqcap B = \bot$, in case B has a bottom element.

We shall use a special notation for the case where B has two elements. We shall write $a \sqcup b$ and $a \sqcap b$ for $\sqcup\{a, b\}$ and $\sqcap\{a, b\}$, respectively. According to the definitions of least upper bound and greatest lower bound, note that $a \sqcup b = b$ and $a \sqcap b = a$ if $a \sqsubseteq b$.

We shall mainly be interested in those structures for which $a \sqcup b$ and $a \sqcap b$ exist for all pair of elements $a, b \in A$.

Definition 2.2.6 (Semilattices and lattices)
Let A be a non-empty partially ordered set. We call A a $\sqcup$-*semilattice* ("join-semilattice"), if $a \sqcup b$ exist for all pair of elements $a, b \in A$. Dually, we call A a $\sqcap$-*semilattice* ("meet-semilattice"), if $a \sqcap b$ exist for all $a, b \in A$. The partially ordered set A is a *lattice* if it is

simultaneously a $\sqcup$-semilatice and a $\sqcap$-semilattice. It is not difficult to see that if A is a finite lattice, it has top and bottom elements.

If $\sqcup B$ and $\sqcap B$ exist for any subset $B \subseteq A$ (and not only for pairs of elements), then A is called a *complete lattice*. □

Notice that $\langle \wp(A); \subseteq \rangle$ is a complette lattice, where joins are realised by unions and meets by intersections.

It is worth noting that finite lattices are also complete lattices: If B is a (necessarily finite) subset of A, and so $B = \{b_1, b_2, \ldots, b_n\}$, then we have $\sqcup B = (\cdots(b_1 \sqcup b_2) \sqcup \cdots) \sqcup b_n$. We proceed dually for $\sqcap B$.

Definition 2.2.7 (Properties of maps)
Let A be a partially ordered set. We call a map $f : A \to A$ *monotone* (*on* A) if f preserves the underlying order; formally, for all $a, b \in A$, we must have that

$$a \sqsubseteq b \quad \text{implies} \quad f(a) \sqsubseteq f(b).$$

If $\langle a_i \rangle_n$ denotes the *ascending chain*

$$a_1 \sqsubseteq \cdots \sqsubseteq a_n,$$

then a map is said to *preserve joins of chains* if and only if

$$f(\bigsqcup_{i \geq n} a_i) = \bigsqcup_{i \geq n} f(a_i).$$

□

We shall be interested in expressing the solutions of sets of constraints as particular elements arising as solutions of fixpoint equations.

Definition 2.2.8 (Fixpoint)
Let $f : A \to A$ be a function. An element $a \in A$ is called a *fixpoint* of f, if

$$f(a) = a.$$

□

In particular, we shall be looking for the smallest fixpoint (solution), which always exists for monotone maps defined on complete lattices.

Theorem 2.2.9 (Knaster-Tarski Fixpoint Theorem)
Let A be a complete lattice and $f : A \to A$ a monotone map. Then,

$$h = \sqcap\{a \in A \mid f(a) \sqsubseteq a\}$$

is a least fixpoint of f.

Proof. Let $H = \{a \in A \mid f(a) \sqsubseteq a\}$, and so $h = \sqcap H$. We shall prove that $h = f(h)$ by showing that $f(h) \sqsubseteq h$ and $h \sqsubseteq f(h)$ respectively. Note that for all $a \in H$, we have $h \sqsubseteq a$. It follows that $f(h) \sqsubseteq f(a) \sqsubseteq a$ by monotonicity of f. Therefore, $f(h)$ is a lower bound of H, so $f(h) \sqsubseteq h$. Because f is monotone, we have that $f(f(h)) \sqsubseteq f(h)$, so $f(h) \in H$ by definition, and hence, $h \sqsubseteq f(h)$, as required. □

There is a simple iterative method to compute least fixpoints of monotone maps that also preserve joins of chains. This method is suggested in the following theorem.

Theorem 2.2.10
Let A be a complete lattice and $f : A \to A$ a monotone map preserving joins of chains. Then,

$$\mu(f) = \bigsqcup_{n \geq 0} f^n(\bot)$$

is the least fixpoint of f.

Proof. We first observe that $h = \sqcup_{n \geq 0} f^n(\bot)$ always exists and is the limit of the ascending chain

$$\bot \sqsubseteq f(\bot) \sqsubseteq \ldots \sqsubseteq f^n(\bot) \sqsubseteq f^{n+1}(\bot) \ldots .$$

It is not difficult to see that h is a fixpoint. Indeed,

$$\begin{aligned} f(\bigsqcup_{n \geq 0} f^n(\bot)) &= \bigsqcup_{i \geq 0} f(f^n(\bot)) && \text{(since } f \text{ preserves joins of chains)} \\ &= \bigsqcup_{i \geq 1} f^n(\bot) \\ &= \bigsqcup_{i \geq 0} f^n(\bot) && \text{(since } \bot \sqsubseteq f^n(\bot) \text{ for all } n\text{).} \end{aligned}$$

To prove that h is indeed the least fixpoint, let h' be any fixpoint of f. Then, by induction, $f^n(h') = h'$ for all n. By monotonicity, since $\bot \sqsubseteq h'$, we have $f^n(\bot) \sqsubseteq f^n(h') = h'$ for all n. Therefore, by construction of h,we must have that $h \sqsubseteq h'$, so h is the least fixpoint. □

Chapter 3

Linearity analysis

We begin our study of annotated type systems for the static analysis of structural properties by first presenting a simple version of *linearity analysis*. Linearity analysis is aimed at discovering when values are used exactly once, or in a *linear* fashion, as opposed to any number of times, or in an *intuitionistic* or *non-linear* fashion.

3.0.1 An intermediate linear language

The intermediate language of linearity analysis we present here arises quite naturally as a reformulation of a suitable fragment of Barber and Plotkin's DILL [5] (Dual Intuitinistic Linear Logic) in terms of an annotated type system. The annotations play the role of syntactic markers, indicating the presence or absence of the exponential '!' in types and terms. This encoding of linear types using annotations seems to have been first proposed by Wadler [65]. The fragment we study here corresponds to his type system of 'standard types', which allows the exponential to appear in the places where Girard's standard translation from intuitionistic types into linear types would allow an exponential. We refer to this fragment as the 'functional programming' fragment of linear logic, since it allows the encoding of both intuitionistic and linear functions without the need for explicit promotion and dereliction terms, using the familiar syntax of typed functional languages extended with annotations. Proof-theoretically, the terms of the functional programming fragment may be viewed as encoding intuitionistic linear logic proofs that have the same structure as the intuitionistic proofs we would have obtained if we erased the annotations from the terms. These proofs, and their suggested translations, which are known under the name of *decorations*, have been independently studied by Danos, Joinet and Schellinx [26]. We show here the view of static analysis, which is aimed at finding the optimal set of annotations for an intuitionistic term, in the sense that if an exponential can be avoided in a translation, it does not belong to the optimal set. We shall first show that this optimal, or best, set exists by considering the space of all decorations of a source language (intuitionistic) term, and proving that it forms a structure that admits a smallest decoration.

We differ from Wadler in that we use side-conditions to encode the context restrictions required by linear logic, instead of explicit constraint set in the rules. We shall encounter constraint sets when we discuss annotation quantification and annnotation inference. Our formulation is closely related to the linear fragment of Bierman's usage type system [13], especially in the fact that we consider a set of more or less abstract typing rules, whose concrete semantics is completed by specifying an external domain of annotations; in the case of

linearity analysis, this domain is a 2-point annotation lattice. The given order on annotations encodes the inclusion relationship existing between linear and intuitionistic contexts[1]. We shall exploit this relationship to the advantage of static analysis in the next chapter, when we shall discuss an extension of linearity analysis with subtyping.

3.0.2 An application to inlining

As we pointed out in the introduction, a candidate application for this sort of analysis is the compiler optimisation technique known as *inlining*. Informally, in a functional language compiler, inlining consists in replacing a reference to a definition by the definition body itself[2]. An interesting case is when a definition is known to be used exactly once, in which case it can safely be inlined without risking code inflation or the unnecessary recomputation of its body[3]. Many compilers perform some sort of occurrence analysis to attempt to discover obvious cases of single use. Linearity analysis may therefore be helpul to also uncover the less obvious cases, thus allowing for a more aggressive inlining strategy. It is important to remark that we are not suggesting that linearity should be used as the single inlining criterion. Indeed, most of the benefit of inlining comes from giving a priority to, for instance, small functions that are called from several call sites [63]. We shall use the annotations provided explicitly in our intermediate linear language to formalise a very simple inlining transformation. Our main purpose is to give support to our claim that structural analysis may be used to reason about some interesting source language transformations, so our formalisation does not cover many important aspects of inlining that should be considered in a real implementation [51].

As we shall later see, because our theory is proved sound independently of the reduction strategy chosen, it is therefore very conservative, especially for lazy evaluation strategies. Theories better suited for optimising lazy languages, for instance, have been described in [62, 68, 35].

3.0.3 Organisation

This chapter is organised as follows:

- Section 3.1 reviews intuitionistic linear logic. The aim of this section is to introduce the syntax and static semantics of DILL as the underlying foundations of linearity analysis.
- Section 3.2 introduces NLL, our simplest annotated type system of linearity analysis, and provides some examples.
- Section 3.3 informally comments on the relationship existing between NLL terms and linear decorations.
- Section 3.4 introduces a syntax-directed version of NLL. We first consider a slightly modified version of the contraction rule and introduce some new notation (whose relevance will become evident in the context of the more general framework).
- Section 3.5 studies some important type-theoretic properties and establishes the semantic correctness of the analysis.

[1] A context is simply a term with a hole, like an evaluation context.

[2] The definition can later be removed altogether if it is not used anywhere else. We shall be able to discover some trivial cases of non-usage using non-occurrence analysis, covered in Section 7.4 on page 149.

[3] In the literature, definitions that are referenced from a single call site are usually referred to as 'singletons'.

- Section 3.6 proves the existence of an optimal analysis, thus concluding our discussion on the correctness of linearity analysis.
- Section 3.7 formalises a very simple inlining optimisation as an immediate application of linearity analysis.

We shall leave the problem of how to devise an algorithm for inferring linearity properties, as well as other related pragmatic issues, to Chapter 6.

3.1 A brief review of DILL

In this section we review the type theory obtained by assigning terms to the intuitionistic fragment of Barber and Plotkin's own formulation of linear logic and known as DILL [5]. Other equivalent formulations exist, with different motivations and historical background [10, 12].

3.1.1 Syntax and typing rules

The grammar for types, ranged over by σ and τ, is shown below:

$$\begin{array}{rcll} \sigma & ::= & \mathbb{G} & \text{Ground type} \\ & \mid & \sigma \multimap \sigma & \text{Linear function space} \\ & \mid & \sigma \otimes \sigma & \text{Tensor product} \\ & \mid & !\sigma & \text{Exponential type} \end{array}$$

Intuitively, $\sigma \multimap \tau$ stands for the type of linear functions with domain σ and codomain τ; and $\sigma \otimes \tau$ is the type of linear pairs with first component of type σ and second component of type τ. The "banged" or "shrieked" type $!\sigma$ is reserved for intuitionistic values, which can be freely duplicated or erased. Intuitionistic functions have type $!\sigma \multimap \tau$, making explicit the fact that the argument of the function may be used several times, or none at all. Intuitionistic pairs have, therefore, type $!\sigma \otimes !\tau$.

The set Λ_{DILL} of preterms, again ranged over by M and N, is defined by the following grammar rules:

$$\begin{array}{rcll} M & ::= & \pi & \text{Primitive} \\ & \mid & x & \text{Variable} \\ & \mid & \lambda x{:}\sigma.M & \text{Function abstraction} \\ & \mid & MM & \text{Function application} \\ & \mid & \langle M, M \rangle & \text{Pairing} \\ & \mid & \mathsf{let}\ \langle x, x \rangle = M\ \mathsf{in}\ M & \text{Unpairing} \\ & \mid & \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M & \text{Conditional} \\ & \mid & \mathsf{fix}\ x{:}\sigma.M & \text{Fixpoint} \\ & \mid & !M & \text{Promotion} \\ & \mid & \mathsf{let}\ !x = M\ \mathsf{in}\ M & \text{Dereliction} \end{array}$$

The elementary syntactic notion of substitution $M[\rho]$ can be defined in the usual way. Note that the dereliction term $\mathsf{let}\ !x = M\ \mathsf{in}\ N$ binds x in N, much like a common let.

Unlike other formulations of linear logic, the particular characteristic of DILL is that it distinguishes between linear and intuitionistic variables explicitly by introducing separate typing contexts (hence the term 'dual'). Typing judgments have the form

$$\Gamma\,;\Delta \vdash M : \sigma,$$

$$\frac{}{\Gamma, x:\sigma\,;- \vdash x:\sigma}\ \text{Identity}_I \qquad \frac{}{\Gamma\,; x:\sigma \vdash x:\sigma}\ \text{Identity}_L$$

$$\frac{\Sigma(\pi) = \sigma}{\Gamma\,;- \vdash \pi:\sigma}\ \text{Primitive}$$

$$\frac{\Gamma\,;\Delta, x:\sigma \vdash M:\tau}{\Gamma\,;\Delta \vdash \lambda x{:}\sigma.M : \sigma \multimap \tau}\ {\multimap_\mathcal{I}} \qquad \frac{\Gamma\,;\Delta_1 \vdash M:\sigma \multimap \tau \quad \Gamma\,;\Delta_2 \vdash N:\sigma}{\Gamma\,;\Delta_1,\Delta_2 \vdash MN:\tau}\ {\multimap_\mathcal{E}}$$

$$\frac{\Gamma\,;\Delta_1 \vdash M_1:\sigma_1 \quad \Gamma\,;\Delta_2 \vdash M_2:\sigma_2}{\Gamma\,;\Delta_1,\Delta_2 \vdash \langle M_1, M_2\rangle : \sigma_1 \otimes \sigma_2}\ \otimes_\mathcal{I}$$

$$\frac{\Gamma\,;\Delta_1 \vdash M:\sigma_1 \otimes \sigma_2 \quad \Gamma\,;\Delta_2, x_1:\sigma_1, x_2:\sigma_2 \vdash N:\tau}{\Gamma\,;\Delta_1,\Delta_2 \vdash \mathsf{let}\ \langle x_1, x_2\rangle = M\ \mathsf{in}\ N:\tau}\ \otimes_\mathcal{E}$$

$$\frac{\Gamma\,;\Delta_1 \vdash M:\mathsf{bool} \quad \Gamma\,;\Delta_2 \vdash N_1:\sigma \quad \Gamma\,;\Delta_2 \vdash N_2:\sigma}{\Gamma\,;\Delta_1,\Delta_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2:\sigma}\ \text{Conditional}$$

$$\frac{\Gamma, x:\sigma\,;- \vdash M:\sigma}{\Gamma\,;- \vdash \mathsf{fix}\ x{:}\sigma.M:\sigma}\ \text{Fixpoint}$$

$$\frac{\Gamma\,;- \vdash M:\sigma}{\Gamma\,;- \vdash !M : !\sigma}\ !_\mathcal{I} \qquad \frac{\Gamma\,;\Delta_1 \vdash M : !\sigma \quad \Gamma, x:\sigma\,;\Delta_2 \vdash N:\tau}{\Gamma\,;\Delta_1,\Delta_2 \vdash \mathsf{let}\ !x = M\ \mathsf{in}\ N:\tau}\ !_\mathcal{E}$$

Figure 3.1: DILL typing rules

where Γ conventionally contains declarations for intuitionistic variables and Δ contain declarations for linear variables. We assume that variables in either context are pairwise distinct. The typing rules of DILL are summarised in Figure 3.1.

Except for the Conditional and Fixpoint rules, the other rules are standard from [5]. The Conditional rule is a special case of the rule for sums, whereas the Fixpoint rule is standard from Brauner's work [15]. As before, Σ contains the signatures for constants and primitive operators. We assume all signatures to be linear; so, for instance, $\Sigma(+) = \mathsf{int} \multimap \mathsf{int} \multimap \mathsf{int}$.

A further interesting characteristic of DILL is that the structural rules are implicit, as is clear from the way intuitionistic contexts are handled by the rules. Weakening and Contraction are therefore admissible rules:

$$\frac{\Gamma\,;\Delta \vdash M:\tau}{\Gamma, x:\sigma\,;\Delta \vdash M:\tau}\ \text{Weakening} \qquad \frac{\Gamma, x_1:\sigma, x_2:\sigma\,;\Delta \vdash M:\tau}{\Gamma, x:\sigma\,;\Delta \vdash M[x_1/x, x_2/x]:\tau}\ \text{Contraction}$$

There are two versions of the Identity rule, one for each variable sort. The linear context in the Identity_I rule is constrained to be empty, since no linear variables may be discarded. The same remark applies for the Constant rule. This restriction should not apply to the intuitionistic variables in Γ, which are allowed to be both contracted and weakened.

Functions are by default linear, so $\multimap_\mathcal{I}$ extends the linear context with the function's declared binding, which is constrained to be used exactly once in the function's body.

Pairs are typed using the rule $\otimes_{\mathcal{I}}$, which partitions the linear context into two sub-contexts to ensure that pair components do actually use distinct linear variables. A similar remark applies to the rules $\multimap_{\mathcal{E}}$, $\otimes_{\mathcal{E}}$, $!_{\mathcal{E}}$ and Conditional. Notice that in the Conditional rule, both branches of the conditional share the same linear variables. This is perfectly safe in this case, since only one of the branches will be selected for evaluation.

Intuitionistic values, of the form $!M$, are introduced with the $!_{\mathcal{I}}$ rule. Because intuitionistic variables may be freely duplicated or erased, linear variables are not allowed to occur inside such terms. Intuitionistic values can be deconstructed using the pattern-matching form $\mathsf{let}\ !x = M\ \mathsf{in}\ N$. The $!_{\mathcal{E}}$ rule is the only rule that introduces intuitionistic variable declarations, so x is allowed to be used in a non-linear fashion inside N. For this reason, the rule also ensures that M is a non-linear value by verifying that it has a matching non-linear type.

3.1.2 Reduction

A notion of β-reduction for linear terms is defined in a similar way as we did for our source language. The rewrite rules are the following:

$$\begin{aligned}
(\lambda x{:}\sigma.M)N &\to M[N/x] \\
\mathsf{let}\ \langle x_1, x_2\rangle = \langle M_1, M_2\rangle\ \mathsf{in}\ N &\to N[M_1/x_1, M_2/x_2] \\
\mathsf{let}\ !x = !M\ \mathsf{in}\ N &\to N[M/x] \\
\mathsf{if\ true\ then}\ N_1\ \mathsf{else}\ N_2 &\to N_1 \\
\mathsf{if\ false\ then}\ N_1\ \mathsf{else}\ N_2 &\to N_2 \\
\mathsf{fix}\ x{:}\sigma.M &\to M[\mathsf{fix}\ x{:}\sigma.M/x]
\end{aligned}$$

Notice that unfolding a fixpoint term results in the duplication of its body on the right-hand side. This explains why the linear context in the Fixpoint rule is constrained to be empty. Once again, we assume the existence of a number of δ-rules, that we shall here not explicitly address.

3.1.3 Substitution

Because of the split contexts, DILL admits two different sorts of substitution (cut), depending on the sort of variable that is substituted for:

$$\frac{\Gamma\,;\Delta_1, x:\sigma \vdash M:\tau \quad \Gamma\,;\Delta_2 \vdash N:\sigma}{\Gamma\,;\Delta_1,\Delta_2 \vdash M[N/x]:\tau} \qquad \frac{\Gamma, x:\sigma\,;\Delta \vdash M:\tau \quad \Gamma\,;- \vdash N:\sigma}{\Gamma\,;\Delta \vdash M[N/x]:\tau}$$

Notice that substituting a term for the free occurrences of an intuitionistic variable may result in the duplication or deletion of the substituted term; this is the reason why the intuitionistic substitution rule (right) does not allow any linear variables in its context. The linear substitution rule (left), on the other hand, need not impose any restrictions.

The restriction on the linear context imposed by the intuitionistic substitution is important, and lies at the heart of the operational correctness of the calculus. We shall study reduction later, in the context of a our annotated linear theory.

3.1.4 Girard's translation

Figure 3.2 provides a definition for the well-known Girard's translation [30], mapping FPL terms into the intuitionistic fragment of DILL.

$$
\begin{aligned}
\mathbb{G}^\bullet &= \mathbb{G} \\
(\sigma \to \tau)^\bullet &= !\sigma^\bullet \multimap \tau^\bullet \\
(\sigma_1 \times \sigma_2)^\bullet &= !\sigma_1{}^\bullet \otimes !\sigma_2{}^\bullet
\end{aligned}
$$

$$
\begin{aligned}
x^\bullet &= x \\
\pi^\bullet &= \begin{cases} \lambda x_1{:}!\mathbb{G}_1 \ldots \lambda x_n{:}!\mathbb{G}_n. \\ \quad (\mathsf{let}\ !y_1 = x_1\ \mathsf{and}\ \cdots\ \mathsf{and}\ !y_n = x_n\ \mathsf{in}\ \pi\ y_1 \cdots\ y_n), \\ \qquad \text{if } \Sigma(\pi) = \mathbb{G}_1 \multimap \cdots \multimap \mathbb{G}_n \multimap \mathbb{G}; \\ \pi, \text{ otherwise} \end{cases} \\
(\lambda x{:}\sigma.M)^\bullet &= \lambda x{:}!\sigma^\bullet.\mathsf{let}\ !y = x\ \mathsf{in}\ M[y/x]^\bullet \\
(MN)^\bullet &= M^\bullet\, !N^\bullet \\
\langle M_1, M_2\rangle^\bullet &= \langle !M_1{}^\bullet, !M_2{}^\bullet\rangle \\
(\mathsf{let}\ \langle x_1, x_2\rangle = M\ \mathsf{in}\ N)^\bullet &= \mathsf{let}\ \langle x_1, x_2\rangle = M^\bullet\ \mathsf{in} \\
&\qquad (\mathsf{let}\ !y_1 = x_1\ \mathsf{and}\ !y_2 = x_2\ \mathsf{in} \\
&\qquad\ N[y_1/x_1, y_2/x_2]^\bullet) \\
(\mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2)^\bullet &= \mathsf{if}\ M^\bullet\ \mathsf{then}\ N_1{}^\bullet\ \mathsf{else}\ N_2{}^\bullet \\
(\mathsf{fix}\ x{:}\sigma.M)^\bullet &= \mathsf{fix}\ x{:}\sigma^\bullet.M^\bullet
\end{aligned}
$$

where y,y_1 and y_2 are fresh variables.

Figure 3.2: Girard's translation

For contexts, let $\Gamma^\bullet \equiv (x_1 : \sigma_1, \ldots, x_n : \sigma_n)^\bullet = x_1 : \sigma_1{}^\bullet, \ldots, x_n : \sigma_n{}^\bullet$.

(The notation $\mathsf{let}\ !x_1 = M_1\ \mathsf{and}\ \cdots\ \mathsf{and}\ !x_n = M_n\ \mathsf{in}\ N$ is used, as expected, as an abbreviation for a series of nested derelictions.)

The reader may have noticed that the translation of primitives $\pi^\bullet$ requires the construction of a 'wrapper' function as a result of the fact that we have assumed type signatures to be linear.

Proposition 3.1.1 (Soundness)
$\Gamma \underset{\text{FPL}}{\vdash} M : \sigma$ *implies* $\Gamma^\bullet ; - \underset{\text{DILL}}{\vdash} M^\bullet : \sigma^\bullet$.

Proof. By induction on the derivation of $\Gamma \underset{\text{FPL}}{\vdash} M : \sigma$. □

3.2 The type system NLL

We are now ready to present the syntax and typing rules of NLL, our intermediate linear language. NLL is what we call the 'functional programming' subset of DILL, the minimal setting to discuss translations from our source language into our intermediate linear language. In this minimal fragment, both the linear and intuitionistic logical connectives appear as primitive.

(The intuitionistic implication, for instance, is not definable in terms of the exponential and linear implication.)

3.2.1 Annotation set

Annotated type systems are usually formulated in terms of an ordered annotation set. For the case of linearity analysis, we define the 2-point partially ordered set of annotations

$$\mathbb{A} \equiv \langle \{1, \top\}, \sqsubseteq \rangle.$$

The elements 1 and $\top$, collectively ranged over by a, b and c, are called *annotation constants* or *values*. They are intended as notation for the following structural properties:

1	Linear
$\top$	Intuitionistic

For convenience, we shall write $\mathbb{A}$ for both the annotation poset and the underlying annotation set. When confusion may arise, we shall also qualify our notation with the conventional name of the typed theory as a subscript (as in $\sqsubseteq_{\text{NLL}}$, for instance).

Informally, the order relation $\sqsubseteq$ explicitly encodes the fact that linear resources are special sorts of intuitionistic resources. Hence, we adopt the order relation given by

$$a \sqsubseteq a \qquad 1 \sqsubseteq \top$$

From a static analysis viewpoint, the order relation can be interpreted as specifying that linear annotations should be preferred over intuitionistic annotations in terms of their information content. (The order relation will play a valuable role in the definition of the 'best' analysis.)

3.2.2 Annotated types

The set of *annotated types*, ranged over by σ and τ, is generated by the following grammar rules:

σ	$::=$	$\mathbb{G}$	Ground type
	$\mid$	$\sigma^a \multimap \sigma$	Function space
	$\mid$	$\sigma^a \otimes \sigma^a$	Product

where $\mathbb{G}$ is, as before, one of the ground types int or bool.

An annotated type σ provides an alternative notation for a particular DILL type, whose meaning $[\![\sigma]\!]$ is given by the following equations.

$$[\![\mathbb{G}]\!] = \mathbb{G} \tag{3.1}$$
$$[\![\sigma^a \multimap \tau]\!] = ([\![\sigma]\!])^{\mathrm{p}}_a \multimap [\![\tau]\!] \tag{3.2}$$
$$[\![\sigma^a \otimes \tau^b]\!] = ([\![\sigma]\!])^{\mathrm{p}}_a \otimes ([\![\tau]\!])^{\mathrm{p}}_b \tag{3.3}$$

where

$$(\sigma)^{\mathrm{p}}_1 = \sigma \tag{3.4}$$
$$(\sigma)^{\mathrm{p}}_\top = !\sigma \tag{3.5}$$

So, in syntactic terms, an annotation marks the existence or absence of an exponential.

If σ is an NLL type, we shall write σ° for its *underlying type*, obtained by erasing all annotations:

$$\mathbb{G}^\circ = \mathbb{G} \tag{3.6}$$
$$(\sigma^a \multimap \tau)^\circ = \sigma^\circ \to \tau^\circ \tag{3.7}$$
$$(\sigma^a \otimes \tau^b)^\circ = \sigma^\circ \times \tau^\circ \tag{3.8}$$

3.2.3 Annotated preterms

The set Λ^{NLL} of *annotated preterms*, ranged over by M and N, is generated by the following grammar rules:

$$\begin{array}{rcll} M & ::= & \pi & \text{Primitive} \\ & \mid & x & \text{Variable} \\ & \mid & \lambda x{:}\sigma^a.M & \text{Function abstraction} \\ & \mid & M M & \text{Function application} \\ & \mid & \langle M, M\rangle^{a,a} & \text{Pairing} \\ & \mid & \mathsf{let}\ \langle x, x\rangle = M\ \mathsf{in}\ M & \text{Unpairing} \\ & \mid & \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M & \text{Conditional} \\ & \mid & \mathsf{fix}\ x{:}\sigma.M & \text{Fixpoint} \end{array}$$

The syntax of preterms is almost identical to that of FPL, except for the annotations on λ-bound variables and pairs.

As for types, if M is a NLL preterm, we shall write M° for the *underlying preterm*, obtained by erasing all the annotations. In particular, we have $(\lambda x{:}\sigma^a.M)^\circ = \lambda x{:}\sigma^\circ.M^\circ$ and $(\langle M, N\rangle^{a,a})^\circ = \langle M^\circ, N^\circ\rangle$.

3.2.4 Typing contexts

As usual, typing assertions in linearity analysis are contextual, and take the form of *annotated typing judgments*

$$\Gamma \vdash M : \sigma,$$

where Γ ranges over *annotated typing contexts* of the form

$$\Gamma ::= x_1 : {\sigma_1}^{a_1}, \ldots, x_n : {\sigma_n}^{a_n}$$

As usual, we consider only well-formed contexts.

If $\Gamma \equiv \Gamma', x : \sigma^a$, then $\Gamma(x)$ stands for the pair of base type and annotation associated to x in Γ, written σ^a. We write $|\Gamma(x)| = a$ to obtain the annotation component of the pair, and $\Gamma(x)^\circ$ for the base type component.

Let Γ° stand for the *underlying typing context*, obtained by dropping all annotations:

$$(-)^\circ = - \quad \text{and} \quad (\Gamma, x : \sigma^a)^\circ = \Gamma^\circ, x : \sigma^\circ \tag{3.9}$$

Annotated typing contexts are just a syntactic alternative to DILL contexts in which annotations provide the information necessary to discriminate between linear and non-linear

$$\frac{\Gamma \vdash M : \tau}{\Gamma, x : \sigma^{\top} \vdash M : \tau}\,\text{Weakening} \qquad \frac{\Gamma, x_1 : \sigma^{\top}, x_2 : \sigma^{\top} \vdash M : \tau}{\Gamma, x : \sigma^{\top} \vdash M[x/x_1, x/x_2] : \tau}\,\text{Contraction}$$

Figure 3.3: NLL structural rules

variables. To be more precise, the semantics of an NLL context is the DILL context $[\![\Gamma]\!]$ defined by the equations below.

$$[\![-]\!] = - \,;\, - \tag{3.10}$$

$$[\![\Gamma, x : \sigma^a]\!] = \begin{cases} \Gamma' \,;\, \Delta', x : [\![\sigma]\!] & \text{if } a \equiv 1 \\ \Gamma', x : [\![\sigma]\!] \,;\, \Delta' & \text{if } a \equiv \top \end{cases} \quad \text{where } [\![\Gamma]\!] = \Gamma' \,;\, \Delta' \tag{3.11}$$

3.2.5 Typing rules

Because we only have a single context for both linear and intuitionistic assumptions (labelled 1 and $\top$, respectively) we need to reintroduce the structural rules as shown in Figure 3.3. Notice that, as expected, the structural rules only apply to intuitionistic assumptions. The remaining typing rules for the core language constructs are given in Figure 3.4.

In the $\multimap_{\mathcal{E}}$, $\otimes_{\mathcal{I}}$ and Fixpoint rules, the side-condition $|\Gamma| \sqsupseteq a$ is an abbreviation for the predicate

$$|\Gamma| \sqsupseteq a \stackrel{\text{def}}{=} |\Gamma(x)| \sqsupseteq a, \text{ for all } x \in dom(\Gamma). \tag{3.12}$$

There is a single rule to introduce both the linear and intuitionistic function types; and a single rule to eliminate them. These connectives would have required separate rules if we had used split contexts instead of annotations. Actually, we would have needed two rules for $\sigma \multimap \tau$, and two for $!\sigma \multimap \tau$. For pairs, eight rules would have been necessary, two for each case of tensor product, $\sigma \otimes \tau$, $\sigma \otimes !\tau$, $!\sigma \otimes \tau$ and $!\sigma \otimes !\tau$.

In fact, each rule defined on arbitrary annotation values may be understood as giving rise to a family of DILL-like rules, where each rule in the family corresponds to a given assignment of annotation values. The resulting system is what we have earlier referred to as the ‘functional programming’ fragment of DILL. For completeness, we have summarised the implicational subset of this fragment in Figure 3.5.

Notice that we do not distinguish between linear and intuitionistic function applications, and still, terms correctly encode proofs: The type of the function gives the information necessary to know which version of the application rule should apply at each point.

The only difference between the $\multimap_{\mathcal{E}L}$ and $\multimap_{\mathcal{E}I}$ rules for typing an application term MN is the restriction establishing that the argument N to an intuitionistic function should not contain any linear variables. The justification for this restriction becomes clear once we consider how a typical intuitionistic function application looks like in DILL:

$$\frac{\Gamma_1 \,;\, \Delta \vdash M : !\sigma \multimap \tau \qquad \dfrac{\Gamma_2 \,;\, - \vdash N : \sigma}{\Gamma_2 \,;\, - \vdash !N : !\sigma}\,!_{\mathcal{I}}}{\Gamma_1, \Gamma_2 \,;\, \Delta \vdash M\,!N : \tau}\,\multimap_{\mathcal{E}}$$

$$\frac{}{x : \sigma^a \vdash x : \sigma}\ \text{Identity} \qquad \frac{\Sigma(\pi) = \sigma}{\vdash \pi : \sigma}\ \text{Primitive}$$

$$\frac{\Gamma, x : \sigma^a \vdash M : \tau}{\Gamma \vdash \lambda x{:}\sigma^a.M : \sigma^a \multimap \tau}\ \multimap_{\mathcal{I}}$$

$$\frac{\Gamma_1 \vdash M : \sigma^a \multimap \tau \quad \Gamma_2 \vdash N : \sigma \quad |\Gamma_2| \sqsupseteq a}{\Gamma_1, \Gamma_2 \vdash MN : \tau}\ \multimap_{\mathcal{E}}$$

$$\frac{\Gamma_1 \vdash M_1 : \sigma_1 \quad \Gamma_2 \vdash M_2 : \sigma_2 \quad |\Gamma_1| \sqsupseteq a_1 \quad |\Gamma_2| \sqsupseteq a_2}{\Gamma_1, \Gamma_2 \vdash \langle M_1, M_2 \rangle^{a_1, a_2} : {\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2}}\ \otimes_{\mathcal{I}}$$

$$\frac{\Gamma_1 \vdash M : {\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2} \quad \Gamma_2, x_1 : {\sigma_1}^{a_1}, x_2 : {\sigma_2}^{a_2} \vdash N : \tau}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ \langle x_1, x_2 \rangle = M\ \mathsf{in}\ N : \tau}\ \otimes_{\mathcal{E}}$$

$$\frac{\Gamma_1 \vdash M : \mathsf{bool} \quad \Gamma_2 \vdash N_1 : \sigma \quad \Gamma_2 \vdash N_2 : \sigma}{\Gamma_1, \Gamma_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \sigma}\ \text{Conditional}$$

$$\frac{\Gamma, x : \sigma^\top \vdash M : \sigma \quad |\Gamma| \sqsupseteq \top}{\Gamma \vdash \mathsf{fix}\ x{:}\sigma.M : \sigma}\ \text{Fixpoint}$$

Figure 3.4: NLL typing rules

Types σ $::= \mathbb{G} \mid \sigma \multimap \sigma \mid\ !\sigma \multimap \sigma$

Terms M $::= x \mid \lambda x{:}\sigma.M \mid MM$

$$\frac{}{\Gamma\,;x : \sigma \vdash x : \sigma}\ \text{Identity}_L \qquad \frac{}{\Gamma, x : \sigma\,; - \vdash x : \sigma}\ \text{Identity}_I$$

$$\frac{\Gamma\,; \Delta, x : \sigma \vdash M : \tau}{\Gamma\,; \Delta \vdash \lambda x{:}\sigma.M : \sigma \multimap \tau}\ \multimap_{\mathcal{I}L} \qquad \frac{\Gamma\,; \Delta_1 \vdash M : \sigma \multimap \tau \quad \Gamma\,; \Delta_2 \vdash N : \sigma}{\Gamma\,; \Delta_1, \Delta_2 \vdash MN : \tau}\ \multimap_{\mathcal{E}L}$$

$$\frac{\Gamma, x : \sigma\,; \Delta \vdash M : \tau}{\Gamma\,; \Delta \vdash \lambda x{:}!\sigma.M : !\sigma \multimap \tau}\ \multimap_{\mathcal{I}I} \qquad \frac{\Gamma\,; \Delta \vdash M : !\sigma \multimap \tau \quad \Gamma\,; - \vdash N : \sigma}{\Gamma\,; \Delta \vdash MN : \tau}\ \multimap_{\mathcal{E}I}$$

Figure 3.5: The ‘functional programming’ fragment of DILL

In the case of NLL, this context restriction is formulated a bit differently in terms of the underlying order relation on annotations. The elimination rule for $\sigma^\top \multimap \tau$,

$$\frac{\Gamma_1 \vdash M : \sigma^\top \multimap \tau \quad \Gamma_2 \vdash N : \sigma \quad |\Gamma_2| \sqsupseteq \top}{\Gamma_1, \Gamma_2 \vdash MN : \tau}$$

requires that all annotations in Γ_2 be precisely $\top$, thus forbiding any 1-annotated variables. (We naturally have the same restriction for the Fixpoint rule, as expected.) If we consider the elimination rule for $\sigma^1 \multimap \tau$, the side-condition $|\Gamma_2| \sqsupseteq 1$ translates into no restriction at all, so we retrieve the standard application rule for linear functions.

3.2.6 A remark on primitive operators

We have remained silent regarding the nature of the type $\Sigma(\pi)$ of a primitive operator π. Assuming linear signatures is not entirely satisfying for our intermediate linear language, since we may sometimes need to use an operator in an intuitionistic context. Using wrapper functions to coerce the types of primitive operators, as we have done in Figure 3.2, is a solution that works; but for the moment, it seems best to assume that, for each operator of the source language, there is a host of related operators in the intermediate language differing only on their annotations. We therefore have, for example, $\Sigma(+^{a,b}) = \mathsf{int}^a \multimap \mathsf{int}^b \multimap \mathsf{int}$, for all combinations of a and b. As we would like terms to have unique types, the explicit annotation of operators is necessary, but we shall not be very formal about this; in particular, we omit any operator annotations in the examples. The reason is that a more satisfactory solution will come in the form of annotation subtyping.

3.2.7 Examples

For clarity, we may sometimes use in future examples the following let construct, that should be parsed in the standard way as a function application:

$$\mathsf{let}\ x{:}\sigma^a = M\ \mathsf{in}\ N \stackrel{\text{def}}{=} (\lambda x{:}\sigma^a.N)\ M.$$

As a first illustrative example, we show in Figure 3.6 a (generic) type derivation for the function

$$\mathsf{twice}_{a,b} \stackrel{\text{def}}{=} \lambda f{:}(\sigma^a \multimap \sigma)^\top.\lambda x{:}\sigma^{a \sqcup b}.f(fx)\ :\ (\sigma^a \multimap \sigma)^\top \multimap \sigma^{a \sqcup b} \multimap \sigma,$$

where a and b may take arbitrary annotation values. (The reader may like to verify that any other choice of annotations would violate the conditions imposed by the typing rules.)

$$\dfrac{\dfrac{}{f_1 : (\sigma^a \multimap \sigma)^\top \vdash f_1 : \sigma^a \multimap \sigma}\text{Identity} \quad \dfrac{\dfrac{}{f_2 : (\sigma^a \multimap \sigma)^\top \vdash f_2 : \sigma^a \multimap \sigma}\text{Identity} \quad \dfrac{}{x : \sigma^{a \sqcup b} \vdash x : \sigma}\text{Identity}}{f_2 : (\sigma^a \multimap \sigma)^\top, x : \sigma^{a \sqcup b} \vdash f_2\ x : \sigma}\multimap_{\mathcal{E}}}{f_1 : (\sigma^a \multimap \sigma)^\top, f_2 : (\sigma^a \multimap \sigma)^\top, x : \sigma^{a \sqcup b} \vdash f_1\ (f_2\ x) : \sigma}\multimap_{\mathcal{E}}$$

$$\dfrac{\dfrac{\dfrac{f_1 : (\sigma^a \multimap \sigma)^\top, f_2 : (\sigma^a \multimap \sigma)^\top, x : \sigma^{a \sqcup b} \vdash f_1\ (f_2\ x) : \sigma}{f : (\sigma^a \multimap \sigma)^\top, x : \sigma^{a \sqcup b} \vdash f\ (f\ x) : \sigma}\text{Contraction}}{f : (\sigma^a \multimap \sigma)^\top \vdash \lambda x{:}\sigma^{a \sqcup b}.f\ (f\ x) : \sigma^{a \sqcup b} \multimap \sigma}\multimap_{\mathcal{I}}}{- \vdash \lambda f{:}(\sigma^a \multimap \sigma)^\top.\lambda x{:}\sigma^{a \sqcup b}.f(fx) : (\sigma^a \multimap \sigma)^\top \multimap \sigma^{a \sqcup b} \multimap \sigma}\multimap_{\mathcal{I}}$$

Figure 3.6: Example NLL type derivation

id	$\equiv$	$\lambda x{:}\sigma^a.x$	:	$\sigma^a \multimap \sigma$
inc	$\equiv$	$\lambda x{:}\mathsf{int}^a.x+1$	:	$\mathsf{int}^a \multimap \mathsf{int}$
dup	$\equiv$	$\lambda x{:}\mathsf{int}^\top.x+x$	:	$\mathsf{int}^\top \multimap \mathsf{int}$
pair	$\equiv$	$\lambda x_1{:}{\sigma_1}^{a_1\sqcup b_1}.\lambda x_2{:}{\sigma_2}^{a_2\sqcup b_2}.\langle x_1, x_2\rangle^{b_1,b_2}$	:	${\sigma_1}^{a_1\sqcup b_1} \multimap {\sigma_2}^{a_2\sqcup b_2} \multimap {\sigma_1}^{b_1} \otimes {\sigma_2}^{b_2}$
fst	$\equiv$	$\lambda x{:}({\sigma_1}^a \otimes {\sigma_2}^\top)^b.\mathsf{let}\ \langle y_1, y_2\rangle = x\ \mathsf{in}\ y_1$	:	$({\sigma_1}^a \otimes {\sigma_2}^\top)^b \multimap \sigma_1$
snd	$\equiv$	$\lambda x{:}({\sigma_1}^\top \otimes {\sigma_2}^a)^b.\mathsf{let}\ \langle y_1, y_2\rangle = x\ \mathsf{in}\ y_2$	:	$({\sigma_1}^\top \otimes {\sigma_2}^a)^b \multimap \sigma_2$
apply	$\equiv$	$\lambda f{:}(\sigma^a \multimap \tau)^b.\lambda x{:}\sigma^{c\sqcup a}.f\ x$	:	$(\sigma^a \multimap \tau)^b \multimap \sigma^{c\sqcup a} \multimap \tau$

Figure 3.7: Typing examples of some familiar terms

Notice that the bound variable f is annotated with $\top$ because it is used twice in the body of the function. Also, as required by the $\multimap_{\mathcal{E}}$ rule, any annotation chosen for x, say b, must be greater than a, the annotation of the argument to f. The choice of $a \sqcup b$ is explained by the fact that the inequation $b \sqsupseteq a$ can be substituted by the equation $b = a \sqcup b$. The interest in using the join operator in the examples is that we are not obliged to place any extra side-conditions. In the example, any choice of a and b would result in a valid type derivation[4].

Figure 3.7 provide example typings for some familiar terms.

3.2.8 Reduction

We may conclude the presentation of our linear intermediate language by considering the β-reduction relation induced by FPL, which can be directly defined as the contextual closure of the relation generated by the following axioms:

$$
\begin{aligned}
(\lambda x{:}\sigma^a.M)N &\rightarrow M[N/x] \\
\mathsf{let}\ \langle x_1, x_2\rangle = \langle M_1, M_2\rangle^{a_1,a_2}\ \mathsf{in}\ N &\rightarrow N[M_1/x_1, M_2/x_2] \\
\mathsf{if\ true\ then}\ N_1\ \mathsf{else}\ N_2 &\rightarrow N_1 \\
\mathsf{if\ false\ then}\ N_1\ \mathsf{else}\ N_2 &\rightarrow N_2 \\
\mathsf{fix}\ x{:}\sigma.M &\rightarrow M[\mathsf{fix}\ x{:}\sigma.M/x]
\end{aligned}
$$

The following is a straightforward consequence of the above definition.

Proposition 3.2.1
For any two preterms M and N, $M \rightarrow N$ implies $M^\circ \rightarrow N^\circ$. □

We shall later prove a subject reduction result stating that reducing a well-typed term does always result in another well-typed term.

3.3 Decorations

The most important syntactic characteristic of our own presentation of this fragment of intuitionistic linear logic is perhaps that NLL typings can effectively be regarded as FPL

[4] In Appendix A we provide an alternative formulation of NLL that exploits this idea.

typings ‘decorated’ with extra structural information. This is also true of type derivations in general.

Let the erasure of an NLL typing judgment $\Gamma \vdash M : \sigma$ be defined by

$$(\Gamma \vdash M : \sigma)^\circ = \Gamma^\circ \vdash M^\circ : \sigma^\circ. \tag{3.13}$$

Then, for each NLL typing rule

$$J_1, \ldots, J_n \vdash J$$

there is a counterpart FPL typing rule

$$J_1{}^\circ, \ldots, J_n{}^\circ \vdash J^\circ,$$

obtained by erasing the annotations everywhere.

The following proposition is a straightforward corollary of this observation.

Proposition 3.3.1
If $\Gamma \underset{\text{NLL}}{\vdash} M : \sigma$*, then* $\Gamma^\circ \underset{\text{FPL}}{\vdash} M^\circ : \sigma^\circ$. □

A word on notation and terminology. We shall sometimes write J^* to emphasize that J^* is the annotated version of a FPL typing judgment that has been introduced in the context of the discussion, and which is syntactically equivalent to $(J^*)^\circ \equiv J$. Instead of ‘annotated’, we shall feel free to use the words ‘decorated’ or ‘enriched’. The same conventions apply to other syntactic categories, including terms, types and contexts.

We may sometimes use the term *decoration*, borrowed from the work of Danos, Joinet and Shellinx [26], to refer to decorated type derivations. To be more precise, what they call ‘decoration’ is a translation mapping intuitionistic proofs into intuitionistic linear logic proofs with the provision that the translation should preserve the overall structure of the proof. If $\Pi(J)$ is a type derivation of a source language typing judgment J, we may think of a possible decorated typing judgment $\Pi(J^*)$ as determining a translation

$$\theta^* : \Pi(J) \to \Pi(J^*).$$

By our previous observation, it is clear that $\Pi(J^*)^\circ \equiv \Pi(J)$, so θ^* has the obvious property of preserving the structure of the type derivation.

If terms correspond to proofs under the looking glass of the Curry-Howard correspondence, then NLL is nothing else but the language of linear logic decorations[5].

3.3.1 The problem of static analysis

We are now in a position to give a more precise idea of what we mean by “static analysis of linearity properties”.

> *Linearity analysis consists in finding an optimal decorated typing judgment* J^{opt}*, for an input typing judgment* J *of the source language, such that* $(J^{\text{opt}})^\circ \equiv J$.

[5]Note, however, that since there is no explicit syntax for the structural rules, an NLL term actually stands for a whole class of decorations, that are equivalent modulo certain commuting conversions.

By optimal decorated typing judgment we mean a decorated typing judgment J^* that is the conclusion of an *optimal decoration* $\Pi(J^*)^{\mathrm{opt}}$, in the sense of [26]. Informally, an optimal decoration is a decoration where each occurrence of a $\top$-annotated assumption is unavoidable, as there is an instance of a structural rule somewhere in the derivation that either duplicates or deletes the assumption; or the assumption appears in an intuitionistic context (i.e., a context restricted to have only $\top$-annotated assumptions.)

For our illustrative example, the optimal decoration for $\mathsf{twice}^*_{a,b}$ is clearly

$$\mathsf{twice}^*_{1,1} \equiv \lambda f{:}(\sigma^1 \multimap \sigma)^\top.\lambda x{:}\sigma^1.f(fx) \;:\; (\sigma^1 \multimap \sigma)^\top \multimap \sigma^1 \multimap \sigma.$$

The annotation for f is an example of an unavoidable annotation.

In Section 3.6 we shall provide a formal definition of optimality; and later, in Section 6.1, we shall look at a simple 'type reconstruction' algorithm for finding the optimal decorated typing judgment J^{opt}.

3.4 Towards syntax-directedness

Some important results, like semantic correctness, are very cumbersome to prove for a type system that is not syntax-directed. For this reason, we shall consider an alternative version of NLL without explicit structural rules.

3.4.1 Contraction revisited

In order for our formulation to be as general as possible, we shall first consider a slightly modified version of the contraction rule, the nature of which will become clear in the context of the more general framework discussed in Chapter 7. The syntax-directed version we give next uses the following rule, instead of the one given in Figure 3.3.

$$\frac{\Gamma, x_1 : \sigma^{a_1}, x_2 : \sigma^{a_2} \vdash M : \tau}{\Gamma, x : \sigma^{a_1 + a_2} \vdash M[x/x_1, x/x_2] : \tau}\;\text{Contraction}^+ \tag{3.14}$$

The new rule is defined in terms of a binary operator $+ : \mathbb{A} \times \mathbb{A} \to \mathbb{A}$, called *contraction operator*, and defined by

$$a_1 + a_2 = \top \quad \text{for all } a_1, a_2 \in \mathbb{A}. \tag{3.15}$$

We should first note that the four distinct instance contraction rules obtained by assigning values to a_1 and a_2 are all admissible in NLL. To see this, let us first note that the following rule, which is nothing more than our version of the Transfer rule of DILL, is also admissible[6]:

$$\frac{\Gamma, x : \sigma^1 \vdash M : \tau}{\Gamma, x : \sigma^\top \vdash M : \tau}\;\text{Transfer}$$

The case setting $a_1 = a_2 = \top$ corresponds to our previous contraction rule. The other cases follow directly from the admissibility of the Transfer rule. For instance, the case where

[6] In static analysis, this property is known as *subeffecting*, since it allows annotations to be replaced by less precise ones. Actually, linearity analysis in an instance of what is known as a *subeffecting analysis*, since the subeffecting rule need not be explicitly introduced in the type sytem to ensure conservativity.

$a_1 = a_2 = 1$ is derivable as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma, x_1 : \sigma^1, x_2 : \sigma^1 \vdash M : \tau}{\Gamma, x_1 : \sigma^\top, x_2 : \sigma^1 \vdash M : \tau}\text{Transfer}}{\Gamma, x_1 : \sigma^\top, x_2 : \sigma^\top \vdash M : \tau}\text{Transfer}}{\Gamma, x : \sigma^\top \vdash M[x/x_1, x/x_2] : \tau}\text{Contraction}$$

Because the new contraction rule does not modify the set of derivable typing judgments, it does not add any expressive power to the static analysis either, although it does allow for more 'informative' type derivations. As an illustrative example, we show below two enriched type derivations for the FPL term $\lambda x{:}\sigma.\langle x, x\rangle : \sigma \to \sigma \times \sigma$. With the less general contraction rule, we obtain the type derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x_1 : \sigma^\top \vdash x_1 : \sigma}\text{Identity} \quad \dfrac{}{x_2 : \sigma^\top \vdash x_2 : \sigma}\text{Identity}}{x_1 : \sigma^\top, x_2 : \sigma^\top \vdash \langle x_1, x_2\rangle^{1,1} : \sigma^1 \otimes \sigma^1}\otimes_\mathcal{I}}{x : \sigma^\top \vdash \langle x, x\rangle^{1,1} : \sigma^1 \otimes \sigma^1}\text{Contraction}}{- \vdash \lambda x{:}\sigma^\top.\langle x, x\rangle^{1,1} : \sigma^\top \multimap \sigma^1 \otimes \sigma^1}\multimap_\mathcal{I}$$

With the new contraction rule we obtain a more informative type derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x_1 : \sigma^1 \vdash x_1 : \sigma}\text{Identity} \quad \dfrac{}{x_2 : \sigma^1 \vdash x_2 : \sigma}\text{Identity}}{x_1 : \sigma^1, x_2 : \sigma^1 \vdash \langle x_1, x_2\rangle^{1,1} : \sigma^1 \otimes \sigma^1}\otimes_\mathcal{I}}{x : \sigma^\top \vdash \langle x, x\rangle^{1,1} : \sigma^1 \otimes \sigma^1}\text{Contraction}^+}{- \vdash \lambda x{:}\sigma^\top.\langle x, x\rangle^{1,1} : \sigma^\top \multimap \sigma^1 \otimes \sigma^1}\multimap_\mathcal{I}$$

Notice that we could use the linear instances of the Identity rule in the last derivation. In both cases, the analyses obtained (conclusions) are the same, which is clearly the only important consideration to have in mind.

3.4.2 A syntax-directed version of NLL

We are now ready to provide a syntax-oriented version of the typing rules. Figure 3.8 gives a summary of the rules that have to be modified to obtain this new version. We call this system $\mathrm{NLL}^{\uplus}$.

Weakening is implicit in the new Identity and Constant rules. Likewise, Contraction is implicit in the rules with two premises in the operator $\uplus$ for *merging* two contexts, discussed below. The rules with only a single premise remain unchanged.

Definition 3.4.1 (Context merge)
If Γ_1 and Γ_2 are two contexts, then $\Gamma_1 \uplus \Gamma_2$ is defined as the map

$$(\Gamma_1 \uplus \Gamma_2)(x) = \begin{cases} \Gamma_1(x), & \text{if } x \in \mathit{dom}(\Gamma_1), \text{ but } x \notin \mathit{dom}(\Gamma_2) \\ \Gamma_2(x), & \text{if } x \in \mathit{dom}(\Gamma_2), \text{ but } x \notin \mathit{dom}(\Gamma_1) \\ \sigma^{a_1+a_2}, & \text{if } \Gamma_1(x) = \sigma^{a_1} \text{ and } \Gamma_1(x) = \sigma^{a_2} \end{cases}$$

for all $x \in \mathit{dom}(\Gamma_1) \cup \mathit{dom}(\Gamma_2)$. □

$$\frac{|\Gamma| \sqsupseteq \top}{\Gamma, x : \sigma^a \vdash x : \sigma}\text{Identity} \qquad \frac{|\Gamma| \sqsupseteq \top \quad \Sigma(\pi) = \sigma}{\Gamma \vdash \pi : \sigma}\text{Primitive}$$

$$\frac{\Gamma_1 \vdash M : \sigma^a \multimap \tau \quad \Gamma_2 \vdash N : \sigma \quad |\Gamma_2| \sqsupseteq a}{\Gamma_1 \uplus \Gamma_2 \vdash MN : \tau}\multimap_{\mathcal{E}}$$

$$\frac{\Gamma_1 \vdash M_1 : \sigma_1 \quad \Gamma_2 \vdash M_2 : \sigma_2 \quad |\Gamma_1| \sqsupseteq a_1 \quad |\Gamma_2| \sqsupseteq a_2}{\Gamma_1 \uplus \Gamma_2 \vdash \langle M_1, M_2\rangle^{a_1,a_2} : {\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2}}\otimes_{\mathcal{I}}$$

$$\frac{\Gamma_1 \vdash M : {\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2} \quad \Gamma_2, x_1 : {\sigma_1}^{a_1}, x_2 : {\sigma_2}^{a_2} \vdash N : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{let}\ \langle x_1, x_2\rangle = M\ \mathsf{in}\ N : \tau}\otimes_{\mathcal{E}}$$

$$\frac{\Gamma_1 \vdash M : \mathsf{bool} \quad \Gamma_2 \vdash N_1 : \sigma \quad \Gamma_2 \vdash N_2 : \sigma}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \sigma}\text{Conditional}$$

Figure 3.8: Modified syntax-directed typing rules for $\text{NLL}^{\uplus}$

Intuitively, the merge operator behaves like context union, except that duplicate variables have their annotations combined using the contraction operator. In particular, we have that $\Gamma_1 \uplus \Gamma_2$ behaves like Γ_1, Γ_2 whenever Γ_1 and Γ_2 are disjoint.

The merge is clearly undefined if both contexts map the same variable to different base types. Therefore, the rules that have a combined context $\Gamma_1 \uplus \Gamma_2$ in the conclusion (i.e., with more than two premises), are assumed to implicitly verify that $\Gamma_1(x) = \Gamma_2(x)$ for all $x \in \mathit{dom}(\Gamma_1) \cap \mathit{dom}(\Gamma_2)$.

It is useful to define an order relation on contexts differing only on their respective annotations, as follows:

$$\Gamma_1 \sqsubseteq \Gamma_2 \stackrel{\text{def}}{=} {\Gamma_1}^{\circ} \subseteq {\Gamma_2}^{\circ} \text{ and } |\Gamma_1(x)| \sqsubseteq |\Gamma_2(x)| \text{ for all } x \in \mathit{dom}(\Gamma_1).$$

Proposition 3.4.2 (Properties of $\uplus$)
The merge operator satisfies the properties enumerated below, for suitable contexts Γ_1, Γ_2 and Γ_3.

a. $\Gamma_1 \uplus \Gamma_2 = \Gamma_2 \uplus \Gamma_1$

b. $(\Gamma_1 \uplus \Gamma_2) \uplus \Gamma_3 = \Gamma_1 \uplus (\Gamma_2 \uplus \Gamma_3)$

c. $\Gamma_1 \sqsubseteq \Gamma_1 \uplus \Gamma_2$ *and* $\Gamma_2 \sqsubseteq \Gamma_1 \uplus \Gamma_2$

□

The commutativity and associativity of $\uplus$ (Properties 3.4.2a and 3.4.2b) are direct consequences of the commutativity and associativity of $+$. Property 3.4.2c clearly points at the fact that the merge of two contexts results in a context that is less precise in terms of static information.

In order to convince the reader that NLL and $\text{NLL}^{\uplus}$ are indeed equivalent, we shall prove the following lemmas.

Lemma 3.4.3
$\Gamma \underset{\text{NLL}}{\vdash} M : \sigma$ *implies* $\Gamma \underset{\text{NLL}^{\uplus}}{\vdash} M : \sigma$.

Proof. Most clearly, the core rules of NLL are special cases of those of NLL$^{\uplus}$. In particular, for the typing rules with two premises, we have that Γ_1, Γ_2 is well-formed if Γ_1 and Γ_2 are disjoint, and therefore equivalent to $\Gamma_1 \uplus \Gamma_2$.

We are left to prove that Weakening and Contraction are admissible in NLL$^{\uplus}$. This is obtained easily by induction on the derivation of $\Gamma \underset{\text{NLL}}{\vdash} M : \sigma$. □

To prove the other direction of the implication, we need the following syntactic lemma.

Lemma 3.4.4
If $\Gamma \vdash M : \sigma$, *then* $\Gamma[\rho] \vdash M[\rho] : \sigma$, *where* ρ *is a renaming substitution verifying* $dom(\rho) \not\subseteq FV(M)$.

Proof. Easy induction on the derivation of $\Gamma \vdash M : \sigma$. □

Lemma 3.4.5
$\Gamma \underset{\text{NLL}^{\uplus}}{\vdash} M : \sigma$ *implies* $\Gamma \underset{\text{NLL}}{\vdash} M : \sigma$.

Proof. By induction on the derivation of $\Gamma \underset{\text{NLL}^{\uplus}}{\vdash} M : \sigma$.

We consider only two prototypical cases: the Identity rule and the $\otimes_{\mathcal{I}}$ rule; the arguments for the other cases fit the same pattern.

- The Identity rule

$$\frac{|\Gamma| \sqsupseteq \top}{\Gamma, x : \sigma^a \vdash x : \sigma}$$

is derivable in NLL by repeatedly weakening all variables in Γ:

$$\dfrac{\dfrac{}{x : \sigma^a \vdash x : \sigma}\text{ Identity}}{\Gamma, x : \sigma^a \vdash x : \sigma}\text{ Weakening}$$

The condition $|\Gamma| \sqsupseteq \top$ is there to ensure that Weakening in indeed applicable.

- For the $\otimes_{\mathcal{I}}$ rule

$$\frac{\Gamma_1 \vdash M_1 : \sigma_1 \quad \Gamma_2 \vdash M_2 : \sigma_2 \quad |\Gamma_1| \sqsupseteq a_1 \quad |\Gamma_2| \sqsupseteq a_2}{\Gamma_1 \uplus \Gamma_2 \vdash \langle M_1, M_2 \rangle^{a_1, a_2} : {\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2}}$$

we have that $\Gamma_1 \uplus \Gamma_2 = \Gamma_1, \Gamma_2$ whenever Γ_1 and Γ_2 are disjoint, so the only interesting case is when both contexts have some variables in common. Therefore, we show that the above rule is provable in NLL assuming that $dom(\Gamma_1) \cap dom(\Gamma_2) \neq \emptyset$.

Let $\Gamma_1 = \Gamma_1', \Gamma_1''$ and $\Gamma_2 = \Gamma_2', \Gamma_2''$, where Γ_1'' and Γ_2'' share all the common variables (and so Γ_1' and Γ_2' are disjoint). By definition of context merge, $\Gamma_1 \uplus \Gamma_2 = \Gamma_1', \Gamma_2', \Gamma''$ with $\Gamma''(x) = \sigma^{a_1 + a_2}$ if and only if $\Gamma_1''(x) = \sigma^{a_1}$ and $\Gamma_2''(x) = \sigma^{a_2}$. (In other words, Γ'' combines the annotations of the common variables in Γ_1 and Γ_2.)

By the induction hypothesis and Lemma 3.4.4, we have in NLL that $\Gamma_1', \Gamma_1''[\rho_1] \vdash M_1[\rho_1] : \sigma_1$ and $\Gamma_2', \Gamma_2''[\rho_2] \vdash M_2[\rho_2] : \sigma_2$, where ρ_1 and ρ_2 are renaming substitutions, with

$\rho_1(x) = x_1$ and $\rho_2(x) = x_2$, for all $x \in dom(\Gamma'')$ and fresh variables x_1 and x_2. The renamings ensure that $\Gamma''_1[\rho_1]$ and $\Gamma''_2[\rho_2]$ are disjoint in order to apply the $\otimes_{\mathcal{I}}$ rule of NLL. We then recover the original names together with the combined annotations in Γ'' by carefully applying Contraction several times. Each application contracts $x_1 \in dom(\Gamma''_1[\rho_1])$ and $x_2 \in dom(\Gamma''_2[\rho_2])$ into $x \in dom(\Gamma'')$, as expected. The type derivation (omitting the intermediate steps) would look like this:

$$\dfrac{\dfrac{\Gamma'_1, \Gamma''_1[\rho_1] \vdash M_1[\rho_1] : \sigma_1 \quad \Gamma'_2, \Gamma''_2[\rho_2] \vdash M_2[\rho_2] : \sigma_2}{\Gamma'_1, \Gamma'_2, \Gamma''_1[\rho_1], \Gamma''_2[\rho_2] \vdash \langle M_1[\rho_1], M_2[\rho_2]\rangle^{a_1,a_2} : {\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2}}\otimes_{\mathcal{I}}}{\Gamma'_1, \Gamma'_2, \Gamma'' \vdash \langle M_1, M_2\rangle^{a_1,a_2} : {\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2}}\text{Contraction}$$

The same argument applies to all the other rules with two premises.

□

We have seen how to transform a type system containing explicit structural rules into an equivalent one where the Contraction rule is implicit in the rules involving two premises and the Weakening rule is implicit in the axiom rules. Because this transformation does only depend on the way contexts are used and not on the nature of the type rules themselves, we shall implicitly assume its validity for other type systems. In general, if $\mathcal{L}$ is an intermediate language, we shall write $\mathcal{L}^{\uplus}$ to refer to its syntax-directed version.

3.5 Type-theoretic properties

In this section, we study some basic typing properties of NLL, the most important of which will imply the semantic correctness of linearity analysis with respect to the operational semantics of the source language.

3.5.1 Some elementary properties

We start by observing that in NLL, every term that is typeable has a unique type. This property, as well as the remaining properties in this subsection, are easily proved by induction on derivations.

Proposition 3.5.1 (Unique Typing)
If $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$, then $\sigma \equiv \tau$.

The Typing Uniqueness property is not preserved if we decide to omit any of the linearity annotations on functions or pairs. However, we may decide for practical reasons to consider terms with redundant annotations (and base type information) in order to simplify the definition of the compiler optimisations. An example is provided by the let construct, used extensively throughout in several examples.

Proposition 3.5.2 (Single Occurrence)
If $\Gamma, x : \sigma^1 \vdash M : \tau$, then x occurs exactly once in M[7]. □

[7] For a precise definition of what we mean by 'occurs exactly once', refer to Figure 7.4 on page 147.

The following Annotation Weakening property provides an interesting interpretation of the order relation as the inclusion of annotated term contexts. (As we shall see in the following chapter, Annotation Weakening may be understood as a 'rudimentary' form of subtyping.)

Proposition 3.5.3 (Annotation Weakening)
The following rule is admissible in NLL*:*

$$\frac{\Gamma, x : \sigma^1 \vdash M : \tau}{\Gamma, x : \sigma^\top \vdash M : \tau}\ \text{Transfer}$$

□

We may also encounter the Transfer rule written in a slightly different, but equivalent, form:

$$\frac{\Gamma_1 \vdash M : \sigma \quad \Gamma_1 \sqsubseteq \Gamma_2}{\Gamma_2 \vdash M : \sigma}\ \text{Transfer}$$

3.5.2 Embedding FPL into NLL

A theory of static analysis should be expressive enough to provide an analysis, no matter how approximate it may be, for every input term of the source language.

This is rather obvious in our case, as there is always a 'worst' analysis corresponding to the annotated version of Girard's translation. We write $(-^\bullet)$ for this translation, and define it on FPL types as follows:

$$\mathbb{G}^\bullet = \mathbb{G} \tag{3.16}$$

$$(\sigma \to \tau)^\bullet = (\sigma^\bullet)^\top \multimap \tau^\bullet \tag{3.17}$$

$$(\sigma \times \tau)^\bullet = (\sigma^\bullet)^\top \otimes (\tau^\bullet)^\top \tag{3.18}$$

For typing contexts, we let

$$(-)^\bullet = - \quad \text{and} \quad (\Gamma, x : \sigma)^\bullet = \Gamma^\bullet, x : (\sigma^\bullet)^\top. \tag{3.19}$$

Terms are translated in the obvious way, and in particular we have that $(\lambda x{:}\sigma.M)^\bullet = \lambda x{:}(\sigma^\bullet)^\top.M^\bullet$, $\langle M_1, M_2\rangle^\bullet = \langle {M_1}^\bullet, {M_2}^\bullet\rangle^{\top,\top}$ and $\pi^\bullet = \pi^{\top,\ldots,\top}$ if π is an operator.

We formalise the completeness of the analysis by stating the following proposition.

Proposition 3.5.4 (Completeness)
$\Gamma \underset{\text{FPL}}{\vdash} M : \sigma$ *implies* $\Gamma^\bullet \underset{\text{NLL}}{\vdash} M^\bullet : \sigma^\bullet$.

Proof. Obvious, from the fact that the typing rules of NLL, restricted to intuitionistic annotations only, are in clear correspondence with the typing rules of FPL. □

The following statement establishes a rather immediate relationship existing between the erasing and embedding functors.

Proposition 3.5.5
Let J be any valid FPL *typing judgment. Then,* $J \equiv (J^\bullet)^\circ$. □

This last observation applies to type derivations as well.

3.5.3 Substitution

We now show that substitution is well-behaved with respect to typing under certain reasonable restrictions. This property will play a fundamental role in the proof of the semantic correctness of our intermediate language, as we shall see next.

Lemma 3.5.6 (Substitution)
The following rule is admissible in NLL.

$$\frac{\Gamma_1, x : \sigma^a \vdash M : \tau \quad \Gamma_2 \vdash N : \sigma \quad |\Gamma_2| \sqsupseteq a}{\Gamma_1, \Gamma_2 \vdash M[N/x] : \tau}\text{Substitution}$$

Proof. We shall actually prove a more general property for $\text{NLL}^{\uplus}$:

$$\frac{\Gamma_1, x : \sigma^a \vdash M : \tau \quad \Gamma_2 \vdash N : \sigma \quad |\Gamma_2| \sqsupseteq a}{\Gamma_1 \uplus \Gamma_2 \vdash M[N/x] : \tau}$$

We proceed by induction on the structure of M. We assume $\Gamma_2 \vdash N : \sigma$.

- $M \equiv x$.

  Immediate, because of the fact that $x[N/x] \equiv N$ and, by the Identity rule, $\tau \equiv \sigma$.

- $M \equiv y$ and $y \not\equiv x$.

  In this case, we must have that $\Gamma_1, x : \sigma^\top \vdash y : \tau$ where $\Gamma_1(y) = \tau^b$ for some b. The result follows from the fact that $y[N/x] \equiv y$. This same reasoning applies when $M \equiv k$ as well.

- $M \equiv \lambda y{:}\tau_1^b.M'$.

  Suppose $\Gamma_1, x : \sigma^a \vdash \lambda y{:}\tau_1^b.M' : \tau_1^b \multimap \tau_2$, with $\tau \equiv \tau_1^b \multimap \tau_2$, because $\Gamma_1, x : \sigma^a, y : \tau_1^b \vdash M' : \tau_2$. Applying the induction hypothesis to the latter and the assumption $\Gamma_2 \vdash N : \sigma$, we obtain $(\Gamma_1, y : \tau_1^b) \uplus \Gamma_2 \vdash M'[N/x] : \tau_2$. Assuming that $y \notin dom(\Gamma_2)$ by α-equivalence, we can now apply the $\multimap_{\mathcal{I}}$ rule to obtain $\Gamma_1 \uplus \Gamma_2 \vdash \lambda y{:}\tau_1^b.M'[N/x] : \tau_1^b \multimap \tau_2$. The desired result follows from the fact that, in this case, $\lambda y{:}\tau_1^b.M'[N/x] \equiv (\lambda y{:}\tau_1^b.M')[N/x]$.

  This same reasoning applies to the fixpoint construct.

- $M \equiv M'N'$.

  Suppose $\Gamma_1, x : \sigma^a \vdash M'N' : \tau$ because $\Gamma_1' \vdash M' : \tau_1^b \multimap \tau$ and $\Gamma_1'' \vdash N' : \tau_1$ and $|\Gamma_1''| \sqsupseteq b$, with $\Gamma_1, x : \sigma^a = \Gamma_1' \uplus \Gamma_1''$. There are three sub-cases to consider, corresponding to whether x appears free in Γ_1', Γ_1'', or both.

  - $x \in dom(\Gamma_1')$, but $x \notin dom(\Gamma_1'')$.
    We can now apply the induction hypothesis to $\Gamma_1' \vdash M' : \tau_1^b \multimap \tau$ and the assumption $\Gamma_2 \vdash N : \sigma$ to conclude $\Gamma_1' \uplus \Gamma_2 \vdash M'[N/x] : \tau_1^b \multimap \tau$. By the $\multimap_{\mathcal{E}}$ rule, we have that $(\Gamma_1' \uplus \Gamma_2) \uplus \Gamma_1'' \vdash (M'[N/x])N' : \tau$ from our previous conclusion and the sequent $\Gamma_1'' \vdash N' : \tau_1$. The desired result, $\Gamma_1' \uplus \Gamma_1'' \uplus \Gamma_2 \vdash (M'N')[N/x] : \tau$, follows from commutativity and associativity of $\uplus$, and the fact that in our case $(M'[N/x])N' \equiv (M'N')[N/x]$.

– $x \in dom(\Gamma''_1)$, but $x \notin dom(\Gamma'_1)$.

Similarly, we can obtain $\Gamma''_1 \uplus \Gamma_2 \vdash N'[N/x] : \tau_1$ by applying the induction hypothesis to $\Gamma''_1 \vdash N' : \tau_1$ and $\Gamma_2 \vdash N : \sigma$. Since we have $|\Gamma''_1 \uplus \Gamma_2| \sqsupseteq b$ from the fact that, by assumption $|\Gamma_2| \sqsupseteq a$ and $|\Gamma''_1, x : \sigma^a| \sqsupseteq b$ (hence, $|\Gamma_2| \sqsupseteq b$), we can apply the $\multimap_{\mathcal{E}}$ rule to our previous conclusion and $\Gamma'_1 \vdash M' : \tau_1^b \multimap \tau$ to derive $\Gamma'_1 \uplus (\Gamma''_1 \uplus \Gamma_2) \vdash M'(N'[N/x]) : \tau$, which implies the desired conclusion.

– $x \in dom(\Gamma'_1)$ and $x \in dom(\Gamma''_1)$.

In this case, we must have that $\Gamma_1(x) = \sigma^{a_1+a_2}$ where $\Gamma'_1(x) = \sigma^{a_1}$ and $\Gamma''_1(x) = \sigma^{a_2}$, for some a_1 and a_2, with $a \equiv a_1 + a_2 = \top$, in our linear theory. Since $|\Gamma_2| \sqsupseteq a_1$ and $|\Gamma_2| \sqsupseteq a_2$, because of the assumption $|\Gamma_2| \sqsupseteq a$, we can apply the induction hypothesis twice, as we did in our previous two sub-cases, to obtain $\Gamma'_1 \uplus \Gamma_2 \vdash M'[N/x] : \tau_1^b \multimap \tau$ and $\Gamma''_1 \uplus \Gamma_2 \vdash N'[N/x] : \tau_1$. From $|\Gamma_2| \sqsupseteq a_2$ and the assumption $|\Gamma''_1, x : \sigma^{a_2}| \sqsupseteq b$, we deduce that $|\Gamma''_1 \uplus \Gamma_2| \sqsupseteq b$, and so we can apply the $\multimap_{\mathcal{E}}$ rule to our previous two conclusions to obtain $(\Gamma'_1 \uplus \Gamma_2) \uplus (\Gamma''_1 \uplus \Gamma_2) \vdash M'[N/x]N'[N/x] : \tau$. The desired conclusion follows from the properties of $\uplus$ and substitution. In particular, note that $\Gamma_2 \uplus \Gamma_2 = \Gamma_2$, since $|\Gamma_2| \sqsupseteq \top$.

The same reasoning applies to the other typing rules with more than two premises.

□

3.5.4 Semantic correctness

Having proved the Substitution Lemma, we are now in a position to establish the correctness of our analysis with respect to the notion of reduction induced by the source language. The correctness argument states that reducing an annotated program can never result in an annotated program that is ill-typed, thus ensuring the validity of the analysis throughout evaluation.

Theorem 3.5.7 (Subject Reduction)
Whenever $\Gamma \underset{\text{NLL}}{\vdash} M : \sigma$ *and* $M \rightarrow N$, *then* $\Gamma \underset{\text{NLL}}{\vdash} N : \sigma$.

Proof. We show this for $\text{NLL}^{\uplus}$ by induction on $\rightarrow$-derivations.

- $M \equiv (\lambda x{:}\tau^a.M')N'$ and $N \equiv M'[N'/x]$.

  Suppose $\Gamma \vdash (\lambda x{:}\tau^a.M')N' : \sigma$ because $\Gamma' \vdash \lambda x{:}\tau^a.M' : \tau^a \multimap \sigma$ and $\Gamma'' \vdash N' : \tau$ with $\Gamma \equiv \Gamma' \uplus \Gamma''$, since a derivation for M must necessarily end with an application of $\multimap_{\mathcal{E}}$. We also have that $|\Gamma''| \sqsupseteq a$. By $\multimap_{\mathcal{I}}$, we have that $\Gamma', x : \tau^a \vdash M' : \sigma$ must justify $\Gamma' \vdash \lambda x{:}\tau^a.M' : \tau^a \multimap \sigma$. The annotation restrictions on Γ'' ensure that the Substitution Lemma is indeed applicable to $\Gamma', x : \tau^a \vdash M' : \sigma$ and $\Gamma'' \vdash N' : \tau$ to obtain $\Gamma' \uplus \Gamma'' \vdash M'[x/N'] : \sigma$.

- $M \equiv \mathsf{let}\ \langle x_1, x_2\rangle = \langle M'_1, M'_2\rangle^{a_1,a_2}\ \mathsf{in}\ N'$ and $N \equiv N'[M'_1/x_1][M'_2/x_2]$.

  A derivation for M must necessarily end with an application of the $\otimes_{\mathcal{E}}$ rule. Suppose $\Gamma \vdash \mathsf{let}\ \langle x_1, x_2\rangle = \langle M'_1, M'_2\rangle^{a_1,a_2}\ \mathsf{in}\ N' : \sigma$ because $\Gamma' \vdash \langle M'_1, M'_2\rangle^{a_1,a_2} : \tau_1^{a_1} \otimes \tau_2^{a_2}$ and $\Gamma'', x_1 : \tau_1^{a_1}, x_2 : \tau_2^{a_2} \vdash N' : \sigma$ with $\Gamma \equiv \Gamma' \uplus \Gamma''$. We also know that the first premise can only be justified with an application of $\otimes_{\mathcal{I}}$ to the premises $\Gamma'_1 \vdash M'_1 : \tau_1$ and $\Gamma'_2 \vdash M'_2 : \tau_2$ with $\Gamma' \equiv \Gamma'_1 \uplus \Gamma'_2$. Because the rule requires that $|\Gamma'_1| \sqsupseteq a_1$ and $|\Gamma'_2| \sqsupseteq a_2$, we can apply

the Substitution Lemma to $\Gamma'', x_1 : \tau_1^{a_1}, x_2 : \tau_2^{a_2} \vdash N' : \sigma$ and $\Gamma'_1 \vdash M'_1 : \tau_1$ to obtain $\Gamma'_1 \uplus (\Gamma'', x_2 : \tau_2^{a_2}) \vdash N'[M'_1/x_1] : \sigma$ and again to the latter and $\Gamma'_2 \vdash M'_2 : \tau_2$ to obtain $\Gamma'_1 \uplus \Gamma'_2 \uplus \Gamma'' \vdash N'[M'_1/x_1][M'_2/x_2] : \sigma$.

- $M \equiv \mathsf{if}\ k\ \mathsf{then}\ N'_1\ \mathsf{else}\ N'_2$ and either $N \equiv N'_1$ or $N \equiv N'_2$.

  Immediate from the fact that N'_1 and N'_2 have type σ under the Conditional rule.

- $M \equiv \mathsf{fix}\ x{:}\sigma.M'$ and $N \equiv M'[\mathsf{fix}\ x{:}\sigma.M'/x]$.

  By the Fixpoint rule, suppose that $\Gamma \vdash \mathsf{fix}\ x{:}\sigma.M' : \sigma$ because $\Gamma, x : \sigma^\top \vdash M' : \sigma$ with $|\Gamma| \sqsupseteq \top$. We can now apply the Substitution Lemma to premise and conclusion to obtain $\Gamma \uplus \Gamma \vdash M'[\mathsf{fix}\ x{:}\sigma.M'/x] : \sigma$, as required. (Note that $\Gamma \uplus \Gamma = \Gamma$, in this case where $|\Gamma| \sqsupseteq \top$.)

- $M \equiv \mathcal{C}[M']$ and $N \equiv \mathcal{C}[N']$ with $M' \to N'$.

  Suppose $\Gamma \vdash \mathcal{C}[M'] : \sigma$ because $\Gamma', x : \tau^a \vdash \mathcal{C}[x] : \sigma$ and $\Gamma'' \vdash M' : \tau$ with $\Gamma \equiv \Gamma' \uplus \Gamma''$ and $|\Gamma''| \sqsupseteq a$. By the induction hypothesis, we have that $\Gamma'' \vdash N' : \tau$. The required conclusion $\Gamma \vdash \mathcal{C}[N'] : \sigma$ easily follows from the Substitution Lemma.

□

Using the results of Theorem 3.5.7 and Proposition 3.5.2, we can attempt a *contextual* natural language definition of the notion of usage implied by linearity analysis.

Definition 3.5.8 (Linear usage)
Let $P[x{:}\sigma]$ be a well-typed program with a distinguished 'hole' $x{:}\sigma$. We may say that x has *linear usage* in P if no reduction strategy exists that may duplicate or erase x. □

Because we have not committed ourselves to a particular evaluation strategy, Theorem 3.5.7 implies that the static information provided by linearity analysis is correct for any reduction strategy, so we can in principle apply the analysis to both call-by-value and call-by-need languages. The price to pay for this level of generality is a certain loss in the expressivity of the analysis, as we are not allowed to use any information specific to a given evaluation strategy.

More concretely, consider the following simple annotated program:

$$\mathsf{let}\ x{:}\mathsf{int}^\top = 1 + 2\ \mathsf{in}\ \mathsf{fst}\ \langle x, x\rangle^{1,\top}$$

Because x occurs twice, Contraction forces a $\top$ annotation for x. We also have that the pair is non-linear on its second component because fst discards it. This analysis is compatible with our intuitive understanding of 'usage' in a call-by-value language: Before applying fst, the variable x is evaluated twice as part of the evaluation of the pair[8]. In contrast to this, x is evaluated only once in a call-by-need language, after fst returns it as result. Assigning a linear annotation to x could be more profitable in this case, although it would be completely wrong from a 'logical' viewpoint. Actually, it would allow us to suggest the inlining of x to obtain

$$\mathsf{fst}\ \langle 1 + 2, 1 + 2\rangle^{1,\top},$$

[8] If you think in terms of transitions in an abstract machine like Krivine's, each evaluation of x corresponds to accessing the closure associated to x in the environment. The $\top$ annotation for x is compatible with the fact that x is accessed twice.

although it would perhaps not be a good idea to allow inlining in this case to avoid duplicating code; but that is a different story. For call-by-need languages, observations like this one have triggered some interesting research. A good example is the usage analyser used by the Glasgow Haskell Compiler [68, 67].

3.5.5 Considering η-reduction

If we were interested in having a reduction system that includes a notion of η-reduction, we should notice that adding the rule

$$\lambda x{:}\sigma^a.M\ x \to M \qquad \text{if } x \notin FV(M) \tag{3.20}$$

to NLL results in a system that is operationally unsound, in the sense that Theorem 3.5.7 is no longer valid. Indeed, taking $M \equiv \lambda y{:}\sigma^1.y$ and $a \equiv \top$, we see that

$$\lambda x{:}\sigma^\top.(\lambda y{:}\sigma^1.y)\ x\ :\ \sigma^\top \multimap \sigma, \quad \text{but} \quad \lambda y{:}\sigma^1.y\ :\ \sigma^1 \multimap \sigma;$$

so redex and reduct do not have equivalent types[9].

There is no trouble, however, in allowing a restricted linear instance of the rule above:

$$\lambda x{:}\sigma^1.M\ x \to M \qquad \text{if } x \notin FV(M) \tag{3.21}$$

The more generic η-rule is nonetheless desirable in our intermediate language if we would like transformations in the source language to remain valid in the intermediate language. A solution to this problem will be provided by subtyping in the next chapter.

3.6 Optimal typings

In Subsection 3.5.2, we have shown that for each source language term M there is always a worst analysis, noted $M^\bullet$, which corresponds to the (not very useful) decoration providing no structural information at all. Linearity analysis has, therefore, at least one solution.

In this section, we shall show that there is also a best or *optimal analysis*. A standard method to prove the existence of an optimal analysis consists in showing that the set of all decorated typings forms an ordered set that admits a smallest element [48]. As we shall see, for the case of our simple linearity analysis, the space of all analyses (decorations) forms a complete lattice. The order relation used is analogous to the *sub-decoration* relation considered by Danos and Schellinx [26].

We begin by defining an order relation on typing judgments that we shall use to compare analyses in terms of their information contents. Intuitively, if ${J_1}^*$ and ${J_2}^*$ are two decorated typing judgments,

$${J_1}^* \sqsubseteq {J_2}^*$$

should somehow express the fact that linearity information in ${J_1}^*$ must be more precise than that in ${J_2}^*$. The order relation that compares corresponding annotations on both typing judgments seems to be a good candidate.

[9] This should not come as a surprise if we understand NLL reductions as 'syntactic sugar' of more verbose DILL reductions. With this in mind, it is clear that the intuitionistic instance of the above η-rule does not correspond to any legal reduction sequence in DILL.

Definition 3.6.1 (Sub-decoration order)
If $J_1{}^* \equiv \Gamma_1 \vdash M_1 : \sigma_1$ and $J_2{}^* \equiv \Gamma_2 \vdash M_2 : \sigma_2$ are two enriched typing judgments, then let $J_1{}^* \sqsubseteq J_2{}^*$ be the reflexive and transitive closure of the relation generated by the rule

$$\frac{\Gamma_1 \sqsubseteq \Gamma_2 \quad M_1 \sqsubseteq M_2 \quad \sigma_1 \sqsubseteq \sigma_2}{(\Gamma_1 \vdash M_1 : \sigma_1) \sqsubseteq (\Gamma_2 \vdash M_2 : \sigma_2)} \tag{3.22}$$

For any two decorated types, terms or contexts, the relation is defined by simply comparing annotations at corresponding positions. Below, we show the rules that define the sub-decoration order on types.

$$\frac{}{\mathbb{G} \sqsubseteq \mathbb{G}} \tag{3.23}$$

$$\frac{\sigma_1 \sqsubseteq \tau_1 \quad \sigma_2 \sqsubseteq \tau_2 \quad a_1 \sqsubseteq a_2}{\sigma_1{}^{a_1} \multimap \tau_1 \sqsubseteq \sigma_2{}^{a_2} \multimap \tau_2} \tag{3.24}$$

$$\frac{\sigma_1 \sqsubseteq \tau_1 \quad \sigma_2 \sqsubseteq \tau_2 \quad a_1 \sqsubseteq a_2 \quad b_1 \sqsubseteq b_2}{\sigma_1{}^{a_1} \otimes \tau_1{}^{b_1} \sqsubseteq \sigma_2{}^{a_2} \otimes \tau_2{}^{b_2}} \tag{3.25}$$

□

The purpose of showing the above rules is simply to note that, unlike the subtyping relation of Section 4.1, $\sqsubseteq$ is covariant everywhere, including function domains.

Definition 3.6.2 (Decoration space)
The *decoration space* associated to a source language typing J is written $\mathfrak{D}_{\text{NLL}}(J)$ and defined to be the set of NLL typings J^* that are decorated versions of J:

$$\mathfrak{D}_{\text{NLL}}(J) \stackrel{\text{def}}{=} \{J^* \mid (J^*)^\circ = J\}. \tag{3.26}$$

□

It is not difficult to prove that the decoration space forms a complete lattice under the sub-decoration order. The proof is somewhat tedious, so we shall only cover some representative cases. We shall find a more convenient (and interesting) way of proving the existence of solutions in the context of annotation inference.

We should first remark that the set of all the decorated types of a given underlying type also forms a complete lattice.

Lemma 3.6.3
Given a source type σ, the set of all decorated types

$$\{\tau \mid (\tau)^\circ \equiv \sigma\}$$

ordered with $\sqsubseteq$ forms a complete lattice. The same can be said of the set of all decorated terms and contexts.

Proof. Note that $\mathbb{A}_{\text{NLL}}$ is itself a complete lattice, and that $\sqsubseteq$, if we abstract over the underlying syntactic structure of a type or context, is morally an extension of $\sqsubseteq$ to products of annotations. □

Theorem 3.6.4 (Complete decoration lattice)
For any give source typing judgment J, the structure $\langle \mathfrak{D}_{\mathrm{NLL}}(J); \sqsubseteq \rangle$ forms a complete lattice.

Proof. It is clear that by Lemma 3.5.4, $J^\bullet \in \mathfrak{D}_{\mathrm{NLL}}(J)$, and $J^* \sqsubseteq J^\bullet$ for every $J^* \in \mathfrak{D}_{\mathrm{NLL}}(J)$, so $J^\bullet$ always exists and is the top element of our set of decorated typings.

It only remains to prove that meets exist for arbitrary non-empty subsets. We shall prove this by induction on the derivations of J in the source type system. In any case, let $D = \{{J_i}^* \mid i \in I\}$ be a non-empty subset of $\mathfrak{D}_{\mathrm{NLL}}(J)$ indexed by elements of I, with ${J_i}^* \equiv \Gamma_i \vdash M_i : \sigma_i$. We shall prove that $\sqcap D$ exists for some representative cases only; the other cases can be proved similarly.

- $J \equiv {\Gamma_i}^\circ, x : {\sigma_i}^\circ \vdash x : {\sigma_i}^\circ$.

  Each element of D in this case has the form $\Gamma_i, x : \sigma_i^{a_i} \vdash x : \sigma_i$, where $|\Gamma_i| = \top$. Lemma 3.6.3 guarantees the existence of $\sqcap\sigma_i$ and $\sqcap\Gamma_i$, and so that of $\sqcap D = \sqcap\Gamma_i, x : \sqcap{\sigma_i}^a \vdash x : \sqcap\sigma_i$, where $a \equiv \sqcap a_i$. (We should note that $|\sqcap\Gamma_i| = \top$.)

- $J \equiv {\Gamma_i}^\circ \vdash \lambda x{:}{\sigma_i}^\circ.{M_i}^\circ : {\sigma_i}^\circ \to {\tau_i}^\circ$.

  Suppose $\Gamma_i \vdash \lambda x{:}{\sigma_i}^{a_i}.M_i : {\sigma_i}^{a_i} \multimap \tau_i$ because $\Gamma_i, x : {\sigma_i}^{a_i} \vdash M_i : \tau_i$. Clearly, the latter is an element of $\mathfrak{D}_{\mathrm{NLL}}({\Gamma_i}^\circ, x : {\sigma_i}^\circ \vdash {M_i}^\circ : {\tau_i}^\circ)$; therefore, by the induction hypothesis, a meet exists for D and is defined component-wise as $\sqcap\Gamma_i, x : \sqcap{\sigma_i}^a \vdash \sqcap M_i : \sqcap\tau_i$, where $a \equiv \sqcap a_i$. Applying $\multimap_{\mathcal{I}}$ to the meet, we can conclude $\sqcap\Gamma_i \vdash \lambda x{:}\sqcap{\sigma_i}^a.\sqcap M_i : \sqcap{\sigma_i}^a \multimap \sqcap\tau_i$, which by definition equals $\sqcap D$.

- $J \equiv {\Gamma_i}^\circ \vdash {M_i}^\circ {N_i}^\circ : {\tau_i}^\circ$.

  Suppose $\Gamma_i \vdash M_i\ N_i : \tau_i$ because $\Gamma'_i \vdash M_i : {\sigma_i}^{a_i} \multimap \tau_i$ and $\Gamma''_i \vdash N_i : \sigma_i$ where $\Gamma_i = \Gamma'_i, \Gamma''_i$ and $|\Gamma''| \sqsupseteq a_i$. Clearly, both premises are, respectively, elements of $\mathfrak{D}_{\mathrm{NLL}}({\Gamma'_i}^\circ \vdash {M_i}^\circ : {\sigma_i}^\circ \to {\tau_i}^\circ)$ and $\mathfrak{D}_{\mathrm{NLL}}({\Gamma'_i}^\circ \vdash {N_i}^\circ : {\sigma_i}^\circ)$. By the induction hypothesis, twice, we have that $\sqcap\Gamma'_i \vdash \sqcap M_i : \sqcap{\sigma_i}^{\sqcap a_i} \multimap \sqcap\tau_i$ and $\sqcap\Gamma''_i \vdash \sqcap N_i : \sqcap\sigma_i$ must define the meets of these decoration spaces. Because the annotation set of linearity analysis is a lattice, we have $|\Gamma''| \sqsupseteq a_i$ implies $|\sqcap\Gamma''_i| \sqsupseteq \sqcap a_i$; hence, by $\multimap_{\mathcal{E}}$, we can conclude $\sqcap\Gamma'_i, \sqcap\Gamma''_i \vdash \sqcap M_i\ \sqcap N_i : \tau_i$.

□

We are now able to simply characterise the optimal typing as the meet of the whole decoration space:

$$J^{\mathrm{opt}} \stackrel{\mathrm{def}}{=} \sqcap(\mathfrak{D}_{\mathrm{NLL}}(J)). \tag{3.27}$$

The proof of the above theorem relies heavily on the fact that $\mathbb{A}_{\mathrm{NLL}}$ must itself be a complete lattice; or, in other words, that there is a natural choice in the form of a 'best' annotation. This condition is necessary to prove Lemma 3.6.3, which guarantees the existence of a 'best' type among an arbitrary subset of decorated types. This will not be true for more complex posets of structural properties, where there is not one (canonical) smallest annotation, but many possible minimal annotations from which to choose. We can still prove a weaker theorem by relaxing our definition of $\sqsubseteq$; but we shall come back to this problem later, where our motivations will also be made clearer. The stronger result, though, will remain true for all theories based on 2-point posets, like the theories for affine and neededness analysis of Section 7.3.

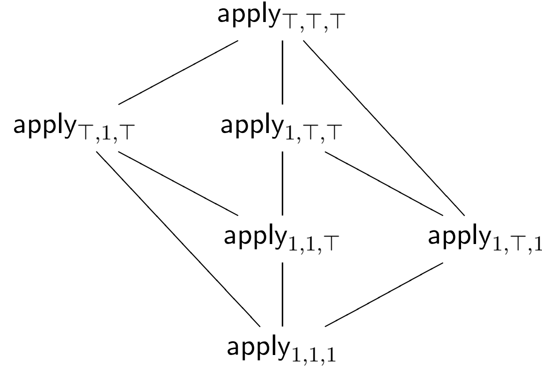


Figure 3.9: Decoration space for the apply function

As an example, Figure 3.9 provides a pictorial representation of the space of solutions for the apply function. Each solution is abbreviated

$$\mathsf{apply}_{a,b,c} \equiv \lambda f{:}(\sigma^a \multimap \tau)^b.\lambda x{:}\sigma^c.f\,x.$$

Recall that $\mathsf{apply}_{a,b,c}$ is a valid decoration for all a, b, c such that $a \sqsubseteq c$. The worst decorated term is precisely $\mathsf{apply}_{\top,\top,\top}$; the best is, quite luckily, $\mathsf{apply}_{1,1,1}$.

3.7 Applications

Many variants of intuitionistic linear logic (or some suitable fragment of it) have been proposed, with the hope of coming up with more efficient implementation techniques for functional languages. All the techniques proposed rely on the fact that linear logic can be used to faithfully distinguish between shared and non-shared resources. The idea is that the property 'linear' can be used as an approximation of the property 'non-shared'. As it turns out, this approximation is unsafe for most functional language implementations. The reasons depend on the details of what 'sharing' is supposed to mean for a given implementation, so the problems encountered, even if they present some similarities, may differ in many respects.

As a sound application of linear logic, inlining does not suffer from the semantical gap mentioned above, as it is formulated at a fairly high-level of abstraction, depending only on properties of the intermediate language like the Substitution Lemma.

After formalising and discussing the inlining optimisation, we shall briefly comment on some related work concerning some applications to sharing and single-threading.

3.7.1 Inlining

It is straightforward to formalise inlining as a single-step reduction relation on annotated terms that substitutes linear uses of definitions by their corresponding definition bodies.

Let $\overset{\text{inl}}{\leadsto}$ stand for this relation, defined as the contextual closure of the rewrite rules of Figure 3.10.

Inlining a whole program corresponds to the process of iteratively applying the rewrite rules in any order until no more linear redexes are found. It is not difficult to see that

$$(\lambda x{:}\sigma^1.M)N \overset{\text{inl}}{\rightsquigarrow} M[N/x]$$
$$\mathsf{let}\ \langle x_1, x_2\rangle = \langle M_1, M_2\rangle^{1,1}\ \mathsf{in}\ N \overset{\text{inl}}{\rightsquigarrow} N[M_1/x_1][M_2/x_2]$$
$$\mathsf{let}\ \langle x_1, x_2\rangle = \langle M_1, M_2\rangle^{1,\top}\ \mathsf{in}\ N \overset{\text{inl}}{\rightsquigarrow} \mathsf{let}\ x_2 = M_2\ \mathsf{in}\ N[M_1/x_1]$$
$$\mathsf{let}\ \langle x_1, x_2\rangle = \langle M_1, M_2\rangle^{\top,1}\ \mathsf{in}\ N \overset{\text{inl}}{\rightsquigarrow} \mathsf{let}\ x_1 = M_1\ \mathsf{in}\ N[M_2/x_2]$$
$$\mathsf{let}\ x{:}\sigma^1 = M\ \mathsf{in}\ N \overset{\text{inl}}{\rightsquigarrow} N[M/x]$$

Figure 3.10: The inlining optimisation relation

the inlining relation is confluent and strongly normalising, so the process must eventually terminate with the same completely inlined program.

Note that $\overset{\text{inl}}{\rightsquigarrow} \subseteq \rightarrow$, hence the correctness of the inlining transformation follows as a corollary of subject reduction.

Proposition 3.7.1 (Correctness of $\overset{\text{inl}}{\rightsquigarrow}$)
If $\Gamma \underset{\text{NLL}}{\vdash} M : \sigma$ *and* $M \overset{\text{inl}}{\rightsquigarrow} N$, *then* $\Gamma \underset{\text{NLL}}{\vdash} N : \sigma$. □

By the Single Occurrence property (Proposition 3.5.2), we know that the substitutions on the right-hand side of the rules will not (syntactically) duplicate any terms.

As an example, we apply inlining to optimise the following input FPL program. (For reasons of clarity, we have omitted any base type information.)

$$\begin{array}{l}
\mathsf{let\ uncurry} = \lambda f.\lambda x.\mathsf{let}\ \langle x_1, x_2\rangle = x\ \mathsf{in}\ f\,x_1\,x_2\ \mathsf{in}\\
\quad \mathsf{let\ sum} = \lambda y_1.\lambda y_2.y_1 + y_2\ \mathsf{in}\\
\quad\quad \mathsf{let\ n} = 1 + 2\ \mathsf{in}\\
\quad\quad\quad \mathsf{uncurry\ sum}\ \langle 3, \mathsf{sum\ n}\ 1\rangle
\end{array}$$

Applying our analysis to the example will output the decorated version shown below. We have decided to omit any base type information, leaving only the annotations on bound variables and pairs. (The reader may like to find the decoration corresponding to the annotations shown.)

$$\begin{array}{l}
\mathsf{let\ uncurry}{:}1 = \lambda f{:}1.\lambda x{:}1.\mathsf{let}\ \langle x_1, x_2\rangle = x\ \mathsf{in}\ f\,x_1\,x_2\ \mathsf{in}\\
\quad \mathsf{let\ sum}{:}\top = \lambda y_1{:}1.\lambda y_2{:}1.y_1 + y_2\ \mathsf{in}\\
\quad\quad \mathsf{let\ n}{:}1 = 1 + 2\ \mathsf{in}\\
\quad\quad\quad \mathsf{uncurry\ sum}\ \langle 3, \mathsf{sum\ n}\ 1\rangle^{1,1}
\end{array}$$

Except for the sum function which is used twice in the body of the innermost let, all other variables are linear.

As a strategy to apply the inlining transformation, we choose to always reduce the leftmost-outermost redex first (and inside functions, as well). Therefore, we start reducing the first

and third let to obtain

$$\begin{array}{l}\mathsf{let\ sum}{:}\top = \lambda y_1.\lambda y_2.y_1 + y_2\ \mathsf{in}\\ \quad (\lambda f{:}1.\lambda x{:}1.\mathsf{let}\ \langle x_1, x_2\rangle = x\ \mathsf{in}\ f\,x_1\,x_2)\ \mathsf{sum}\ \langle 3, \mathsf{sum}\ (1+2)\ 1\rangle^{1,1}\end{array}$$

The transformation proceeds inside the body of the let, with the following reduction sequence:

$$\begin{array}{ll} & (\lambda f{:}1.\lambda x{:}1.\mathsf{let}\ \langle x_1, x_2\rangle = x\ \mathsf{in}\ f\,x_1\,x_2)\ \mathsf{sum}\ \langle 3, \mathsf{sum}\ (1+2)\ 1\rangle^{1,1}\\ \overset{\mathrm{inl}}{\leadsto} & (\lambda x{:}1.\mathsf{let}\ \langle x_1, x_2\rangle = x\ \mathsf{in\ sum}\,x_1\,x_2)\ \langle 3, \mathsf{sum}\ (1+2)\ 1\rangle^{1,1}\\ \overset{\mathrm{inl}}{\leadsto} & \mathsf{let}\ \langle x_1, x_2\rangle = \langle 3, \mathsf{sum}\ (1+2)\ 1\rangle^{1,1}\ \mathsf{in\ sum}\,x_1\,x_2\\ \overset{\mathrm{inl}}{\leadsto} & \mathsf{sum}\,3\,(\mathsf{sum}\ (1+2)\ 1)\end{array}$$

We are left after inlining with a much shorter program:

$$\mathsf{let\ sum}{:}\top = \lambda y_1.\lambda y_2.y_1 + y_2\ \mathsf{in\ sum}\,3\,(\mathsf{sum}\ (1+2)\ 1).$$

We should remark that many opportunities for inlining would be lost if we restricted ourselves to the rewrite rules of Figure 3.10. For instance, the following rule turns a binary function call into a unary function call, with the second argument inlined in the function body:

$$(\lambda x_1{:}\sigma_1^\top.\lambda x_2{:}\sigma_2^1.M)\,N_1\,N_2 \overset{\mathrm{inl}}{\leadsto} (\lambda x_1{:}\sigma_1^\top.M[N_2/x_2])\,N_1$$

Many other rules, not necessarily related to inlining, would indeed be important in order to actually reveal redexes that might otherwise be hidden[10]. We shall here content ourselves with the study of the properties that enable the different optimisations, leaving out any details concerning how this optimisations may actually be performed in an actual compiler.

3.7.2 Limitations

As soon as we begin to try out our linearity analysis on realistic examples, we quickly find out that it is not as good as we thought, even in a call-by-value setting. For example, our analysis forbids the inlining of the outer let-definition of the following example term:

$$\begin{array}{l}\mathsf{let}\ x{:}\mathsf{int} = M\ \mathsf{in}\\ \quad \mathsf{let}\ y{:}\mathsf{int} = x + 1\ \mathsf{in}\ y + y\end{array}$$

Because y is non-linear, the Substitution rule requires that x be non-linear too, as it appears in the context needed to type $x+1$. It is however not difficult to see that applying the inlining transformation to obtain

$$\mathsf{let}\ y{:}\mathsf{int} = x + 1\ \mathsf{in}\ y + y$$

would not risk the duplication of the (possibly expensive) computation of M: the term $M+1$ would first be reduced to an integer value before the substitution in the body of the let takes place.

[10]These and other practical issues related to the art of compiler design have been successfully treated elsewhere; [58], for instace, discusses optimising translations for the Haskell language at length, and provides many practical examples.

It is clear that a more accurate analysis that would detect cases like the one shown would not only have to be able to distinguish between computations and values, but also know when computations are transformed into values—in other words, the reduction strategy chosen.

An elegant solution to this problem would consist in translating our ideas into a more general framework, like Moggi's computational calculus, and derive better analyses specifically tailored for particular reduction strategies by studying translations into this calculus. This is a matter of further work, as discussed in Subsection 8.2.2.

3.7.3 Sharing and single-threading

Barendsen and Smetsers considered a typing system of 'uniqueness types', which allows them to infer single-threaded uses of values [29]. Implementations can 'reuse' single-threaded cells by destructively updating their contents once used. Altering the contents of a single-threaded array, for instance, can be implemented more efficiently by destructively updating its contents in-place; it is not necessary to duplicate the array first, since we can be sure it is not shared.

As we remarked in the introduction, it is relatively well-known now that the notion of usage provided by linear logic does not correctly scale down to lower-level notions, like that of sharing values in a particular implementation of the reduction strategy of the calculus. Depending on the details of the implementation, 'used only once' may not necessarily imply 'not shared' [18, 61]. We stumbled upon this same semantical gap ourselves when designing an abstract machine for an intermediate language based on DILL [2]. The abstract machine handled intuitionistic and linear resources differently, using two separate environments and two separate sorts of substitution (linear and intuitionistic). Since it was formulated at a sufficiently higher-level of abstraction, it was easily proved correct with respect to the reduction rules of linear logic. We then considered the problem of implementing linear substitution using destructive updating of linear variables in-place. It did not took us long to find a simple first-order counter-example showing how linear variables would become 'indirectly' shared in our implementation. What we needed was a single-threaded analyser, but we were so eager to find an application for linear logic…

To motivate the nature of this mismatch in intuitive terms, suppose we have a function of type $(\sigma_1{}^1 \otimes \sigma_2{}^1)^\top \multimap \tau$. The annotations tell us already that the function may use its argument, which is a pair, an unknown number of times, possibly many. In other words, the function may share its argument. However, how this may be compatible with the linear annotations on the pair components? A pair is just an aggregate structure, so if the whole structure is shared, then each component must also be shared. The existence of such a type is already problematic according to this interpretation. Wadler [65], for example, already recognised this problem and corrected it by actually forbiding such types in typings, observing that the resulting system is probably too weak to be of any practical use. Indeed, his type system would give the function the weaker type $(\sigma_1{}^\top \otimes \sigma_2{}^\top)^\top \multimap \tau$.

A number of people have worked on annotated type systems based on ideas coming from linear logic, but the actual relationship with linear logic is only superficial: Their systems cannot be in general understood as a term-assignment system for linear logic, or some suitable fragment of it. A successful attempt at crafting a less conservative and, hence, more useful single-usage type system for the call-by-need Haskell language has been published in a series of papers [62, 68, 67]. The application intended was to avoid the updating linear closures in their graph-reduction implementation of Haskell. (Mogensen [45], proposed some refinements to the early analysis of Turner et al. [62], although he did not prove his analysis correct.)

Chapter 4

Annotation subtyping

In this chapter, we study an extension of linearity analysis with a notion of subsumption that is induced by a subtype relation between annotated types.

Subsumption allows an intuitionistic context to be used where a linear context is expected. In particular, because functions are instances of contexts, subsumption allows a linear function to be given a non-linear functional type, and therefore to be used in a context that expects a non-linear function. Formally, we write this fact

$$\sigma^1 \multimap \tau \leq \sigma^\top \multimap \tau,$$

for any two types σ and τ. Likewise, a context that expects a linear pair can be fed with a pair that is non-linear in one or both components.

Subsumption is important as it increases the expressive power of the type system. For instance, suppose that a context expects a function of type $\sigma^\top \multimap \tau$. Without subsumption, only functions having $\top$-annotated bound variables (i.e., of the form $\lambda x{:}\sigma^\top.M$) can be used in such a context. This *dependency* between the type of the context and that of the candidate function is alleviated with subsumption: The bound variable of a candidate function can retain its linear annotation, and still be given the (less precise) type $\sigma^\top \multimap \tau$, in order to conform to the type of the including context. Notice that inlining inspects the annotations on bound variables, so subtyping clearly opens the door for better optimised programs.

Annotation subtyping can be regarded as providing a partial solution to the 'poisoning problem', informally discussed in Subsection 1.4.1, as it allows terms of non-ground type to be assigned distinct annotated types. It also provides a simple criterion for assigning annotated types to definitions in modules, with the aim of augmenting the accuracy of the analysis across separately compiled modules. We shall discuss these two related problems in more detail in Section 5.1.

Many modern annotated type systems include a notion of subsumption in one way or the other. Annotation subtyping for usage type systems is an idea that seems to have sprung into existence only recently. A pratical example of a static analysis of affine properties including a notion of subsumption in the same spirit as the one we consider here may be found in [68].

4.0.4 Organisation

We have organised the contents on this chapter as follows:

- Section 4.1 considers NLL$^{\leq}$, our extension of NLL with subtyping. We also motivate the usefulness of the extensions by means of an example.

$$\frac{}{\mathbb{G} \leq \mathbb{G}}$$

$$\frac{\sigma_2 \leq \sigma_1 \quad \tau_1 \leq \tau_2 \quad a_1 \sqsubseteq a_2}{{\sigma_1}^{a_1} \multimap \tau_1 \leq {\sigma_2}^{a_2} \multimap \tau_2}$$

$$\frac{\sigma_1 \leq \sigma_2 \quad \tau_1 \leq \tau_2 \quad a_2 \sqsubseteq a_1 \quad b_2 \sqsubseteq b_1}{{\sigma_1}^{a_1} \otimes {\tau_1}^{b_1} \leq {\sigma_2}^{a_2} \otimes {\tau_2}^{b_2}}$$

Figure 4.1: Subtyping relation on types

- Section 4.2 shows that the extension is sound.
- Section 4.3 introduces $\mathrm{NLL}^{\mu\leq}$, a restriction of $\mathrm{NLL}^{\leq}$ to its minimum typings. This variant will play an important role when we consider type inference algorithms for linearity analysis with subtyping.
- Section 4.4 proves the semantic correctness of $\mathrm{NLL}^{\leq}$ with respect to reduction by stating the corresponding Substitution Lemma and Subject Reduction Theorem.

4.1 The Subsumption rule

Let $\mathrm{NLL}^{\leq}$ refer to the linear theory NLL extended with the following *subsumption* rule.

$$\frac{\Gamma \vdash M : \sigma \quad \sigma \leq \tau}{\Gamma \vdash M : \tau}\,\text{Subsumption}$$

Subsumption states that if a term has type σ, it also has supertype τ. A type σ is a *subtype* of type τ (and, conversely, τ is a *supertype* of type σ) if $\sigma \leq \tau$ is derivable using the inference rules of Figure 4.1.

The rules are standard. The subtyping relation is contravariant on function domains and covariant on function codomains, whereas, for pairs types, it is covariant on both component types. For annotations this situation is reversed: the relation is covariant on function domain annotations, whereas, for pair types, it is contravariant on both component type annotations.

From a type-theoretic viewpoint, the notion of subtyping we have just introduced is known as *shape conformant subtyping*, since the relation is invariant on the 'shape' of the underlying types[1].

Proposition 4.1.1 (Shape conformance)
If $\sigma \leq \tau$, then $\sigma^\circ = \tau^\circ$. □

[1] Gustavsson defines his analysis as an extension of an underlying type system able to accomodate recursive types, general subtyping as well as type-parametric polymorphism [35]. His extension is based on a formulation based on constrained types, which is ideal for type inference. The author thought such a formulation would obscure the presentation of structural analysis, and preferred a 'minimal' approach, so that the reader can better appreciate how the intermediate language relates to the source language using the notion of (structural) decoration.

The orientation of the relation on annotations may seem unnatural to the reader, but note that annotation subtyping derives from the inclusion of contexts, and so the relation appears reversed on annotations. As we have seen in the previous chapter, the inclusion of contexts appears in the formulation of the Transfer rule, which can be understood as a 'rudimentary' form of subtyping, taking place at the left of the turnstile[2].

4.1.1 Inlining revisited

If we look at the axioms of the inlining relation of Figure 3.10, it is easy to see that any inlining decisions ultimately depend on the annotations given to the bound variables. The binder $\mathsf{let}\ \langle x_1, x_2\rangle = M\ \mathsf{in}\ N$ does not explicitly carry any annotations on its bound variables: these seem unnecessary to define the inlining relation, as they can be deduced from the annotations of the matching pair, as in the axiom

$$\mathsf{let}\ \langle x_1, x_2\rangle = \langle M_1, M_2\rangle^{1,1}\ \mathsf{in}\ N \overset{\mathrm{inl}}{\rightsquigarrow} N[M_1/x_1, M_2/x_2].$$

The correctness of this observation derives from the $\otimes_{\mathcal{E}}$ rule. However, with subtyping, it is possible for the bound variables to have annotations in the typing context different from those of the matching pair, as the following example derivation shows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{-\vdash 0 : \mathsf{int}} \quad \dfrac{}{-\vdash 1 : \mathsf{int}}}{-\vdash \langle 0,1\rangle^{\top,\top} : \mathsf{int}^{\top} \otimes \mathsf{int}^{\top}}}{-\vdash \langle 0,1\rangle^{\top,\top} : \mathsf{int}^{1} \otimes \mathsf{int}^{\top}}\ \text{Subsumption} \qquad \dfrac{\dfrac{}{x_1 : \mathsf{int}^1 \vdash x_1 : \mathsf{int}}}{x_1 : \mathsf{int}^1, x_2 : \mathsf{int}^{\top} \vdash x_1 : \mathsf{int}}}{-\vdash \mathsf{let}\ \langle x_1, x_2\rangle = \langle 0,1\rangle^{\top,\top}\ \mathsf{in}\ x_1 : \mathsf{int}}\ \otimes_{\mathcal{E}}$$

The analysis clearly discovers that x_1 is used once inside the let, but this information is not reflected in the final annotated term. For this reason, we shall henceforth annotate all bound variables explicitly.

We first change the syntax of the unpairing construct as follows:

$$\begin{array}{rcll} M & ::= & \text{as defined in Section 3.2, except for} & \\ & & \mathsf{let}\ \langle x, x\rangle^{a,a} = M\ \mathsf{in}\ M & \text{Unpairing} \end{array}$$

The modified construct is typed in the obvious way, according to the following rule:

$$\dfrac{\Gamma_1 \vdash M : {\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2} \quad \Gamma_2, x_1 : {\sigma_1}^{a_1}, x_2 : {\sigma_2}^{a_2} \vdash N : \tau}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ \langle x_1, x_2\rangle^{a_1,a_2} = M\ \mathsf{in}\ N : \tau}\ \otimes_{\mathcal{E}}$$

The inlining relation is corrected accordingly to inspect the annotations of the bound variables, as shown in Figure 4.2.

[2]Wansbrough seems to have preferred the more 'natural' reading by letting $\top \sqsubseteq 1$ [66]. We prefer $1 \sqsubseteq \top$ as this is also the order suggested by the sub-decoration order relation we somehow inherited from early work on linear decorations.

$$(\lambda x{:}\sigma^1.M)N \overset{\text{inl}}{\rightsquigarrow} M[N/x]$$
$$\mathsf{let}\ \langle x_1, x_2\rangle^{1,1} = \langle M_1, M_2\rangle^{a,b}\ \mathsf{in}\ N \overset{\text{inl}}{\rightsquigarrow} N[M_1/x_1, M_2/x_2]$$
$$\mathsf{let}\ \langle x_1, x_2\rangle^{1,\top} = \langle M_1, M_2\rangle^{a,b}\ \mathsf{in}\ N \overset{\text{inl}}{\rightsquigarrow} \mathsf{let}\ x_2 = M_2\ \mathsf{in}\ N[M_1/x_1]$$
$$\mathsf{let}\ \langle x_1, x_2\rangle^{\top,1} = \langle M_1, M_2\rangle^{a,b}\ \mathsf{in}\ N \overset{\text{inl}}{\rightsquigarrow} \mathsf{let}\ x_1 = M_1\ \mathsf{in}\ N[M_2/x_2]$$
$$\mathsf{let}\ x{:}\sigma^1 = M\ \mathsf{in}\ N \overset{\text{inl}}{\rightsquigarrow} N[M/x]$$

Figure 4.2: The revised inlining relation

4.1.2 An illustrative example

To illustrate the use of subtyping, we shall compare two optimal analyses, with and without subtyping, of the following input program:

$$\begin{array}{l}\mathsf{let}\ \mathsf{p} = \langle 0, 1\rangle\ \text{in}\\ \quad \mathsf{let}\ \mathsf{fst} = \lambda x{:}\mathsf{int}\times\mathsf{int}.\mathsf{let}\ \langle x_1, x_2\rangle = x\ \mathsf{in}\ x_1\ \mathsf{in}\\ \quad\quad \mathsf{let}\ \mathsf{snd} = \lambda x{:}\mathsf{int}\times\mathsf{int}.\mathsf{let}\ \langle x_1, x_2\rangle = x\ \mathsf{in}\ x_2\ \mathsf{in}\\ \quad\quad\quad (\mathsf{fst}\ \mathsf{p}) + (\mathsf{snd}\ \mathsf{p})\end{array}$$

Without subsumption, we obtain the following optimal analysis:

$$\begin{array}{l}\mathsf{let}\ \mathsf{p}{:}\top = \langle 0, 1\rangle^{\top,\top}\ \mathsf{in}\\ \quad \mathsf{let}\ \mathsf{fst}{:}1 = \lambda x{:}(\mathsf{int}^\top \otimes \mathsf{int}^\top)^1.\mathsf{let}\ \langle x_1, x_2\rangle = x\ \mathsf{in}\ x_1\ \mathsf{in}\\ \quad\quad \mathsf{let}\ \mathsf{snd}{:}1 = \lambda x{:}(\mathsf{int}^\top \otimes \mathsf{int}^\top)^1.\mathsf{let}\ \langle x_1, x_2\rangle = x\ \mathsf{in}\ x_2\ \mathsf{in}\\ \quad\quad\quad (\mathsf{fst}\ \mathsf{p}) + (\mathsf{snd}\ \mathsf{p})\end{array}$$

(Once again, we have omitted any base type information for let-bound variables.) Notice that p has $\top$ as its annotation since it is used twice. The components of $\langle 0, 1\rangle$ must also be annotated with $\top$ since the first component is discarded in $(\mathsf{snd}\ \mathsf{p})$ and the second component is discarded in $(\mathsf{fst}\ \mathsf{p})$. The optimal typings for fst and snd are

$$\begin{array}{rcl}\mathsf{fst} & : & (\mathsf{int}^1 \otimes \mathsf{int}^\top)^1 \multimap \mathsf{int}\\ \mathsf{snd} & : & (\mathsf{int}^\top \otimes \mathsf{int}^1)^1 \multimap \mathsf{int}\end{array}$$

but since p must necessarily have type $\mathsf{int}^\top \otimes \mathsf{int}^\top$, then fst and snd must have domain types matching the type of p, that is $(\mathsf{int}^\top \otimes \mathsf{int}^\top)^1 \multimap \mathsf{int}$.

With subtyping, we can make use of the following relationships

$$\begin{array}{rcl}\mathsf{int}^\top \otimes \mathsf{int}^\top & \leq & \mathsf{int}^1 \otimes \mathsf{int}^\top\\ \mathsf{int}^\top \otimes \mathsf{int}^\top & \leq & \mathsf{int}^\top \otimes \mathsf{int}^1\end{array}$$

to obtain a more accurate analysis, as shown below.

$$\begin{array}{l}\mathsf{let}\ \mathsf{p}{:}\top = \langle 0,1\rangle^{\top,\top}\ \mathsf{in}\\ \quad \mathsf{let}\ \mathsf{fst}{:}1 = \lambda x{:}(\mathsf{int}^1 \otimes \mathsf{int}^\top)^1.\mathsf{let}\ \langle x_1,x_2\rangle^{1,\top} = x\ \mathsf{in}\ x_1\ \mathsf{in}\\ \qquad \mathsf{let}\ \mathsf{snd}{:}1 = \lambda x{:}(\mathsf{int}^\top \otimes \mathsf{int}^1)^1.\mathsf{let}\ \langle x_1,x_2\rangle^{\top,1} = x\ \mathsf{in}\ x_2\ \mathsf{in}\\ \qquad\quad (\mathsf{fst}\ \mathsf{p}) + (\mathsf{snd}\ \mathsf{p})\end{array}$$

The analysis is able to detect that x_1 is used once in the body of fst. This is reflected in the annotation of the pair pattern $\langle x_1, x_2\rangle^{1,\top}$. A similar remark applies to snd. Figure 4.3 shows two possible derivations of (fst p) that show how this becomes possible.

Notice that if we had decided to inline one occurrence of p, as in for instance $(\mathsf{fst}\ \langle 0,1\rangle^{\top,\top})$, the revised inlining relation would have allowed us to rewrite the expression completely. The last step is the more interesting one:

$$\mathsf{let}\ \langle x_1,x_2\rangle^{1,\top} = \langle 0,1\rangle^{\top,\top}\ \mathsf{in}\ x_1 \overset{\text{inl}}{\rightsquigarrow} 0.$$

4.1.3 Digression: context narrowing

We could have chosen an alternative presentation of $\text{NLL}^{\leq}$ where the Subsumption rule would have been replaced by the following Context Narrowing rule[3], that we introduce here as a property that will prove useful in the sequel.

Lemma 4.1.2 (Context Narrowing)
The following rule is admissible in $\text{NLL}^{\leq}$.

$$\frac{\Gamma, x : {\sigma_1}^a \vdash M : \tau \quad \sigma_2 \leq \sigma_1}{\Gamma, x : {\sigma_2}^a \vdash M : \tau}$$

Proof. Easy induction on the derivations of $\Gamma, x : {\sigma_1}^a \vdash M : \tau$. The key case is provided by a derivation consisting of a single application of the Identity rule:

$$\frac{}{x : {\sigma_1}^a \vdash x : \sigma_1}\ \text{Identity}$$

The conclusion follows from subsumption as shown by the following derivation:

$$\frac{\dfrac{}{x : {\sigma_2}^a \vdash x : \sigma_2}\ \text{Identity} \qquad \sigma_2 \leq \sigma_1}{x : {\sigma_2}^a \vdash x : \sigma_1}\ \text{Subsumption}$$

□

[3]Some authors prefer the name 'Bound Weakening' for this rule. The name comes from the duality that exists between the Context Narrowing rule and the Subsumption rule, which has the effect of 'widening' the type of the premise.

Derivation 1:

$$
\dfrac{
\dfrac{}{\mathsf{fst} : ((\mathsf{int}^1 \otimes \mathsf{int}^\top)^1 \multimap \mathsf{int})^1 \vdash \mathsf{fst} : (\mathsf{int}^1 \otimes \mathsf{int}^\top)^1 \multimap \mathsf{int}}\,\text{Identity}
\qquad
\dfrac{\dfrac{}{\mathsf{p} : (\mathsf{int}^\top \otimes \mathsf{int}^\top)^\top \vdash \mathsf{p} : \mathsf{int}^\top \otimes \mathsf{int}^\top}\,\text{Identity}}{\mathsf{p} : (\mathsf{int}^\top \otimes \mathsf{int}^\top)^\top \vdash \mathsf{p} : \mathsf{int}^1 \otimes \mathsf{int}^\top}\,\text{Subsumption}
}{
\mathsf{fst} : ((\mathsf{int}^1 \otimes \mathsf{int}^\top)^1 \multimap \mathsf{int})^1, \mathsf{p} : (\mathsf{int}^\top \otimes \mathsf{int}^\top)^\top \vdash \mathsf{fst}\ \mathsf{p} : \mathsf{int}
}\,\multimap_{\mathcal{E}}
$$

Derivation 2:

$$
\dfrac{
\dfrac{\dfrac{}{\mathsf{fst} : ((\mathsf{int}^1 \otimes \mathsf{int}^\top)^1 \multimap \mathsf{int})^1 \vdash \mathsf{fst} : (\mathsf{int}^1 \otimes \mathsf{int}^\top)^1 \multimap \mathsf{int}}\,\text{Identity}}{\mathsf{fst} : ((\mathsf{int}^1 \otimes \mathsf{int}^\top)^1 \multimap \mathsf{int})^1 \vdash \mathsf{fst} : (\mathsf{int}^\top \otimes \mathsf{int}^\top)^1 \multimap \mathsf{int}}\,\text{Subsumption}
\qquad
\dfrac{}{\mathsf{p} : (\mathsf{int}^\top \otimes \mathsf{int}^\top)^\top \vdash \mathsf{p} : \mathsf{int}^\top \otimes \mathsf{int}^\top}\,\text{Identity}
}{
\mathsf{fst} : ((\mathsf{int}^1 \otimes \mathsf{int}^\top)^1 \multimap \mathsf{int})^1, \mathsf{p} : (\mathsf{int}^\top \otimes \mathsf{int}^\top)^\top \vdash \mathsf{fst}\ \mathsf{p} : \mathsf{int}
}\,\multimap_{\mathcal{E}}
$$

Figure 4.3: Optimal decoration for (fst p)

4.2 Soundness

A first obvious observation is that NLL$^{\leq}$ is a conservative extension of NLL.

Proposition 4.2.1 (Conservativity)
$\Gamma \underset{\text{NLL}}{\vdash} M : \sigma$ implies $\Gamma \underset{\text{NLL}^{\leq}}{\vdash} M : \sigma$. □

To prove the correctness of the extended type system with subtyping, we shall take the standard approach and, at the end of the chapter, prove a Substitution Lemma. In this section, we shall briefly motivate the correctness of the theory in a different way, by providing a translation from NLL$^{\leq}$ terms into NLL terms, and showing that this translation is invariant with respect to reduction[4].

We begin by giving an operational interpretation of subtyping as a 'retyping' or coercion function

$$[\![\sigma \leq \tau]\!] : \sigma^1 \multimap \tau,$$

mapping terms of type σ into terms of type τ, for every σ that is a subtype of τ[5]. This retyping function is easily defined by induction on the definition of the subtype relation, as follows:

$$\begin{aligned}
[\![\mathbb{G} \leq \mathbb{G}]\!] &\stackrel{\text{def}}{=} \lambda x{:}\mathbb{G}^1.x \\
[\![{\sigma_1}^{a_1} \multimap \tau_1 \leq {\sigma_2}^{a_2} \multimap \tau_2]\!] &\stackrel{\text{def}}{=} \lambda f{:}({\sigma_1}^{a_1} \multimap \tau_1)^1. \\
&\qquad \lambda x{:}{\sigma_2}^{a_2}.[\![\tau_1 \leq \tau_2]\!]\ (f\ ([\![\sigma_2 \leq \sigma_1]\!]\ x)) \\
[\![{\sigma_1}^{a_1} \otimes {\tau_1}^{b_1} \leq {\sigma_2}^{a_2} \otimes {\tau_2}^{b_2}]\!] &\stackrel{\text{def}}{=} \lambda x{:}({\sigma_1}^{a_1} \otimes {\tau_1}^{b_1})^1.\mathsf{let}\ \langle x_1, x_2\rangle^{a_1,b_1} = x\ \mathsf{in} \\
&\qquad \langle [\![\sigma_1 \leq \sigma_2]\!]\ x_1, [\![\tau_1 \leq \tau_2]\!]\ x_2\rangle^{a_2,b_2}
\end{aligned}$$

The following two propositions state, respectively, that the coercion function has the expected type, and that its erasure behaves like the identity on source language terms of the appropriate type. (We note that if $\sigma \leq \tau$, then $\sigma^\circ \equiv \tau^\circ$, so $[\![\sigma \leq \tau]\!]^\circ$ has at least the type of the identity.)

Proposition 4.2.2
If $\Gamma \underset{\text{NLL}}{\vdash} M : \sigma$ and $\sigma \leq \tau$ for some τ, then $\Gamma \underset{\text{NLL}}{\vdash} [\![\sigma \leq \tau]\!]\ M : \tau$.

Proof. It suffices to check that $- \vdash [\![\sigma \leq \tau]\!] : \sigma^1 \multimap \tau$. The proposition follows by a simple application of $\multimap_{\mathcal{E}}$. □

Proposition 4.2.3
If $\Gamma \underset{\text{NLL}}{\vdash} M : \sigma$, then $[\![\sigma \leq \tau]\!]^\circ\ M^\circ \twoheadrightarrow M^\circ$ for any τ. □

Using the above coercion function, we now provide a translation $[\![-]\!]$ mapping NLL$^{\leq}$ type derivations into NLL type derivations.

We define $[\![\Pi(\Gamma \vdash M : \sigma)]\!]$ by induction on the structure of Π. We recursively translate subderivations, replacing the subterms in the conclusion of the translation by the corresponding subterms appearing in the conclusion of the subderivations just translated. The only

[4]This translation could be the basis of a straightforward semantics of NLL$^{\leq}$ in terms of DILL.
[5]We could have equally chosen $\sigma^\top \multimap \tau$ to be the type of the retyping function.

$$\frac{\Gamma_1 \vdash M : {\sigma_1}^a \multimap \tau \quad \Gamma_2 \vdash N : \sigma_2 \quad \sigma_2 \leq \sigma_1 \quad |\Gamma_2| \sqsupseteq a}{\Gamma_1, \Gamma_2 \vdash MN : \tau} \multimap_{\mathcal{E}}$$

$$\frac{\Gamma_1 \vdash M : {\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2} \quad \Gamma_2, x_1 : {\tau_1}^{b_1}, x_2 : {\tau_2}^{b_2} \vdash N : \tau \quad a_i \sqsupseteq b_i \quad \sigma_i \leq \tau_i \quad (i = 1, 2)}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ \langle x_1, x_2 \rangle^{b_1, b_2} = M\ \mathsf{in}\ N : \tau} \otimes_{\mathcal{E}}$$

$$\frac{\Gamma_1 \vdash M : \mathsf{bool} \quad \Gamma_2 \vdash N_1 : \sigma_1 \quad \Gamma_2 \vdash N_2 : \sigma_2}{\Gamma_1, \Gamma_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \sigma_1 \sqcup \sigma_2} \text{Conditional}$$

Figure 4.4: Modified rules for $\text{NLL}^{\mu\leq}$

interesting case is the translation of a derivation ending in an application of the Subsumption rule:

$$\left[\!\!\left[\frac{\Pi(\Gamma \vdash M : \sigma)}{\Gamma \vdash M : \tau}\right]\!\!\right] = \frac{\Pi(\Gamma \vdash M' : \sigma)}{\Gamma \vdash [\![\sigma \leq \tau]\!]\ M' : \tau} \tag{4.1}$$

where $[\![\Pi(\Gamma \vdash M : \sigma)]\!] = \Pi(\Gamma \vdash M' : \sigma)$.

Notice that a $\text{NLL}^{\leq}$ term M may have different possible translations, corresponding to the different possibilities that exist of applying the Subsumption rule in their associated type derivations. So, let $[\![M]\!]_\Gamma$ stand for any translation M' of a term M typeable in context Γ:

$$[\![M]\!]_\Gamma \stackrel{\text{def}}{=} M' \quad \text{if } [\![\Pi(\Gamma \underset{\text{NLL}^{\leq}}{\vdash} M : \sigma)]\!] = \Gamma \underset{\text{NLL}}{\vdash} M' : \sigma,$$

for some type derivation Π.

The following statement can be easily proved by induction and Proposition 4.2.3.

Proposition 4.2.4 (Soundness)
For suitable Γ and M, if $[\![M]\!]_\Gamma = M'$, then $M'^{\circ} \twoheadrightarrow M^{\circ}$. □

4.3 Minimum typing

The Unique Typing property is obviously not verified in the presence of subtyping. However, a related Minimum Typing property can be proved for this system. This property states that every term of $\text{NLL}^{\leq}$ that has a type, has also a minimum type.

This property is important as it provides a criterion for choosing among the set of enriched types available for a term. As a matter of fact, we shall consider a subset of $\text{NLL}^{\leq}$, that we call $\text{NLL}^{\mu\leq}$, having unique types and such that if a term has a type in this system, it has the same type in $\text{NLL}^{\leq}$ and, moreover, it is the smallest such type. The type system $\text{NLL}^{\mu\leq}$ is obtained from $\text{NLL}^{\leq}$ by dropping the subsumption rule and replacing the elimination rules with the rules shown in Figure 4.4[6].

[6]Following the notation adopted for constraint sets in Chapter 6, any restrictions on annotations will be written as inequations of the form $a \sqsupseteq b$.

The notation $\sigma_1 \sqcup \sigma_2$, used in the Conditional rule, stands for the join of σ_1 and σ_2 with respect to the subtyping order. Notice that in the $\multimap_{\mathcal{E}}$ rule, the conditions $a_i \sqsupseteq b_i$ and $\sigma_i \leq \tau_i$ (for $i = 1, 2$) imply $\sigma_1^{a_1} \otimes \sigma_2^{a_2} \leq \tau_1^{b_1} \otimes {\tau_2}^{b_2}$.

The following three lemmas prove some basic results about NLL$^{\mu\leq}$. We begin by showing that NLL$^{\mu\leq}$ typings are also NLL$^{\leq}$ typings.

Lemma 4.3.1
If $\Gamma \underset{\text{NLL}^{\mu\leq}}{\vdash} M : \sigma$, *then* $\Gamma \underset{\text{NLL}^{\leq}}{\vdash} M : \sigma$.

Proof. It is straightforward to show that the modified rules of NLL$^{\mu\leq}$ are derivable in NLL$^{\leq}$. We show the derivations for the elimination rules below.

- For the $\multimap_{\mathcal{E}}$ rule:

$$\dfrac{\Gamma_1 \vdash M : {\sigma_1}^a \multimap \tau \qquad \dfrac{\Gamma_2 \vdash N : \sigma_2}{\Gamma_2 \vdash N : \sigma_1}\text{Subsumption}}{\Gamma_1, \Gamma_2 \vdash MN : \tau}$$

- For the $\otimes_{\mathcal{E}}$ rule:

$$\dfrac{\dfrac{\Gamma_1 \vdash M : {\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2}}{\Gamma_1 \vdash M : {\tau_1}^{b_1} \otimes {\tau_2}^{b_2}}\text{Subsumption} \qquad \Gamma_2, x_1 : {\tau_1}^{b_1}, x_2 : {\tau_2}^{b_2} \vdash N : \tau}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ \langle x_1, x_2\rangle^{b_1, b_2} = M\ \mathsf{in}\ N : \tau}$$

- For the Conditional rule:

$$\dfrac{\Gamma_1 \vdash M : \mathsf{bool} \qquad \dfrac{\Gamma_2 \vdash N_1 : \sigma_1}{\Gamma_2 \vdash N_1 : \sigma_1 \sqcup \sigma_2}\text{Subsumption} \qquad \dfrac{\Gamma_2 \vdash N_1 : \sigma_2}{\Gamma_2 \vdash N_1 : \sigma_1 \sqcup \sigma_2}\text{Subsumption}}{\Gamma_1, \Gamma_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \sigma_1 \sqcup \sigma_2}$$

□

The remaining two lemmas state that typings in NLL$^{\mu\leq}$ are unique and smaller, respectively, than typings in NLL$^{\leq}$.

Lemma 4.3.2 (Unique Typing)
If $\Gamma \underset{\text{NLL}^{\mu\leq}}{\vdash} M : \sigma$ *and* $\Gamma \underset{\text{NLL}^{\mu\leq}}{\vdash} M : \tau$, *then* $\sigma \equiv \tau$.

Proof. Easy induction on the derivations of $\Gamma \vdash M : \sigma$. □

Lemma 4.3.3 (Smaller Typing)
If $\Gamma \underset{\text{NLL}^{\leq}}{\vdash} M : \sigma$, *then* $\Gamma \underset{\text{NLL}^{\mu\leq}}{\vdash} M : \tau$ *for some* $\tau \leq \sigma$.

Proof. We proceed by induction on NLL$^{\leq}$ derivations of $\Gamma \vdash M : \sigma$. Only the key cases are shown.

- $\dfrac{}{x : \sigma^a \vdash x : \sigma}$
  This case is obvious; we just let $\tau \equiv \sigma$.

- $$\frac{\Gamma, x : \sigma^a \vdash M : \tau_1}{\Gamma \vdash \lambda x{:}\sigma^a.M : \sigma^a \multimap \tau_1}$$

  By the induction hypothesis, we have that $\Gamma, x : \sigma^a \vdash M : \tau_0$ is derivable for some $\tau_0 \leq \tau_1$. Therefore, applying the $\multimap_{\mathcal{I}}$ rule, we can obtain $\lambda x{:}\sigma^a.M : \sigma^a \multimap \tau_0$; and, because $\leq$ is covariant on function codomains, we have $\sigma^a \multimap \tau_0 \leq \sigma^a \multimap \tau_1$, as expected.

- $$\frac{\Gamma_1 \vdash M : {\sigma_1}^{a_1} \multimap \tau_1 \quad \Gamma_2 \vdash N : \sigma_1}{\Gamma_1, \Gamma_2 \vdash MN : \tau_1}$$

  Applying the induction hypothesis twice, we obtain $\Gamma_1 \vdash M : {\sigma_0}^{a_0} \multimap \tau_0$ and $\Gamma_2 \vdash N : \sigma_0'$, with ${\sigma_0}^{a_0} \multimap \tau_0 \leq {\sigma_1}^{a_1} \multimap \tau_1$ and $\sigma_0' \leq \sigma_1$. Because subtyping is contravariant on function domains, we have that $\sigma_1 \leq \sigma_0$, and hence $\sigma_0' \leq \sigma_0$. Also, since it must be the case that $|\Gamma_2| \sqsupseteq a_1$, from $a_1 \sqsupseteq a_0$, we deduce that $|\Gamma_2| \sqsupseteq a_0$. Therefore, we can apply the $\multimap_{\mathcal{E}}$ rule to conclude $\Gamma_1, \Gamma_2 \vdash MN : \tau_0$, and $\tau_0 \leq \tau_1$.

- $$\frac{\Gamma_1 \vdash M : \mathsf{bool} \quad \Gamma_2 \vdash N_1 : \sigma_1 \quad \Gamma_2 \vdash N_2 : \sigma_1}{\Gamma_1, \Gamma_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \sigma_1}$$

  By the induction hypothesis, twice, we have that $\Gamma_2 \vdash N_1 : \sigma_0'$ and $\Gamma_2 \vdash N_2 : \sigma_0''$, with $\sigma_0' \leq \sigma_1$ and $\sigma_0'' \leq \sigma_1$. We can therefore apply the Conditional rule to conclude $\Gamma_1, \Gamma_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \sigma_0' \sqcup \sigma_0''$, and $\sigma_0' \sqcup \sigma_0'' \leq \sigma_1$ by definition.

- $$\frac{\Gamma, x : {\sigma_1}^{\top} \vdash M : \sigma_1}{\Gamma \vdash \mathsf{fix}\ x{:}\sigma_1.M : \sigma_1}$$

  Applying the induction hypothesis, we obtain $\Gamma, x : {\sigma_1}^{\top} \vdash M : \sigma_0$ with $\sigma_0 \leq \sigma_1$. By Context Narrowing (Lemma 4.1.2), we have that $\Gamma, x : {\sigma_0}^{\top} \vdash M : \sigma_0$. We can therefore apply the Fixpoint rule and conclude $\Gamma \vdash \mathsf{fix}\ x{:}\sigma_0.M : \sigma_0$.

- $$\frac{\Gamma \vdash M : \sigma_1}{\Gamma \vdash M : \tau_1}$$

  By the induction hypothesis, we know that $\Gamma \vdash M : \sigma_0$ for some $\sigma_0 \leq \sigma_1$. Since $\sigma_1 \leq \tau_1$ by Subsumption, we may conclude $\sigma_0 \leq \tau_1$, as desired.

□

Using these lemmas, we are now ready to prove the following Minimum Typing property for $\mathrm{NLL}^{\leq}$.

Theorem 4.3.4 (Minimum Typing)
If $\Gamma \underset{\mathrm{NLL}^{\leq}}{\vdash} M : \sigma$, then there exists τ such that $\Gamma \underset{\mathrm{NLL}^{\leq}}{\vdash} M : \tau$, and, for every other σ' for which $\Gamma \underset{\mathrm{NLL}^{\leq}}{\vdash} M : \sigma'$, then $\tau \leq \sigma'$.

Proof. Suppose that $\Gamma \vdash M : \sigma$ in $\mathrm{NLL}^{\leq}$. By Lemma 4.3.3, we know that $\Gamma \vdash M : \tau$ for some $\tau \leq \sigma$ is derivable in $\mathrm{NLL}^{\mu\leq}$. From Lemma 4.3.1, it follows that $\Gamma \vdash M : \tau$ must also derivable in $\mathrm{NLL}^{\leq}$. Again, by Lemma 4.3.3, if $\Gamma \vdash M : \sigma'$ is derivable in $\mathrm{NLL}^{\leq}$, then $\Gamma \vdash M : \tau'$ is derivable in $\mathrm{NLL}^{\mu\leq}$ with $\tau' \leq \sigma'$. By Lemma 4.3.2, we must have that $\tau \equiv \tau'$, and hence $\tau \leq \sigma'$. □

$$\frac{|\Gamma| \sqsupseteq \top}{\Gamma, x : \sigma^a \vdash x : \sigma}\ \text{Identity} \qquad \frac{|\Gamma| \sqsupseteq \top \quad \Sigma(\pi) = \sigma}{\Gamma \vdash \pi : \sigma}\ \text{Primitive}$$

$$\frac{\Gamma, x : \sigma^a \vdash M : \tau}{\Gamma \vdash \lambda x{:}\sigma^a.M : \sigma^a \multimap \tau} \multimap_{\mathcal{I}}$$

$$\frac{\Gamma_1 \vdash M : {\sigma_1}^a \multimap \tau \quad \Gamma_2 \vdash N : \sigma_2 \quad \sigma_2 \leq \sigma_1 \quad |\Gamma_2| \sqsupseteq a}{\Gamma_1 \uplus \Gamma_2 \vdash MN : \tau} \multimap_{\mathcal{E}}$$

$$\frac{\Gamma_1 \vdash M_1 : \sigma_1 \quad \Gamma_2 \vdash M_2 : \sigma_2 \quad |\Gamma_1| \sqsupseteq a_1 \quad |\Gamma_2| \sqsupseteq a_2}{\Gamma_1 \uplus \Gamma_2 \vdash \langle M_1, M_2 \rangle^{a_1,a_2} : {\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2}} \otimes_{\mathcal{I}}$$

$$\frac{\Gamma_1 \vdash M : {\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2} \quad \Gamma_2, x_1 : {\tau_1}^{b_1}, x_2 : {\tau_2}^{b_2} \vdash N : \tau \quad a_i \sqsupseteq b_i \quad \sigma_i \leq \tau_i \quad (i = 1, 2)}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{let}\ \langle x_1, x_2 \rangle^{b_1,b_2} = M\ \mathsf{in}\ N : \tau} \otimes_{\mathcal{E}}$$

$$\frac{\Gamma_1 \vdash M : \mathsf{bool} \quad \Gamma_2 \vdash N_1 : \sigma_1 \quad \Gamma_2 \vdash N_2 : \sigma_2}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \sigma_1 \sqcup \sigma_2}\ \text{Conditional}$$

$$\frac{\Gamma, x : \sigma^{\top} \vdash M : \sigma \quad |\Gamma| \sqsupseteq \top}{\Gamma \vdash \mathsf{fix}\ x{:}\sigma.M : \sigma}\ \text{Fixpoint}$$

Figure 4.5: The typing rules of $\mathrm{NLL}^{\mu\leq\uplus}$

The system $\mathrm{NLL}^{\mu\leq}$ will be the basis of the annotation inference algorithm studied in the following chapter. Actually, the development of the following chapter is directly based on a syntax-directed version of it. Figure 4.5 summarises the typing rules of this system, that we call $\mathrm{NLL}^{\mu\leq\uplus}$. Lemmas 3.4.3 and 3.4.5 provide the template proofs for the equivalence to $\mathrm{NLL}^{\mu\leq}$.

4.4 Semantic correctness

Like NLL, typings in $\mathrm{NLL}^{\leq}$ are preserved by the reduction rules. It is easier to state the corresponding Substitution Lemma and Subject Reduction Theorem for $\mathrm{NLL}^{\mu\leq}$ first. For $\mathrm{NLL}^{\leq}$, these properties follow as corollaries, as we shall soon explain.

Lemma 4.4.1 (Substitution for $\mathrm{NLL}^{\mu\leq}$**)**
The following rule is admissible.

$$\frac{\Gamma_1, x : {\sigma_1}^a \vdash M : \tau \quad \Gamma_2 \vdash N : \sigma_2 \quad |\Gamma_2| \sqsupseteq a \quad \sigma_2 \leq \sigma_1}{\Gamma_1, \Gamma_2 \vdash M[N/x] : \tau}\ \text{Substitution}$$

Proof. Basically, a trivial modification of Lemma 3.5.6. □

Theorem 4.4.2 (Subject Reduction for $\mathrm{NLL}^{\mu\leq}$)
If $\Gamma \underset{\mathrm{NLL}^{\mu\leq}}{\vdash} M : \sigma$ *and* $M \to N$*, then* $\Gamma \underset{\mathrm{NLL}^{\mu\leq}}{\vdash} N : \sigma$.

Proof. Basically, a trivial modification of Theorem 3.5.7. □

Theorem 4.4.3 (Subject Reduction for $\mathrm{NLL}^{\leq}$)
If $\Gamma \underset{\mathrm{NLL}^{\leq}}{\vdash} M : \sigma$ *and* $M \to N$*, then* $\Gamma \underset{\mathrm{NLL}^{\leq}}{\vdash} N : \sigma$.

Proof. Assume $\Gamma \vdash M : \sigma$ holds in $\mathrm{NLL}^{\leq}$. By Lemma 4.3.3, we know $\Gamma \vdash M : \tau$ holds in $\mathrm{NLL}^{\mu\leq}$ for some $\tau \leq \sigma$. Assuming $M \to N$, by Subject Reduction for $\mathrm{NLL}^{\mu\leq}$ and Lemma 4.3.1, we know $\Gamma \vdash N : \tau$ must be the case in $\mathrm{NLL}^{\mu\leq}$. The required conclusion $\Gamma \vdash N : \sigma$ follows by Subsumption. □

We can use a similar argument to prove the admissibility of the Substitution for $\mathrm{NLL}^{\leq}$.

4.4.1 Subject reduction for η-reduction

As we argued in Subsection 3.5.5, extending our notion of reduction with the η-reduction axiom

$$\lambda x{:}\sigma^a.M\ x \to M \quad \text{if } x \notin FV(M) \tag{η}$$

compromises NLL's Subject Reduction property. Fortunately, this property can be recovered for η-reduction for our linear type theory with subtyping, as stated by the following proposition.

Proposition 4.4.4 (Subject Reduction for η)
If $\Gamma \underset{\mathrm{NLL}^{\leq}}{\vdash} \lambda x{:}\sigma^a.M\ x : \sigma^a \multimap \tau$ *with* $x \notin FV(M)$*, then* $\Gamma \underset{\mathrm{NLL}^{\leq}}{\vdash} M : \sigma^a \multimap \tau$.

Proof. A derivation for the left-hand side of the implication must be as shown:

$$\dfrac{\dfrac{\Pi(\Gamma \vdash M : \sigma^b \multimap \tau) \qquad \dfrac{}{x : \sigma^a \vdash x : \sigma}\text{Identity} \qquad a \sqsupseteq b}{\Gamma, x : \sigma^a \vdash M\ x : \tau}\multimap_{\mathcal{E}}}{\Gamma \vdash \lambda x{:}\sigma^a.M\ x : \sigma^a \multimap \tau}\multimap_{\mathcal{I}}$$

Clearly, the critical case is when a and b have distinct values. In this case, subsumption is needed to obtain the required type:

$$\dfrac{\Pi(\Gamma \vdash M : \sigma^b \multimap \tau) \quad \sigma^b \multimap \tau \leq \sigma^a \multimap \tau}{\Gamma \vdash M : \sigma^a \multimap \tau}\text{Subsumption}$$

□

Chapter 5

Annotation polymorphism

In the previous chapter, we have looked at an extension of linearity analysis with a notion of subsumption over annotated types. As we have observed, the resulting analysis is more expressive from a static analysis viewpoint, as it allows terms of non-ground type to be assigned many distinct annotated types. The annotations in terms are not required to match the annotations in types precisely. Indeed, the usage of subsumption implies that the annotations of bound variables are necessarily more accurate. An interesting question is whether this 'degree of independence' gained can be carried farther.

In this chapter, we extend the analysis of our previous chapter with *general annotation polymorphism*. Roughly speaking, with general annotation polymorphism, a term cannot only be assigned the types in its subtyping family, but also all the types in its decoration family[1]. What this suggests is that the analysis of a term and the analysis of the contexts where that term is used can be approached separately. This was not possible with our previous versions of linearity analysis, because of the strong interplay between the annotations of a term and its uses.

Annotation polymorphism provides a satisfactory solution to the 'poisoning problem', informally discussed in Subsection 1.4.1. As we pointed out in the introduction, this is nothing more than the consequence of the fact that linearity analysis, as described so far, is monomorphic on annotations.

The main motivation for having annotation polymorphism is to serve as basis for the *accurate* static analysis of linearity properties across module boundaries, so we shall begin by looking at this problem in more detail.

Our approach is more general compared to other similar systems, in the sense that we are interested in having general rules for introducing and eliminating quantified types, and not just specific rules that match our type inference algorithm. In the following chapter, we shall derive a type inference algorithm that assigns quantified types to definitions as a restriction of the more general system we introduce here.

5.0.2 Organisation

The contents of this chapter are organised as follows:

[1] We use the term 'general annotation polymorphism' instead of simply 'annotation polymorphism', as restricted versions of general annotation polymorphism exist in the literature. An example is Wansbrough's simple usage polymorphism [67].

- Section 5.1 explains in more detail why annotation polymorphism is necessary for those languages supporting separately compiled modules.
- Section 5.2 introduces the syntax and typing rules of $\mathrm{NLL}^{\forall}$, our version of NLL with annotation polymorphism.
- Section 5.3 introduces $\mathrm{NLL}^{\forall\leq}$, a system that mixes both annotation subtyping and annotation polymorphism.
- Section 5.4 lists some type-theoretic properties of $\mathrm{NLL}^{\forall\leq}$ and establishes its semantical correctness.
- Section 5.5 introduces $\mathrm{NLL}^{\forall\mathrm{let}\leq}$, a subset of $\mathrm{NLL}^{\forall\leq}$ that restricts annotation polymorphism to let-definitions only. This system will provide the minimal setting needed to discuss modular linearity analysis.
- Section 5.6 argues that annotation polymorphism is powerful enough to emulate subtyping.
- Section 5.7 finally shows the semantic correctness of an extended version of $\mathrm{NLL}^{\forall\leq}$ that includes type-parametric polymorphism in the style of System F.

5.1 Separate compilation and optimality

Terms in modular languages may contain free variables that refer to definitions in either the same module, or in separately compiled (external) modules, and for which only the types are known at compilation time. When compiling a program, the bodies of any external definitions used are usually not available to the static analyser, so unless the properties inferred for these definitions are saved, the static analyser has no other possibility other than assuming the worst. 'Assuming the worst' refers here to adopting the worst decoration for the type of an external definition as the only safe strategy to fill in the missing information. Formally, if σ is the type of an external definition bound to the identifier x, and $M[x]$ is a term containing x, the static analyser must assume $x : \sigma^{\bullet}$ in the analysis of M. Without any knowledge on the structure of the definition bound to x, any other structural assumptions would necessarily be unsound. The result is an analysis that has degraded to the point of uselessness.

It seems then that saving the inferred properties of definitions (in the module interface, for instance[2]) is compulsory. However, saving precomputed optimal types, or any other type for that matter, does not work. A trivial counter-example is all that is needed to motivate the problem.

Assume there is a module containing the following simple definition:

$$\mathsf{let\ origin} = \langle 0, 0 \rangle.$$

(We suppose that modules are simply lists of bindings, associating a variable name to a term. We leave any details for later.)

The optimal decoration for such a definition is

$$- \vdash \langle 0, 0 \rangle^{1,1} : \mathsf{int}^1 \otimes \mathsf{int}^1.$$

[2]Note that this means that client modules must be recompiled if the annotated type of the module has changed with respect to the annotated type in the interface, even if the underlying types remain the same.

Now, suppose the compiler comes across the expression

$$\mathsf{let}\ \langle x_1, x_2\rangle = \mathsf{origin}\ \mathsf{in}\ x_1.$$

The variable origin occurs in a context of (at best) type $\mathsf{int}^1 \otimes \mathsf{int}^\top$ (because x_2 is discarded in the body of the let). However, this type is incompatible with the optimal type $\mathsf{int}^1 \otimes \mathsf{int}^1$ precomputed for origin. Notice that subsumption cannot help to alleviate the problem, since

$$\mathsf{int}^1 \otimes \mathsf{int}^\top \leq \mathsf{int}^1 \otimes \mathsf{int}^1,$$

not the reverse. The static analyser has then got stuck.

As the example above shows, the optimal property of a definition may be too restrictive, so it cannot be generally used in practice. The problem here is that we do not have any contextual information regarding the use of a definition, at least not at compilation time. An accurate analysis certainly depends on the availability of this information. (We assume that we are not interested in deferring compilation until the whole application has been assembled, so we are inevitably left at a point where some important information is missing.)

We should remark that accuracy is not always compromised. Suppose our compiler takes a safe decision and decorates origin as follows:

$$- \vdash \langle 0, 0\rangle^{\top,\top} : \mathsf{int}^\top \otimes \mathsf{int}^\top.$$

Thanks to subsumption, origin could be used in a context of type $\mathsf{int}^1 \otimes \mathsf{int}^1$, possibly allowing some interesting inlining optimisations to take place.

Instead of the optimal property of a definition, what we should be really be looking for is the property that is sufficiently general as to not compromise typeability, and sufficiently precise as to not compromise accuracy. Notice that the worst decoration for origin works because of subsumption, but this is generally not the case if we consider other examples where higher-order functions are involved[3]. (Notice that for first-order languages, both the decoration and subtyping families coincide, so the strategy that assigns the smallest type in the subtyping family is enough to ensure typeability.)

As another example, consider the following decorated module definition:

$$\mathsf{let\ apply} = \lambda f{:}(\mathsf{int}^\top \multimap \mathsf{int})^1.\lambda x{:}\mathsf{int}^\top.f\ x.$$

We clearly cannot assign f the more accurate type $\mathsf{int}^1 \multimap \mathsf{int}$, because it would incorrectly constrain the applications of apply to only linear functions. But we have lost the information necessary to fully inline programs like

$$\begin{array}{l}\mathsf{let\ inc} = \lambda y{:}\mathsf{int}^1.y + 1\ \mathrm{in}\\ \quad \mathsf{apply\ inc}\ 3\end{array}$$

where apply is used in the context of a linear function. Indeed, after two steps of inlining (and renaming of bound variables), we are left with

$$(\lambda x{:}\mathsf{int}^\top.(\lambda y{:}\mathsf{int}^1.y)\ x)\ 3.$$

[3]The Substitution Lemma of our simple linearity analysis (Lemma 3.5.6) states that the types of both the context hole and the substituted term have to be equivalent, so the only gain must necessarily come from the side of the Subsumption rule, that relaxes this restriction by imposing that the types should be in the subtype relation.

Even if x is morally linear, as witnessed by the annotation of y, inlining cannot proceed because we have been forced to give x an annotation that is compatible with that given to the domain of f. The same loss of accuracy would be observed if apply appeared as a local definition, but used in different contexts:

$$\begin{array}{l} \mathsf{let\ apply} = \lambda f{:}(\mathsf{int}^\top \multimap \mathsf{int})^\top.\lambda x{:}\mathsf{int}^\top.f\ x\ \mathsf{in} \\ \quad \mathsf{let\ inc} = \lambda x{:}\mathsf{int}^1.x\ \mathrm{in} \\ \quad\quad \mathsf{let\ dup} = \lambda x{:}\mathsf{int}^\top.x + x\ \mathrm{in} \\ \quad\quad\quad \mathsf{apply\ inc\ (apply\ dup\ }4). \end{array}$$

Here, apply is used in both a linear and an intuitionistic context, and because the type system of linearity analysis does not allow apply to have more than one type, we must content ourselves with assigning to its definition the weakest of the types. A solution to this problem might consist in adding intersection types to the type system of linearity analysis, thus allowing apply to be assigned 'simultaneously' the two types

$$(\mathsf{int}^\top \multimap \mathsf{int})^\top \multimap \mathsf{int}^\top \multimap \mathsf{int} \quad \text{and} \quad (\mathsf{int}^1 \multimap \mathsf{int})^\top \multimap \mathsf{int}^1 \multimap \mathsf{int}.$$

However, this would hardly help in providing a solution to the problem we started with, for which no contextual information is available.

Annotation polymorphism provides a more satisfactory solution to modular static analysis, as we see next. It would allow, for instance, the definition module of apply above to be decorated as shown:

$$\mathsf{let\ apply} = \Lambda \mathsf{p}_1, \mathsf{p}_2 \,|\, \mathsf{p}_2 \sqsupseteq \mathsf{p}_1.\lambda f{:}(\mathsf{int}^{\mathsf{p}_1} \multimap \mathsf{int})^\top.\lambda x{:}\mathsf{int}^{\mathsf{p}_2}.f\ x.$$

The compiler would also need to save the type of such a function in the module interface, that we could write using a similar notation:

$$\mathsf{apply}\ :\ \forall \mathsf{p}_1, \mathsf{p}_2 \,|\, \mathsf{p}_2 \sqsupseteq \mathsf{p}_1.(\mathsf{int}^{\mathsf{p}_1} \multimap \mathsf{int})^\top \multimap \mathsf{int}^{\mathsf{p}_2} \multimap \mathsf{int}.$$

It is clear that the two substitution instances required to accurately analyse the examples above arise as substitution instances of this polymorphic type.

5.2 The type system

Having discussed our motivations, we are now ready to describe an extension of our intermediate linear language with a notion of annotation polymorphism.

5.2.1 Types

The types of the new language, ranged over by ϕ and ψ, extend the types of NLL (Section 3.2), as the following grammar rules show.

$$\begin{array}{rcll} \phi & ::= & \mathbb{G} & \text{Ground type} \\ & | & \phi^t \multimap \phi & \text{Linear function space} \\ & | & \phi^t \otimes \phi^t & \text{Tensor product} \\ & | & \forall \overline{\mathsf{p}_i} \,|\, \Theta.\phi & \text{Generalised type} \\ \\ t & ::= & a & \text{Annotation value} \\ & | & \mathsf{p} & \text{Annotation parameter} \\ & | & t + t & \text{Contraction of annotations} \end{array}$$

Types carry annotations drawn from a set $\mathbb{T}$ of *annotation terms*, which include not only annotation values as before, but also *annotation parameters*, or any explicit combination of annotation terms using the contraction operator $+$. We assume an infinite supply $\mathbb{P}$ of annotation parameters.

A type that contains only annotation values, as before, will be called a *simple type*, and we shall use σ and τ to range over them.

The new type construct, written

$$\forall \overline{\mathsf{p}_i} \,|\, \Theta.\phi,$$

stands for a *generalised type*[4], and consists of a set of *quantified annotation parameters* $\overline{\mathsf{p}_i}$, a constraint set Θ, and a type ϕ. Annotation generalisation, or quantification, relies on a mechanism for providing a range of values for the quantified annotation parameters, which in our case takes the form of a set of constraint inequations. The notation $\overline{\mathsf{p}_i}$ is used here to stand for the indexed set $\{\mathsf{p}_i\}_{i \leq n}$, for some n. We shall abbreviate indexed sets similarly for other syntactic elements. Also, whenever we see fit, we shall write sets as comma-separated sequences, as we have done for contexts.

A *constraint set* Θ is a (possibly empty) finite set of inequations of the following form:

$$\Theta ::= t_1 \sqsupseteq t'_1, \ldots, t_n \sqsupseteq t'_n.$$

No restrictions whatsoever are placed on constraint sets; in particular, constraint sets are allowed to be inconsistent.

Intuitively speaking, a generalised type may be understood as a compact description for a family, or set, of types. For instance, the family denoted by the generalised type $\forall \mathsf{p} \,|\, \mathsf{p} \sqsupseteq 1.\mathsf{int}^{\mathsf{p}} \multimap \mathsf{bool}$, involves two types, $\mathsf{int}^1 \multimap \mathsf{bool}$ and $\mathsf{int}^\top \multimap \mathsf{bool}$, each of which could stand for the type required by two uses of the same function in two different contexts.

The notation '$\overline{\mathsf{p}_i} \mid \Theta$', which is usually found in definitions of sets by comprehension, suggests that Θ should not only be understood as a 'system' of constraints (for which a 'solution' must be found), but also as a logical predicate. In fact, even if we have established here the general form of this predicate, it is perhaps interesting to point out that its internal structure is not very important, as long as it denotes a logical predicate. We have been careful to remain as general as possible in this sense, so all the properties of the extended type system do not actually depend on the precise nature of Θ. We shall feel free to write $\forall \overline{\mathsf{p}_i}.\phi$ as an abbreviation for $\forall \overline{\mathsf{p}_i} \,|\, \emptyset.\phi$, to recall the syntax of universal quantification.

[4]The term 'qualified type' may also have been appropriate in this context, since generalised types describe families of types, however we have preferred to use the term that is familiar in context analysis.

5.2.2 Preterms

The set of preterms $\Lambda_{\text{NLL}^\forall}$, ranged over by M and N, extends the preterms of NLL as follows:

$$
\begin{array}{lcll}
M & ::= & \pi & \text{Primitive} \\
 & | & x & \text{Variable} \\
 & | & \lambda x{:}\phi^t.M & \text{Function abstraction} \\
 & | & M M & \text{Function application} \\
 & | & \langle M, M\rangle^{t,t} & \text{Pairing} \\
 & | & \mathsf{let}\ \langle x, x\rangle^{t,t} = M\ \mathsf{in}\ M & \text{Unpairing} \\
 & | & \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M & \text{Conditional} \\
 & | & \mathsf{fix}\ x{:}\phi.M & \text{Fixpoint} \\
 & | & \Lambda\overline{\mathsf{p}_i}\,|\,\Theta.M & \text{Generalised (pre)term} \\
 & | & M\ \vartheta & \text{Specialised (pre)term}
\end{array}
$$

As for types, preterms also carry annotation terms. We also extend the syntax of the language with two new language constructs, $\Lambda\overline{\mathsf{p}_i}\,|\,\Theta.M$ and $M\vartheta$, corresponding to a notion of functional abstraction over a set of named annotation parameters $\overline{\mathsf{p}_i}$, together with its matching notion of application. In fact, we shall also refer to these as Λ-*abstraction* and Λ-*application*, respectively. The operand of the application ϑ denotes an annotation substitution, defined below.

We may write $M[\mathsf{p}_1, \ldots, \mathsf{p}_n]$ to explicitly indicate that $\mathsf{p}_1, \ldots, \mathsf{p}_n$ actually occur (free) in the preterm M (see Subsection 5.2.3).

We assume Λ-application to be left-associative, so

$$M\ \vartheta_1\ \vartheta_2 \ldots \vartheta_n \stackrel{\text{def}}{=} (\ldots((M\ \vartheta_1)\ \vartheta_2)\ldots)\ \vartheta_n.$$

We must not forget to specify how term-substitution $M[\rho]$ should behave for the new constructs, so we define

$$(\Lambda\overline{\mathsf{p}_i}\,|\,\Theta.M)[\rho] \stackrel{\text{def}}{=} \Lambda\overline{\mathsf{p}}\,|\,\Theta.M[\rho] \quad \text{if } \overline{\mathsf{p}_i} \not\subseteq \cup_{x\in dom(\rho)} FA(\rho(x)) \tag{5.1}$$

$$(M\ \vartheta)[\rho] \stackrel{\text{def}}{=} (M[\rho])\ \vartheta \tag{5.2}$$

Before showing the reader the rules of the type system per se, it seems wise to first give a detailed definition of the basic syntactic notions of set of free variables and substitution when these involve annotation parameters. These definitions, although boring, are important as they expose quite clearly the binding role of generalised types, and may help the reader to correctly 'parse' the rest of the chapter.

5.2.3 Set of free annotation parameters

If $\Theta \equiv \overline{t_i \sqsupseteq t'_i}$ is a constraint set, we define the set of free annotation parameters of Θ, as follows[5]:

$$FA(\overline{t_i \sqsupseteq t'_i}) \stackrel{\text{def}}{=} \bigcup_i (FA(t_i) \cup FA(t'_i)),$$

[5]Actually, the language of annotation terms does not include any binding constructs, so all annotation parameters are free. We might think of extensions where this would not be the case [34].

where $FA(t)$ denotes the set of free annotation parameters in t, inductively defined by

$$FA(a) \stackrel{\text{def}}{=} \emptyset$$
$$FA(\mathsf{p}) \stackrel{\text{def}}{=} \{\mathsf{p}\}$$
$$FA(t_1 + t_2) \stackrel{\text{def}}{=} FA(t_1) \cup FA(t_2).$$

Likewise, the set $FA(\phi)$ of free annotation parameters in a type ϕ stands for all the annotation parameters occurring in ϕ. The only special case to consider is

$$FA(\forall \overline{\mathsf{p}_i} \,|\, \Theta.\phi) \stackrel{\text{def}}{=} (FA(\Theta) \cup FA(\phi))\backslash\{\overline{\mathsf{p}_i}\}.$$

Similarly, for the set of free annotation parameters of a preterm M, $FA(M)$, we have that $FA(\Lambda\overline{\mathsf{p}} \,|\, \Theta.M) = (FA(\Theta) \cup FA(M))\backslash\{\overline{\mathsf{p}}\}$.

In a generalised type, as is the case for other binding constructs in the language, the name given to the quantified annotation parameters should not be important, so we shall regard $\forall \overline{\mathsf{p}_i} \,|\, \Theta.\phi$ and $\forall \overline{\mathsf{q}_i} \,|\, \Theta[\overline{\mathsf{q}_i}/\overline{\mathsf{p}_i}].\phi[\overline{\mathsf{q}_i}/\overline{\mathsf{p}_i}]$ as syntactically equivalent, where $\overline{\mathsf{q}_i}$ are fresh annotation parameters not occurring anywhere else in the type. We shall therefore work with α-equivalence classes of types, and preterms as well, and assume that annotation parameters are implicitly renamed as necessary.

5.2.4 Annotation substitution

An *annotation substitution* is any partial function

$$\vartheta : \mathbb{P} \rightharpoonup \mathbb{T}$$

assigning annotation terms to annotation parameters. We shall use the notation

$$\langle t_1/\mathsf{p}_1, \ldots, t_n/\mathsf{p}_n \rangle$$

to stand for the (finite) annotation substitution ϑ mapping p_i into $\vartheta(\mathsf{p}_i) = t_i$, as expected. The special empty annotation substitution will be written $\langle\rangle$.

We write $t[\vartheta]$, $\Theta[\vartheta]$, $\phi[\vartheta]$ and $M[\vartheta]$ for the 'simultaneous' substitution of $\vartheta(\mathsf{p})$ for the (free) occurrences of $\mathsf{p} \in dom(\vartheta)$ in t, Θ, ϕ and M, respectively. Their definitions are detailed in Figure 5.1.

The notation $\vartheta\backslash\overline{\mathsf{p}_i}$ is used to refer to the map that is the same as ϑ, but restricted to the domain $dom(\vartheta)\backslash\overline{\mathsf{p}_i}$ (hence, $\mathsf{p}_i[\vartheta\backslash\overline{\mathsf{p}_i}] = \mathsf{p}_i$ according to the definition above). The condition $\overline{\mathsf{p}_i} \not\subseteq img(\vartheta)$ is standard; it ensures that no p_i becomes incorrectly bound by a $\forall$ or Λ during substitution. In the last equation, $\vartheta \circ \vartheta'$ stands for the composition of the substitutions ϑ and ϑ', defined by

$$(\vartheta \circ \vartheta')(\mathsf{p}) \stackrel{\text{def}}{=} (\mathsf{p}[\vartheta'])[\vartheta].$$

We shall feel free to write $\phi[t/\mathsf{p}]$ as an abbreviation for $\phi[\vartheta]$, where ϑ is such that $dom(\vartheta) = \{\mathsf{p}\}$ and $\vartheta(\mathsf{p}) = t$.

An annotation substitution θ mapping annotation parameters into annotation values,

$$\theta : \mathbb{P} \rightharpoonup \mathbb{A},$$

will be called a *ground substitution*. The terms *valuation* and *annotation assignment* will be used as synonyms.

$$a[\vartheta] \stackrel{\text{def}}{=} a$$

$$\mathsf{p}[\vartheta] \stackrel{\text{def}}{=} \begin{cases} \vartheta(\mathsf{p}) & \text{if } \mathsf{p} \in dom(\vartheta) \\ \mathsf{p} & \text{otherwise} \end{cases}$$

$$(t_1 + t_2)[\vartheta] \stackrel{\text{def}}{=} t_1[\vartheta] + t_2[\vartheta]$$

$$(\overline{t_i \sqsupseteq t'_i})[\vartheta] \stackrel{\text{def}}{=} \overline{t_i[\vartheta] \sqsupseteq t'_i[\vartheta]}$$

$$\mathbb{G}[\vartheta] \stackrel{\text{def}}{=} \mathbb{G}$$

$$(\phi^t \multimap \psi)[\vartheta] \stackrel{\text{def}}{=} \phi[\vartheta]^{t[\vartheta]} \multimap \psi[\vartheta]$$

$$(\phi^t \otimes \psi^{t'})[\vartheta] \stackrel{\text{def}}{=} \phi[\vartheta]^{t[\vartheta]} \otimes \psi[\vartheta]^{t'[\vartheta]}$$

$$(\forall \overline{\mathsf{p}_i} \,|\, \Theta.\phi)[\vartheta] \stackrel{\text{def}}{=} \forall \overline{\mathsf{p}_i} \,|\, \Theta[\vartheta \backslash \overline{\mathsf{p}_i}].\phi[\vartheta \backslash \overline{\mathsf{p}_i}], \quad \text{if } \overline{\mathsf{p}_i} \not\subseteq img(\vartheta)$$

$$\pi[\vartheta] \stackrel{\text{def}}{=} \pi$$

$$x[\vartheta] \stackrel{\text{def}}{=} x$$

$$(\lambda x{:}\phi^t.M)[\vartheta] \stackrel{\text{def}}{=} \lambda x{:}\phi[\vartheta]^{t[\vartheta]}.M[\vartheta]$$

$$(MN)[\vartheta] \stackrel{\text{def}}{=} (M[\vartheta])\ (N[\vartheta])$$

$$(\langle M_1, M_2 \rangle^{t_1,t_2})[\vartheta] \stackrel{\text{def}}{=} \langle M_1[\vartheta], M_2[\vartheta] \rangle^{t_1[\vartheta],t_2[\vartheta]}$$

$$(\mathsf{let}\ \langle x_1, x_2 \rangle^{t_1,t_2} = M\ \mathsf{in}\ N)[\vartheta] \stackrel{\text{def}}{=} \mathsf{let}\ \langle x_1, x_2 \rangle^{t_1[\vartheta],t_2[\vartheta]} = M[\vartheta]\ \mathsf{in}\ N[\vartheta]$$

$$(\mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2)[\vartheta] \stackrel{\text{def}}{=} \mathsf{if}\ M[\vartheta]\ \mathsf{then}\ N_1[\vartheta]\ \mathsf{else}\ N_2[\vartheta]$$

$$(\mathsf{fix}\ x{:}\phi.M)[\vartheta] \stackrel{\text{def}}{=} \mathsf{fix}\ x{:}\phi[\vartheta].M[\vartheta]$$

$$(\Lambda \overline{\mathsf{p}} \,|\, \Theta.M)[\vartheta] \stackrel{\text{def}}{=} \Lambda \overline{\mathsf{p}_i} \,|\, \Theta[\vartheta \backslash \overline{\mathsf{p}_i}].M[\vartheta \backslash \overline{\mathsf{p}_i}] \quad \text{if } \overline{\mathsf{p}_i} \not\subseteq img(\vartheta)$$

$$(M\ \vartheta')[\vartheta] \stackrel{\text{def}}{=} (M[\vartheta])\ (\vartheta \circ \vartheta')$$

Figure 5.1: Annotation substitution

Definition 5.2.1 (Annotation term evaluation)
We shall write $\theta^*(t)$ for the *evaluation* of t under θ, defined by

$$\theta^*(a) \stackrel{\text{def}}{=} a$$
$$\theta^*(\mathsf{p}) \stackrel{\text{def}}{=} \theta(\mathsf{p})$$
$$\theta^*(t_1 + t_2) \stackrel{\text{def}}{=} \theta^*(t_1) + \theta^*(t_2).$$

Alternatively, we can define $\theta^*(t)$ as the extension of θ to annotation terms. For this reason, we shall drop the distinction and generally write $\theta(t)$ to mean $\theta^*(t)$ when necessary. □

Notice the difference existing between $\theta^*(t)$ and $t[\theta]$; if $t \equiv \mathsf{p} + \mathsf{q}$ and $\theta \equiv \langle 1/\mathsf{p}, \top/\mathsf{q}\rangle$, whereas $\theta^*(t) = \top$, we have $t[\theta] = 1 + \top$. It would perhaps have been wiser to distinguish explicitly between the two uses of + in the syntax, however the use of the contraction operator as a function will only be relevant in connection with the evaluation of terms.

5.2.5 Constraint set satisfaction

We shall mostly be interested in valuations that are solutions to the inequations in a given constraint set.

Definition 5.2.2 (Solution, satisfaction)
A valuation θ is a *solution* of a constraint set Θ, if each constraint is independently verified with respect to the assignments in θ. Formally, we define the predicate

$$\theta \models \Theta \stackrel{\text{def}}{=} \theta(\mathsf{p}) \sqsupseteq \theta(t), \text{ for all } \mathsf{p} \sqsupseteq t \text{ in } \Theta,$$

and equivalently say that θ *satisfies* Θ. □

According to the above definition,

$$\theta \models \emptyset,$$

for any θ. Also, if we write Θ, Θ' for the union (disjunction) of the constraint sets Θ and Θ', then

$$\theta \models \Theta, \Theta' \quad \text{whenever} \quad \theta \models \Theta \text{ and } \theta \models \Theta',$$

for all suitable θ (i.e., for all θ such that $FA(\Theta) \cup FA(\Theta') \subseteq dom(\theta)$).

It is implicit in the definition of satisfaction that for $\theta \models \Theta$ to be properly defined, $FA(\Theta) \subseteq dom(\theta)$; otherwise, the substitutions $\theta(\mathsf{p})$ and $\theta(t)$ would be meaningless. We shall say in this case that θ *covers* Θ. The same terminology will be employed for other constructs of the language, including types, terms, typing contexts, and even whole typing judgments.

We shall write $[\Theta]$ for the *solution space* of Θ (i.e., the set of valuations that satisfy Θ). That is,

$$[\Theta] \stackrel{\text{def}}{=} \{\theta \mid \theta \models \Theta\}.$$

5.2.6 Constraint implication

If P is any predicate on annotation terms (which may perhaps contain some free annotation parameters), we define

$$\Theta \triangleright P \stackrel{\text{def}}{=} \text{for all } \theta,\ \text{if } \theta \models \Theta \text{ then } \theta(P) \text{ holds},$$

where $\theta(P)$ is defined by replacing the occurrences of t in P by $\theta(t)$, as expected. The assertion $\Theta \triangleright P$ should be read "P is valid in the context of Θ". In our case, P will more commonly stand for a structural assertion, so Θ will effectively give the possible set of annotation values for which the structural assertion is valid.

We have admitted two readings of Θ: as a set of constraints, and as a predicate (i.e., comprising the conjunction of constraint inequalities). As a predicate, we note that $\theta(\Theta)$ is just a synonym for $\theta \models \Theta$ (since $\theta(\Theta) \equiv \theta(\overline{t'_i \sqsupseteq t''_i}) = \overline{\theta(t'_i) \sqsupseteq \theta(t''_i)}$). Therefore,

$$\Theta \triangleright \Theta' \quad \text{is equivalent to} \quad \theta \models \Theta \text{ implies } \theta \models \Theta' \text{ for all } \theta.$$

It is this use of '$\triangleright$' that has deserved in the literature the name of constraint implication. Notice that $\Theta \triangleright \Theta'$ actually implies $[\Theta] \subseteq [\Theta']$, which establishes the semantics of constraint implication when constraint sets are interpreted as solution sets.

We shall end here our discussion on constraint sets by giving a list of properties that will prove useful for the sequel.

Proposition 5.2.3 (Some properties of $\triangleright$)
The following properties hold for any constraint sets Θ, Θ' and Θ'':

a. $\Theta \triangleright \Theta$.

b. $\Theta \triangleright \Theta'$ *and* $\Theta' \triangleright \Theta''$ *imply* $\Theta \triangleright \Theta''$.

c. $\Theta, \Theta' \triangleright \Theta$.

d. $\Theta \triangleright P'$ *and* $\Theta \triangleright P''$ *imply* $\Theta \triangleright P', P''$.

e. $\Theta \triangleright P$ *implies* $\Theta[\vartheta] \triangleright P[\vartheta]$.

f. $\Theta, \Theta' \triangleright P$ *and* $\Theta \triangleright \Theta'[\vartheta]$ *implies* $\Theta \triangleright P[\vartheta]$, *where the annotation substitution* ϑ *is such that* $dom(\vartheta) = FA(\Theta') \setminus FA(\Theta)$.

Proof. Notice how $\triangleright$ behaves precisely like logical implication, so we could think of many more interesting properties other than the ones given. (The comma ',' in P', P'' is supposed to denote disjunction.)

We give a proof of property (f), which is the only non-obvious property. From $\Theta, \Theta' \triangleright P$ and (e) we deduce $\Theta[\vartheta], \Theta'[\vartheta] \triangleright P[\vartheta]$. Also, $\Theta \triangleright \Theta[\vartheta], \Theta'[\vartheta]$, since $\Theta[\vartheta] \equiv \Theta$ (as $FA(\Theta) \cap dom(\vartheta) = \emptyset$ because $dom(\vartheta) = FA(\Theta') \setminus FA(\Theta)$) and $\Theta \triangleright \Theta'[\vartheta]$ by assumption; hence, from (b), we conclude $\Theta \triangleright P[\vartheta]$. □

5.2.7 The typing rules

We know of at least two different approaches for introducing annotation quantification into a type system of structural properties like NLL. The approach we fully discuss in this chapter is the first approach, based on the presentation of NLL where the context restrictions appear explicitly as rule side-conditions, and which is also the one we have been dealing with until now. A second approach is based on the presentation of Appendix A, which is closer in spirit to Bierman's monomorphic type system [13], albeit less expressive. The second approach is more elegant from a logical viewpoint, as it is more compact; the first one is closer to the algorithms of annotation inference we shall investigate in the following chapter.

We call $\mathrm{NLL}^{\forall}$ the type system that extends NLL with annotation polymorphism. The new system can be easily recognised because of the form of its typing judgments:

$$\Theta\,;\Gamma \vdash M : \phi.$$

Any typing declarations in Γ are allowed to be annotated with arbitrary annotation terms:

$$\Gamma ::= x_1 : {\phi_1}^{t_1}, \ldots, x_n : {\phi_n}^{t_n}.$$

It will be useful to have a notation for the free annotation variables in a typing context. Therefore, we define

$$\mathit{FA}(-) \stackrel{\mathrm{def}}{=} \emptyset \quad \text{and} \quad \mathit{FA}(\Gamma, x : \phi^t) \stackrel{\mathrm{def}}{=} \mathit{FA}(\Gamma) \cup \mathit{FA}(\phi) \cup \mathit{FA}(t). \tag{5.3}$$

The typing rules of the new system are shown in Figure 5.2.

The basic idea is that the set of constraints Θ specifies the range of annotation values for the annotation parameters occurring free in the rest of the typing judgment, and for which it is assumed to be valid. As such, it provides an 'interpretation' against which it is possible to verify the side-conditions. This explains our adoption of the notation $\Theta \triangleright |\Gamma| \sqsupseteq t$, which generalises our old notation for side-conditions in an obvious way:

$$\Theta \triangleright |\Gamma| \sqsupseteq t \;\stackrel{\mathrm{def}}{=}\; \Theta \triangleright |\Gamma(x)| \sqsupseteq t, \text{ for all } x \in \mathit{dom}(\Gamma). \tag{5.4}$$

We should remark that the side-condition $\Theta \triangleright t \sqsupseteq \top$ in the Weakening rule is not really necessary; we could as well have replaced t by $\top$ directly in the conclusion of the rule. However, we would like to be able to write sequents like

$$\mathsf{p} \sqsupseteq \top\,; x : \phi^{\mathsf{p}} \vdash 0 : \mathsf{int},$$

which would be forbidden if we did not allow annotation parameters in discardable typing declarations. The same remark applies to the Fixpoint and Contraction rules. The use of inequations in the side-conditions of the structural rules is convenient to our discussion of annotation inference, in the next chapter.

5.2.8 Introducing and eliminating generalised types

Except for $\forall_{\mathcal{I}}$ and $\forall_{\mathcal{E}}$, it is not difficult to see that the remaining typing rules adapt the typing rules of NLL to typing judgments containing free annotation parameters.

$$\frac{}{\Theta\,;x:\phi^t\vdash x:\phi}\,\text{Identity}\qquad\frac{\Sigma(\pi)=\sigma}{\Theta\,;-\vdash\pi:\sigma}\,\text{Primitive}$$

$$\frac{\Theta\,;\Gamma,x:\phi^t\vdash M:\psi}{\Theta\,;\Gamma\vdash\lambda x{:}\phi^t.M:\phi^t\multimap\psi}\multimap_{\mathcal{I}}$$

$$\frac{\Theta\,;\Gamma_1\vdash M:\phi^t\multimap\psi\quad\Theta\,;\Gamma_2\vdash N:\phi\quad\Theta\rhd|\Gamma_2|\sqsupseteq t}{\Theta\,;\Gamma_1,\Gamma_2\vdash MN:\psi}\multimap_{\mathcal{E}}$$

$$\frac{\Theta\,;\Gamma_1\vdash M_1:\phi_1\quad\Theta\,;\Gamma_2\vdash M_2:\phi_2\quad\Theta\rhd|\Gamma_1|\sqsupseteq t_1\quad\Theta\rhd|\Gamma_2|\sqsupseteq t_2}{\Theta\,;\Gamma_1,\Gamma_2\vdash\langle M_1,M_2\rangle^{t_1,t_2}:{\phi_1}^{t_1}\otimes{\phi_2}^{t_2}}\otimes_{\mathcal{I}}$$

$$\frac{\Theta\,;\Gamma_1\vdash M:{\phi_1}^{t_1}\otimes{\phi_2}^{t_2}\quad\Theta\,;\Gamma_2,x_1:{\phi_1}^{t_1},x_2:{\phi_2}^{t_2}\vdash N:\psi}{\Theta\,;\Gamma_1,\Gamma_2\vdash\mathsf{let}\ \langle x_1,x_2\rangle^{t_1,t_2}=M\ \mathsf{in}\ N:\psi}\otimes_{\mathcal{E}}$$

$$\frac{\Theta\,;\Gamma_1\vdash M:\mathsf{bool}\quad\Theta\,;\Gamma_2\vdash N_1:\phi\quad\Theta\,;\Gamma_2\vdash N_2:\phi}{\Theta\,;\Gamma_1,\Gamma_2\vdash\mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2:\phi}\,\text{Conditional}$$

$$\frac{\Theta\,;\Gamma,x:\phi^t\vdash M:\phi\quad\Theta\rhd|\Gamma,x:\phi^t|\sqsupseteq\top}{\Theta\,;\Gamma\vdash\mathsf{fix}\ x{:}\phi.M:\phi}\,\text{Fixpoint}$$

$$\frac{\Theta,\Theta'\,;\Gamma\vdash M:\phi\quad\overline{\mathsf{p}_i}\not\subseteq FA(\Theta\,;\Gamma)\quad\Theta'\backslash\overline{\mathsf{p}_i}=\emptyset}{\Theta\,;\Gamma\vdash\Lambda\overline{\mathsf{p}_i}\,|\,\Theta'.M:\forall\overline{\mathsf{p}_i}\,|\,\Theta'.\phi}\forall_{\mathcal{I}}$$

$$\frac{\Theta\,;\Gamma\vdash M:\forall\overline{\mathsf{p}_i}\,|\,\Theta'.\phi\quad\Theta\rhd\Theta'[\vartheta]\quad dom(\vartheta)=\overline{\mathsf{p}_i}}{\Theta\,;\Gamma\vdash M\ \vartheta:\phi[\vartheta]}\forall_{\mathcal{E}}$$

$$\frac{\Theta\,;\Gamma\vdash M:\psi\quad\Theta\rhd t\sqsupseteq\top}{\Theta\,;\Gamma,x:\phi^t\vdash M:\psi}\,\text{Weakening}$$

$$\frac{\Theta\,;\Gamma,x_1:\phi^{t_1},x_2:\phi^{t_2}\vdash M:\psi\quad\Theta\rhd t\sqsupseteq t_1+t_2}{\Theta\,;\Gamma,x:\phi^t\vdash M[x/x_1,x/x_2]:\psi}\,\text{Contraction}$$

Figure 5.2: The type system $\text{NLL}^{\forall}$

There are two typing rules that deal with quantification per se. Generalised types are introduced in type derivations with the following rule:

$$\frac{\Theta, \Theta' \,;\, \Gamma \vdash M : \phi \quad \overline{\mathsf{p}_i} \not\subseteq FA(\Theta\,;\,\Gamma) \quad \Theta' \backslash \overline{\mathsf{p}_i} = \emptyset}{\Theta\,;\,\Gamma \vdash \Lambda\overline{\mathsf{p}_i} \,|\, \Theta'.M : \forall\overline{\mathsf{p}_i} \,|\, \Theta'.\phi} \; \forall_{\mathcal{I}}$$

Its meaning is fairly intuitive, although its side-conditions deserve to be briefly explained. A generalised term $\Lambda\overline{\mathsf{p}_i} \,|\, \Theta'.M$ has type $\forall\overline{\mathsf{p}_i} \,|\, \Theta'.\phi$, if M has type ϕ under the interpretation given by considering all the inequations in both Θ and Θ'. The condition $\overline{\mathsf{p}_i} \not\subseteq FA(\Theta\,;\,\Gamma)$ is standard in logic, and means that none of the quantified annotation parameters may appear outside the scope of the Λ-binder. The condition

$$\Theta' \backslash \overline{\mathsf{p}_i} = \emptyset$$

states that all inequations in Θ' must involve a quantified annotation parameter, which is a simple way of guaranteeing that no inequations involving only unbound annotation parameters wrongly leave the scope of Θ. If this was allowed to happen, we would have a mechanism for relaxing the restrictions on some of the free annotation parameters, by moving inequations inside the scope of Λ-binders.

The notation $\Theta\backslash\mathbb{P}$, where $\mathbb{P}$ is any set of annotation parameters, denotes the set of inequations in Θ that do not involve any annotation parameter $\mathsf{p} \in \mathbb{P}$. If we define $\Theta\!\restriction\!\mathbb{P}$ to stand for the inequations in Θ that do involve some $\mathsf{p} \in \mathbb{P}$, as detailed in the following equations, then $\Theta\backslash\mathbb{P}$ can be simply defined as its complement.

$$\begin{aligned} \emptyset\!\restriction\!\mathbb{P} &= \emptyset \\ (\Theta, \mathsf{p} \sqsupseteq t)\!\restriction\!\mathbb{P} &= \begin{cases} (\Theta\!\restriction\!\mathbb{P}), \mathsf{p} \sqsupseteq t, & \text{if, for some } \mathsf{p}' \in \mathbb{P},\ \mathsf{p}' \in FA(\mathsf{p} \sqsupseteq t); \\ (\Theta\!\restriction\!\mathbb{P}) & \text{otherwise.} \end{cases} \end{aligned}$$

Taken together, both side-conditions imply that Θ' is uniquely determined by the choice of $\overline{\mathsf{p}_i}$: it suffices to take, from the available set of inequations, those inequations involving all $\mathsf{p} \in \overline{\mathsf{p}_i}$. For this reason, we may sometimes refer to these as the 'separation conditions' of the $\forall_{\mathcal{E}}$ rule[6].

Using this rule, we can assign to our example apply function of Section 5.1, an annotated polymorphic type, from which the two types required to accurately analyse the example arise as type instances. The type derivation that supports this claim is shown below.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{-\,;\,f{:}(\mathsf{int}^{\mathsf{p}} \multimap \mathsf{int})^{\top} \vdash f{:}\mathsf{int}^{\mathsf{p}} \multimap \mathsf{int}}\,\text{Identity} \qquad \dfrac{}{-\,;\,x{:}\mathsf{int}^{\mathsf{p}} \vdash x{:}\mathsf{int}}\,\text{Identity}}{-\,;\,f{:}(\mathsf{int}^{\mathsf{p}} \multimap \mathsf{int})^{\top}, x{:}\mathsf{int}^{\mathsf{p}} \vdash f\ x : \mathsf{int}}\,\multimap_{\mathcal{E}}}{-\,;\,- \vdash \lambda f{:}(\mathsf{int}^{\mathsf{p}} \multimap \mathsf{int})^{\top}.\lambda x{:}\mathsf{int}^{\mathsf{p}}.f\ x : (\mathsf{int}^{\mathsf{p}} \multimap \mathsf{int})^{\top} \multimap \mathsf{int}^{\mathsf{p}} \multimap \mathsf{int}}\,\multimap_{\mathcal{I}}}{-\,;\,- \vdash \Lambda\mathsf{p} \,|\, \emptyset.\lambda f{:}(\mathsf{int}^{\mathsf{p}} \multimap \mathsf{int})^{\top}.\lambda x{:}\mathsf{int}^{\mathsf{p}}.f\ x : \forall\mathsf{p} \,|\, \emptyset.(\mathsf{int}^{\mathsf{p}} \multimap \mathsf{int})^{\top} \multimap \mathsf{int}^{\mathsf{p}} \multimap \mathsf{int}}\,\forall_{\mathcal{I}}$$

[6] There seem to be many possible ways of presenting the side-conditions of the $\forall_{\mathcal{I}}$ rule; our choice admits a 'deterministic' reading that is appropriate for the annotation inference algorithms of the next chapter. Alternatively, we might have chosen to remove the side-condition $\Theta\backslash\overline{\mathsf{p}_i} = \emptyset$ and replace Θ in the conclusion of the rule by $\Theta, \Theta'\backslash\overline{\mathsf{p}_i}$. This modification ensures that inequations not involving any p_i do effectively go out of scope, while 'hiding out' all inequations involving $\overline{\mathsf{p}_i}$ from the conclusion constraints, as necessary. In [59], for instance, the conclusion constraints would have read $\Theta, \exists\overline{\mathsf{p}_i}.\Theta'$, where $\exists$ is introduced as an existential quantification operator for constraint sets (predicates). In either case, the set of relevant free annotation parameters and constraint inequations for a given typing judgment are the same.

(The side-condition of the application of the $\multimap_{\mathcal{E}}$ rule, not shown, should read $\emptyset \triangleright \mathsf{p} \sqsupseteq \mathsf{p}$.)

The application of a Λ-abstraction (specialisation) is typed using the following rule:

$$\frac{\Theta\,;\Gamma \vdash M : \forall \overline{\mathsf{p}_i}\,|\,\Theta'.\phi \quad \Theta \triangleright \Theta'[\vartheta] \quad dom(\vartheta) = \overline{\mathsf{p}_i}}{\Theta\,;\Gamma \vdash M\ \vartheta : \phi[\vartheta]}\ \forall_{\mathcal{E}}$$

The rule states that if M has generalised type $\forall \overline{\mathsf{p}_i}\,|\,\Theta'.\phi$ and ϑ is a given annotation substitution with domain $\overline{\mathsf{p}_i}$, then $M\ \vartheta$ has type $\phi[\vartheta]$, provided that the constraint set obtained by applying ϑ to the inequations in Θ', is valid under the interpretation given by Θ.

Using the $\forall_{\mathcal{E}}$ rule, for instance, if we wanted to use apply in the context of a non-linear function, we would first need to obtain a non-linear type instance, as follows:

$$\frac{\begin{array}{c}\vdots \\ -\,;- \vdash \mathsf{apply} : \forall \mathsf{p}\,|\,\emptyset.(\mathsf{int}^{\mathsf{p}} \multimap \mathsf{int})^{\top} \multimap \mathsf{int}^{\mathsf{p}} \multimap \mathsf{int} \qquad \emptyset \triangleright \emptyset\end{array}}{-\,;- \vdash \mathsf{apply}\ \langle \top/\mathsf{p}\rangle : (\mathsf{int}^{\top} \multimap \mathsf{int})^{\top} \multimap \mathsf{int}^{\top} \multimap \mathsf{int}}\ \forall_{\mathcal{E}}$$

5.2.9 A 'most general' example decoration

As an example, Figure 5.3 shows an annotated-polymorphic decoration for the FPL function

$$\mathsf{curry} \stackrel{\text{def}}{=} \lambda f{:}((\sigma_1 \times \sigma_2) \to \tau).\lambda x_1{:}\sigma_1.\lambda x_2{:}\sigma_2.f\ \langle x_1, x_2\rangle,$$

of type $((\sigma_1 \times \sigma_2) \to \tau) \multimap \sigma_1 \to \sigma_2 \to \tau$.

The applications of $\multimap_{\mathcal{E}}$ and $\otimes_{\mathcal{I}}$ impose, respectively, the following side-conditions:

$$\Theta \triangleright \mathsf{p}_5 \sqsupseteq \mathsf{p}_1, \mathsf{p}_6 \sqsupseteq \mathsf{p}_2 \quad \text{and} \quad \Theta \triangleright \mathsf{p}_5 \sqsupseteq \mathsf{p}_3, \mathsf{p}_6 \sqsupseteq \mathsf{p}_3.$$

These are clearly verified by Θ, since Θ literally includes them all as part of its definition.

The reader may have noticed that the $\text{NLL}^{\forall}$ decoration of curry carries in it all the information necessary to 'generate' all the corresponding NLL decorations of the same function, which arise as particular instances of it. This observation lies at the heart of the strategy we shall develop in the following chapter to design 'complete' annotation inference algorithms.

5.2.10 Reduction

As we have changed the syntax to allow annotation parameters in the types and terms of the intermediate language, we must update our definition of β-reduction accordingly. We define the reduction relation on $\text{NLL}^{\forall}$ terms as the contextual closure of the following rewrite rules:

$$\begin{aligned}
(\lambda x{:}\phi^t.M)N &\to M[N/x] \\
\mathsf{let}\ \langle x_1, x_2\rangle = \langle M_1, M_2\rangle^{t_1,t_2}\ \mathsf{in}\ N &\to N[M_1/x_1, M_2/x_2] \\
\mathsf{if\ true\ then}\ N_1\ \mathsf{else}\ N_2 &\to N_1 \\
\mathsf{if\ false\ then}\ N_1\ \mathsf{else}\ N_2 &\to N_2 \\
\mathsf{fix}\ x{:}\phi.M &\to M[\mathsf{fix}\ x{:}\phi.M/x] \\
(\Lambda \overline{\mathsf{p}_i}\,|\,\Theta.M)\ \vartheta &\to M[\vartheta]
\end{aligned}$$

The last rewrite rule takes care of reducing the explicit application of Λ-abstractions, a standard rule as may be found in other calculi having explicit syntactic quantification constructs.

$$\dfrac{\dfrac{}{\Theta\,;\, f{:}{\phi_f}^{\mathsf{p}_4} \vdash f{:}\phi_f}\ \text{Identity} \qquad \dfrac{\dfrac{}{\Theta\,;\, x_1{:}{\phi_1}^{\mathsf{p}_5} \vdash x_1{:}\phi_1}\ \text{Identity} \qquad \dfrac{}{\Theta\,;\, x_2{:}{\phi_1}^{\mathsf{p}_6} \vdash x_2{:}\phi_2}\ \text{Identity}}{\Theta\,;\, x_1{:}{\phi_1}^{\mathsf{p}_5}, x_2{:}{\phi_2}^{\mathsf{p}_6} \vdash \langle x_1, x_2\rangle^{\mathsf{p}_1,\mathsf{p}_2} : {\phi_1}^{\mathsf{p}_1} \otimes {\phi_2}^{\mathsf{p}_2}}\ \otimes_{\mathcal{I}}}{\Theta\,;\, f{:}{\phi_f}^{\mathsf{p}_4}, x_1{:}{\phi_1}^{\mathsf{p}_5}, x_2{:}{\phi_2}^{\mathsf{p}_6} \vdash f\ \langle x_1, x_2\rangle^{\mathsf{p}_1,\mathsf{p}_2} : \psi}\ \multimap_{\mathcal{E}}$$

$$\dfrac{\Theta\,;\, f{:}{\phi_f}^{\mathsf{p}_4}, x_1{:}{\phi_1}^{\mathsf{p}_5}, x_2{:}{\phi_2}^{\mathsf{p}_6} \vdash f\ \langle x_1, x_2\rangle^{\mathsf{p}_1,\mathsf{p}_2} : \psi}{\Theta\,;\, - \vdash \lambda f{:}{\phi_f}^{\mathsf{p}_4}.\lambda x_1{:}{\phi_1}^{\mathsf{p}_5}.\lambda x_2{:}{\phi_2}^{\mathsf{p}_6}.f\ \langle x_1, x_2\rangle^{\mathsf{p}_1,\mathsf{p}_2} : {\phi_f}^{\mathsf{p}_4} \multimap {\phi_1}^{\mathsf{p}_5} \multimap {\phi_2}^{\mathsf{p}_6} \multimap \psi}\ \multimap_{\mathcal{I}}$$

$$\dfrac{\Theta\,;\, - \vdash \lambda f{:}{\phi_f}^{\mathsf{p}_4}.\lambda x_1{:}{\phi_1}^{\mathsf{p}_5}.\lambda x_2{:}{\phi_2}^{\mathsf{p}_6}.f\ \langle x_1, x_2\rangle^{\mathsf{p}_1,\mathsf{p}_2} : {\phi_f}^{\mathsf{p}_4} \multimap {\phi_1}^{\mathsf{p}_5} \multimap {\phi_2}^{\mathsf{p}_6} \multimap \psi}{-\,;\, - \vdash \Lambda\overline{\mathsf{p}_i} \mid \Theta.\lambda f{:}{\phi_f}^{\mathsf{p}_4}.\lambda x_1{:}{\phi_1}^{\mathsf{p}_5}.\lambda x_2{:}{\phi_2}^{\mathsf{p}_6}.f\ \langle x_1, x_2\rangle^{\mathsf{p}_1,\mathsf{p}_2} : \forall\overline{\mathsf{p}_i} \mid \Theta.{\phi_f}^{\mathsf{p}_4} \multimap {\phi_1}^{\mathsf{p}_5} \multimap {\phi_2}^{\mathsf{p}_6} \multimap \psi}\ \forall_{\mathcal{I}}$$

For the derivation above, we have

$$\begin{aligned}
\phi_f &\equiv ({\phi_1}^{\mathsf{p}_1} \otimes {\phi_2}^{\mathsf{p}_2})^{\mathsf{p}_3} \multimap \psi \\
\overline{\mathsf{p}_i} &\equiv \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, \mathsf{p}_4, \mathsf{p}_5, \mathsf{p}_6 \\
\Theta &\equiv \mathsf{p}_5 \sqsupseteq \mathsf{p}_1, \mathsf{p}_6 \sqsupseteq \mathsf{p}_2, \mathsf{p}_5 \sqsupseteq \mathsf{p}_3, \mathsf{p}_6 \sqsupseteq \mathsf{p}_3.
\end{aligned}$$

Figure 5.3: An example NLL$^{\forall}$ type derivation

$$\frac{}{\Theta \vdash \mathbb{G} \leq \mathbb{G}}$$

$$\frac{\Theta \vdash \phi_2 \leq \phi_1 \quad \Theta \vdash \psi_1 \leq \psi_2 \quad \Theta \triangleright t_1 \sqsubseteq t_2}{\Theta \vdash {\phi_1}^{t_1} \multimap \psi_1 \leq {\phi_2}^{t_2} \multimap \psi_2}$$

$$\frac{\Theta \vdash \phi_1 \leq \phi_2 \quad \Theta \vdash \psi_1 \leq \psi_2 \quad \Theta \triangleright t_2 \sqsubseteq t_1 \quad \Theta \triangleright t_2' \sqsubseteq t_1'}{\Theta \vdash {\phi_1}^{t_1} \otimes {\psi_1}^{t_1'} \leq {\phi_2}^{t_2} \otimes {\psi_2}^{t_2'}}$$

$$\frac{\Theta, \Theta' \vdash \phi_1 \leq \phi_2 \quad \overline{\mathsf{p}} \not\subseteq FA(\Theta)}{\Theta' \vdash \forall \overline{\mathsf{p}} \,|\, \Theta'.\phi_1 \leq \forall \overline{\mathsf{p}} \,|\, \Theta'.\phi_2}$$

Figure 5.4: Subtyping relation for $\mathrm{NLL}^{\forall\leq}$

5.3 Subtyping annotation polymorphism

Until now, we have discussed annotation polymorphism in the context of NLL without subtyping. In this section, we consider the theory to its full extent, therefore including both a notion of annotation subtyping and quantification.

Let $\mathrm{NLL}^{\forall\leq}$ refer to the typing system $\mathrm{NLL}^{\forall}$ extended with the following Subsumption rule:

$$\frac{\Theta \,;\Gamma \vdash M : \phi \quad \Theta \vdash \phi \leq \psi}{\Theta \,;\Gamma \vdash M : \psi}\ \text{Subsumption}$$

Because ϕ and ψ may contain free annotation parameters, we have adopted a 'contextual' definition of the subtyping relation. The fact that $\phi \leq \psi$ hold with respect to the set of constraints Θ is written in the form of a *subtyping judgment*

$$\Theta \vdash \phi \leq \psi.$$

The set of valid subtyping judgments is inductively defined by the inference rules of Figure 5.4. Notice how the new inference rules generalise the rules of Figure 4.1 to accomodate the fact that types may now contain free annotation parameters.

The meaning of the subtyping rule for generalised types is quite intuitive: A term of type $\forall \overline{\mathsf{p}} \,|\, \Theta'.\phi_1$ may be used in a context with a hole of type $\forall \overline{\mathsf{p}} \,|\, \Theta'.\phi_2$, if any term specialisation of type $\phi_1[\vartheta]$ may be used in any context specialisation of type $\phi_2[\vartheta]$, for suitable ϑ.

To reduce the notational clutter, we may sometimes choose to write $\phi \leq \psi$ as an abbreviation for $- \vdash \phi \leq \psi$.

5.3.1 Soundness

Following the development of Section 4.2, it is easy to provide an interpretation for $\Theta \vdash \phi \leq \psi$ in terms of a coercion function of type $\phi^1 \multimap \psi$ in context Θ. It suffices to upgrade the definition of this function by replacing σ and τ by ϕ and ψ, respectively, and introduce distinct

annotation parameters for the annotations. The equation that deals with quantification is given as follows:

$$[\![\forall\overline{\mathsf{p}}\,|\,\Theta.\phi_1 \le \forall\overline{\mathsf{p}}\,|\,\Theta.\phi_2]\!] \stackrel{\text{def}}{=} \lambda x{:}(\Lambda\overline{\mathsf{p}}\,|\,\Theta)^1.\Lambda\overline{\mathsf{p}}\,|\,\Theta.[\![\phi_1 \le \phi_2]\!]\ (x\ \langle\rangle).$$

(We recall that $\langle\rangle$ is the identity with respect to syntactic annotation substitution.)

Proposition 5.3.1
$\Theta\,;- \vdash [\![\phi \le \psi]\!] : \phi^1 \multimap \psi$.

Proof. Easy induction on the definition of $[\![\phi \le \psi]\!]$. □

The subtyping relation is related to constraint implication and annotation substitution as shown by the following propositions. These are required to prove Lemma 5.3.4, which states that annotation substitution is well-behaved with respect to the subtyping relation, a property that will be useful to prove a similar result for typings in $\text{NLL}^{\forall\le}$.

Proposition 5.3.2
If $\Theta \vdash \phi \le \psi$ *and* $\Theta' \triangleright \Theta$*, then* $\Theta' \vdash \phi \le \psi$

Proof. Easy induction on the structure of ϕ. □

Proposition 5.3.3
If $\Theta \vdash \phi \le \psi$*, then* $\Theta[\vartheta] \vdash \phi[\vartheta] \le \psi[\vartheta]$.

Proof. Easy induction on the structure of ϕ. □

Lemma 5.3.4
If $\Theta, \Theta' \vdash \phi \le \psi$ *and* $\Theta \triangleright \Theta'[\vartheta]$*, then* $\Theta \vdash \phi[\vartheta] \le \psi[\vartheta]$*, where* $dom(\vartheta) = FA(\Theta')\backslash FA(\Theta)$.

Proof. By induction on the structure of ϕ.

- $\phi \equiv \mathbb{G}$.

  The result follows trivially by the definition of subtyping, since $\mathbb{G}[\vartheta] = \mathbb{G}$.

- $\phi \equiv {\phi_1}^{t_1} \multimap \phi_2$.

  We must have $\Theta, \Theta' \vdash {\phi_1}^{t_1} \multimap \psi_1 \le {\phi_2}^{t_2} \multimap \psi_2$ because $\Theta, \Theta' \vdash \phi_2 \le \phi_1$, $\Theta, \Theta' \vdash \psi_1 \le \psi_2$ and $\Theta, \Theta' \triangleright t_1 \sqsubseteq t_2$.

  Assuming $\Theta \triangleright \Theta'[\vartheta]$, by the induction hypothesis, twice, it follows that $\Theta \vdash \phi_2[\vartheta] \le \phi_1[\vartheta]$ and $\Theta \vdash \psi_1[\vartheta] \le \psi_2[\vartheta]$ must hold. Also, $\Theta \triangleright t_1[\vartheta] \sqsubseteq t_2[\vartheta]$ can be deduced from the fact that if $\Theta, \Theta' \triangleright P$ holds, for any predicate P, $\Theta' \triangleright P[\vartheta]$ must also hold, provided that $\Theta \triangleright \Theta'[\vartheta]$ and $dom(\vartheta) = FA(\Theta')\backslash FA(\Theta)$. The required conclusion, $\Theta \vdash ({\phi_1}^{t_1} \multimap \psi_1)[\vartheta] \le ({\phi_2}^{t_2} \multimap \psi_2)[\vartheta]$, clearly follows from the definition of annotation substitution.

- $\phi \equiv \forall\overline{\mathsf{p}}\,|\,\Theta''.\phi_1$.

  In this case, we must have $\Theta, \Theta' \vdash \forall\overline{\mathsf{p}}\,|\,\Theta''.\phi_1 \le \forall\overline{\mathsf{p}}\,|\,\Theta''.\phi_2$ because $\Theta, \Theta', \Theta'' \vdash \phi_1 \le \phi_2$, where $\overline{\mathsf{p}} \not\subseteq FA(\Theta, \Theta')$.

  By Proposition 5.3.3, we have $\Theta[\vartheta], \Theta'[\vartheta], \Theta''[\vartheta] \vdash \phi_1[\vartheta] \le \phi_2[\vartheta]$ must hold. From the fact that $\Theta[\vartheta] = \Theta$ (since $dom(\vartheta) \not\subseteq FA(\Theta)$) and assuming $\Theta \triangleright \Theta'[\vartheta]$, we deduce $\Theta, \Theta''[\vartheta] \triangleright \Theta[\vartheta], \Theta'[\vartheta], \Theta''[\vartheta]$. Then, by constraint strenghening (Proposition 5.3.2), $\Theta, \Theta''[\vartheta] \vdash \phi_1[\vartheta] \le \phi_2[\vartheta]$ must also hold. Because $\overline{\mathsf{p}} \not\subseteq FA(\Theta, \Theta')$ is a condition hypothesis, we can conclude, by definition of subtyping, that $\Theta \vdash (\forall\overline{\mathsf{p}}\,|\,\Theta''.\phi_1)[\vartheta] \le (\forall\overline{\mathsf{p}}\,|\,\Theta''.\phi_2)[\vartheta]$.

5.4 Type-theoretic properties

We shall now list some type-theoretic properties of interest. First of all, we extend the erasure ($^\circ$) mapping of Section 3.2 to the new types in the obvious way. In particular, we should have

$$(\forall \overline{\mathsf{p}_i} \,|\, \Theta.\phi)^\circ \stackrel{\text{def}}{=} \phi^\circ \qquad (\Lambda \overline{\mathsf{p}_i} \,|\, \Theta.M)^\circ \stackrel{\text{def}}{=} (M\ \vartheta)^\circ \stackrel{\text{def}}{=} M^\circ.$$

The following typing soundness proposition states that well-typed typing judgments correspond to well-typed typing judgments in the source language.

Proposition 5.4.1
If $\Theta\,;\Gamma \underset{\text{NLL}^{\forall\leq}}{\vdash} M : \phi$, *then* $\Gamma^\circ \underset{\text{FPL}}{\vdash} M^\circ : \phi^\circ$. □

Also, any reductions in the extended intermediate language correspond to legal reductions in the source language.

Proposition 5.4.2
For any two preterms M *and* N, $M \to N$ *implies* $M^\circ \to N^\circ$ *or* $M^\circ = N^\circ$. □

The special case $M^\circ = N^\circ$ arises whenever a Λ-redex is reduced, since $((\Lambda \overline{\mathsf{p}_i} \,|\, \Theta.M)\ \vartheta)^\circ = M[\vartheta]^\circ = M^\circ$.

As far as typings are concerned, it is clear that $\text{NLL}^{\forall\leq}$ is a conservative extension of $\text{NLL}^{\leq}$.

Proposition 5.4.3
If $\Gamma \underset{\text{NLL}^{\leq}}{\vdash} M : \sigma$, *then* $-\,;\Gamma \underset{\text{NLL}^{\forall\leq}}{\vdash} M : \sigma$. □

Moreover, the fragment of $\text{NLL}^{\forall\leq}$ restricted to $\text{NLL}^{\leq}$ terms and contexts (i.e., without free annotation parameters and quantification) proves the same typings as $\text{NLL}^{\leq}$ does.

Lemma 5.4.4
For simple Γ *and* M, *if* $-\,;\Gamma \underset{\text{NLL}^{\forall\leq}}{\vdash} M : \sigma$, *then* $\Gamma \underset{\text{NLL}^{\leq}}{\vdash} M : \sigma$. □

As it was the case for the constrained subtyping relation we introduced earlier, the constrained typing judgments of the generalised theory relate to constraint strengthening and substitution as stated below.

Proposition 5.4.5
If $\Theta\,;\Gamma \vdash M : \phi$, *then* $\Theta[\vartheta]\,;\Gamma[\vartheta] \vdash M[\vartheta] : \phi[\vartheta]$

Proof. Easy induction on the derivation of $\Theta\,;\Gamma \vdash M : \phi$. □

Lemma 5.4.6 (Constraint Strengthening)
If $\Theta\,;\Gamma \vdash M : \phi$ *and* $\Theta' \triangleright \Theta$, *then* $\Theta'\,;\Gamma \vdash M : \phi$.

Proof. By induction on the derivation of $\Theta\,;\Gamma \vdash M : \phi$. Its proofs depends on the fact that if $\Theta \triangleright P$ holds for a predicate P, and $\Theta' \triangleright \Theta$, then $\Theta' \triangleright P$ also holds. □

For the correctness proofs of the annotation inference algorithms we shall be looking at in the next chapter, we shall repeatedly make use of the following trivial corollary of the above lemma.

Proposition 5.4.7
If $\Theta\,;\Gamma \vdash M : \phi$*, then* $\Theta, \Theta'\,;\Gamma \vdash M : \phi$*.*

Proof. From Lemma 5.4.6 and the fact that $\Theta, \Theta' \triangleright \Theta$. □

The correctness of NLL$^{\forall\leq}$ also depends on the following important property of annotation substitutions, which we have already encountered as a property of subtyping judgments in the form of Lemma 5.3.4, and which says that any valid typing judgments obtained by replacing the annotation parameters with a set of annotation values that satisfy the requirements in Θ are also valid.

Lemma 5.4.8 (Annotation Substitution)
The following is an admissible rule.

$$\frac{\Theta, \Theta'\,;\Gamma \vdash M : \phi \quad \Theta \triangleright \Theta'[\vartheta] \quad dom(\vartheta) = FA(\Theta')\backslash FA(\Theta)}{\Theta\,;\Gamma[\vartheta] \vdash M[\vartheta] : \phi[\vartheta]}\ \vartheta\text{-Substitution}$$

Proof. By induction on the derivation of $\Theta, \Theta'\,;\Gamma \vdash M : \phi$. We prove the lemma for the the base case, including the less obvious inductive cases. Assume in each case that $\Theta \triangleright \Theta'[\vartheta]$.

- $$\frac{}{\Theta, \Theta'\,; x : \phi^t \vdash x : \phi}$$

  Trivial, since $\Theta\,; x : \phi[\vartheta]^{t[\vartheta]} \vdash x : \phi[\vartheta]$ is a valid typing judgment.

- $$\frac{\Theta, \Theta', \Theta''\,;\Gamma \vdash M : \phi \quad \overline{\mathsf{p}_i} \not\subseteq FA(\Theta, \Theta'\,;\Gamma) \quad \Theta''\backslash\overline{\mathsf{p}_i} = \emptyset}{\Theta\,;\Gamma \vdash \Lambda\overline{\mathsf{p}_i}\,|\,\Theta''.M : \forall\overline{\mathsf{p}_i}\,|\,\Theta''.\phi}$$

  From $\Theta, \Theta', \Theta''\,;\Gamma \vdash M : \phi$, it follows that $\Theta[\vartheta], \Theta'[\vartheta], \Theta''[\vartheta]\,;\Gamma[\vartheta] \vdash M[\vartheta] : \phi[\vartheta]$ by Proposition 5.4.5. From the fact that $\Theta = \Theta[\vartheta]$, since $dom(\vartheta)$ does not include $FA(\Theta)$ by assumption and $\Theta \triangleright \Theta'[\vartheta]$, we can deduce $\Theta[\vartheta], \Theta''[\vartheta] \triangleright \Theta[\vartheta], \Theta'[\vartheta], \Theta''[\vartheta]$, so $\Theta, \Theta''[\vartheta]\,;\Gamma[\vartheta] \vdash M[\vartheta] : \phi[\vartheta]$ must hold by constraint strengthening (Lemma 5.4.6). Applying $\forall_{\mathcal{I}}$, we may finally conclude $\Theta\,;\Gamma[\vartheta] \vdash \Lambda\overline{\mathsf{p}_i}\,|\,\Theta''[\vartheta].M[\vartheta] : \forall\overline{\mathsf{p}_i}\,|\,\Theta''[\vartheta].\phi[\vartheta]$. Notice that we have $\Lambda\overline{\mathsf{p}_i}\,|\,\Theta''[\vartheta].M[\vartheta] = (\Lambda\overline{\mathsf{p}_i}\,|\,\Theta''.M)[\vartheta]$ and $\forall\overline{\mathsf{p}_i}\,|\,\Theta''[\vartheta].\phi[\vartheta] = (\forall\overline{\mathsf{p}_i}\,|\,\Theta''.\phi)[\vartheta]$ as required, since $\vartheta \upharpoonright \overline{\mathsf{p}_i} = \vartheta$. (We have also implicitly assumed, without loss of generality, that $\mathsf{p}_i \notin img(\vartheta)$, by α-equivalence.)

- $$\frac{\Theta, \Theta', \Theta''\,;\Gamma \vdash M : \forall\overline{\mathsf{p}_i}\,|\,\Theta''.\phi \quad \Theta, \Theta' \triangleright \Theta''[\vartheta'] \quad dom(\vartheta') = \overline{\mathsf{p}_i}}{\Theta, \Theta'\,;\Gamma \vdash M\ \vartheta' : \phi[\vartheta']}$$

  By the induction hypothesis, we must have $\Theta, \Theta', \Theta''\,;\Gamma[\vartheta] \vdash M[\vartheta] : (\forall\overline{\mathsf{p}_i}\,|\,\Theta''.\phi)[\vartheta]$. Assuming by α-equivalence that $\overline{\mathsf{p}_i}$ do not occur anywhere outside $\forall\overline{\mathsf{p}_i}\,|\,\Theta''.\phi$, we can safely suppose that $(\forall\overline{\mathsf{p}_i}\,|\,\Theta''.\phi)[\vartheta] = \forall\overline{\mathsf{p}_i}\,|\,\Theta''[\vartheta].\phi[\vartheta]$. Notice that from the assumption $\Theta, \Theta' \triangleright \Theta''[\vartheta']$ it must follow that $\Theta[\vartheta], \Theta'[\vartheta] \triangleright \Theta''[\vartheta' \circ \vartheta]$, and so $\Theta \triangleright \Theta''[\vartheta' \circ \vartheta]$ must be the case by constraint strengthening, since $\Theta = \Theta[\vartheta]$ (as shown above) and $\Theta \triangleright \Theta'[\vartheta]$ by assumption. Applying $\forall_{\mathcal{E}}$, we obtain the required conclusion, $\Theta\,;\Gamma[\vartheta] \vdash M[\vartheta]\ (\vartheta' \circ \vartheta) : \phi[\vartheta' \circ \vartheta]$. We note that, by definition of substitution, $M[\vartheta]\ (\vartheta' \circ \vartheta) = (M\ \vartheta')[\vartheta]$ and $= \phi[\vartheta' \circ \vartheta] = (\phi[\vartheta'])[\vartheta]$.

□

$$\frac{\Theta\,;\Gamma_1 \vdash M : {\phi_1}^t \multimap \psi \quad \Theta\,;\Gamma_2 \vdash N : \phi_2 \quad \Theta \triangleright |\Gamma_2| \sqsupseteq t \quad \Theta \vdash \phi_2 \leq \phi_1}{\Theta\,;\Gamma_1, \Gamma_2 \vdash MN : \psi}\multimap_{\mathcal{E}}$$

$$\frac{\Theta\,;\Gamma_1 \vdash M : {\phi_1}^{t_1} \otimes {\phi_2}^{t_2} \quad \Theta\,;\Gamma_2, x_1 : {\psi_1}^{t'_1}, x_2 : {\psi_2}^{t'_2} \vdash N : \psi \quad \begin{array}{c}\Theta \triangleright t_i \sqsupseteq t'_i \\ \Theta \vdash \phi_i \leq \psi_i \\ (i = 1, 2)\end{array}}{\Theta\,;\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ \langle x_1, x_2 \rangle^{t'_1, t'_2} = M\ \mathsf{in}\ N : \psi}\otimes_{\mathcal{E}}$$

$$\frac{\Theta\,;\Gamma_1 \vdash M : \mathsf{bool} \quad \Theta\,;\Gamma_2 \vdash N_1 : \phi_1 \quad \Theta\,;\Gamma_2 \vdash N_2 : \phi_2 \quad \Theta \triangleright \phi = \phi_1 \sqcup \phi_2}{\Theta \triangleright \Gamma_1, \Gamma_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \phi}\text{Conditional}$$

Figure 5.5: Modified rules for $\mathrm{NLL}^{\forall\mu\leq}$

We note that if $\Gamma[\vartheta]$ and $M[\vartheta]$ are simple, then $\Gamma[\vartheta] \vdash M[\vartheta] : \phi[\vartheta]$ is, by Lemma 5.4.4, also valid in $\mathrm{NLL}^{\leq}$.

The following proposition generalises the Annotation Weakening property to typing judgments containing arbitrary annotation terms.

Proposition 5.4.9 (Annotation Weakening)
The following rule is admissible.

$$\frac{\Theta\,;\Gamma, x : \phi^t \vdash M : \psi \quad \Theta \triangleright t \sqsubseteq t'}{\Theta\,;\Gamma, x : \phi^{t'} \vdash M : \psi}\text{Transfer}$$

□

5.4.1 Minimum typings

The new syntax introduced to deal with (general) annotation polymorphism ensures that typings remain unique for the system without subtyping.

Proposition 5.4.10 (Unique Typing)
If $\Theta\,;\Gamma \underset{\mathrm{NLL}^{\forall}}{\vdash} M : \phi$ *and* $\Theta\,;\Gamma \underset{\mathrm{NLL}^{\forall}}{\vdash} M : \psi$, *then* $\phi \equiv_\alpha \psi$. □

For the system with subtyping, we can prove a Minimum Typing property, as we did for our monomorphic linearity analysis in Section 4.3. We therefore introduce a related type system of minimum types, called $\mathrm{NLL}^{\forall\mu\leq}$, and state the following three basic lemmas, following our previous presentation for $\mathrm{NLL}^{\mu\leq}$.

As before, $\mathrm{NLL}^{\forall\mu\leq}$ is obtained from $\mathrm{NLL}^{\forall\leq}$ by dropping the Subsumption rule and by replacing the elimination rules by the ones in Figure 5.5.

Lemma 5.4.11
If $\Theta\,;\Gamma \underset{\mathrm{NLL}^{\forall\mu\leq}}{\vdash} M : \phi$, *then* $\Theta\,;\Gamma \underset{\mathrm{NLL}^{\forall\leq}}{\vdash} M : \phi$.

Proof. A straightforward adaptation of Lemma 4.3.1. □

Lemma 5.4.12 (Unique Typing)
If $\Theta\,;\Gamma \underset{\mathrm{NLL}^{\forall\mu\leq}}{\vdash} M : \phi$ *and* $\Theta\,;\Gamma \underset{\mathrm{NLL}^{\forall\mu\leq}}{\vdash} M : \psi$*, then* $\phi \equiv_\alpha \psi$*.*

Proof. Easy induction on the derivations of $\Theta\,;\Gamma \underset{\mathrm{NLL}^{\forall\mu\leq}}{\vdash} M : \phi$. □

Lemma 5.4.13 (Smaller Typing)
If $\Theta\,;\Gamma \underset{\mathrm{NLL}^{\forall\leq}}{\vdash} M : \phi$*, then* $\Theta\,;\Gamma \underset{\mathrm{NLL}^{\forall\mu\leq}}{\vdash} M : \psi$ *for some* ψ *with* $\Theta \vdash \psi \leq \phi$*.*

Proof. By induction on $\mathrm{NLL}^{\forall\leq}$ derivations of $\Theta\,;\Gamma \vdash M : \phi$ and a straightforward adaptation of Lemma 4.3.3. We show the cases corresponding to the quantification rules.

- $$\frac{\Theta,\Theta'\,;\Gamma \vdash M : \phi \quad \overline{\mathsf{p}_i} \not\subseteq FA(\Theta\,;\Gamma) \quad \Theta'\backslash\overline{\mathsf{p}_i} = \emptyset}{\Theta\,;\Gamma \vdash \Lambda\overline{\mathsf{p}_i}\,|\,\Theta'.M : \forall\overline{\mathsf{p}_i}\,|\,\Theta'.\phi}\,\forall_{\mathcal{I}}$$

  By the induction hypothesis, we have $\Theta,\Theta'\,;\Gamma \vdash M : \phi_0$ for some ϕ_0 satisfying $\Theta,\Theta' \vdash \phi_0 \leq \phi$. Since $\overline{\mathsf{p}_i} \not\subseteq FA(\Theta)$, we may conclude from $\forall_{\mathcal{I}}$ and the subtyping rule for quantified types that $\Theta\,;\Gamma \vdash \Lambda\overline{\mathsf{p}_i}\,|\,\Theta'.M : \forall\overline{\mathsf{p}_i}\,|\,\Theta'.\phi_0$ and $\forall\overline{\mathsf{p}}\,|\,\Theta'.\phi_0 \leq \forall\overline{\mathsf{p}}\,|\,\Theta'.\phi$, as required.

- $$\frac{\Theta\,;\Gamma \vdash M : \forall\overline{\mathsf{p}_i}\,|\,\Theta'.\phi \quad \Theta \triangleright \Theta'[\vartheta] \quad dom(\vartheta) = \overline{\mathsf{p}_i}}{\Theta\,;\Gamma \vdash M\ \vartheta : \phi[\vartheta]}\,\forall_{\mathcal{E}}$$

  By the induction hypothesis, we have $\Theta\,;\Gamma \vdash M : \psi$ with $\Theta \vdash \psi \leq \forall\overline{\mathsf{p}_i}\,|\,\Theta'.\phi$. By the definition of subtyping, we must have $\psi \equiv \forall\overline{\mathsf{p}_i}\,|\,\Theta'.\phi_0$, where $\Theta,\Theta' \vdash \phi_0 \leq \phi$. Since $\Theta \triangleright \Theta'[\vartheta]$, we can apply Lemma 5.4.8 in order to conclude $\Theta \vdash \phi_0[\vartheta] \leq \phi[\vartheta]$, as needed. Note that $\Theta\,;\Gamma \vdash M\ \vartheta : \phi_0[\vartheta]$ easily follows by $\forall_{\mathcal{E}}$.

□

Using these lemmas, we can prove the following Minimum Typing property, reasoning along the same lines of the proof of Theorem 4.3.4.

Theorem 5.4.14 (Minimum Typing)
If $\Theta\,;\Gamma \underset{\mathrm{NLL}^{\forall\leq}}{\vdash} M : \phi$*, then there exists* ψ *such that* $\Theta\,;\Gamma \underset{\mathrm{NLL}^{\forall\leq}}{\vdash} M : \psi$*, and, for every other* ϕ' *for which* $\Gamma \underset{\mathrm{NLL}^{\forall\leq}}{\vdash} M : \phi'$*, then* $\psi \leq \phi'$*.* □

5.4.2 Semantic correctness

As is the case for $\mathrm{NLL}^{\leq}$, the static information in terms is preserved across reductions. We shall follow the development of Section 4.4, so it suffices to prove the corresponding Substitution Lemma and Subject Reduction for $\mathrm{NLL}^{\forall\mu\leq}$.

As before, the proofs use $\mathrm{NLL}^{\forall\mu\leq\uplus}$, the syntax-directed version of $\mathrm{NLL}^{\forall\mu\leq}$. We have defined the merge operator $\uplus$ for simple types only—in Definition 3.4.1. We can easily update its definition to types with more complex annotations as follows.

Definition 5.4.15 (Context merge)
If Γ_1 and Γ_2 are two contexts, then $\Gamma_1 \uplus \Gamma_2$ is defined as the map

$$(\Gamma_1 \uplus \Gamma_2)(x) = \begin{cases} \Gamma_1(x), & \text{if } x \in dom(\Gamma_1)\text{, but } x \notin dom(\Gamma_2) \\ \Gamma_2(x), & \text{if } x \in dom(\Gamma_2)\text{, but } x \notin dom(\Gamma_1) \\ \phi^{t_1+t_2}, & \text{if } \Gamma_1(x) = \phi^{t_1} \text{ and } \Gamma_1(x) = \phi^{t_2} \end{cases}$$

for all $x \in dom(\Gamma_1) \cup dom(\Gamma_2)$. □

All the properties of the context merge operator of Proposition 3.4.2 hold, as these only depend on the general properties of the contraction operator $+$, not on what it actually does on annotation values.

We are now ready to state the following Substitution Lemma.

Lemma 5.4.16 (Substitution for $\mathrm{NLL}^{\forall\mu\leq\uplus}$**)**
The following rule is admissible.

$$\frac{\Theta\,;\Gamma_1, x : {\phi_1}^t \vdash M : \psi \quad \Theta\,;\Gamma_2 \vdash N : \phi_2 \quad \Theta \rhd |\Gamma_2| \sqsupseteq t \quad \Theta \vdash \phi_2 \leq \phi_1}{\Theta\,;\Gamma_1 \uplus \Gamma_2 \vdash M[N/x] : \psi}\ \text{Substitution}$$

Proof. By induction on the structure of M. This lemma is basically an adaptation of Lemma 3.5.6. □

Theorem 5.4.17 (Subject Reduction for $\mathrm{NLL}^{\forall\mu\leq}$**)**
If $\Theta\,;\Gamma \underset{\mathrm{NLL}^{\forall\mu\leq}}{\vdash} M : \phi$ *and* $M \to N$*, then* $\Theta\,;\Gamma \underset{\mathrm{NLL}^{\forall\mu\leq}}{\vdash} N : \phi$*.*

Proof. Easy induction on $\to$-derivations, and basically an adaptation of Theorem 3.5.7. The interesting case consists in showing how Λ-redex reductions preserve typings.

- $M \equiv (\Lambda\overline{\mathsf{p}_i}\,|\,\Theta'.M')\ \vartheta$ and $N \equiv M'[\vartheta]$.

  A derivation for M must have the following structure:

$$\cfrac{\cfrac{\Theta,\Theta'\,;\Gamma \vdash M' : \phi' \quad \overline{\mathsf{p}_i} \not\subseteq FA(\Theta\,;\Gamma) \quad \Theta' \backslash \overline{\mathsf{p}_i} = \emptyset}{\Theta\,;\Gamma \vdash \Lambda\overline{\mathsf{p}_i}\,|\,\Theta'.M' : \forall\overline{\mathsf{p}_i}\,|\,\Theta'.\phi'}\ \forall_{\mathcal{I}} \qquad \begin{array}{c}\Theta \rhd \Theta'[\vartheta] \\ dom(\vartheta) = \overline{\mathsf{p}_i}\end{array}}{\Theta\,;\Gamma \vdash (\Lambda\overline{\mathsf{p}_i}\,|\,\Theta'.M)\ \vartheta : \phi'[\vartheta]}\ \forall_{\mathcal{E}}$$

  From $\Theta,\Theta'\,;\Gamma \vdash M' : \phi'$ and $\Theta \rhd \Theta'[\vartheta]$, it follows by Lemma 5.4.8 that $\Theta\,;\Gamma[\vartheta] \vdash M'[\vartheta] : \phi'[\vartheta]$. We can finally deduce $\Theta\,;\Gamma \vdash M'[\vartheta] : \phi'[\vartheta]$ from the fact that $\Gamma[\vartheta] = \Gamma$ since $dom(\vartheta) = \overline{\mathsf{p}_i}$ and $\overline{\mathsf{p}_i} \not\subseteq FA(\Gamma)$.

□

5.4.3 A word on contextual analysis

Much like type polymorphism, annotation polymorphism allows a term to be assigned different types for different contexts. These types are related to one another in the sense that they all belong to the same type family. Each type in the family corresponds to a structural assertion—a valid statement about the structural behaviour of the term. As we argued in the introduction, without annotation polymorphism, we would be obliged to choose the *weakest* of the structural assertions (structural properties) that is compatible with all the contexts in which the term is used. In the worst case, the weakest property is the property that gives no information at all (i.e., decorated with $\top$ everywhere).

It would not therefore be wrong to say that a polymorphic static analysis is, in some degree, *context-independent*. It is useful to think of a context as having the active role of *picking*, from the properties available, the one that best suits its purposes (or, in technical terms, the

$$(\lambda x{:}\phi^1.M)N \overset{\text{inl}}{\rightsquigarrow} M[N/x]$$
$$\mathsf{let}\ \langle x_1, x_2\rangle^{1,1} = \langle M_1, M_2\rangle^{t_1,t_2}\ \mathsf{in}\ N \overset{\text{inl}}{\rightsquigarrow} N[M_1/x_1][M_2/x_2]$$
$$\mathsf{let}\ \langle x_1, x_2\rangle^{1,t} = \langle M_1, M_2\rangle^{t_1,t_2}\ \mathsf{in}\ N \overset{\text{inl}}{\rightsquigarrow} \mathsf{let}\ x_2 = M_2\ \mathsf{in}\ N[M_1/x_1]$$
$$\mathsf{let}\ \langle x_1, x_2\rangle^{t,1} = \langle M_1, M_2\rangle^{t_1,t_2}\ \mathsf{in}\ N \overset{\text{inl}}{\rightsquigarrow} \mathsf{let}\ x_1 = M_1\ \mathsf{in}\ N[M_2/x_2]$$
$$\mathsf{let}\ x{:}\phi^1 = M\ \mathsf{in}\ N \overset{\text{inl}}{\rightsquigarrow} N[M/x]$$
$$(\Lambda\overline{\mathsf{p}_i} \mid \Theta.M)\ \vartheta \overset{\text{inl}}{\rightsquigarrow} M[\vartheta]$$

Figure 5.6: Final version of the inlining relation

strongest structural assertion that satisfies the annotation restrictions). In the sequel, we use the term *contextual analysis* to refer to a static analysis of structural properties that uses annotation polymorphism to achieve the degree of independence needed. Otherwise, we shall employ the term *non-contextual analysis*.

5.4.4 Inlining revisited again

We shall complete the specification of the rewrite rules for the inlining transformation we introduced in Subsection 3.7.1 for the case involving Λ-redexes, which may contain important static information for the optimiser, but in an 'indirect' form.

As a simple illustrative example, consider the following inlining sequence:

$$\begin{aligned}&(\lambda x{:}(\mathsf{int}^1 \otimes \mathsf{int}^1)^1.\mathsf{let}\ \langle y_1, y_2\rangle^{1,1} = x\ \mathsf{in}\ y_1 + y_2)\ (\Lambda\mathsf{p}_1, \mathsf{p}_2.\langle 2, 5\rangle^{\mathsf{p}_1,\mathsf{p}_2})\ 1\ 1)\\ &\overset{\text{inl}}{\rightsquigarrow} \mathsf{let}\ \langle y_1, y_2\rangle^{1,1} = \underline{(\Lambda\mathsf{p}_1, \mathsf{p}_2.\langle 2, 5\rangle^{\mathsf{p}_1,\mathsf{p}_2})\ 1\ 1)}\ \mathsf{in}\ y_1 + y_2.\end{aligned}$$

It is easy to see that inlining cannot proceed unless we reduce the polymorphic pair (shown underlined) first. This observation calls for an update of the inlining transformation of Figure 4.2, by adopting Λ-redex reduction as a rewrite rule. The final version of the rewrite rules is shown in Figure 5.6. Its correctness should be clear for the reasons outlined in Subsection 3.7.1.

Applying the new rewrite rules, inlining can proceed as expected:

$$\begin{aligned}&\mathsf{let}\ \langle y_1, y_2\rangle^{1,1} = \underline{(\Lambda\mathsf{p}_1, \mathsf{p}_2.\langle 2, 5\rangle^{\mathsf{p}_1,\mathsf{p}_2})\ 1\ 1)}\ \mathsf{in}\ y_1 + y_2\\ &\overset{\text{inl}}{\rightsquigarrow} \mathsf{let}\ \langle y_1, y_2\rangle^{1,1} = \langle 2, 5\rangle^{1,1}\ \mathsf{in}\ y_1 + y_2\\ &\overset{\text{inl}}{\rightsquigarrow} 2 + 5.\end{aligned}$$

As the example shows, the extra expressivity gained does not come completely for free. There is a price to pay in the form of a more expensive inlining algorithm that constructs instances of generalised terms 'on the fly'[7].

[7]The example shows that augmenting the accuracy of the analysis leads to a more complex instrumented

5.5 Towards modular linearity analysis

The first prototype of linearity analysis we have been experimenting with implements a restricted form of annotation polymorphism, that we coined *let-based annotation polymorphism*. This restricted form of annotation polymorphism provides a simple (and elegant) framework we can use to derive appropriate annotation inference algorithms for modular languages, a problem that we discussed in some detail in the introduction.

5.5.1 Let-based annotation polymorphism

Much like ML-style type-parametric polymorphism [44], let-based annotation polymorphism is so called because it allows only local definitions, introduced using a construct similar to the let construct of ML, to be assigned generalised types. By our previous discussion, this enables each occurrence of a let-bound variable to have a different annotated type. We shall refer to the type system that introduces annotation polymorphism in this way as $\mathrm{NLL}^{\forall\mathrm{let}\leq}$.

A system like $\mathrm{NLL}^{\forall\mathrm{let}\leq}$ is useful to discuss at this stage, for many reasons.

- First of all, the strategy for inferring annotated types we shall describe for this restricted language will be the same as for modular languages: Both local and module definitions may be treated likewise.
- Secondly, as we shall later see, let-based annotation polymorphism can be implemented, surprisingly, as a simple extension of the annotation inference algorithm for $\mathrm{NLL}^{\leq}$. Also, it is the ideal setting on which to base an extension of the traditional Hindley/Milner type inference algorithm, used by many modern functional languages, of which ML is only an example. (The interested reader is referred to [36] for a detailed description of the algorithm, as well as for any related historical background.)
- Finally, let-based polymporphism seems to constitute a good trade-off between the expressivity gained by introducing contextual analysis into the picture (although in a rather 'controlled' way) and the complexity needed to deal with the extra syntax, as well as the size of the constraint sets involved[8].

5.5.2 Retricted quantification rules

As far as the syntax is concerned, $\mathrm{NLL}^{\forall\mathrm{let}\leq}$ distinguishes between 'standard' types, ranged over by φ and ϱ, and which may not contain any quantifiers, and generalised types, which are considered separately and called in this context *annotated type schemes*. The syntax of types is summarised as follows:

Types	$\varphi ::= \mathbb{G} \mid \varphi^t \multimap \varphi \mid \varphi^t \otimes \varphi^t$
Annotated type schemes	$\forall \overline{\mathsf{p}_i} \mid \Theta.\varphi$

In an annotated type scheme, Θ stands, as usual, for a set of constraints.

(intermediate) language. To obtain the structural information it needs, the optimiser must partially reduce intermediate terms at compile time, which is what static analysis by type inference is supposed to avoid (besides the computation of fixpoints). Ultimately, if we are not careful enough, we may end up with an instrumentation complexity comparable to that obtained through abstract interpretation.

[8] The author does not personally think that type systems enabling the full power of annotation polymorphism would prove useful in practice, although, naturally, this still remains to be seen.

$$\frac{\Theta \triangleright \Theta'[\vartheta] \quad dom(\vartheta) = \overline{\mathsf{p}_i}}{\Theta\,;x : (\forall\overline{\mathsf{p}_i} \,|\, \Theta'.\varphi)^t \vdash x\;\vartheta : \varphi[\vartheta]}\ \text{Identity}^{\forall}$$

$$\frac{\Theta,\Theta'\,;\Gamma_1 \vdash M : \varphi \quad \Theta\,;\Gamma_2, x : (\forall\overline{\mathsf{p}_i} \,|\, \Theta'.\varphi)^t \vdash N : \varrho \quad \begin{array}{c}\overline{\mathsf{p}_i} \not\subseteq FA(\Theta\,;\Gamma_1)\\ \Theta'\backslash\overline{\mathsf{p}_i} = \emptyset\\ \Theta \triangleright |\Gamma_1| \sqsupseteq t\end{array}}{\Theta\,;\Gamma_1,\Gamma_2 \vdash \mathsf{let}\ x{:}(\forall\overline{\mathsf{p}_i} \,|\, \Theta'.\varphi)^t = \Lambda\overline{\mathsf{p}_i} \,|\, \Theta'.M\ \mathsf{in}\ N : \varrho}\ \text{Let}$$

Figure 5.7: Restricted quantification rules for $\text{NLL}^{\forall\text{let}\leq}$

The syntax of preterms and typing rules are those of $\text{NLL}^{\forall\leq}$ (where we should be careful to replace ϕ and ψ by φ and ϱ, respectively), except that general annotation polymorphism is introduced using the following constructs, which must be typed according to the rules shown in Figure 5.7:

$\mathsf{let}\ x{:}(\forall\overline{\mathsf{p}_i} \,|\, \Theta.\varphi)^t = \Lambda\overline{\mathsf{p}_i} \,|\, \Theta.M\ \mathsf{in}\ N$	(Generalised) let
$x\;\vartheta$	Let-bound variable specialisation

Although we have not made the distinction syntactically, only let-bound variables may be applied to annotation substitutions. Notice that we have written $\Lambda\overline{\mathsf{p}_i} \,|\, \Theta.M$ for the definition bound to x in the let construct, to suggest that definitions may be given annotated polymorphic types.

The correctness of the type system follows from the fact that, by construction, $\text{NLL}^{\forall\text{let}\leq}$ is a conservative extension of $\text{NLL}^{\forall\leq}$, and the following proposition, which shows that the Identity$^{\forall}$ and Let rules are derivable in $\text{NLL}^{\forall\leq}$.

Proposition 5.5.1 (Soundness of $\text{NLL}^{\forall\text{let}\leq}$)
$\Theta\,;\Gamma \underset{\text{NLL}^{\forall\text{let}\leq}}{\vdash} M : \varphi$ *implies* $\Theta\,;\Gamma \underset{\text{NLL}^{\forall\leq}}{\vdash} M : \varphi$.

Proof. Immediate from consideration of the following derivations:

$$\frac{\dfrac{}{\Theta\,;x : (\forall\overline{\mathsf{p}_i} \,|\, \Theta'.\varphi)^t \vdash x : \forall\overline{\mathsf{p}_i} \,|\, \Theta'.\varphi}\ \text{Identity} \qquad \Theta \triangleright \Theta'[\vartheta] \quad dom(\vartheta) = \overline{\mathsf{p}_i}}{\Theta\,;x : (\forall\overline{\mathsf{p}_i} \,|\, \Theta'.\varphi)^t \vdash x\;\vartheta : \varphi[\vartheta]}\ \forall_{\mathcal{E}}$$

$$\frac{\dfrac{\Theta\,;\Gamma_2, x : (\forall\overline{\mathsf{p}_i} \,|\, \Theta'.\varphi)^t \vdash N : \varrho}{\Theta\,;\Gamma_2 \vdash \lambda x{:}(\forall\overline{\mathsf{p}_i} \,|\, \Theta'.\varphi)^t.N : (\forall\overline{\mathsf{p}_i} \,|\, \Theta'.\varphi)^t \multimap \varrho}\ {\multimap_{\mathcal{I}}} \qquad \dfrac{\Theta,\Theta'\,;\Gamma_1 \vdash M : \varphi}{\Theta\,;\Gamma_1 \vdash \Lambda\overline{\mathsf{p}_i} \,|\, \Theta'.M : \forall\overline{\mathsf{p}_i} \,|\, \Theta'.\varphi}\ \forall_{\mathcal{E}}}{\Theta\,;\Gamma_1,\Gamma_2 \vdash (\lambda x{:}(\forall\overline{\mathsf{p}_i} \,|\, \Theta'.\varphi)^t.N)\ (\Lambda\overline{\mathsf{p}_i} \,|\, \Theta'.M)}\ {\multimap_{\mathcal{E}}}$$

For the last derivation, we have used the fact that $\mathsf{let}\ x = M\ \mathsf{in}\ N$ is interpreted in $\text{NLL}^{\forall}$ as an abbreviation for $(\lambda x{:}\phi^t.M)\ N$, for suitable ϕ and t. Also, notice that the side-conditions of the Let rule (omitted in the derivation for reasons of space), ensure the applicability of the $\multimap_{\mathcal{E}}$ and $\forall_{\mathcal{E}}$ rules. $\square$

As an example, the following is a decoration of the example we discussed in Section 5.1:

$$\begin{array}{l}\mathsf{let\ apply} = \Lambda \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3 \,|\, \mathsf{p}_3 \sqsupseteq \mathsf{p}_1 . \lambda f{:}(\mathsf{int}^{\mathsf{p}_1} \multimap \mathsf{int})^{\mathsf{p}_2}.\lambda x{:}\mathsf{int}^{\mathsf{p}_3}.f\ x\ \mathsf{in}\\ \quad \mathsf{let\ inc} = \Lambda \mathsf{p}_4.\lambda x{:}\mathsf{int}^{\mathsf{p}_4}.x\ \mathsf{in}\\ \quad\quad \mathsf{let\ dup} = \Lambda \emptyset.\lambda x{:}\mathsf{int}^{\top}.x + x\ \mathsf{in}\\ \quad\quad\quad \mathsf{apply}_{1,1,1}\ \mathsf{inc}_1\ (\mathsf{apply}_{\top,1,\top}\ \mathsf{dup}_{\emptyset}\ 4),\end{array}$$

where we have used the following abbreviations:

$$\begin{aligned}\mathsf{apply}_{a,b,c} &\stackrel{\mathrm{def}}{=} \mathsf{apply}\ \langle a/\mathsf{p}_1, b/\mathsf{p}_2, c/\mathsf{p}_3\rangle\\ \mathsf{inc}_a &\stackrel{\mathrm{def}}{=} \mathsf{inc}\ \langle a/\mathsf{p}_4\rangle\\ \mathsf{dup}_{\emptyset} &\stackrel{\mathrm{def}}{=} \mathsf{dup}\ \langle\rangle\end{aligned}$$

The decoration shown is not any decoration, but the optimal decoration, in the sense that all $\top$ annotations in the specialised terms are unavoidable.

Because annotation quantification is introduced in definitions only, a complete inlining strategy must be able to generate the necessary specialisations. This is easily achieved by replacing the general specialisation rule of Figure 5.6, by the following rewrite rule:

$$\mathsf{let}\ x{:}(\forall\overline{\mathsf{p}_i} \,|\, \Theta'.\varphi)^1 = \Lambda\overline{\mathsf{p}_i} \,|\, \Theta'.M\ \mathsf{in}\ N[x\ \vartheta] \stackrel{\mathrm{inl}}{\leadsto} N[M[\vartheta]],$$

where $N[x\ \vartheta]$ stands for the term N containing a single occurrence of the subterm $x\ \vartheta$, which gets replaced in the right-hand side by the specialised term $M[\vartheta]$.

5.6 Emulating the Subsumption rule

Another hint on the expressive power of general annotation polymorphism is given by the fact that any decoration of a source language term that requires the use of Subsumption can be substituted by an alternative decoration that does not require it, but that instead 'emulates' it using the tools provided by general annotation polymorphism.

From the point of view of static analysis, for the two decorations to have the same 'value', it is necessary that they convey (in the terms) the same static information. The basic idea will consist in showing that for any decoration M_1 in NLL$^{\leq}$, say of type σ, it is possible to construct a NLL$^{\forall}$ decoration M_2 of type ϕ, where σ arises as an instance of ϕ. (Therefore, ϕ contains the static information necessary to 'generate' σ.)

To be more precise about what we mean by 'instance', we shall use the notation $\phi \prec \psi$ to indicate that ϕ is a *type instance* of ψ. The relation $\prec$ can be defined as the reflexive contextual closure of the axiom rule[9]:

$$\frac{b \sqsupseteq a}{\phi[b/\mathsf{p}] \prec \forall \mathsf{p} \sqsupseteq a.\phi}$$

We begin by defining, in Figure 5.8, two functions on simple types, $(-)^\sharp$ and $(-)^\flat$, that we shall be needing for our main construction. Intuitively, if σ is a simple type (that is not a ground type), $\sigma^\sharp$ translates to a generalised type that has all supertypes of σ as its instances; and, conversely, $\sigma^\flat$ has all subtypes of σ as its instances.

[9] If $[\![\phi]\!]$ stands for the obvious interpretation of a type ϕ as a family (set) of simple types, the predicate $\sigma \prec \phi$ is nothing more but a synonym for $\sigma \in [\![\phi]\!]$.

$$\mathbb{G}^\sharp \stackrel{\text{def}}{=} \mathbb{G}^\flat \stackrel{\text{def}}{=} \mathbb{G}$$

$$(\sigma^a \multimap \tau)^\sharp \stackrel{\text{def}}{=} \forall \mathsf{p}.(\sigma^\flat)^t \multimap \tau^\sharp, \quad \text{where } t \equiv \begin{cases} \mathsf{p}, & \text{if } a \equiv 1; \\ \top, & \text{if } a \equiv \top \end{cases}$$

$$(\sigma^a \multimap \tau)^\flat \stackrel{\text{def}}{=} \forall \mathsf{p}.(\sigma^\sharp)^t \multimap \tau^\flat, \quad \text{where } t \equiv \begin{cases} 1, & \text{if } a \equiv 1; \\ \mathsf{p}, & \text{if } a \equiv \top \end{cases}$$

$$({\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2})^\sharp \stackrel{\text{def}}{=} \forall \mathsf{p}_1, \mathsf{p}_2.({\sigma_1}^\sharp)^{t_1} \otimes ({\sigma_2}^\sharp)^{t_2}, \quad \text{where } t_i \equiv \begin{cases} 1, & \text{if } a_i \equiv 1; \\ \mathsf{p}_i, & \text{if } a_i \equiv \top \end{cases} \text{ for } i = 1, 2$$

$$({\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2})^\flat \stackrel{\text{def}}{=} \forall \mathsf{p}_1, \mathsf{p}_2.({\sigma_1}^\flat)^{t_1} \otimes ({\sigma_2}^\flat)^{t_2}, \quad \text{where } t_i \equiv \begin{cases} \mathsf{p}_i, & \text{if } a_i \equiv 1; \\ \top, & \text{if } a_i \equiv \top \end{cases} \text{ for } i = 1, 2$$

Figure 5.8: Definition of $\sigma^\sharp$ and $\sigma^\flat$

The following proposition formally states the relationship between subtyping and the translations just defined.

Proposition 5.6.1
If $\sigma \leq \tau$*, then* $\tau \prec \sigma^{\sharp}$ *and* $\sigma \prec \tau^{\flat}$*.*

Proof. By induction on the structure of the derivation of $\sigma \leq \tau$. □

We define in Figures 5.9 and 5.10 a translation $(-)^{\dagger}$ transforming NLL$^{\leq}$ type derivations into NLL$^{\forall}$ type derivations. We only cover the cases for the $\{\multimap, \otimes\}$-fragment of the language. The other cases follow a similar pattern.

The correctness of the translation can be easily established by induction on the structure of an NLL$^{\leq}$ derivation.

Lemma 5.6.2 (Correctness)
$\Pi(\Gamma_1 \underset{\mathrm{NLL}^{\leq}}{\vdash} M_1 : \sigma)^{\dagger} = \Pi(-\,; \Gamma_2 \underset{\mathrm{NLL}^{\forall}}{\vdash} M_2 : \sigma^{\sharp})$.
Moreover, it is clear by construction that ${\Gamma_1}^{\circ} \equiv {\Gamma_2}^{\circ}$ *and* ${M_1}^{\circ} \equiv {M_2}^{\circ}$*.* □

The above lemma, and the fact that $\sigma \prec \sigma^{\sharp}$ by Proposition 5.6.1, justify the following statement.

Theorem 5.6.3 (Subsumption emulation)
If $\Gamma_1 \underset{\mathrm{NLL}^{\leq}}{\vdash} M_1 : \sigma$*, then there exists* Γ_2*,* M_2 *and* ϕ*, such that* ${\Gamma_1}^{\circ} \equiv {\Gamma_2}^{\circ}$ *and* ${M_1}^{\circ} \equiv {M_2}^{\circ}$ *and* $\sigma \prec \phi$*, for which* $\Gamma_2 \underset{\mathrm{NLL}^{\forall}}{\vdash} M_2 : \phi$*.* □

It is important to remark that the above theorem should not be taken to imply that subtyping is not useful. Not only it is quite helpful in practice, as it can be used to reduce the number of inequations and annotation parameters to be considered during annotation inference [67], but also because, without subtyping, any source language transformations based on η-reduction—as implied by Proposition 4.4.4—would be unsound!

$$\left(\frac{}{x : \sigma^a \vdash x : \sigma}\right)^\dagger \stackrel{\text{def}}{=} \frac{}{- \,;\, x : (\sigma^\sharp)^a \vdash x : \sigma^\sharp}$$

$$\left(\frac{\Pi(\Gamma, x : \sigma^a \vdash M : \tau)}{\Gamma \vdash \lambda x{:}\sigma^a.M : \sigma^a \multimap \tau}\right)^\dagger \stackrel{\text{def}}{=} \frac{\dfrac{\dfrac{\Pi(- \,;\, \Gamma', x : \phi^a \vdash M' : \psi)}{\mathsf{p} \sqsupseteq 1 \,;\, \Gamma', x : \phi^t \vdash M' : \psi}\,(*)}{\mathsf{p} \sqsupseteq 1 \,;\, \Gamma' \vdash \lambda x{:}\phi^t.M' : \phi^t \multimap \psi}}{- \,;\, \Gamma' \vdash \Lambda\mathsf{p}.\lambda x{:}\phi^t.M' : \forall\mathsf{p}.\phi^t \multimap \psi}$$

where $\Pi(\Gamma, x : \sigma^a \vdash M : \tau)^\dagger = \Pi(- \,;\, \Gamma', x : \phi^a \vdash M' : \psi)$ and $t = \begin{cases} \mathsf{p}, & \text{if } a \equiv 1; \\ \top, & \text{if } a \equiv \top. \end{cases}$. The step marked (*) is justified by Lemma 5.4.6 and Transfer.

$$\left(\frac{\Pi(\Gamma_1 \vdash M : \sigma^a \multimap \tau) \quad \Pi(\Gamma_2 \vdash N : \sigma)}{\Gamma_1, \Gamma_2 \vdash MN : \tau}\right)^\dagger \stackrel{\text{def}}{=} \frac{\dfrac{\Pi(- \,;\, \Gamma_1' \vdash M' : \forall\mathsf{p}.\phi^t \multimap \psi)}{- \,;\, \Gamma_1' \vdash M'\ a : \phi^a \multimap \psi} \qquad \Pi(- \,;\, \Gamma_2' \vdash N' : \phi)}{- \,;\, \Gamma_1', \Gamma_2' \vdash (M'\ a)\ N : \psi}$$

where $\Pi(\Gamma_1 \vdash M : \sigma^a \multimap \tau)^\dagger = \Pi(- \,;\, \Gamma_1' \vdash M' : \forall\mathsf{p}.\phi^t \multimap \psi)$ and $\Pi(\Gamma_2 \vdash N : \sigma)^\dagger = \Pi(- \,;\, \Gamma_2' \vdash N' : \phi)$.

Figure 5.9: Definition of $(-^\dagger)$ translation

$$\left(\dfrac{\Pi(\Gamma_1 \vdash M_1 : \sigma_1) \quad \Pi(\Gamma_2 \vdash M_2 : \sigma_2)}{\Gamma_1, \Gamma_2 \vdash \langle M_1, M_2 \rangle^{a_1, a_2} : {\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2}}\right)^{\dagger} \stackrel{\text{def}}{=} \dfrac{\dfrac{\dfrac{\Pi(- \,;\Gamma'_1 \vdash M'_1 : \phi_1)}{\mathsf{p}_1 \sqsupseteq 1, \mathsf{p}_2 \sqsupseteq 1\,; \Gamma'_1 \vdash M'_1 : \phi_1}\,(*) \qquad \dfrac{\Pi(- \,;\Gamma'_2 \vdash M'_2 : \phi_2)}{\mathsf{p}_1 \sqsupseteq 1, \mathsf{p}_2 \sqsupseteq 1\,; \Gamma'_2 \vdash M'_2 : \phi_2}\,(*)}{\mathsf{p}_1 \sqsupseteq 1, \mathsf{p}_2 \sqsupseteq 1\,; \Gamma'_1, \Gamma'_2 \vdash \langle M'_1, M'_2 \rangle^{t_1, t_2} : {\phi_1}^{t_1} \otimes {\phi_2}^{t_2}}}{- \,; \Gamma'_1, \Gamma'_2 \vdash \Lambda \mathsf{p}_1, \mathsf{p}_2 . \langle M'_1, M'_2 \rangle^{t_1, t_2} : \forall \mathsf{p}_1, \mathsf{p}_2 . {\phi_1}^{t_1} \otimes {\phi_2}^{t_2}}$$

where $\Pi(\Gamma_i \vdash M_i : \sigma_i)^{\dagger} = \Pi(- \,; \Gamma'_i \vdash M'_i : \phi_i)$ and $t_i = \begin{cases} 1, & \text{if } a_i \equiv 1; \\ \mathsf{p}_i, & \text{if } a_i \equiv \top, \end{cases}$ for $i = 1, 2$. The steps marked (*) are justified by Lemma 5.4.6.

$$\left(\dfrac{\Pi(\Gamma_1 \vdash M : {\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2}) \quad \Pi(\Gamma_2, x_1 : {\sigma_1}^{a_1}, x_2 : {\sigma_2}^{a_2} \vdash N : \tau)}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ \langle x_1, x_2 \rangle = M\ \mathsf{in}\ N : \tau}\right)^{\dagger} \stackrel{\text{def}}{=} \dfrac{\dfrac{\Pi(- \,; \Gamma'_1 \vdash M' : \forall \mathsf{p}_1, \mathsf{p}_2 . {\phi_1}^{t_1} \otimes {\phi_2}^{t_2})}{- \,; \Gamma'_1 \vdash M'\ a_1\ a_2 : {\phi_1}^{a_1} \otimes {\phi_2}^{a_2}} \qquad \Pi(- \,; \Gamma'_2, x_1 : {\phi_1}^{a_1}, x_2 : {\phi_2}^{a_2} \vdash N' : \psi)}{- \,; \Gamma'_1, \Gamma'_2 \vdash \mathsf{let}\ \langle x_1, x_2 \rangle = M'\ \mathsf{in}\ N' : \psi}$$

where $\Pi(\Gamma_1 \vdash M : {\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2})^{\dagger} = \Pi(- \,; \Gamma'_1 \vdash M' : \forall \mathsf{p}_1, \mathsf{p}_2 . {\phi_1}^{t_1} \otimes {\phi_2}^{t_2})$ and $\Pi(\Gamma_2, x_1 : {\sigma_1}^{a_1}, x_2 : {\sigma_2}^{a_2} \vdash N : \tau)^{\dagger} = \Pi(- \,; \Gamma'_2, x_1 : {\phi_1}^{a_1}, x_2 : {\phi_2}^{a_2} \vdash N' : \psi)$.

$$\left(\dfrac{\Pi(\Gamma \vdash M : \sigma)}{\Gamma \vdash M : \tau}\right)^{\dagger} \stackrel{\text{def}}{=} \Pi(- \,; \Gamma' \vdash M' : \phi) \qquad \text{if } \sigma \leq \tau,$$

where $\Pi(\Gamma \vdash M : \sigma)^{\dagger} = \Pi(- \,; \Gamma' \vdash M' : \phi)$.

Figure 5.10: Definition of $(-^{\dagger})$ translation (continued)

5.7 Adding type-parametric polymorphism

The language we have been using so far is monomorphic on base types. We shall now consider a type-parametric polymorphic version of the language, and prove the semantic correctness of the obtained intermediate language. The extension is standard, based on Girard's second order λ-calculus System F. (A detailed introduction may be found in [53, Part V].)

The correctness argument depends on proving a key syntactic lemma stating that decorations are invariant with respect to type substitution. This means that the problem of analysing the structural properties of type-parametric polymorphic is equivalent to (the simpler) problem of analysing the structural properties of any of its monomorphic instances.

5.7.1 Syntax and typing rules

For the discussion that follows, let ϕ and ψ range over the set of extended types, possibly containing some type parameters, collectively ranged over by α. Some of these parameters may be bound by a universal type quantifier, written as shown below:

$$\begin{array}{rcll} \phi & ::= & \text{(same as Subsection 5.2.1)} & \\ & | & \forall\alpha.\phi & \text{Type quantification} \end{array}$$

Likewise, we extend the syntax of terms, ranged over by M and N, with two new constructs corresponding to the mechanisms of type abstraction and application:

$$\begin{array}{rcll} M & ::= & \text{(same as Subsection 5.2.2)} & \\ & | & \Lambda\alpha.M & \text{Type abstraction} \\ & | & M\ \phi & \text{Type application} \end{array}$$

The new constructs are typed according to the following introduction and elimination rules:

$$\frac{\Gamma \vdash M : \phi \quad \alpha \notin \mathit{FTP}(\Gamma)}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\phi}\,\forall_{\mathcal{I}} \qquad \frac{\Gamma \vdash M : \forall\alpha.\phi}{\Gamma \vdash M\ \psi : \phi[\psi/\alpha]}\,\forall_{\mathcal{E}}$$

The set-valued function $\mathit{FTP}(\Gamma)$ returns the free type parameters occurring in the types in Γ. Let

$$\mathit{FTP}(\Gamma) = \mathit{FTP}(\Gamma^{\circ}),$$

where the latter is defined by

$$\mathit{FTP}(-) = \emptyset \quad \text{and} \quad \mathit{FTP}(\Gamma, x : \sigma) = \mathit{FTP}(\Gamma) \cup \mathit{FTP}(\sigma)$$

The set of free type parameters of a source type σ is inductively defined by the following equations:

$$\begin{aligned} \mathit{FTP}(\mathbb{G}) &= \emptyset \\ \mathit{FTP}(\sigma \to \tau) = \mathit{FTP}(\sigma \times \tau) &= \mathit{FTP}(\sigma) \cup \mathit{FTP}(\tau) \\ \mathit{FTP}(\forall\alpha.\sigma) &= \mathit{FTP}(\sigma)\backslash\{\alpha\} \end{aligned}$$

In order to take subtyping into account, we need to extend the subtyping relation with the following two rules:

$$\frac{}{\alpha \leq \alpha} \qquad \frac{\phi \leq \psi}{\forall\alpha.\phi \leq \forall\alpha.\psi} \tag{5.5}$$

We should also add the following reduction axiom, that takes care of type applications:

$$(\Lambda\alpha.M)\ \phi \rightarrow M[\phi/\alpha] \tag{5.6}$$

5.7.2 Correctness

Having introduced the syntax, we are now ready to state the following Type-substitution Invariance property.

Lemma 5.7.1 (Type-substitution Invariance)
If $\Theta\,;\Gamma \vdash M : \phi$, *then* $\Theta\,;\Gamma[\psi/\alpha] \vdash M[\psi/\alpha] : \phi[\psi/\alpha]$, *for any* ψ.

Proof. Easy induction on the derivation of $\Theta\,;\Gamma \vdash M : \phi$. □

It is then straightforward to prove that our extended intermediate language is semantically correct.

Theorem 5.7.2 (Subject Reduction)
If $\Theta\,;\Gamma \vdash M : \phi$ *and* $M \rightarrow N$, *then* $\Theta\,;\Gamma \vdash N : \phi$.

Proof. We only consider the following case:

- $M \equiv (\Lambda\alpha.M')\ \phi$ and $N \equiv M'[\phi/\alpha]$.

  A derivation for $(\Lambda\alpha.M')\ \phi$ must necessarily have the following structure:

$$\dfrac{\dfrac{\Pi(\Theta\,;\Gamma \vdash M' : \psi')}{\Theta\,;\Gamma \vdash \Lambda\alpha.M' : \forall\alpha.\psi'}\ \forall_{\mathcal{I}}}{\Theta\,;\Gamma \vdash (\Lambda\alpha.M')\ \phi : \psi'[\phi/\alpha]}\ \forall_{\mathcal{E}}$$

  Applying Lemma 5.7.1 to $\Theta\,;\Gamma \vdash M' : \psi'$, it immediately follows that $\Theta\,;\Gamma \vdash M'[\phi/\alpha] : \psi'[\phi/\alpha]$, as required. (Note that $\alpha \notin FTP(\Gamma)$.)

□

Chapter 6

Annotation inference

A key element in the formulation of any type-based static analysis is, undoubtedly, the type system itself. We have ensured that the different type theories of linearity analysis we have proposed in the previous chapters have the 'right' properties, thus setting the scene for the matter of discussion of the present chapter: *annotation inference algorithms*.

For the simple case of linearity analysis, an annotation inference algorithm is a computer program that takes as input a pair $\langle \Gamma, M \rangle$, comprising a source language context and term, and outputs another pair $\langle \Gamma^*, M^* \rangle$, where $- \,; \Gamma^* \vdash M^* : \phi$ is the $\mathrm{NLL}^{\leq}$ optimal decoration of the source typing $\Gamma \vdash M : \phi^\circ$ (and recalling that, in this case, $(M^*)^\circ = M$ and $(\Gamma^*)^\circ = \Gamma$). Notice that we do not refer to this algorithm as a 'type inference' algorithm, for the simple reason that we shall not be interested in algorithms that infer decorated types from terms carrying no type information at all, and which may possibly be ill-typed. We therefore assume that our algorithm takes as input a well-typed term, and that this term already carries base type information, as is the case for our prototypical functional language. Our algorithms therefore concentrate on the (simpler) task of inferring optimal annotations, leaving the task of inferring base type information to an early stage of the compilation process[1].

We shall have a look at two annotation inference algorithms, which we shall prove sound and complete with respect to their associated type theories. We begin by describing an annotation inference algorithm for $\mathrm{NLL}^{\leq}$, the theory of simple linearity analysis. We could have addressed this issue earlier, but the reason why we have waited until now has to do with the fact that we shall be 'reusing' some of the tools (and results) of part of the framework belonging to the type theory of annotation polymorphism. Using these same tools, we shall describe a second annotation inference algorithm, but this time based on $\mathrm{NLL}^{\forall\mathrm{let}\leq}$, to be able to assign families of annotated types to local definitions.

Based on the annotation inference algorithm for $\mathrm{NLL}^{\forall\mathrm{let}\leq}$, we shall describe a strategy of linearity analysis for definitions in modules.

6.0.3 A two-stage process

The annotation inference algorithms we discuss in this chapter are based on the idea that all the decorations of a source typing $\Gamma \vdash M : \sigma$ can be represented *within* our linear language extended with annotation polymorphism, as a typing $\Theta \,; \Gamma^* \vdash M^* : \phi$, where

[1] During the optimisation phase, it might be useful to perform several passes of annotation inference, so having a separate annotation inference algorithm is always handy.

- M^* and Γ^* contain only annotation parameters, and
- Θ contains the context restrictions guaranteeing that each substitution instance $\Gamma^*[\theta] \vdash M^*[\theta] : \phi[\theta]$ is a valid decoration.

In Subsection 5.2.7, we have provided a hint of this idea through the example of Figure 5.3: The context restrictions give rise to a number of inequations on annotation parameters, which, all together, trivially determine the 'minimum' set of inequations required to satisfy the context restrictions.

Therefore, annotation inference will be formulated as a two-stage process:

- Firstly, we infer the constraint inequations Θ necessary to find a type ϕ for an input well-typed context-term pair $\langle \Gamma, M \rangle$. The algorithm basically reconstructs the type derivation for a 'template' of M, M^*, containing only annotation parameters. The algorithm is driven by the structure of M^*, so this is where the syntax-directed version of $\mathrm{NLL}^{\forall\leq}$ comes into play.
- Secondly, we find the optimal solution of the obtained constraint set. As we shall see, this optimal solution always exists, and the substitution instance obtained is the meet of the decoration space of the input pair.

The process just described is how traditionally annotation inference is handled for annotated type systems [48, Chapter 5]. The only difference with other presentations is in the fact that we reason about $\mathrm{NLL}^{\leq}$ decorations and annotation inference using the tools of $\mathrm{NLL}^{\forall\leq}$.

6.0.4 Organisation

The contents of this chapter are organised as follows:

- Section 6.1 presents an algorithm for inferring constraint inequations for our simple linearity analysis. We prove the correctness of the algorithm by establishing soundness and completness results with respect to the decoration space of a given input typing.
- Section 6.2 discusses the possibility of finding the least solution of a constraint set using fixpoints. We shall also describe a simple graph-based algorithm for finding the optimal solution of a constraint set over our 2-point lattice of linearity properties.
- Section 6.3 presents an extension of the algorithm of Section 6.1 with let-based annotation polymorphism, and proves soundness and completeness.
- Section 6.4 applies the techniques developed in Section 6.3 to propose a strategy for modular linearity analysis.

6.1 Simple annotation inference

The notation we use to specify the legal *runs* (or executions) of the algorithm for inferring constraint inequations is the following:

$$\Theta\,; \Delta \vdash M \Rightarrow X : \phi,$$

where the context Δ and source language term M are the inputs of the algorithm, and the constraint set Θ, the intermediate language term X and type ϕ are the outputs. We introduce here the use of Δ and X to range over contexts and terms, respectively, that are allowed to contain only annotation parameters.

The basic idea is that if $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$ is a legal run of the algorithm, $\Theta\,;\Delta \vdash X : \phi$ is a valid $\mathrm{NLL}^{\forall\leq}$ typing judgment (Lemma 6.1.9). Moreover, all suitable substitution instances $\Delta[\theta] \vdash X[\theta] : \phi[\theta]$, for all $\theta \models \Theta$ denote all the valid decorations of $\Gamma \vdash M : \sigma$, where $\Delta^\circ = \Gamma$ and $X^\circ = M$, for some $\sigma = \phi^\circ$[2]. Thus, our correctness criteria is given by the following two conditions:

$$\begin{aligned}\{\Delta[\theta] \vdash X[\theta] : \phi[\theta] \mid \theta \models \Theta \text{ and } FA(X) \cup FA(\Delta) \subseteq dom(\theta)\} \\ = \mathfrak{D}_{\mathrm{NLL}^{\leq}}(\Delta^\circ \underset{\mathrm{FPL}}{\vdash} X^\circ : \phi^\circ)\end{aligned} \tag{6.1}$$

and

$$X^\circ = M. \tag{6.2}$$

Conditions (6.1) and (6.2) constitute our correctness criteria, which will be the matter of Subsection 6.1.2 (Theorems 6.1.10 and 6.1.11).

The arrow '$\Rightarrow$' naturally suggests the translation of the source language term M into an intermediate language term X of type ϕ. It would perhaps have been better to write Θ to the right of the arrow, and not to the left, but we have preferred a notation that recalls the typing judgments of our linear theory extended with annotation polymorphism, to remind the reader that they are both intimately related.

Definition 6.1.1 (Well-formed run)
An assertion $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$ determines a *well-formed run* if there is a proof of it, using the rules of Figures 6.3 and 6.4 on pages 119 and 120. □

As usual, the requirement 'p fresh' states that the annotation parameter p should not appear free anywhere else in the rule. Similarly, by $\phi = \mathit{fresh}(\sigma)$, we mean that ϕ corresponds to a type containing only annotation parameters, where each annotation parameter is fresh, and such that $\phi^\circ \equiv \sigma$.

The notation $(\phi_1 \leq \phi_2) = \Theta$ should be understood as stating that Θ is the result of a function taking two arguments, ϕ_1 and ϕ_2, comprising the inequations needed to make ϕ_1 a subtype of ϕ_2. Figure 6.1 provides a recursive definition of this function, following the structure of the types.

We state its correctness in a slightly more general fashion in the following proposition.

Proposition 6.1.2 (Correctness of $(- \leq -)$)
If $(\phi_1 \leq \phi_2) = \Theta$, then $\Theta \vdash \phi_1 \leq \phi_2$. Moreover, if σ_1 and σ_2 are any two types, where $\sigma_1 \leq \sigma_2$ with ${\sigma_1}^\circ \equiv {\phi_1}^\circ$ and ${\sigma_2}^\circ \equiv {\phi_2}^\circ$, then there exists $\theta \models \Theta$, such that $\sigma_1 \equiv \psi_1[\theta]$ and $\sigma_2 \equiv \psi_2[\theta]$.
□

An algorithm for inferring constraints suitable for NLL (i.e., without subtyping) can be easily obtained by modifying the definition of $(- \leq -)$; it suffices to add the 'mirror' inequation $\mathsf{q} \sqsupseteq \mathsf{p}$ along side each occurrence of $\mathsf{p} \sqsupseteq \mathsf{q}$, thus making the types equal.[3].

[2] Thus, the inferred constraint set Θ effectively contains the information necessary to generate the whole decoration space.

[3] In fact, this is precisely what our prototype implementation of linearity analysis does when the 'subtyping option' is disabled.

$$\frac{}{(\mathbb{G} \leq \mathbb{G}) = \emptyset}$$

$$\frac{(\phi_2 \leq \phi_1) = \Theta_1 \quad (\psi_1 \leq \psi_2) = \Theta_2}{({\phi_1}^{\mathsf{p}_1} \multimap \psi_1 \leq {\phi_2}^{\mathsf{p}_2} \multimap \psi_2) = \Theta_1, \Theta_2, \mathsf{p}_2 \sqsupseteq \mathsf{p}_1}$$

$$\frac{(\phi_1 \leq \phi_2) = \Theta_1 \quad (\psi_1 \leq \psi_2) = \Theta_2}{({\phi_1}^{\mathsf{p}_1} \otimes {\psi_1}^{\mathsf{q}_1} \leq {\phi_2}^{\mathsf{p}_2} \otimes {\psi_2}^{\mathsf{q}_2}) = \Theta_1, \Theta_2, \mathsf{p}_1 \sqsupseteq \mathsf{p}_2, \mathsf{q}_1, \sqsupseteq \mathsf{q}_2}$$

Figure 6.1: Generating subtyping constraints

$$split(-, M_1, M_2) = (-, -, \emptyset)$$

$$split((\Delta, x{:}\phi^{\mathsf{p}}), M_1, M_2) = \begin{cases} ((\Delta_1', x{:}\phi^{\mathsf{p}}), \Delta_2', \Theta), \\ \qquad \text{if } x \in FV(M_1), \text{ but } x \notin FV(M_2); \\ (\Delta_1', (\Delta_2', x{:}\phi^{\mathsf{p}}), \Theta), \\ \qquad \text{if } x \in FV(M_2), \text{ but } x \notin FV(M_1); \\ ((\Delta_1', x{:}\phi^{\mathsf{p}_1}), (\Delta_2', x{:}\phi^{\mathsf{p}_2}), (\Theta, \mathsf{p} \sqsupseteq \mathsf{p}_1 + \mathsf{p}_2)), \\ \qquad \text{otherwise;} \end{cases}$$

$$\text{and where } split(\Delta, M_1, M_2) = (\Delta_1', \Delta_2', \Theta).$$

Figure 6.2: A general definition of $split(-, -, -)$

The *split* function does the opposite of context merge; it takes as input a typing context Δ and two terms M_1 and M_2, and produces a triple $(\Delta_1, \Delta_2, \Theta)$ as output, where Θ contains the inequations needed to ensure that Δ is the merge of Δ_1 and Δ_2 (Definition 5.4.15). The typing contexts Δ_1 and Δ_2 are then used by our inference algorithm to reconstruct the type derivations of M_1 and M_2, respectively; so we must also ensure that each Δ_i contains, at least, the typing assertions for the free variables in its corresponding M_i.

The following definition states the properties that any definition of context splitting must satisfy.

Definition 6.1.3 (Properties of $split(-, -, -)$)
If $split(\Delta, M_1, M_2) = (\Delta_1, \Delta_2, \Theta)$, then

- $\Theta \triangleright \Delta \sqsupseteq \Delta_1 \uplus \Delta_2$; and
- $FV(M_1) \subseteq dom(\Delta_1)$ and $FV(M_2) \subseteq dom(\Delta_2)$.

It is not necessary to require that Δ be precisely $\Delta_1 \uplus \Delta_2$, although this will be true for the definition of *split* we shall be using for linearity analysis.

$$\frac{\Delta \equiv \overline{x_i : \phi_i{}^{\mathsf{p}_i}}}{\overline{\mathsf{p}_i \sqsupseteq \top} \,;\, \Delta, x : \phi^{\mathsf{p}} \vdash x \Rightarrow x : \phi}$$

$$\frac{\Sigma(\pi) = \phi \quad \Delta \equiv \overline{x_i : \phi_i{}^{\mathsf{p}_i}}}{\overline{\mathsf{p}_i \sqsupseteq \top} \,;\, \Delta \vdash \pi \Rightarrow \pi : \phi}$$

$$\frac{\Theta \,;\, \Delta, x : \phi^{\mathsf{p}} \vdash M \Rightarrow X : \psi \quad \phi = \mathit{fresh}(\sigma) \quad \mathsf{p}\ \text{fresh}}{\Theta \,;\, \Delta \vdash \lambda x{:}\sigma.M \Rightarrow \lambda x{:}\phi^{\mathsf{p}}.X : \phi^{\mathsf{p}} \multimap \psi}$$

$$\frac{\Theta_2 \,;\, \Delta_1 \vdash M \Rightarrow X : \phi_1{}^{\mathsf{p}} \multimap \psi \quad \Theta_3 \,;\, \Delta_2 \vdash N \Rightarrow Y : \phi_2 \quad \mathit{split}(\Delta, M, N) = (\Delta_1, \Delta_2, \Theta_1) \quad \begin{array}{r} \Delta_2 \equiv \overline{x_i : \phi_i{}^{\mathsf{q}_i}} \\ (\phi_2 \leq \phi_1) = \Theta_4 \end{array}}{\Theta_1, \Theta_2, \Theta_3, \Theta_4, \overline{\mathsf{q}_i \sqsupseteq \mathsf{p}} \,;\, \Delta \vdash MN \Rightarrow XY : \psi}$$

$$\frac{\Theta_2 \,;\, \Delta_1 \vdash M_1 \Rightarrow X_1 : \phi_1 \quad \Theta_3 \,;\, \Delta_2 \vdash M_2 \Rightarrow X_2 : \phi_2 \quad \mathit{split}(\Delta, M, N) = (\Delta_1, \Delta_2, \Theta_1) \quad \begin{array}{r} \Delta_1 \equiv \overline{x_{1,i} : \phi_{1,i}{}^{\mathsf{q}_{1,i}}} \\ \Delta_2 \equiv \overline{x_{2,i} : \phi_{2,i}{}^{\mathsf{q}_{2,i}}} \end{array}}{\Theta_1, \Theta_2, \Theta_3, \overline{\mathsf{q}_{1,i} \sqsupseteq \mathsf{p}_1}, \overline{\mathsf{q}_{2,i} \sqsupseteq \mathsf{p}_2} \,;\, \Delta \vdash \langle M_1, M_2 \rangle \Rightarrow \langle X_1, X_2 \rangle^{\mathsf{p}_1, \mathsf{p}_2} : \phi_1{}^{\mathsf{p}_1} \otimes \phi_2{}^{\mathsf{p}_2}}$$

Figure 6.3: Inferring constraint inequations for simple linearity analysis

$$\frac{\Theta_2\,;\Delta_1 \vdash M \Rightarrow X : \mathsf{bool} \quad \Theta_3\,;\Delta_2 \vdash N_1 \Rightarrow Y_1 : \phi_1 \quad \Theta_4\,;\Delta_2 \vdash N_2 \Rightarrow Y_2 : \phi_2 \quad \begin{array}{l} \phi = \mathit{fresh}(\phi_1{}^{\circ}) \\ (\phi_1 \leq \phi) = \Theta_5 \\ (\phi_2 \leq \phi) = \Theta_6 \\ \mathit{split}(M, \langle N_1, N_2 \rangle) = (\Delta_1, \Delta_2, \Theta_1) \end{array}}{\Theta_1, \Theta_2, \Theta_3, \Theta_4, \Theta_5, \Theta_6\,;\Delta \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 \Rightarrow \mathsf{if}\ X\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2 : \phi}$$

$$\frac{\Theta_1\,;\Delta, x : \phi_1{}^{\mathsf{p}} \vdash M \Rightarrow X : \phi_2 \quad (\phi_1 \leq \phi_2) = \Theta_2 \quad \Delta \equiv \overline{x_i : \psi_i^{\mathsf{q}_i}} \quad \phi_1 = \mathit{fresh}(\sigma) \quad \mathsf{p}\ \text{fresh}}{\Theta_1, \Theta_2, \overline{\mathsf{q}_i \sqsupseteq \top}, \mathsf{p} \sqsupseteq \top\,;\Delta \vdash \mathsf{fix}\ x{:}\sigma.M \Rightarrow \mathsf{fix}\ x{:}\phi_1.X : \phi_2}$$

$$\frac{\Theta_2\,;\Delta_1 \vdash M \Rightarrow X : \phi_1{}^{\mathsf{p}_1} \otimes \phi_2{}^{\mathsf{p}_2} \quad \Theta_3\,;\Delta_2, x_1 : \phi_3{}^{\mathsf{p}_3}, x_2 : \phi_4{}^{\mathsf{p}_4} \vdash N \Rightarrow Y : \psi \quad \begin{array}{l} \mathsf{p}_3, \mathsf{p}_4\ \text{fresh} \\ (\phi_1{}^{\mathsf{p}_1} \otimes \phi_2{}^{\mathsf{p}_2} \leq \phi_3{}^{\mathsf{p}_3} \otimes \phi_4{}^{\mathsf{p}_4}) = \Theta_4 \\ \mathit{split}(\Delta, M, N) = (\Delta_1, \Delta_2, \Theta_1) \end{array}}{\Theta_1, \Theta_2, \Theta_3, \Theta_4\,;\Delta \vdash \mathsf{let}\ \langle x_1, x_2 \rangle = M\ \mathsf{in}\ N \Rightarrow \mathsf{let}\ \langle x_1, x_2 \rangle^{\mathsf{p}_3, \mathsf{p}_4} = X\ \mathsf{in}\ Y : \psi}$$

Figure 6.4: Inferring constraint inequations for simple linearity analysis (continued)

An inductive definition of context splitting satisfying the properties of Definition 6.1.3 is shown in Figure 6.2.

Notice that, in the last equation, it is possible to simplify $\mathsf{p} \sqsupseteq \mathsf{p}_1 + \mathsf{p}_2$ to $\mathsf{p} \sqsupseteq \top$. However, the definition above is more general, so it will work as well for other structural analyses.

We shall now turn to the properties satisfied by our inference algorithm. We begin by observing that each run of the algorithm is unique for a given input $\langle \Delta, M \rangle$.

Proposition 6.1.4 (Determinacy)
If $\Theta\,; \Delta \vdash M \Rightarrow X : \phi$ *and* $\Theta'\,; \Delta \vdash M \Rightarrow X' : \phi'$, *then* $X \equiv X'$, $\phi \equiv \phi'$ *and* $\Theta \equiv \Theta'$ □

Notice that, whenever $\Theta\,; \Delta \vdash M \Rightarrow X : \phi$, then $\Delta^\circ \vdash M : \phi^\circ$ is true in FPL, so the algorithm of Figure 6.3 can be understood as an extension of a simple type reconstruction algorithm for FPL[4]. Which constraints should be considered at each stage, is dictated by a slightly modified syntax-directed version of $\mathrm{NLL}^{\forall\mu\leq}$, as we explain below.

6.1.1 Relaxing the conditional rule

We shall now establish the correctness of our algorithm for inferring constraint inequations. Essentially, this amounts to showing that Conditions 6.1 and 6.2, stated at the beginning of this section, are verified. Actually, we shall show soundness with respect to a slightly modified version of $\mathrm{NLL}^{\mu\leq}$, called $\mathrm{NLL}^{\nu\leq}$. This intermediate type system has the same rules as its sibling, except for the conditional rule, shown below:

$$\frac{\Gamma_1 \vdash M : \mathsf{bool} \quad \Gamma_2 \vdash N_1 : \sigma_1 \quad \Gamma_2 \vdash N_2 : \sigma_2 \quad \begin{array}{c}\sigma_1 \leq \sigma \\ \sigma_2 \leq \sigma\end{array}}{\Gamma_1, \Gamma_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \sigma}\ \text{Conditional}$$

It is clear that this rule generalises that of $\mathrm{NLL}^{\forall\mu\leq}$, since $\sigma_1 \leq \sigma_1 \sqcup \sigma_2$ and $\sigma_2 \leq \sigma_1 \sqcup \sigma_2$. The intermediate system is also a type system of minimum types with respect to $\mathrm{NLL}^{\leq}$, except that it does not have unique types.

Proposition 6.1.5
If $\Gamma \underset{\mathrm{NLL}^{\leq}}{\vdash} M : \sigma$, *then* $\Gamma \underset{\mathrm{NLL}^{\nu\leq}}{\vdash} M : \tau$ *for some* τ *with* $\tau \leq \sigma$. □

By the above observation, we see clearly that the three type systems of interest verify $\mathrm{NLL}^{\mu\leq} \subseteq \mathrm{NLL}^{\nu\leq} \subseteq \mathrm{NLL}^{\leq}$.

The reason for introducing an intermediate type system is to avoid complicating our language of annotation terms with a $\sqcup$ operator to handle least upper bounds of annotations. Note that this precision is not necessary in practice; we are interested in the smallest annotation assignment with respect to the sub-decoration order, not the subtyping order, which is necessarily less interesting for applying optimisations.

As suggested in a previous discussion, the proof of soundness will relate the well-formed runs of the algorithm to the set of typings of the corresponding polymorphic version, in our case $\mathrm{NLL}^{\forall\nu\leq}$, which can similarly be defined in terms of $\mathrm{NLL}^{\forall\mu\leq}$, by modifying the conditions

[4] Indeed, if we erase Θ and X from the runs, and substitute all occurrences of ϕ by their respective erasures ϕ°, the result is (not surprisingly) the traditional type-checking algorithm of the simply-typed λ-calculus.

on the types of the branches of the conditional rule, as shown:

$$\frac{\Theta\,;\Gamma_1 \vdash M : \mathsf{bool} \quad \Theta\,;\Gamma_2 \vdash N_1 : \sigma_1 \quad \Theta\,;\Gamma_2 \vdash N_2 : \sigma_2 \quad \begin{array}{c}\Theta \rhd \phi_1 \leq \phi \\ \Theta \rhd \phi_2 \leq \phi\end{array}}{\Theta\,;\Gamma_1, \Gamma_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \phi}\ \text{Conditional}$$

As expected, the following proposition states the relationship between the intermediate polymorphic version and $\mathrm{NLL}^{\forall\leq}$.

Proposition 6.1.6
If $\Theta\,;\Gamma \underset{\mathrm{NLL}^{\forall\leq}}{\vdash} M : \phi$, *then* $\Theta\,;\Gamma \underset{\mathrm{NLL}^{\forall\nu\leq}}{\vdash} M : \psi$ *for some* ψ *with* $\psi \leq \phi$. □

6.1.2 Correctness

By looking at the inference rules of Figure 6.3, it is quite easy to see that if we erase the annotations in the translated intermediate language term, we retrieve the original source language term we started from. This settles Condition 6.2.

Proposition 6.1.7
If $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$, *then* $X^\circ = M$. □

The following proposition states three simple syntactic invariants regarding annotation parameters.

Proposition 6.1.8
If $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$, *then*

a. $FA(\phi) \subseteq FA(\Delta) \cup FA(X)$,

b. $FA(\Delta) \cap FA(X) = \emptyset$, *and*

c. *neither* Δ *nor* X *contain duplicate annotation parameters.*

Proof. Easy induction on the structure of M. □

We shall prove Condition 6.1 by stating soundness and completeness with respect to $\mathrm{NLL}^{\nu\leq}$ typings.

To prove soundness, we have to show that if $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$ is a well-formed run of the algorithm, then $\Delta[\theta] \vdash X[\theta] : \phi[\theta]$ is a valid $\mathrm{NLL}^{\nu\leq}$ typing, for all covering solutions θ of Θ. By *covering solution* we mean that θ must cover the components of the typing judgment, namely Δ, X and ϕ.

We first show how the runs of the algorithm are related to $\mathrm{NLL}^{\forall\nu\leq}$ typings.

Lemma 6.1.9 (Soundness for $\mathrm{NLL}^{\forall\nu\leq}$)
If $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$, *then* $\Theta\,;\Delta \underset{\mathrm{NLL}^{\forall\nu\leq}}{\vdash} X : \phi$.

Proof. By induction on the structure of M. As usual, we reason in terms of the syntax-directed version of $\mathrm{NLL}^{\forall\nu\leq}$. We prove the lemma for some prototypical cases only. In each case, we assume the premises of the matching rule.

- $M \equiv x$.

  Note that $\Theta\,;\Delta, x : \phi^{\mathsf{p}} \vdash x : \phi$ since Θ is defined as $\overline{\mathsf{p}_i \sqsupseteq \top}$ so that the structural condition $\Theta \triangleright |\Delta| \sqsupseteq \top$, which equals $\Theta \triangleright |\overline{x_i : \phi_i{}^{\mathsf{p}_i}}| \sqsupseteq \top$, trivially holds.

- $M \equiv \lambda x{:}\sigma.M'$.

  In this case, we have $\Theta\,;\Delta \vdash \lambda x{:}\sigma.M' \Rightarrow \lambda x{:}\phi^{\mathsf{p}}.X' : \phi^{\mathsf{p}} \multimap \psi$ because $\Theta\,;\Delta, x : \phi^{\mathsf{p}} \vdash M' \Rightarrow X' : \psi$ where $\phi = \mathit{fresh}(\sigma)$. From the latter, we may conclude $\Theta\,;\Delta \vdash X' : \psi$ by the induction hypothesis. The required conclusion $\Theta\,;\Delta \vdash \lambda x{:}\phi^{\mathsf{p}}.X' : \phi^{\mathsf{p}} \multimap \psi$ directly follows by $\multimap_{\mathcal{I}}$.

- $M \equiv M'N'$.

  We have $\Theta\,;\Delta \vdash M'N' \Rightarrow X'Y' : \psi$ because $\Theta_2\,;\Delta_1 \vdash M' \Rightarrow X' : \phi_1{}^{\mathsf{p}} \multimap \psi$ and $\Theta_3\,;\Delta_2 \vdash N' \Rightarrow Y' : \phi_2$, where $\mathit{split}(\Delta, M', N') = (\Delta_1, \Delta_2, \Theta_1)$ and $(\phi_2 \leq \phi_1) = \Theta_4$, with $\Delta_2 \equiv \overline{x_i : \phi_i{}^{\mathsf{q}_i}}$ and $\Theta \equiv \Theta_1, \Theta_2, \Theta_3, \Theta_4, \overline{\mathsf{q}_i \sqsupseteq \mathsf{p}}$. By the induction hypothesis and constraint strengthening, we may conclude $\Theta\,;\Delta_1 \vdash X' : \phi_1{}^{\mathsf{p}} \multimap \psi$ and $\Theta\,;\Delta_1 \vdash Y' : \phi_2$. Note that the following conditions hold:

  $$\begin{aligned} \Theta \vdash \phi_2 \leq \phi_1, &\quad \text{by Proposition 6.1.2 and constraint strengthening;} \\ \Delta = \Delta_1 \uplus \Delta_2, &\quad \text{by the definition of } \uplus \text{ of Figure 6.2; and} \\ \Theta \triangleright |\Delta_2| \sqsupseteq \mathsf{p}, &\quad \text{since } \Theta \text{ contains } \overline{\mathsf{q}_i \sqsupseteq \mathsf{p}}. \end{aligned}$$

  These conditions are sufficient to apply $\multimap_{\mathcal{E}}$ in order to derive $\Theta\,;\Delta \vdash X'Y' : \psi$ as needed.

- $M \equiv \mathsf{if}\ M'\ \mathsf{then}\ N'\ \mathsf{else}\ N''$.

  The proof is similar to that for the application. Notice that the steps required to prove this case require the modified conditional rule of the intermediate system $\mathrm{NLL}^{\forall\nu\leq}$.

□

Using the above lemma, inclusion into the decoration space follows as a corollary of Lemmas 6.1.9, 5.4.4 and 5.4.8.

Theorem 6.1.10 (Soundness)
If $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$, *then* $\Delta[\theta] \underset{\mathrm{NLL}^{\nu\leq}}{\vdash} X[\theta] : \phi[\theta]$, *for all covering* $\theta \models \Theta$.

Proof. Indeed, by Lemma 6.1.9, $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$ implies that $\Theta\,;\Delta \vdash X : \phi$ holds in $\mathrm{NLL}^{\forall\nu\leq}$. According to Lemma 5.4.8, $-\,;\Delta[\theta] \vdash X[\theta] : \phi[\theta]$ must also be true, for all $\theta \models \Theta$. Therefore, by Lemma 5.4.4, and the fact that $\Delta[\theta]$ and $X[\theta]$ are simple by construction, it follows that $\Delta[\theta] \vdash X[\theta] : \phi[\theta]$ is valid in $\mathrm{NLL}^{\nu\leq}$. □

To prove completeness, we must show that if $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$ is a well-formed run, then any alternative decoration of $\Delta^\circ \vdash X^\circ : \phi^\circ$ can be rewritten as $\Delta[\theta] \vdash X[\theta] : \phi[\theta]$, for some suitable solution θ of Θ.

Theorem 6.1.11 (Completeness)
If $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$ *and* $\Gamma \vdash N : \sigma$ *is any* $\mathrm{NLL}^{\nu\leq}$ *decoration of* $\Delta^\circ \vdash X^\circ : \phi^\circ$, *then there exists a covering solution* $\theta \models \Theta$, *such that* $\Gamma \equiv \Delta[\theta]$, $N \equiv X[\theta]$ *and* $\sigma \equiv \phi[\theta]$.

Proof. By induction on the structure of M. Again, we consider some prototypical cases only, and reason with respect to the syntax-directed version of the theory.

- $M \equiv x$.

  We have $\Theta\,;\Delta, x : \phi^{\mathsf{p}} \vdash x \Rightarrow x : \phi$ where $\Theta \equiv \overline{\mathsf{p}_i \sqsupseteq \top}$ and $\Delta \equiv \overline{x_i : \phi_i{}^{\mathsf{p}_i}}$. A decoration must have the form $\Gamma, x : \sigma^a \vdash x : \sigma$, subject to the condition that $|\Gamma| \sqsupseteq \top$.

  By construction of the algorithm (Proposition 6.1.8), Δ and ϕ contain distinct annotation parameters at all positions, and share none, so there trivially exists a suitable covering θ, making $\Gamma \equiv \Delta[\theta]$ and $\sigma \equiv \phi[\theta]$. We require $\theta \models \Theta$, so $\theta(\mathsf{p}_i) \sqsupseteq \top$ for all p_i, but this is already the case, since $|\Gamma| \sqsupseteq \top$ implies $\theta(\mathsf{p}_i) = \top$.

- $M \equiv \lambda x{:}\sigma.M'$.

  In this case, we have $\Theta\,;\Delta \vdash \lambda x{:}\sigma.M' \Rightarrow \lambda x{:}\phi_1^{\mathsf{p}}.X' : \phi_1^{\mathsf{p}} \multimap \phi_2$ because $\Theta\,;\Delta, x : \phi_1{}^{\mathsf{p}} \vdash M' \Rightarrow X' : \phi_2$, where $\phi_1 = \mathit{fresh}(\sigma)$.

  Any decoration has the form $\Gamma \vdash \lambda x{:}\tau_1{}^a.N' : \tau_1{}^a \multimap \tau_2$, provided that $\Gamma, x : \tau_1{}^a \vdash N' : \tau_2$. From the induction hypothesis applied to the latter, we know there exists $\theta \models \Theta$, such that $\Gamma, x : \tau_1{}^a \equiv (\Delta, x : \phi^{\mathsf{p}})[\theta]$, $N' \equiv X'[\theta]$ and $\tau_2 \equiv \phi_2[\theta]$. By the definition of annotation substitution, it clearly follows that $\Gamma \equiv \Delta[\theta]$, $\lambda x{:}\tau_1{}^a.N' \equiv (\lambda x{:}\phi_1^{\mathsf{p}}.X')[\theta]$ and $\tau_1{}^a \multimap \tau_2 \equiv (\phi_1^{\mathsf{p}} \multimap \phi_2)[\theta]$.

- $M \equiv M'M''$.

  We have $\Theta\,;\Delta \vdash M'M'' \Rightarrow X'X'' : \psi$ because $\Theta_2\,;\Delta_1 \vdash M' \Rightarrow X' : \phi_1{}^{\mathsf{p}} \multimap \psi$ and $\Theta_3\,;\Delta_2 \vdash M'' \Rightarrow X'' : \phi_2$, where $\mathit{split}(\Delta, M', M'') = (\Delta_1, \Delta_2, \Theta_1)$, $(\phi_2 \leq \phi_1) = \Theta_4$, $\Delta_2 \equiv \overline{x_i : \phi_i{}^{\mathsf{q}_i}}$ and $\Theta \equiv \Theta_1, \Theta_2, \Theta_3, \Theta_4, \overline{\mathsf{q}_i \sqsupseteq \mathsf{p}}$.

  A decoration must have the form $\Gamma_1 \uplus \Gamma_2 \vdash N'N'' : \tau$, provided that $\Gamma_1 \vdash N' : \sigma_1{}^a \multimap \tau$ and $\Gamma_1 \vdash N'' : \sigma_2$, subject to the conditions that $\sigma_2 \leq \sigma_1$ and $|\Gamma_2| \sqsupseteq a$. By the induction hypothesis, twice, applied to the decoration premises, it is clear there exist $\theta_2 \models \Theta_2$ and $\theta_3 \models \Theta_3$, such that

$$\Gamma_1 \equiv \Delta_1[\theta_2], \quad N' \equiv X'[\theta_2], \quad \sigma_1{}^a \multimap \tau \equiv (\phi_1{}^{\mathsf{p}} \multimap \psi)[\theta_2]; \tag{6.3}$$

$$\Gamma_2 \equiv \Delta_2[\theta_3], \quad N'' \equiv X''[\theta_3], \quad \sigma_2 \equiv \phi_2[\theta_3]. \tag{6.4}$$

  Take $\theta = \theta_1' \cup \theta_2' \cup \theta_3'$, where $\theta_2' = \theta_2 \upharpoonright (\mathit{FA}(X') \cup \mathit{FA}(\Delta_1))$, $\theta_3' = \theta_3 \upharpoonright (\mathit{FA}(X'') \cup \mathit{FA}(\Delta_2))$ and θ_1' is such that $\theta_1'(\mathsf{p}) = \top$, for all p such that $\mathsf{p} \sqsupseteq \mathsf{p}_1 + \mathsf{p}_2$ is in Θ_1. The union is here meant to stand for the union of annotation substitutions as relation sets, so $\mathit{dom}(\theta) = \mathit{dom}(\theta_1') \cup \mathit{dom}(\theta_2') \cup \mathit{dom}(\theta_3')$. (The restrictions on the domains of θ_1' and θ_2', the definition of *split* and Proposition 6.1.8 ensure that the union is well-defined.)

  It is clear that the syntactic equivalences (6.3) and (6.4) also apply to θ. We therefore have $M'M'' \equiv (X'X'')[\theta]$ and $\tau \equiv \psi[\theta]$. Also, $\Gamma_1 \uplus \Gamma_2 \equiv (\Delta_1 \uplus \Delta_2)[\theta]$, since the definition of *split* ensures that $\Delta[\theta] = \Delta_1[\theta] \uplus \Delta_2[\theta]$ for all $\theta \models \Theta_1$, which follows by definition of θ. Notice that, in general, $\theta \models \Theta$, as required. We know that $\theta \models \Theta_1, \Theta_2, \Theta_3$ by construction. The fact that $\theta \models \Theta_4, \overline{\mathsf{q}_i \sqsupseteq \mathsf{p}}$ is a consequence of the subtyping and structural condition hypotheses on decorations and Proposition 6.1.2.

□

Notice that what syntactic completeness really asserts is that $\Theta\,;\Delta \vdash X : \phi$ is 'principal', although in a slightly different sense.

6.1.3 Avoiding splitting contexts

Splitting contexts as shown would be rather too time-consuming; the annotation inference algorithm would undoubtedly spend most its time computing variable-occurrence predicates. There are at least two well-known 'tricks' to avoid splitting contexts. The first one was proposed for the implementation of the type-checking algorithm of the linear language Lilac [42], and is also the one we have used in our implementation of linearity analysis. We shall however briefly illustrate the second approach [62], which assumes that pre-computed occurrence information is available for bound variables—roughly, as an integer recording the number of times a variable occurs in its scope[5].

An alternative definition, not requiring the splitting of contexts, is given without proof in Figures 6.5 and 6.6.

Notice that the contexts Δ in the conclusions of the rules are now shared. We use the occurrence count associated to the bound variables (when they are introduced) to generate the proper constraints; in particular, if a variable occurs any number of times, excluding one, it must be annotated as non-linear, which explains the use of the inequation $\mathsf{p} \sqsupseteq |n|$. The notation $|n|$ stands for the 'meaning' of an occurrence count in terms of our annotation lattice:

$$|n| = \begin{cases} \top, & \text{if } n \neq 1; \\ 1, & \text{otherwise.} \end{cases}$$

Naturally, we avoid adding any restrictions for the case of variables and constants. A precise definition of the *occurs* function can be found in Figure 7.4, on page 147.

[5]Actually, both approaches rely on precisely the same information, except that, in the case of Lilac, this information is computed incrementally during type inference.

$$\frac{}{\emptyset\,;\Delta, x:\phi^{\mathsf{p}} \vdash x \Rightarrow x:\phi}$$

$$\frac{\Sigma(\pi) = \phi}{\emptyset\,;\Delta \vdash \pi \Rightarrow \pi : \phi}$$

$$\frac{\Theta\,;\Delta, x:\phi^{\mathsf{p}} \vdash M \Rightarrow X:\psi \quad \phi = \mathit{fresh}(\sigma) \quad \mathsf{p}\ \text{fresh} \quad n = \mathit{occurs}(x,X)}{\Theta, \mathsf{p} \sqsupseteq |n|\,;\Delta \vdash \lambda x{:}\sigma.M \Rightarrow \lambda x{:}\phi^{\mathsf{p}}.X : \phi^{\mathsf{p}} \multimap \psi}$$

$$\frac{\Theta_1\,;\Delta \vdash M \Rightarrow X:{\phi_1}^{\mathsf{p}} \multimap \psi \quad \Theta_2\,;\Delta \vdash N \Rightarrow Y:\phi_2 \quad \begin{array}{c}\Delta_2{\restriction}\mathit{FA}(Y) = \overline{x_i : {\phi_i}^{\mathsf{q}_i}} \\ (\phi_2 \leq \phi_1) = \Theta_3\end{array}}{\Theta_1, \Theta_2, \Theta_3, \overline{\mathsf{q}_i \sqsupseteq \mathsf{p}}\,;\Delta \vdash MN \Rightarrow XY : \psi}$$

$$\frac{\Theta_1\,;\Delta \vdash M_1 \Rightarrow X_1:\phi_1 \quad \Theta_2\,;\Delta \vdash M_2 \Rightarrow X_2:\phi_2 \quad \begin{array}{c}\Delta{\restriction}\mathit{FA}(X_1) = \overline{x_{1,i} : {\phi_{1,i}}^{\mathsf{q}_{1,i}}} \\ \Delta{\restriction}\mathit{FA}(X_2) = \overline{x_{2,i} : {\phi_{2,i}}^{\mathsf{q}_{2,i}}}\end{array}}{\Theta_1, \Theta_2, \overline{\mathsf{q}_{1,i} \sqsupseteq \mathsf{p}_1}, \overline{\mathsf{q}_{2,i} \sqsupseteq \mathsf{p}_2}\,;\Delta \vdash \langle M_1, M_2\rangle \Rightarrow \langle X_1, X_2\rangle^{\mathsf{p}_1,\mathsf{p}_2} : {\phi_1}^{\mathsf{p}_1} \otimes {\phi_2}^{\mathsf{p}_2}}$$

Figure 6.5: Inferring constraint inequations for simple linearity analysis without context splitting

$$\frac{\Theta_1\,;\Delta \vdash M \Rightarrow X : \mathsf{bool} \quad \Theta_2\,;\Delta \vdash N_1 \Rightarrow Y_1 : \phi_1 \quad \Theta_3\,;\Delta \vdash N_2 \Rightarrow Y_2 : \phi_2 \quad \begin{array}{l} \phi = \mathit{fresh}(\phi_1{}^{\circ}) \\ (\phi_1 \le \phi) = \Theta_4 \\ (\phi_2 \le \phi) = \Theta_5 \end{array}}{\Theta_1, \Theta_2, \Theta_3, \Theta_4, \Theta_5\,;\Delta \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 \Rightarrow \mathsf{if}\ X\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2 : \phi}$$

$$\frac{\Theta_1\,;\Delta, x : \phi_1{}^{\mathsf{p}} \vdash M \Rightarrow X : \phi_2 \quad (\phi_1 \le \phi_2) = \Theta_2 \quad \Delta \equiv \overline{x_i : \psi_i^{\mathsf{q}_i}} \quad \phi_1 = \mathit{fresh}(\sigma) \quad \mathsf{p}\ \text{fresh}}{\Theta_1, \Theta_2, \overline{\mathsf{q}_i \sqsupseteq \top}, \mathsf{p} \sqsupseteq \top\,;\Delta \vdash \mathsf{fix}\ x{:}\sigma.M \Rightarrow \mathsf{fix}\ x{:}\phi_1.X : \phi_2}$$

$$\frac{\Theta_1\,;\Delta_1 \vdash M \Rightarrow X : \phi_1{}^{\mathsf{p}_1} \otimes \phi_2{}^{\mathsf{p}_2} \quad \Theta_2\,;\Delta_2, x_1 : \phi_3{}^{\mathsf{p}_3}, x_2 : \phi_4{}^{\mathsf{p}_4} \vdash N \Rightarrow Y : \psi \quad \begin{array}{c} \mathsf{p}_3, \mathsf{p}_4\ \text{fresh} \\ (\phi_1{}^{\mathsf{p}_1} \otimes \phi_2{}^{\mathsf{p}_2} \le \phi_3{}^{\mathsf{p}_3} \otimes \phi_4{}^{\mathsf{p}_4}) = \Theta_3 \\ n_1 = \mathit{occurs}(x_1, Y) \\ n_2 = \mathit{occurs}(x_2, Y) \end{array}}{\Theta_1, \Theta_2, \Theta_3, \mathsf{p}_3 \sqsupseteq |n_1|, \mathsf{p}_4 \sqsupseteq |n_2|\,;\Delta \vdash \mathsf{let}\ \langle x_1, x_2 \rangle = M\ \mathsf{in}\ N \Rightarrow \mathsf{let}\ \langle x_1, x_2 \rangle^{\mathsf{p}_3,\mathsf{p}_4} = X\ \mathsf{in}\ Y : \psi}$$

Figure 6.6: Inferring constraint inequations for simple linearity analysis without context splitting (continued)

6.2 Solving constraint inequations

We have given an algorithm for computing the set of constrains that characterises the decoration space of an input typing judgment. We are now left with the task of showing the reader how to solve the constraint inequations to find the optimal solution.

6.2.1 Characterising the least solution

We should first remark that the algorithm for inferring constraint inequations only generates inequations of the form $\mathsf{p} \sqsupseteq t$, where t is either $\top$ or an annotation parameter $\mathsf{q} \not\equiv \mathsf{p}$ (provided that we replace all occurrences of terms of the form $\mathsf{p}_1 + \mathsf{p}_2$ by $\top$). A constraint set Θ formed from inequations of this particular form is not only always consistent, but, in fact, the space of all its solutions, $[\Theta] = \{\theta \mid \theta \models \Theta\}$, forms a complete lattice with respect to the 'natural' order, defined by

$$\theta_1 \sqsubseteq \theta_2 \stackrel{\text{def}}{=} \theta_1(\mathsf{p}) \sqsubseteq \theta_2(\mathsf{p}), \text{ for all } \mathsf{p} \in \mathit{dom}(\theta_1).$$

This fact is stated in the following proposition.

Proposition 6.2.1 (Complete solution lattice)
For all constraint sets $\Theta \equiv \overline{\mathsf{p}_i \sqsupseteq t_i}$, $\langle [\Theta]; \sqsubseteq \rangle$ forms a non-empty complete lattice.

Proof. It is obvious that $\top_{[\Theta]} = \langle \top/\mathsf{p}_i \rangle_{i \geq n}$, for all $\mathsf{p}_i \in \mathbb{P}$, is the greatest element of the solution set. Clearly, $\top_{[\Theta]} \models \Theta$ and $\theta \sqsubseteq \top_{[\Theta]}$, for any θ in the solution set.

Let $\Sigma = \{\theta_i \mid i \in I\}$ be a non-empty subset of the solution set, indexed by elements in I. We show that the meet, defined element-wise,

$$\theta(\mathsf{p}) \stackrel{\text{def}}{=} \sqcap_{i \in I} \theta_i(\mathsf{p}),$$

satifies Θ. Indeed, for each inequation $\mathsf{p} \sqsupseteq t$ and $\theta_i \in \Sigma$, since $\theta_i \models \Theta$, we have $\theta_i(\mathsf{p}) \sqsupseteq \theta_i(t)$, and so $\sqcap_{i \in I} \theta_i(\mathsf{p}) \sqsupseteq \sqcap_{i \in I} \theta_i(t)$. Therefore, $\theta(\mathsf{p}) \sqsupseteq \sqcap_{i \in I} \theta_i(t)$; and, because $\theta_i(t) \sqsupseteq \theta(t)$ implies $\sqcap_{i \in I} \theta_i(t) \sqsupseteq \theta(t)$, we deduce $\theta(\mathsf{p}) \sqsupseteq \theta(t)$. $\square$

Much like Theorem 3.6.4, the proof of the above statement depends fundamentally on the fact that the 2-point annotation set we started with is itself complete.

Since the solution space of a constraint set Θ forms a complete lattice, we are interested in an effetive procedure for computing its meet

$$\theta^{\text{opt}} \stackrel{\text{def}}{=} \sqcap [\Theta].$$

A standard way to proceed, in cases like this, consists in showing how θ^{opt} may be alternatively characterised as the least solution of a fixpoint equation for some suitable map

$$\mathcal{F}_\Theta : (P \to \mathbb{A}) \to (P \to \mathbb{A}) \quad \text{where} \quad P = \mathit{FA}(\Theta),$$

defined over the complete lattice $\langle [P \to \mathbb{A}]; \sqsubseteq \rangle$ of ground annotation substitutions ordered according to the sub-decoration order, which must satisfy the monotonicity and ascending chain conditions. These conditions ensure that least fixpoint exist and that they can be computed using the iterative method of Theorem 2.2.10.

If Θ is a given constraint set, it is not difficult to see that the fixpoints of the map

$$\mathcal{F}_\Theta(\theta)(\mathsf{p}) \stackrel{\text{def}}{=} \bigsqcup \{\theta(t) \mid \mathsf{p} \sqsupseteq t \text{ is in } \Theta\} \tag{6.5}$$

are indeed all the ground substitutions θ satisfying Θ. As required, $\mathcal{F}_\Theta$ is monotone. Indeed,

$$\begin{aligned}\theta_1 \sqsubseteq \theta_2 &\Leftrightarrow \forall \mathsf{p}.\theta_1(\mathsf{p}) \sqsubseteq \theta_2(\mathsf{p}) \\ &\Rightarrow \forall \mathsf{p}.\bigsqcup\{\theta_1(t) \mid \mathsf{p} \sqsupseteq t \text{ is in } \Theta\} \sqsubseteq \bigsqcup\{\theta_2(t) \mid \mathsf{p} \sqsupseteq t \text{ is in } \Theta\} \\ &\Leftrightarrow \forall \mathsf{p}.\mathcal{F}_\Theta(\theta_1)(\mathsf{p}) \sqsubseteq \mathcal{F}_\Theta(\theta_2)(\mathsf{p}) \\ &\Leftrightarrow \mathcal{F}_\Theta(\theta_1) \sqsubseteq \mathcal{F}_\Theta(\theta_2).\end{aligned}$$

The fact that our map preserves the joins of ascending chains follows from the fact $\mathcal{F}_\Theta$ is defined on a finite lattice and monotonicity.

Hence, by Theorem 2.2.10, the least solution can be constructed as follows:

$$\mu(\mathcal{F}_\Theta) = \bigsqcup_{i \geq 0} \mathcal{F}_\Theta{}^i(\overline{\langle \bot/\mathsf{p}_i \rangle}), \tag{6.6}$$

where $\overline{\mathsf{p}_i} = FA(\Theta)$.

The following table depicts the fixpoint approximations $\theta_i \equiv \mathcal{F}_\Theta{}^i(\theta_0)$ for $0 \leq i \leq 3$, where

$$\Theta \equiv \mathsf{p}_5 \sqsupseteq \mathsf{p}_1, \mathsf{p}_6 \sqsupseteq \mathsf{p}_2, \mathsf{p}_5 \sqsupseteq \mathsf{p}_3, \mathsf{p}_6 \sqsupseteq \mathsf{p}_3, \mathsf{p}_3 \sqsupseteq \top, \tag{6.7}$$

which is the constraint set associated to all the decorations of the example of Figure 5.3, except for the added constraint $\mathsf{p}_3 \sqsupseteq \top$. We start with $\theta_0(\mathsf{p}) = 1 \equiv \bot$ for all $\mathsf{p} \in dom(\Theta) = \{\mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, \mathsf{p}_5, \mathsf{p}_6\}$.

	p_1	p_2	p_3	p_5	p_6
θ_0	1	1	1	1	1
θ_1	1	1	$\top$	1	1
θ_2	1	1	$\top$	$\top$	$\top$
θ_3	1	1	$\top$	$\top$	$\top$

Notice that $\theta_{i+1} = \theta_i$ for all $i \geq 2$, so $\mu(\mathcal{F}_\Theta) = \theta_2$ is our desired least solution.

6.2.2 Digression: decorations as closures

It is not difficult to see that the following functional

$$\mathcal{F}^\circ_\Theta(\theta) = \bigsqcup_{i \geq 0} \mathcal{F}_\Theta{}^i(\theta), \tag{6.8}$$

defines a closure operator that maps any ground substitution θ to the smallest substitution $\theta' \sqsupseteq \theta$ satisfying Θ. For this reason, we refer to $\mathcal{F}^\circ_\Theta(\theta)$ as the Θ-*closure* of θ. We therefore have a mechanism that allows us to obtain the least solution compatible with both an initial set of assignments (i.e., those in θ) and the constraint set Θ[6].

6.2.3 A graph-based algorithm for computing the least solution

There are general algorithms, varying in their degree of efficiency, for computing the least solution of a set of constraints. We shall not be studying any of them here, as well-documented versions can be found elsewhere in the literature (for instance, [48] provides a survey of

[6]This fact was suggested to the author in a private communication with Paul-André Melliès.

some of them.) Another reason is that computing the optimal solution for the linear case is straightforward, due to the simple form of the inequations.

Notice that, in all the inference algorithms, the inequations used are of the general form $\mathsf{p} \sqsupseteq \top$ or $\mathsf{p} \sqsupseteq \mathsf{q}$. A simple algorithm would use a directed graph as a representation of Θ, having annotation parameters as nodes. Each time the algorithm generates a new constraint, the graph is updated as follows:

- For $\mathsf{p} \sqsupseteq \top$, label the node associated to p with $\top$.
- For $\mathsf{p} \sqsupseteq \mathsf{q}$, add an edge going from q to p.

In both cases, if the nodes did not already appear in the graph, they must be created.

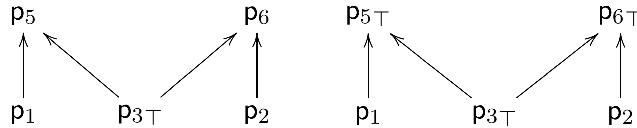
Once the inference algorithm terminates successfully with a complete Θ as result, we can compute the optimal solution in the following way:

- We must first 'close' the graph of Θ, by labeling with $\top$ all the nodes that are reachable from a $\top$-labeled node. (We could have alternatively modified our updating process above, propagating labels as required, each time we add a new label or an edge.)
- The optimal solution can now be defined by letting

$$\theta^{\text{opt}}(\mathsf{p}) \stackrel{\text{def}}{=} \begin{cases} \top, & \text{if } \mathsf{p} \text{ has } \top \text{ as label;} \\ 1, & \text{otherwise} \end{cases}$$

  for all p free for Θ[7].

For our sample constraint set (6.7) above, the first part of the algorithm would generate the graph shown below on the left. The graph on the right corresponds to its closure.



The correctness of our simple graph-based algorithm is easily established upon consideration of the following two facts:

- The $\top$-labeled nodes of the generated graph, after the first stage of the algorithm, correspond to the assignments in θ_1, provided that we interpret the unlabeled nodes as being implicitly labeled 1.
- A single propagation step consists in labeling $\top$ all nodes p, such that $\mathsf{p} \sqsupseteq \mathsf{q}$ is an inequation of Θ, if q was so labeled in a previous single propagation step. This is precisely what happens to the assignments in θ_i when we compute $\theta_{i+1} = \mathcal{F}_\Theta(\theta_i)$.

[7]Because our implementation of linearity analysis was intended for experimental purposes, we provided a mechanism so that programmers could suggest some initial annotation values. Therefore, we start with a dependency graph where some of the nodes are already labeled, with either 1 or $\top$. When we 'close' the graph, we must also label with 1 all the nodes from which a 1-labeled node is reachable. The implementation signals an 'annotation clash' error, whenever the algorithm attempts to label a node that already been labeled with a different annotation. This may happen, since the initial annotation-assignment given by the programmer, say θ_0, may be inconsistent with the constraints inferred. Equivalently, we might choose to detect this sort of inconsistency after annotation inference by checking that there is no $\mathsf{p} \in \mathit{dom}(\theta_0)$, for which $\theta^{\text{opt}}(\mathsf{p}) \sqsupset \theta_0(\mathsf{p})$.

Input $\langle \Gamma, M \rangle$
Output $\langle \Gamma', M' \rangle$

Step 1 From $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$, obtain the constraint set Θ and the translation X. The input context Δ is obtained from $\Delta = \mathit{fresh}(\Gamma)$.

Step 2 Find the optimal solution θ^{opt} of Θ by computing $\theta^{\mathrm{opt}} = \mu(\mathcal{F}_\Theta)$.

Step 3 Extend θ^{opt} to cover Δ and X by letting

$$\theta^{\mathrm{opt}'}(\mathsf{p}) = \begin{cases} \theta^{\mathrm{opt}}(\mathsf{p}), & \text{if } \mathsf{p} \in \mathit{dom}(\theta^{\mathrm{opt}}); \\ 1, & \text{otherwise,} \end{cases}$$

for all $\mathsf{p} \in \mathit{FA}(\Delta) \cup \mathit{FA}(X)$.

Step 4 Output $\Gamma' = \Delta[\theta^{\mathrm{opt}'}]$ and $M' = X[\theta^{\mathrm{opt}'}]$.

Figure 6.7: Annotation inference algorithm for linearity analysis

6.2.4 Putting it all together

Now that we have both an algorithm for inferring constraint inequations, and a generic method for finding the least solution, we can sum up the whole process of annotation inference into a single algorithm. This is done in Figure 6.7.

The extension $\theta^{\mathrm{opt}'}$ of θ^{opt} is necessary to cover all free annotation parameters in Δ and X that are not mentioned in Θ (otherwise, Lemma 6.1.10 would fail to be true). It is clear that $\theta^{\mathrm{opt}'}$ is the smallest such extension.

6.3 Let-based annotation inference

It is now about time to tell the reader how our ideas concerning annotation polymorphism might be put into practice, by showing a more powerful annotation inference algorithm capable of inferring qualified types for language definitions. As we have already discussed in Section 5.1, our motivations are driven by the need to have a 'compositional' static analysis strategy that does not limit itself to stand-alone programs.

6.3.1 Preliminary remarks

Our annotation algorithm will translate source terms into the intermediate language terms of $\mathrm{NLL}^{\forall\mathrm{let}\leq}$, introduced in Section 5.5. We recall that, in $\mathrm{NLL}^{\forall\mathrm{let}\leq}$, only local let-definitions are allowed to have qualified types, and that, as a consequence, only let-bound variables need ever be specialised. This restriction, which is defined at the level of the syntax of terms, is helpful as it tell us where we should find Λ-abstractions and applications in the translated term, and gives us an idea of the general shape of the decorations we shall be dealing with. Giving a general definition of the decoration space is less obvious in the case of annotation polymorphism, because of the occurrence of constraint sets inside the terms. Instead, we

$$\frac{\vartheta \equiv \overline{\langle \mathsf{p}_i'/\mathsf{p}_i \rangle} \quad \mathsf{p}_i' \text{ fresh} \quad \Delta \equiv \overline{x_i : {\phi_i}^{\mathsf{q}_i}}}{\Theta[\vartheta], \overline{\mathsf{q}_i \sqsupseteq \top} \,;\, \Delta, x : (\forall \overline{\mathsf{p}_i} \,|\, \Theta.\phi)^{\mathsf{p}} \vdash x \Rightarrow x\ \vartheta : \phi[\vartheta]}$$

$$\frac{\Theta_2 \,;\, \Delta_1 \vdash M \Rightarrow X : \phi \quad \Theta_3 \,;\, \Delta_2, x : (\forall \overline{\mathsf{p}_i} \,|\, \Theta_4.\phi)^{\mathsf{p}} \vdash N \Rightarrow Y : \psi}{\Theta_1, \Theta_3, \Theta_5, \overline{\mathsf{q}_i \sqsupseteq \mathsf{p}} \,;\, \Delta \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N \Rightarrow \mathsf{let}\ x{:}\forall \overline{\mathsf{p}_i} \,|\, \Theta_4.\phi = \Lambda \overline{\mathsf{p}_i} \,|\, \Theta_4.X\ \mathsf{in}\ Y : \psi}$$

where

$$\begin{aligned} \mathit{split}(\Delta, M, N) &= (\Delta_1, \Delta_2, \Theta_1) \\ \Delta_1 &\equiv \overline{x_i : {\phi_i}^{\mathsf{q}_i}} \\ \overline{\mathsf{p}_i} &= \mathit{FA}(\phi) \backslash \mathit{FA}(\Delta_1) \\ \Theta_4 &= \Theta_2 \restriction \overline{\mathsf{p}_i} \\ \Theta_5 &= \Theta_2 \backslash \Theta_4 \end{aligned}$$

and p fresh

Figure 6.8: Extra rules for let-based annotation inference

shall content ourselves with proving syntactic soundness and completeness with respect to NLL$^{\forall\mathrm{let}\nu\leq}$ typings having no free annotation parameters, which is the only natural condition we shall impose on decorations.

6.3.2 Extending the simple inference algorithm

An algorithm for inferring constraint inequations suitable for NLL$^{\forall\mathrm{let}\leq}$ need not be defined from scratch. As we show next, it suffices to extend the algorithm for simple linearity analysis of Figure 6.3, with the two extra rules shown in Figure 6.8.

The rule that handles let-bound variables translates a bound variable x, of type $\forall \overline{\mathsf{p}_i} \,|\, \Theta.\phi$, into a specialisation $x\ \vartheta$, where ϑ is a renaming annotation substitution $\langle \mathsf{p}_1'/\mathsf{p}_1, \ldots, \mathsf{p}_n'/\mathsf{p}_n \rangle$ for the free parameters $\mathsf{p}_1, \ldots, \mathsf{p}_n$ of ϕ. The idea is to let each use of x have its own type $\phi[\vartheta]$, so we introduce fresh annotation parameters at each use. Naturally, any constraints acting on some of the p_i's must also be reflected on their corresponding p_i''s; this explains the introduction of the 'raw' substitution $\Theta[\vartheta]$. (It is easy to see that $\overline{\mathsf{p}_i[\vartheta] \sqsupseteq t_i[\vartheta]}$ will result in a constraint set of the same form if $\theta(\mathsf{p}) = \mathsf{p}'$, for every $\mathsf{p} \in \overline{\mathsf{p}_i}$.) The rule will also ensure that all typing declarations in Δ get $\top$-annotated, as expected from a rule that handles variables.

A local definition $\mathsf{let}\ x = M\ \mathsf{in}\ N$ is translated into

$$\mathsf{let}\ x{:}\forall \overline{\mathsf{p}_i} \,|\, \Theta_4.\phi = \Lambda \overline{\mathsf{p}_i} \,|\, \Theta_4.X\ \mathsf{in}\ Y,$$

where X and Y are obtained from M and N, respectively. The translation of N is considered in a context where x has generalised type $\forall \overline{\mathsf{p}_i} \,|\, \Theta_4.\phi$. The rule is fairly standard, and can be easily understood in terms of the Let rule of NLL$^{\forall\mathrm{let}\leq}$. The only point that may not be clear is how Θ_4 is built from the inequations in Θ_2 of X.

If our algorithm is sound, the translation of X should imply the validity of $\Theta_2 \,;\, \Delta_1 \vdash X : \phi$. By $\forall_{\mathcal{I}}$, we can conclude $\Theta_5 \,;\, \Delta_1 \vdash X : \forall \overline{\mathsf{p}_i} \,|\, \Theta_4.\phi$, if we are able to express Θ_2 as a union Θ_4, Θ_5,

where Θ_4 does not bind any parameters free in Δ, and Θ_4 and Θ_5 satisfy the separation condition. Hence, we take Θ_2 and split it into two: we form Θ_4 by taking all the inequations in Θ_2 that bind any free parameters in ϕ, but which are not free in Δ_1 (namely, $\overline{\mathsf{p}_i}$), and leave the remaining inequations in Θ_5.

It is easy to see that, for the extension of our simple algorithm, all runs are unique.

Proposition 6.3.1 (Determinacy)
If $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$ *and* $\Theta'\,;\Delta \vdash M \Rightarrow X' : \phi'$, *then* $X \equiv X'$, $\phi \equiv \phi'$ *and* $\Theta \equiv \Theta'$. □

6.3.3 Correctness

Following the development of Subsection 6.1.2, we shall now prove soundness and completeness. We begin by observing that the erasure of the translated terms gives us back the input term.

Proposition 6.3.2
If $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$, *then* $X^\circ = M$. □

To prove soundness, we first show how the runs of the algorithm are related to typings in the intermediate type theory $\mathrm{NLL}^{\forall\mathrm{let}\nu\leq}$.

Lemma 6.3.3
If $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$, *then* $\Theta\,;\Delta \underset{\mathrm{NLL}^{\forall\mathrm{let}\nu\leq}}{\vdash} X : \phi$.

Proof. By induction on the structure of M. This proof is basically an extension of the proof of Lemma 6.1.9, so we only show the cases where annotation polymorphism is involved. As always, we reason with respect to the syntax-directed version of the theory.

- $M \equiv x$.

  There are two cases to consider. We have already considered the case not involving annotation polymorphism in the proof of Lemma 6.1.9, on page 122; we consider the polymorphic case here.

  Assume $\Theta\,;\Delta, x : (\forall\overline{\mathsf{p}_i}\,|\,\Theta.\phi)^{\mathsf{p}} \vdash x \Rightarrow x\,\vartheta : \phi[\vartheta]$, where $\Theta \equiv \Theta[\vartheta], \overline{\mathsf{q}_i \sqsupseteq \top}$, $\vartheta \equiv \overline{\langle \mathsf{p}'_i/\mathsf{p}_i\rangle}$, for $\overline{\mathsf{p}'_i}$ fresh, and $\Delta \equiv \overline{x_i : \phi_i{}^{\mathsf{q}_i}}$.

  For this run to be sound, $\Theta\,;\Delta, x : (\forall\overline{\mathsf{p}_i}\,|\,\Theta.\phi)^{\mathsf{p}} \vdash x\,\vartheta : \phi[\vartheta]$ must be valid. The necessary conditions are given by the Identity$^\forall$ rule of $\mathrm{NLL}^{\forall\mathrm{let}\leq}$, and trivially verified in our case. Indeed, we have $dom(\vartheta) = dom(\overline{\langle \mathsf{p}'_i/\mathsf{p}_i\rangle}) = \overline{\mathsf{p}_i}$ and $\Theta \equiv \Theta[\vartheta], \overline{\mathsf{q}_i \sqsupseteq \top} \triangleright \Theta[\vartheta]$. (The inequations $\overline{\mathsf{q}_i \sqsupseteq \top}$ account for the structural condition $\Theta \triangleright |\Delta| \sqsupseteq \top$, which is required by the syntax-directed version.)

- $M \equiv \mathsf{let}\ x = M'\ \mathsf{in}\ N'$.

  In this case, we must have

  $$\Theta\,;\Delta \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N \Rightarrow \mathsf{let}\ x{:}\forall\overline{\mathsf{p}_i}\,|\,\Theta_4.\phi = \Lambda\overline{\mathsf{p}_i}\,|\,\Theta_4.X\ \mathsf{in}\ Y : \psi,$$

  because $\Theta_2\,;\Delta_1 \vdash M \Rightarrow X : \phi$ and $\Theta_3\,;\Delta_2, x : (\forall\overline{\mathsf{p}_i}\,|\,\Theta_4.\phi)^{\mathsf{p}} \vdash N \Rightarrow Y : \psi$, where $\Theta \equiv (\Theta_1, \Theta_3, \Theta_5, \overline{\mathsf{q}_i \sqsupseteq \mathsf{p}})$, $split(\Delta, M, N) = (\Delta_1, \Delta_2, \Theta_1)$, $\overline{\mathsf{p}_i} = FA(\phi)\backslash FA(\Delta_1)$, $\Theta_4 = \Theta_2{\restriction}\overline{\mathsf{p}_i}$, $\Theta_5 = \Theta_2\backslash\Theta_4$ and $\Delta_1 \equiv \overline{x_i : \phi_i{}^{\mathsf{q}_i}}$.

By the induction hypothesis and constraint strengthening, twice, we can deduce

$$\Theta\,;\Delta_2, x : (\forall\overline{\mathsf{p}_i}\,|\,\Theta_4.\phi)^{\mathsf{p}} \vdash Y : \psi \quad \text{and} \quad \Theta, \Theta_4\,;\Delta_1 \vdash X : \phi.$$

The latter depends on the observation that $\Theta_2 = \Theta_4, \Theta_5$ (since $\Theta_4 = \Theta_2{\restriction}\overline{\mathsf{p}_i}$ and $\Theta_5 = \Theta_2 \backslash \Theta_4$ by construction). The desired conclusion,

$$\Theta\,;\Delta \vdash \mathsf{let}\ x{:}\forall\overline{\mathsf{p}_i}\,|\,\Theta_4.\phi = \Lambda\overline{\mathsf{p}_i}\,|\,\Theta_4.X\ \mathsf{in}\ Y : \psi,$$

follows from the Let rule if the conditions $\overline{\mathsf{p}_i} \not\subseteq FA(\Theta\,;\Delta_1)$, $\Theta_4\backslash\overline{\mathsf{p}_i} = \emptyset$, $\Theta \triangleright |\Delta_1| \sqsupseteq \mathsf{p}$ and $\Delta = \Delta_1 \uplus \Delta_2$ hold true. Except for the first condition, the others follow by consideration of the definitions of Θ_4, Θ and *split*, respectively. We are left to prove that $\overline{\mathsf{p}_i} \not\subseteq FA(\Theta\,;\Delta_1)$.

We know $\overline{\mathsf{p}_i} \not\subseteq FA(\Delta_1)$ and $\overline{\mathsf{p}_i} \not\subseteq FA(\overline{\mathsf{q}_i \sqsupseteq \mathsf{p}})$, since $\overline{\mathsf{p}_i} = FA(\phi)\backslash FA(\Delta_1)$ and $\overline{\mathsf{q}_i} \subseteq FA(\Delta_1)$ by definition. From $\Theta_5 = \Theta_2\backslash\Theta_4$, we also deduce that $\overline{\mathsf{p}_i} \not\subseteq FA(\Theta_5)$. Note that $\overline{\mathsf{p}_i} \not\subseteq FA(\Theta_1)$ if $\overline{\mathsf{p}_i} \not\subseteq FA(\Delta_1, \Delta_2)$ according to the definition of *split*. By Proposition 6.1.8 we know ϕ cannot have any free parameters in common with neither Y nor Δ_2, and since $\overline{\mathsf{p}_i} \subseteq FA(\phi)$, it must be the case that $\overline{\mathsf{p}_i} \not\subseteq FA(\Delta_2)$. The fact that $\overline{\mathsf{p}_i} \not\subseteq FA(\Theta_3)$ also follows from Proposition 6.1.8 and the fact that a constraint set may only refer to the annotation parameters of the sequent where it belongs.

□

Theorem 6.3.4 (Soundness)
If $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$, *then* $-\,;\Delta[\theta] \underset{\mathrm{NLL}^{\forall\mathrm{let}\nu\leq}}{\vdash} X[\theta] : \phi[\theta]$, *for all* $\theta \models \Theta$.

Proof. Follows as a corollary of Lemmas 6.3.3, 5.4.8 and 5.4.4, applied in that order. □

The proof of completeness is a simple extension of that for simple annotation inference.

Theorem 6.3.5 (Completeness)
If $\Theta\,;\Delta \vdash M \Rightarrow X : \phi$ *and* $-\,;\Gamma \vdash N : \varphi$ *is any* $\mathrm{NLL}^{\forall\mathrm{let}\nu\leq}$ *decoration of* $\Delta^\circ \vdash X^\circ : \phi^\circ$, *then there exists* $\theta \models \Theta$, *such that* $\Gamma \equiv \Delta[\theta]$, $N \equiv X[\theta]$ *and* $\varphi \equiv \phi[\theta]$. □

6.3.4 Growing constraint sets

The rule that treats let-bound variables in Figure 6.8 generates, for each use of the definition in its context, at least as many constraints as there are constraints in the qualified type $\forall\Theta.\phi$. With nested definitions, it is clear that the size of constraint sets may grow exponentially. For linearity analysis, this could hardly be problematic in terms of computing time (although perhaps not in terms of space!). The simple graph-based algorithm we have sketched in Subsection 6.2.3 requires only a single linear traversal of the graph, to both propagate the node labels and generate the optimal substitution.

The exponential growth of constraint sets might become a problem for more complex structural analyses, for which clever representations of constraint sets are not enough. One (mostly general) solution to this problem has been proposed by Gustavsson and Svenningsson, which relies on considering an extended term annotation language with, what they call, 'constraint abstractions' and applications [34]. They suggest a constraint solving algorithm for computing least fixpoints in polynomial time.

Another approach would consist in reducing the number of constraints needed by restricting type families to constraints of the form $\mathsf{p} \sqsupseteq \bot$, so that general annotation polymorphism can be replaced with simple annotation polymorphism, which is more efficient to implement [67]. Naturally, simple annotation polymorphism is less powerful that general annotation polymorphism, but Wansbrough and Peyton-Jones seem to have obtained reasonable results with this simpler approach.

Yet another approach would consist in exploiting the fact that, instead of reasoning with inequations of the form $\mathsf{p} \sqsupseteq t$, we can equivalently reason with equations of the form $\mathsf{p} = \mathsf{p} \sqcup t$ (which would imply an extension of our term language). This would actually eliminate constraint sets altogether, so all complexity-related problems instantly vanish. We have presented an equivalent formulation of linearity analysis with annotation polymorphism in Section A.2, page 161. No inference algorithm is described there, but it would not be difficult to derive one from a syntax-directed version of the type system[8].

6.4 Modular linearity analysis

The annotation inference strategies we have discussed until now concern stand-alone programs only. Adapting our ideas to programs composed of several modules is not difficult in our case, using our knowledge on how to handle local definitions using general annotation polymorphism.

It is not necessary to define a language of modules to illustrate our annotation inference strategy; it suffices to specify how a module definition should be analysed, and what type should finally appear in the module interface.

We begin by showing the rule we may use to infer constraint inequations for a module definition, which has been derived from the inference rule for the let of Figure 6.8:

$$\frac{\Theta_1 \,;\Delta \vdash M \Rightarrow X : \phi \quad \overline{\mathsf{p}_i} = FA(\phi) \quad \Theta_2 = \Theta_1 \restriction \overline{\mathsf{p}_i} \quad \Theta_3 = \Theta_1 \backslash \Theta_2}{\Theta_3 \,;\Delta \vdash \mathsf{let}\ x = M \Rightarrow \mathsf{let}\ x = \Lambda \overline{\mathsf{p}_i} \,|\, \Theta_2.X : \forall \overline{\mathsf{p}_i} \,|\, \Theta_2.\phi}$$

According to this rule, if $\mathsf{let}\ x = M$ is a module definition, we compute Θ_1 and the translation X as usual, using the rules for inferring contraint inequations we described in the previous sections. We assume Δ contains the typing declarations necessary to type M, where each typing declaration binds a variable to a *closed* qualified type, as follows:

$$\Delta ::= x_1 : (\forall \overline{\mathsf{p}_{1,i}} \,|\, \Theta_1.\phi_1)^{\mathsf{q}_1}, \ldots, x_n : (\forall \overline{\mathsf{p}_{n,i}} \,|\, \Theta_n.\phi_n)^{\mathsf{q}_n}.$$

The annotation parameters $\mathsf{q}_1, \ldots, \mathsf{q}_n$ are fresh annotation parameters, provided for the sole purpose of running the inference algorithm. Each generalised type $\forall \overline{\mathsf{p}_{i,k}} \,|\, \Theta_i.\phi_i$ is supposed to have been saved by the compiler from previous analyses (perhaps as part of the module interfaces).

We build the translation of M, $\Lambda \overline{\mathsf{p}_i} \,|\, \Theta_2.X$, by restricting Θ_1 to the free annotation parameters in ϕ. The restriction of Θ_2 has been simplified from $\Theta_1 \restriction (FA(\phi) \backslash FA(\Delta))$ to $\Theta_1 \restriction FA(\phi)$, because $FA(\phi)$ and $FA(\Delta)$ cannot have any annotation parameters in common. (We note that all the types in Δ are closed, which leaves us with $FA(\Delta) = \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}$.)

[8] We should remark, however, that the alternative type system presented in Section A.2 is less expressive than $\mathrm{NLL}^{\forall}$.

Finally, we take the optimal decoration of the definition to be

$$\mathsf{let}\ x = \Lambda\overline{\mathsf{p}_i} \,|\, \Theta_2[\theta^{\mathrm{opt}'}].X[\theta^{\mathrm{opt}'}] \ : \ \forall\overline{\mathsf{p}_i} \,|\, \Theta_2[\theta^{\mathrm{opt}'}].\phi[\theta^{\mathrm{opt}'}],$$

where $\theta^{\mathrm{opt}'}$ is the extension of the optimal solution of Θ_3 necessary to cover X, as shown in Figure 6.7. The qualified type obtained is the type that should go in the module interface of the definition.

Notice that $X[\theta^{\mathrm{opt}'}]$ may offer some opportunities for inlining; and many more may be 'revealed', if the compiler chooses to inline any uses of x. But that is a different story.

Chapter 7

Abstract structural analysis

In the previous chapters, we have concentrated on the static analysis of linearity properties. In this chapter, we show how the full type theory of linearity analysis can be generalised into a more abstract static analysis framework.

The generalisation is based on the observation that most of the important type-theoretic properties of linearity analysis can still be proved correct for other annotation lattices, besides the concrete 2-point annotation lattice of linearity analysis. The idea is to be able to define properties (annotations) that stand for different *usage patterns* for the structural rules. The abstract framework provides the basic laws that a given set of properties must obey to validate the type-theoretic properties of interest; in particular, we would like all structural properties to be preserved by source language reductions.

From a proof-theoretic viewpoint, the logics that can be derived from the abstract framework we introduce here, are all logics of multiple modalities, of which Bierman has provided various formulations [14]. Naturally, we are interested in a static analysis framework, so our formulation is different in many respects, and it includes annotation subtyping and polymorphism[1]. Jacobs seems to have been the first to seriously discuss the possibility of using two separate modalities for the structural rules of Weakening and Contraction (instead of the usual single ! modality of linear logic) [39], although other people seem to have had the same idea, inspired by different background motivations [69, 70, 4, 21, 43].

We shall also be commenting on some interesting instances of the abstract framework; these include both affine and relevance analyses. Affine analysis is slightly more interesting for inlining than linearity analysis, and relevance analysis is the 'structural' counterpart of strictness analysis, an important part of optimising compilers for call-by-need languages.

7.0.1 Organisation

The contents of this last chapter are organised as follows:

- Section 7.1 introduces the abstract framework for structural analysis through the notion of abstract annotation structure.

- Section 7.2 provides a summary of the typing properties satisfied by the general framework.

[1] It would not be difficult to formalise the correspondence between structural analysis and a suitable framework of multiple modalities.

- Section 7.3 discusses a number of interesting instances of the general framework, including affine and relevance analysis.
- Section 7.4 discusses dead-code elimination, a very simple optimisation that is enabled by applying a simple non-occurrence analysis.
- Section 7.5 argues that intuitionistic relevance logic can be used in practice to approximate strictness properties.

7.1 Structural analysis

7.1.1 Basic definitions

The formulation of our abstract framework is dependent on the notion of annotation structure, defined below.

Definition 7.1.1 (Annotation structure)
An *annotation structure* consists of a 5-tuple

$$\mathbb{A} \equiv \langle A, \sqsubseteq, \mathbf{0}, \mathbf{1}, + \rangle,$$

where

- $\langle A, \sqsubseteq \rangle$ is a non-empty $\sqcup$-semilattice of annotations;
- $\mathbf{0}, \mathbf{1} \in A$ are two (not necessarily distinct) distinguished elements, used to annotate the Weakening and Identity rules, respectively;
- $+ : A \times A \to A$ is a binary contraction operator, used in the Contraction rule to combine annotations, and which must satisfy the following commutative, associative and distributive properties[2]:

$$a + b = b + a \tag{7.1}$$

$$(a + b) + c = a + (b + c) \tag{7.2}$$

$$a \sqcup (b + c) = (a \sqcup b) + (a \sqcup c) \tag{7.3}$$

□

An annotation structure alone is all that is needed to define a structural analysis.

Definition 7.1.2 (Structural analysis)
A *structural analysis* is fully determined by giving an annotation structure $\mathbb{A} \equiv \langle A, \sqsubseteq, \mathbf{0}, \mathbf{1}, + \rangle$, together with the typing rules of Figure 7.1. □

The typing rules are those of linearity analysis (Figure 5.2), except for the Identity and Weakening rules, which have been modified.

By looking at the typing rules, we can have an approximate idea of the intended meaning of the abstract annotations $\mathbf{0}$ and $\mathbf{1}$, as well as the intended role of the general contraction operator. An informal explanation is given by the table shown below. (Let $x : \phi^a$ stand for any typing hypothesis.)

[2] We note that these properties are those of a commutative ring without identity or inverse, where $+$ stands for addition and $\sqcup$ for multiplication.

$$\frac{\Theta \rhd t \sqsupseteq \mathbf{1}}{\Theta\,;x : \phi^t \vdash x : \phi}\ \text{Identity} \qquad \frac{\Sigma(\pi) = \sigma}{\Theta\,; - \vdash \pi : \sigma}\ \text{Primitive}$$

$$\frac{\Theta\,;\Gamma, x : \phi^t \vdash M : \psi}{\Theta\,;\Gamma \vdash \lambda x{:}\phi^t.M : \phi^t \multimap \psi}\ \multimap_{\mathcal{I}}$$

$$\frac{\Theta\,;\Gamma_1 \vdash M : \phi^t \multimap \psi \quad \Theta\,;\Gamma_2 \vdash N : \phi \quad \Theta \rhd |\Gamma_2| \sqsupseteq t}{\Theta\,;\Gamma_1, \Gamma_2 \vdash MN : \psi}\ \multimap_{\mathcal{E}}$$

$$\frac{\Theta\,;\Gamma_1 \vdash M_1 : \phi_1 \quad \Theta\,;\Gamma_2 \vdash M_2 : \phi_2 \quad \Theta \rhd |\Gamma_1| \sqsupseteq t_1 \quad \Theta \rhd |\Gamma_2| \sqsupseteq t_2}{\Theta\,;\Gamma_1, \Gamma_2 \vdash \langle M_1, M_2\rangle^{t_1,t_2} : {\phi_1}^{t_1} \otimes {\phi_2}^{t_2}}\ \otimes_{\mathcal{I}}$$

$$\frac{\Theta\,;\Gamma_1 \vdash M : {\phi_1}^{t_1} \otimes {\phi_2}^{t_2} \quad \Theta\,;\Gamma_2, x_1 : {\phi_1}^{t_1}, x_2 : {\phi_2}^{t_2} \vdash N : \psi}{\Theta\,;\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ \langle x_1, x_2\rangle^{t_1,t_2} = M\ \mathsf{in}\ N : \psi}\ \otimes_{\mathcal{E}}$$

$$\frac{\Theta\,;\Gamma_1 \vdash M : \mathsf{bool} \quad \Theta\,;\Gamma_2 \vdash N_1 : \phi \quad \Theta\,;\Gamma_2 \vdash N_2 : \phi}{\Theta\,;\Gamma_1, \Gamma_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \phi}\ \text{Conditional}$$

$$\frac{\Theta\,;\Gamma, x : \phi^t \vdash M : \phi \quad \Theta \rhd |\Gamma, x : \phi^\top| \sqsupseteq \top}{\Theta\,;\Gamma \vdash \mathsf{fix}\ x{:}\phi.M : \phi}\ \text{Fixpoint}$$

$$\frac{\Theta, \Theta'\,;\Gamma \vdash M : \phi \quad \overline{\mathsf{p}_i} \not\subseteq FA(\Theta\,;\Gamma) \quad \Theta' \backslash \overline{\mathsf{p}_i} = \emptyset}{\Theta\,;\Gamma \vdash \Lambda\overline{\mathsf{p}_i} \,|\, \Theta'.M : \forall\overline{\mathsf{p}_i} \,|\, \Theta'.\phi}\ \forall_{\mathcal{I}}$$

$$\frac{\Theta\,;\Gamma \vdash M : \forall\overline{\mathsf{p}_i} \,|\, \Theta'.\phi \quad \Theta \rhd \Theta'[\vartheta] \quad dom(\vartheta) = \overline{\mathsf{p}_i}}{\Theta\,;\Gamma \vdash M\ \vartheta : \phi[\vartheta]}\ \forall_{\mathcal{E}}$$

$$\frac{\Theta\,;\Gamma \vdash M : \phi \quad \Theta \vdash \phi \leq \psi}{\Theta\,;\Gamma \vdash M : \psi}\ \text{Subsumption}$$

$$\frac{\Theta\,;\Gamma \vdash M : \psi \quad \Theta \rhd t \sqsupseteq \mathbf{0}}{\Theta\,;\Gamma, x : \phi^t \vdash M : \psi}\ \text{Weakening}$$

$$\frac{\Theta\,;\Gamma, x_1 : \phi^{t_1}, x_2 : \phi^{t_2} \vdash M : \psi \quad \Theta \rhd t \sqsupseteq t_1 + t_2}{\Theta\,;\Gamma, x : \phi^t \vdash M[x/x_1, x/x_2] : \psi}\ \text{Contraction}$$

Figure 7.1: The abstract typing rules of structural analysis

if	then
$a \sqsupseteq \mathbf{0}$	$x : \phi^a$ can be discarded
$a \sqsupseteq \mathbf{1}$	$x : \phi^a$ can be used at least once
$a \sqsupseteq b_1 + b_2$	$x : \phi^a$ can be duplicated

The requirement that $\langle A, \sqsubseteq \rangle$ must be a $\sqcup$-semilattice ensures that A has a top element $\top$, and that well-defined approximations $a \sqcup b$ exist for any pair of annotations a, b.

The top element plays a fundamental role in the typing of the fixpoint construct. Notice that reduction is handled by duplicating the body of fixpoint abstractions, which has the effect of duplicating any $\top$-annotated variables. A requirement is therefore that

$$\top = \top + \top$$

be true in all annotation structures, which follows from the fact that $\top \sqsubseteq \top + \top$. Moreover, both Weakening and Identity are available for the top element, since $\top \sqsupseteq \mathbf{0}$ and $\top \sqsupseteq \mathbf{1}$, so all structural analyses contain an intuitionistic fragment, and, therefore, a worst analysis.

Proposition 7.1.3 (Worst analysis)
If $\Gamma \underset{\text{FPL}}{\vdash} M : \sigma$, *then* $- ; \Gamma^\bullet \vdash M^\bullet : \sigma^\bullet$. □

The commutativity and associativity properties of + stand as 'common sense' properties; whereas commutativity is consistent with the fact that typing contexts are sets (and so the order used to contract annotations should not be relevant), associativity is consistent with the fact that the annotation of a variable resulting from several applications of the Contraction rule should as well be independent of the order chosen.

As we shall soon see, the distributivity property is critical to prove the analysis well-behaved with respect to term substitution, a fundamental property needed to ensure correctness. The distributivity property is also responsible for the admissibility of the Transfer rule, as shown by Proposition 7.2.2.

As a trivial example, the simplest annotation structure is based on the singleton annotation set consisting of only $\top$,

$$\langle \{\top\}, \sqsubseteq, \top, \top, + \rangle,$$

where $\top \sqsubseteq \top$ and $\top + \top = \top$, as required. There is not much we can do with such an analysis. A more interesting annotation structure is the one needed to capture linearity analysis.

Definition 7.1.4 (Linearity analysis)
The annotation structure of *linearity analysis* is given by

$$\mathbb{A}_{\text{NLL}} \equiv \langle \{1, \top\}, \sqsubseteq, \top, 1, + \rangle,$$

with

$$1 \sqsubseteq \top,$$

and

$$\begin{array}{ccccc} 1 & + & 1 & = & \top \\ 1 & + & \top & = & \top \\ \top & + & 1 & = & \top \\ \top & + & \top & = & \top \end{array}$$

□

7.2 Type-theoretic properties

The following is a list of some elementary properties involving typing contexts, which support our intuition on the meaning of the special abstract elements $\mathbf{0}$ and $\mathbf{1}$.

Proposition 7.2.1
The following basic properties are satisfied by the abstract framework.

a. *If* $\Theta\,;\Gamma, x : \phi^{\mathbf{1}} \vdash M : \psi$, *and* $\mathbf{0} \not\sqsupseteq \mathbf{1}$ *then* $x \in FV(M)$.

b. *If* $\Theta\,;\Gamma, x : \phi^{\mathbf{0}} \vdash M : \psi$ *and* $\mathbf{0} \not\sqsupseteq \mathbf{1}$, *then* $x \notin FV(M)$.

c. *If* $\Theta\,;\Gamma, x : \phi^{a} \vdash M : \psi$ *and* $x \in FV(M)$, *then* $a \sqsupseteq \mathbf{1}$.

d. *If* $\Theta\,;\Gamma, x : \phi^{a} \vdash M : \psi$ *and* $x \notin FV(M)$, *then* $a \sqsupseteq \mathbf{0}$.

As usual, the underlying order on annotations is implied in the admissibility of the Transfer rule.

Proposition 7.2.2 (Transfer)
The following rule is admissible for structural analysis.

$$\frac{\Theta\,;\Gamma, x : \phi^{t} \vdash M : \psi \quad \Theta \triangleright t' \sqsupseteq t}{\Theta\,;\Gamma, x : \phi^{t'} \vdash M : \psi}\text{Transfer}$$

Proof. By induction on the derivation of $\Theta\,;\Gamma, x : \phi^{t} \vdash M : \psi$. Alternatively, one can show how type derivations containing applications of the Transfer rule can be transformed into equivalent type derivations (i.e., having the same conclusion) that do not contain them by moving the applications of Transfer upwards in the type derivation.

The critical case is when the typing hypothesis $x{:}\phi^{t}$ interacts when any of the structural rules, in particular the Contraction rule. The distributivity property is necessary to justify how

$$\frac{\dfrac{\Theta\,;\Gamma, x_1 : \phi^{t_1}, x_2 : \phi^{t_2} \vdash M : \psi}{\Theta\,;\Gamma, x : \phi^{t_1+t_2} \vdash M[x/x_1, x/x_2] : \psi}\text{Contraction}}{\Theta\,;\Gamma, x : \phi^{(t_1+t_2)\sqcup t'} \vdash M[x/x_1, x/x_2] : \psi}\text{Transfer}$$

may be transformed into

$$\frac{\dfrac{\Theta\,;\Gamma, x_1 : \phi^{t_1}, x_2 : \phi^{t_2} \vdash M : \psi}{\Theta\,;\Gamma, x_1 : \phi^{(t_1\sqcup t')}, x_2 : \phi^{(t_2\sqcup t')} \vdash M : \psi}\text{Transfer}}{\Theta\,;\Gamma, x : \phi^{(t_1\sqcup t')+(t_2\sqcup t')} \vdash M[x/x_1, x/x_2] : \psi}\text{Contraction}$$

(We have naturally relaxed our notation to enhance clarity. We have used the fact that $t = t \sqcup t'$ is logically equivalent to $t \sqsupseteq t'$, and we have allowed to have more complex annotations, which can be replaced by equivalent side conditions involving a fresh annotation parameter.) $\square$

It is straightforward to adapt the different results obtained for linearity analysis to our abstract framework. Some properties, like Unique Typing (for the theory without subtyping), do not depend on the nature of the annotations used. The same can be said regarding the construction of the syntax-directed versions of the type theories, but we must not forget to first adapt the Identity and Primitive rules to use abstract annotations, as follows:

$$\frac{\Theta \triangleright t \sqsupseteq \mathbf{1} \quad \Theta \triangleright |\Gamma| \sqsupseteq \mathbf{0}}{\Theta\,;\Gamma, x : \phi^t \vdash x : \phi}\,\text{Identity} \qquad \frac{\Sigma(\pi) = \sigma \quad \Theta \triangleright |\Gamma| \sqsupseteq \mathbf{0}}{\Theta\,;\Gamma \vdash \pi : \sigma}\,\text{Primitive}$$

The modified Identity rule is clearly derivable in our abstract framework, a fact needed to adapt Lemma 3.4.5:

$$\frac{\dfrac{\Theta \triangleright t \sqsupseteq \mathbf{1}}{\Theta\,;\Gamma, x : \phi^t \vdash x : \phi}\,\text{Identity} \qquad \Theta \triangleright |\Gamma| \sqsupseteq \mathbf{0}}{\Theta\,; x : \phi^t \vdash x : \phi}\,\text{Weakening}$$

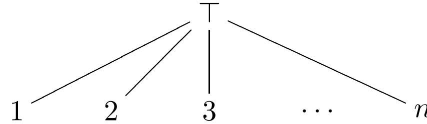
(We proceed similarly for the Primitive rule.)

7.2.1 A non-distributive counter-example

A key property of annotation structures needed to prove the Substitution Lemma is distributivity. Many 'common sense' annotation structures, especially those that rely on precisely counting the occurrences of variables inside terms, are non-distributive and generally violate the substitution property. As a simple example, suppose

$$\mathbb{A} \equiv \langle \mathbb{N} \cup \{\top\}, \sqsubseteq, 0, 1, +\rangle$$

was allowed as an annotation structure, where $\mathbb{N}$ is the set of natural numbers and $n \in \mathbb{N}$ stands for the structural property "occurs exactly n times". We would naturally order properties as shown



and let $+$ stand for the sum of natural numbers, extended cover the case where $\top$ appears as one of the annotations, in which case we would let $a + \top = \top + a = \top$, as expected.

If $\mathbb{A}$ were an annotation lattice, $a \sqsubseteq a + a$ would hold for all annotations a[3]. By the above definition, it is clearly the case that $n \not\sqsubseteq n + n$ for all $n > 0$, a fact that leads to the violation of the term substitution property.

If we are given the two typings

$$x : (\mathsf{int}^1 \otimes \mathsf{int}^1)^2 \vdash \langle x, x\rangle^{1,1} : (\mathsf{int}^1 \otimes \mathsf{int}^1)^1 \otimes (\mathsf{int}^1 \otimes \mathsf{int}^1)^1$$
$$y : \mathsf{int}^2 \vdash \langle y, y\rangle^{1,1} : \mathsf{int}^1 \otimes \mathsf{int}^1$$

[3]This simple fact about annotations poses an intrinsic limit to the 'precision' that can be achieved with structural analysis. We must content ourselves to clearly loose information whenever two annotations are contracted.

the substitution principle states that we can substitute $\langle y, y\rangle^{1,1}$ for x in $\langle x, x\rangle^{1,1}$ if $|y : \mathsf{int}^2| \sqsupseteq 2$ holds, and so is the case. However, the resulting typing judgment

$$y : \mathsf{int}^2 \vdash \langle\langle y, y\rangle, \langle y, y\rangle\rangle^{1,1} : (\mathsf{int}^1 \otimes \mathsf{int}^1)^1 \otimes (\mathsf{int}^1 \otimes \mathsf{int}^1)^1$$

is not provable in the system, since y has retained its original annotation count of 2, but it actually end up having 4 occurrences in the substituted term[4]!

7.2.2 Correctness

We shall now prove that our abstract framework is well-behaved with respect to substitution.

Lemma 7.2.3 (Substitution)
The following rule is admissible for structural analysis.

$$\frac{\Theta\,;\Gamma_1, x : {\phi_1}^t \vdash M : \psi \quad \Theta\,;\Gamma_2 \vdash N : \phi_2 \quad \Theta \rhd |\Gamma_2| \sqsupseteq t \quad \Theta \vdash \phi_2 \leq \phi_1}{\Theta\,;\Gamma_1 \uplus \Gamma_2 \vdash M[N/x] : \psi}\ \text{Substitution}$$

Proof. By induction on the structure of M. We only show one prototypical critical case, which takes place for the rules involving two contexts when the typing hypothesis $x{:}\phi_1$ occurs in both contexts (so the Contraction rule is implicitly involved).

- $M \equiv M'M''$.

  In this case, consider proof Π_1 in Figure 7.3 on page 145. (We have omitted the side-condition $\Theta \vdash \phi_2 \leq \phi_1$.)

  We show that distributivity allows us to rewrite Π_1 as Π_2, where the applications of Substitution are justified by the induction hypothesis. The key step consists in weakening the annotations of Γ_2 using the Transfer rule, by forming $\Gamma_2 \sqcup t_1$ and $\Gamma_2 \sqcup t_2$. The notation $\Gamma \sqcup t$ denotes the replacement of each annotation $t' = |\Gamma(x)|$, for all $x \in \mathit{dom}(\Gamma)$, by $t' \sqcup t$ (by slightly relaxing the notation, as we have done for the proof of the admissibility of Transfer). The idea is to make the structural conditions $\Theta \rhd |\Gamma_2 \sqcup t_1| \sqsupseteq t_1$ and $\Theta \rhd |\Gamma_2 \sqcup t_2| \sqsupseteq t_2$, needed to apply the induction hypothesis to the sub-proofs of $M'[N/x]$ and $M''[N/x]$, trivially hold.

  Also, from $\Theta \rhd |\Gamma_1'', x : {\phi_1}^{t_2}| \sqsupseteq t'$ and $\Theta \rhd |\Gamma_2'| \sqsupseteq t_2$, where $\Gamma_2' = \Gamma_2 \sqcup t_2$, we deduce $\Theta \rhd |\Gamma_1''| \sqsupseteq t'$ and $\Theta \rhd |\Gamma_2'| \sqsupseteq t'$. (The latter follows from the fact that $|\Gamma_2'| \sqsupseteq t_2 \sqsupseteq t'$.) Therefore, by distributivity, it follows that $\Theta \rhd |\Gamma_1'' \uplus \Gamma_2'| \sqsupseteq t'$, which justifies the structural validity of the application of $\multimap_{\mathcal{E}}$.

  It remains to show why $(\Gamma_1' \uplus \Gamma_1'') \uplus \Gamma_2$ and $(\Gamma_1' \uplus (\Gamma_2 \sqcup t_1)) \uplus (\Gamma_1'' \uplus (\Gamma_2 \sqcup t_2))$ are actually the same context. Indeed, by distributivity and the fact that $\Theta \rhd |\Gamma_2| \sqsupseteq t$, we have $\Gamma_2 = \Gamma_2 \sqcup t = \Gamma_2 \sqcup (t_1 + t_2) = (\Gamma_2 \sqcup t_1) \uplus (\Gamma_2 \sqcup t_2)$. The required equivalence follows from the associativity of $\uplus$.

[4] It would not be difficult to define a type system for occurrence analysis where the intuitive semantics of $+$ as the sum of resource counts would give a sound type theory (with respect to the reduction semantics of the underlying language). This requires us to introduce a different notion of substitution. For instance, consider the rule that defines $\Theta\,;\Gamma_1, t \cdot \Gamma_2 \vdash M[x/N] : \psi$ from $\Theta\,;\Gamma_1, x : \phi^t \vdash N : \psi$ and $\Theta\,;\Gamma_2, x : \phi^t \vdash M : \phi$. Here $t \cdot \Gamma_2$ results in a new context where each original annotation $t' = |\Gamma_2(x)|$, for all x, is multiplied by t. However, notice that this notion of substitution suggests updating the annotations as reduction proceeds, so the resulting type theory does not have the same properties as structural analysis does. In particular, its structural annotations would not be invariant with respect to reduction.

$$\frac{\Delta \equiv \overline{x_i : \phi_i^{\mathsf{p}_i}}}{\overline{\mathsf{p}_i \sqsupseteq \mathbf{0}}, \mathsf{p} \sqsupseteq \mathbf{1}\,; \Delta, x : \phi^{\mathsf{p}} \vdash x \Rightarrow x : \phi} \qquad \frac{\Sigma(\pi) = \phi \quad \Delta \equiv \overline{x_i : \phi_i^{\mathsf{p}_i}}}{\overline{\mathsf{p}_i \sqsupseteq \mathbf{0}}\,; \Delta \vdash \pi \Rightarrow \pi : \phi}$$

$$\frac{\vartheta \equiv \overline{\langle \mathsf{p}'_i/\mathsf{p}_i \rangle} \quad \mathsf{p}'_i \text{ fresh} \quad \Delta \equiv \overline{x_i : \phi_i^{\mathsf{q}_i}}}{\Theta[\vartheta], \overline{\mathsf{q}_i \sqsupseteq \mathbf{0}}, \mathsf{p} \sqsupseteq \mathbf{1}\,; \Delta, x : (\forall \Theta.\phi)^{\mathsf{p}} \vdash x \Rightarrow x\ \vartheta : \phi[\vartheta]}$$

Figure 7.2: Modified rules for inferring constraint inequations in structural analysis.

□

It is straightforward now to prove that our abstract theory is correct.

Theorem 7.2.4 (Subject Reduction)
If $\Theta\,; \Gamma \vdash M : \phi$ *and* $M \to N$, *then* $\Theta\,; \Gamma \vdash N : \phi$.

Proof. The proof is essentially that of Theorem 5.4.17. □

7.2.3 Annotation inference

Inferring annotations for structural analysis only requires a simple adaptation of the algorithms for inferring constraint inequations for linearity analysis. We therefore replace the rules for inferring constraint inequations for variables and primitives in Figures 6.3 and 6.8 by the rules shown in Figure 7.2. The modifications match up those required to obtain syntax-directed versions of structural analysis. The resulting annotation inference algorithms can be proved syntactically sound and complete through a simple adaptation of the corresponding theorems for linearity analysis.

The definition of annotation structure we have given in Subsection 7.1.1 does not require that concrete annotation structures have bottom elements. A structural analysis based on an a bottomless annotation structure does not have a unique optimal decoration, but a family of minimum decorations. We also assume the existence of bottom elements when we compute the least solution of a constraint set, so it seems bottom elements have a role to play in the second stage of annotation inference. We therefore assume that, for the second stage of annotation inference, and only if our starting annotation structure $\mathbb{A}$ is not a lattice, we compute solutions with respect to a lifted annotation structure $\mathbb{A}_\bot$, having an artificial $\bot$ element. This affects Steps 2 and 3 of the annotation algorithm of Figure 6.7. The extension $\theta^{\mathrm{opt}'}$ or θ^{opt} must therefore be computed as shown:

$$\theta^{\mathrm{opt}'}(\mathsf{p}) = \begin{cases} \theta^{\mathrm{opt}}(\mathsf{p}), & \text{if } \mathsf{p} \in \mathit{dom}(\theta^{\mathrm{opt}}); \\ \bot, & \text{otherwise.} \end{cases}$$

The reader may like to think of $\bot$ as a structural property conveying *incomplete*, or even *inconsistent*, structural information. In particular, this must mean that it should not be possible to construct a function of type $\phi^\bot \multimap \psi$. In fact, this is the case, and is a simple corollary of Propositions 7.2.1c and 7.2.1d: For any well-typed context $M[x{:}\phi^a] : \psi$, we must have either $a \sqsupseteq \mathbf{0}$ or $a \sqsupseteq \mathbf{1}$, and so $a \neq \bot$[5].

[5] It is however possible to construct a pair of type $\phi^\bot \otimes \psi^\bot$, although there is nothing one can do with it.

The laws of annotation structures justify the transformation of the proof

$$\Pi_1 \equiv \dfrac{\dfrac{\Theta\,;\Gamma_1', x:{\phi_1}^{t_1} \vdash M' : {\psi_1}^{t'} \multimap \psi \qquad \begin{array}{c}\Theta \rhd |\Gamma_1'', x:{\phi_1}^{t_2}| \sqsupseteq t' \\ \Theta\,;\Gamma_1'', x:{\phi_1}^{t_2} \vdash M'':\psi_1\end{array}}{\Theta\,;(\Gamma_1' \uplus \Gamma_1''), x:{\phi_1}^{t} \vdash M'M'':\psi}\multimap_{\mathcal{E}} \qquad \begin{array}{c}\Theta \rhd |\Gamma_2| \sqsupseteq t \\ \Theta\,;\Gamma_2 \vdash N:\phi_2\end{array}}{\Theta\,;(\Gamma_1' \uplus \Gamma_1'') \uplus \Gamma_2 \vdash (M'M'')[N/x]:\psi}\text{Substitution}$$

into the proof

$$\Pi_2 \equiv \dfrac{\dfrac{\Theta\,;\Gamma_1', x:{\phi_1}^{t_1} \vdash M' : {\psi_1}^{t'} \multimap \psi \qquad \dfrac{\Theta\,;\Gamma_2 \vdash N:\phi_2}{\Theta\,;\Gamma_2 \sqcup t_1 \vdash N:\phi_2}\text{Transfer}}{\Theta\,;\Gamma_1' \uplus (\Gamma_2 \sqcup t_1) \vdash M'[N/x] : {\psi_1}^{t'} \multimap \psi}\text{Substitution} \qquad \dfrac{\Theta\,;\Gamma_1'', x:{\phi_1}^{t_2} \vdash M'':\psi_1 \qquad \dfrac{\Theta\,;\Gamma_2 \vdash N:\phi_2}{\Theta\,;\Gamma_2 \sqcup t_2 \vdash N:\phi_2}\text{Transfer}}{\Theta\,;\Gamma_1'' \uplus (\Gamma_2 \sqcup t_2) \vdash M''[N/x]:\psi_1}\text{Substitution}}{\Theta\,;(\Gamma_1' \uplus (\Gamma_2 \sqcup t_1)) \uplus (\Gamma_1'' \uplus (\Gamma_2 \sqcup t_2)) \vdash (M'[N/x])(M''[N/x]):\psi}\multimap_{\mathcal{E}}$$

where $t = t_1 + t_2$, and, in the last proof, $\Theta \rhd |\Gamma_2 \sqcup t_1| \sqsupseteq t_1$, $\Theta \rhd |\Gamma_2 \sqcup t_2| \sqsupseteq t_2$ and $\Theta \rhd |\Gamma_1'' \uplus \Gamma_2| \sqsupseteq t'$.

Figure 7.3: Example critical step in the proof of the substitution property

7.3 Some interesting examples

There are many interesting instances of the abstract framework that may have some practical significance. We briefly review some of these in the following subsections.

7.3.1 Affine analysis

Affine analysis may be understood as a slight variation of linearity analysis aimed at discovering when values are used at most once, instead of precisely once. Affine analysis can be defined in terms of the type system of linearity analysis by allowing Weakening on linear annotations.

Definition 7.3.1 (Affine analysis)
The annotation structure of *affine analysis* is given by

$$\mathbb{A}_{\mathrm{IAL}} \stackrel{\mathrm{def}}{=} \langle \{\top, \leq 1\}, \sqsubseteq, \leq 1, \leq 1, + \rangle,$$

with

$$\leq 1 \sqsubseteq \top,$$

and

$$\begin{array}{ccccc} \leq 1 & + & \leq 1 & = & \top \\ \leq 1 & + & \top & = & \top \\ \top & + & \leq 1 & = & \top \\ \top & + & \top & = & \top \end{array}$$

□

The name of the system, IAL, stands for Intuitionistic Affine Logic.

Notice that the only difference with linearity analysis is that affine analysis sets $\mathbf{0} \equiv \leq 1$, instead of $\top$. The logic underlying the ≤ 1-fragment of IAL is known in the literature under the name of BCK or affine logic; and the calculus that results from such a logic is known under the name of BCK-calculus. (The BCK-calculus can be defined by simply dropping the Contraction rule in the definition of the type system of our source language.)

Affine Logic is an example of a *sub-structural* logic, because it forbids either one of the structural rules. It is a logic of non-reusable information, and its interest dates back to the mid-thirties, and was apparently re-discovered several times by many different people.

Because affine values are, by definition, used at most once, they are good candidates for inlining. This claim is supported by the semantic correctness of the abstract framework, together with the following syntactic property of affine variables.

Proposition 7.3.2 (Affine uses)
If $\Theta\,;\Gamma, x : \phi^{\leq 1} \vdash M : \psi$, *then* $\mathit{occurs}(x, M) \leq 1$.

Proof. Easy induction on the derivation of $\Theta\,;\Gamma, x : \phi^{\leq 1} \vdash M : \psi$. □

The function $\mathit{occurs}(x, M)$ computes the number of times x occurs in M. It is defined inductively on the structure of M in Figure 7.4. Notice that $\mathit{occurs}(x, \mathsf{if}\ \ \mathsf{then}\ x\ \mathsf{else}\ x) = 1$, and not 2, so our notion of 'occurrence' is slightly more semantical in nature, as we do consider the fact that the conditional will evaluate only one of its branches.

$$
\begin{aligned}
occurs(x, \pi) &\stackrel{\text{def}}{=} 0 \\
occurs(x, x) &\stackrel{\text{def}}{=} 1 \\
occurs(x, y) &\stackrel{\text{def}}{=} 0, \quad \text{if } x \not\equiv y \\
occurs(x, \lambda x{:}\phi^t.M) &\stackrel{\text{def}}{=} 0 \\
occurs(x, \lambda y{:}\phi^t.M) &\stackrel{\text{def}}{=} occurs(x, M), \quad \text{if } x \not\equiv y \\
occurs(x, M\ N) &\stackrel{\text{def}}{=} occurs(x, M) + occurs(x, N) \\
occurs(x, \langle M_1, M_2\rangle^{t_1,t_2}) &\stackrel{\text{def}}{=} occurs(x, M_1) + occurs(x, M_2) \\
occurs(x, \mathsf{let}\ \langle x, y\rangle = M\ \mathsf{in}\ N) &\stackrel{\text{def}}{=} occurs(x, M) \\
occurs(x, \mathsf{let}\ \langle y, x\rangle = M\ \mathsf{in}\ N) &\stackrel{\text{def}}{=} occurs(x, M) \\
occurs(x, \mathsf{let}\ \langle y, z\rangle = M\ \mathsf{in}\ N) &\stackrel{\text{def}}{=} occurs(x, M) + occurs(x, N), \quad \text{if } x \not\equiv y \text{ and } x \not\equiv z \\
occurs(x, \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2) &\stackrel{\text{def}}{=} occurs(x, M) + \max\{occurs(x, N_1), occurs(x, N_2)\} \\
occurs(x, \mathsf{fix}\ x{:}\phi.M) &\stackrel{\text{def}}{=} 0 \\
occurs(x, \mathsf{fix}\ y{:}\phi.M) &\stackrel{\text{def}}{=} occurs(x, M), \quad \text{if } x \not\equiv y \\
occurs(x, \Lambda\overline{\mathsf{p}_i} \mid \Theta.M) &\stackrel{\text{def}}{=} occurs(x, M) \\
occurs(x, M\ \vartheta) &\stackrel{\text{def}}{=} occurs(x, M)
\end{aligned}
$$

Figure 7.4: Definition of the $occurs(-,-)$ function

A practical algorithm for inferring constraint inequations for affine analysis (without splitting contexts) can be derived from Figure 6.5 simply by modifying the definition of $|n|$, which should read as follows:

$$|n| = \begin{cases} \top, & \text{if } n > 1; \\ \leq 1, & \text{otherwise.} \end{cases}$$

7.3.2 Relevance analysis

Another interesting example of an analysis based on a sub-structural logic is relevance analysis, which is defined as follows.

Definition 7.3.3 (Relevance analysis)
The annotation structure of *relevance analysis* is given by:

$$\mathbb{A}_{\text{IRL}} \stackrel{\text{def}}{=} \langle \{\top, \geq 1\}, \sqsubseteq, \top, \geq 1, + \rangle,$$

with

$$\geq 1 \sqsubseteq \top$$

and

$$\begin{array}{ccccc} \geq 1 & + & \geq 1 & = & \geq 1 \\ \geq 1 & + & \top & = & \geq 1 \\ \top & + & \geq 1 & = & \geq 1 \\ \top & + & \top & = & \top \end{array}$$

□

The name of the system, IRL, stands for Intuitionistic Relevance Logic[6].

The annotation structures of affine and relevance analysis are almost dual, because the purpose of relevance analysis is to detect values that are used at least once.

Proposition 7.3.4 (Relevance uses)
If $\Theta\,; \Gamma, x : \phi^{\geq 1} \vdash M : \psi$, *then* $occurs(x, M) \geq 1$.

Proof. Easy induction on the derivation of $\Theta\,; \Gamma, x : \phi^{\geq 1} \vdash M : \psi$. □

The logic underlying the ≥ 1-fragment of IRL is known in the literature under the name of relevance logic [3, 28]; and the calculus that results from such a logic is known under the name of λI-calculus [7]. (The λI-calculus is originally untyped; a typed version can be obtained by simply dropping the Weakening rule in the definition of the type system of our source language.)

The type system of IRL may have some interesting applications in the domain of strictness analysis, as suggested by Wright [69]. Some research has been done on usage systems based on relevance logic (or some variants of it), but none of them are refined enough to be practically useful [70, 4, 20, 21]. We shall further discuss strictness analysis in Section 7.5.

[6]The definition of + is actually that of the meet of two annotations, which makes IRL an interesting case from the viewpoint of structural analysis. The obtained structure is clearly a distributive lattice.

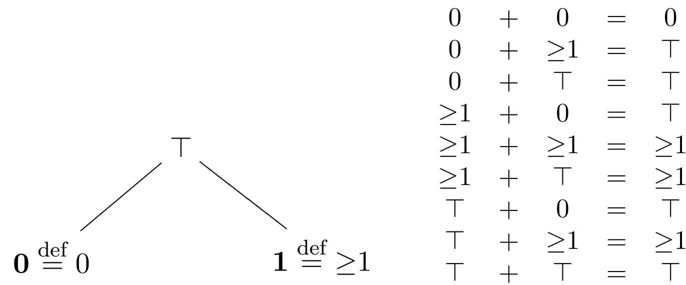


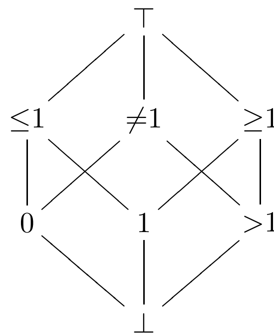
Figure 7.5: An annotation structure for sharing and absence analysis

7.3.3 Combined analyses

There is no reason why we would not be able to combine both affine and strictness analysis into one single combined analysis, or even try out more interesting variations of these two analyses.

Figure 7.5 gives an example of a 'sharing and absence' analysis, suitable for detecting used and unused variables.

There are many possible ways of defining +, but one must always be careful not to violate the upper bound and distributivity properties. The definition we have chosen is consistent with Propositions 7.2.1b and 7.3.4, so $\phi^{\geq 1} \multimap \psi$ is effectively the type of relevant functions and $\phi^{0} \multimap \psi$ is the type of constant functions. (The reader may have noticed that the best we can do in this case is to let + be precisely the least upper bound.) Another well-known annotation lattice is Bierman's lattice, shown below.



Giving a definition of + for this lattice would be a bit long. The only purpose of showing such an interesting lattice is simply to motivate the fact that combined lattices may involve the interaction of many structural properties.

7.4 Dead-code elimination

Eliminating dead code is a simple optimisation that consists in removing those program fragments that will never be evaluated. As a simple illustrative example, consider the following program:

$$\textsf{let}\ \langle x_1, x_2\rangle = \langle 0, 1+2\rangle\ \textsf{in}\ x_1 + 1.$$

Adopting a call-by-need strategy, it is easy to see that the computation '$1+2$', corresponding to the second component of the pair, will never be evaluated, so a clever compiler may choose to leave it out from the final compiled code. The correctness of this observation comes from the fact that x_2 does not occur free in 'x_1+1' and that pair components are substituted for the pair variables unevaluated.

Eliminating the unnecessary computation can be easily achieved for the example above simply by transforming it into the following more compact version:

$$\mathsf{let}\ x_1 = 0\ \mathsf{in}\ x_1 + 1.$$

The transformation is very similar to the inlining transformation on Figure 3.10 in Subsection 3.7.1, except that the substitution need not be performed.

Notice that the above transformation is semantically sound with respect to a call-by-value strategy. In this case, we would be saving not only space, but also computing time, since the value of the computation of '1+2' serves no purpose in the example. For this reason, we use the term *dead-code elimination* to refer to the optimisation that not only takes care of unevaluated code, but also 'unneeded' code. (Some care must be taken, though, in the case of call-by-value: Eliminating unneeded code as we have done above is unsound in the presence of side-effects[7].)

The criterion we choose to detect those cases where dead-code elimination can be applied will be based on *non-occurrence analysis*, which is a rather trivial application of structural analysis.

By Proposition 7.2.1b, $\mathbf{0}$-annotated variables, where $\mathbf{0} \not\sqsupseteq \mathbf{1}$, do not occur in their scope of definition, so context applications can be simplified as we show next. We note that detecting unneeded code must have more to it than non-occurrence analysis; but unfortunately, this is the best we can do in the arena of structural analysis. (If the variable is found to occur in its context, by Proposition 7.2.1c, its usage must be some $a \sqsupseteq \mathbf{1}$, so it is at least affine. This annotation is consistent with the fact that we may choose to evaluate the variable, even if it will be subsequently discarded.)

7.4.1 A simple dead-code elimination transformation

To formalise the dead-code elimination transformation, we shall assume we are given an annotation structure having a null property $0 \not\sqsupseteq \mathbf{1}$, for instance, the annotation structure of sharing and absence analysis of Figure 7.5.

We shall write $\overset{\text{dce}}{\leadsto}$ for the dead-code elimination transformation relation, defined as the contextual closure of the rewrite rules in Figure 7.6.

Proposition 7.4.1 (Correctness)
If $\Theta\,;\Gamma \vdash M : \phi$ and $M \overset{\text{dce}}{\leadsto} N$, then $\Theta\,;\Gamma \vdash M : \phi$.

Proof. Follows from Theorem 7.2.4 and the fact that $\overset{\text{dce}}{\leadsto} \subseteq \rightarrow$. □

It should be better to apply this optimisation before any other optimisations [58]. To see why, consider the following program:

$$\mathsf{let}\ x{:}\mathsf{int}^{\top} = 1 + 2\ \mathsf{in}\ x + ((\lambda y{:}\mathsf{int}^{0}.0)\ x).$$

[7] To see this, replace '1+2' above by some input-output statement. The difference would then be noticeable.

$$(\lambda x{:}\phi^0.M)N \overset{\text{dce}}{\leadsto} M$$
$$\mathsf{let}\ \langle x_1, x_2\rangle^{0,0} = \langle M_1, M_2\rangle^{t_1,t_2}\ \mathsf{in}\ N \overset{\text{dce}}{\leadsto} N$$
$$\mathsf{let}\ \langle x_1, x_2\rangle^{0,t} = \langle M_1, M_2\rangle^{t_1,t_2}\ \mathsf{in}\ N \overset{\text{dce}}{\leadsto} \mathsf{let}\ x_2 = M_2\ \mathsf{in}\ N, \quad \text{if } t \not\equiv 0$$
$$\mathsf{let}\ \langle x_1, x_2\rangle^{t,0} = \langle M_1, M_2\rangle^{t_1,t_2}\ \mathsf{in}\ N \overset{\text{dce}}{\leadsto} \mathsf{let}\ x_1 = M_1\ \mathsf{in}\ N, \quad \text{if } t \not\equiv 0$$
$$\mathsf{let}\ x{:}\phi^0 = M\ \mathsf{in}\ N \overset{\text{dce}}{\leadsto} N$$
$$(\Lambda\overline{\mathsf{p}_i} \,|\, \Theta.M)\ \vartheta \overset{\text{dce}}{\leadsto} M[\vartheta]$$

Figure 7.6: The simple dead-code optimisation relation

Applying the dead-code optimisation to our example would eliminate the 'vacuous' redex, thus eliminating one occurrence of x:

$$\mathsf{let}\ x{:}\mathsf{int}^\top = 1 + 2\ \mathsf{in}\ x + 0.$$

The compiler would then be able to assign (after some amount of re-decorating) a linear type to the transformed function, and apply the inlining transformation, as shown:

$$\mathsf{let}\ x{:}\mathsf{int}^1 = 1 + 2\ \mathsf{in}\ x + 0 \overset{\text{inl}}{\leadsto} (1 + 2) + 0.$$

An obvious improvement in the developement of an actual compiler would consist in reducing the number of re-decoration passes needed to update the structural information of the program. This might be easily implemented by letting each variable occurrence have its own individual annotation, so that the annotation of the variable in the whole context can be computed on-the-fly as the contraction (sum) of all the individual annotations. We may write out this information for our simple example above as shown below. (The notation used should be intuitively clear.)

$$\mathsf{let}\ x{:}\mathsf{int}^{|x_1|+|x_2|} = 1 + 2\ \mathsf{in}\ {x_1}^1 + ((\lambda y{:}\mathsf{int}^0.0)\ {x_2}^\top)$$

Notice how the second occurrence of x, x_2, is given a $\top$ annotation instead of a null annotation. As we have previously discussed, even if x_2 is not needed in its context, in the sense that the information it carries is not required semantically to compute the value of the application, it is however identified as used. This is because there exists a strategy (for instance, call-by-value) that would attempt to evaluate x before computing the application.

7.5 Strictness analysis

As we briefly discussed in Subsection 7.3.2, relevance analysis may be used to detect effective uses of values. Predicting whether functions use their arguments turns out to be important for call-by-need implementations. Evaluated function arguments are handled more efficiently in graph-based implementations than unevaluated ones, so an interesting optimisation consists in evaluating any function arguments that are known to be used before functions are called.

Statistics show that, in practice, most functions written by programmers actually use their arguments[8], so this optimisation plays an important role in the construction of optimising compilers for this family of languages [58].

Knowing whether functions use their arguments has been the main application arena of strictness analysis [47, 17, 8], which has now a very long history. Both strictness and relevance analysis propose two distinct, but related, notions of usage. The notion of usage proposed by relevance analysis is more syntactical in nature than that of strictness analysis, which comes directly from a denotational description of the language.

7.5.1 Approximating strictness properties

Let $\Omega_\sigma \stackrel{\text{def}}{=} \mathsf{fix}\ x{:}\sigma.x$ denote a divergent term; that is, a term for which no reduction exists that leads to a normal form. We say that a source language context $M[x{:}\sigma] : \tau$ is *strict* (*on* x) if the evaluation of $M[\Omega_\sigma/x]$ diverges.

It is clear that divergence is inevitable if, all reduction sequences of $M[x]$, depend on what is substituted for x. This 'material dependence' is precisely what relevance analysis is able to detect. By Theorem 7.2.4 and Proposition 7.3.4, if $M^*[x{:}\phi^{\geq 1}] : \psi$ is a valid IRL decoration of $M[x]$, then there is no reduction sequence that erases x.

The following theorem, that we state here without proof, states that the intuitionistic extension of Belnap's relevance logic [3] provides a sound logical basis for strictness analysis.

Proposition 7.5.1 (Relevance implies strictness)
If $\Theta\,; x : \phi^{\geq 1} \vdash M : \psi$, *then* $M[\Omega_\phi/x]$ *diverges.*

□

(For the above, take $\mathsf{fix}\ x{:}\phi^{\top}.x$ as the definition of Ω_ϕ.)

We should however note that, if relevance implies strictness, the converse is not generally true. (It would be surprising if it were.) Whereas

$$F\ \Omega_\phi \to \Omega_\psi \quad \text{where} \quad F \stackrel{\text{def}}{=} \lambda x{:}\phi.\Omega_\psi,$$

showing that F is clearly strict, it is however not relevant, since x does not occur free in its body. Moreover, by considering a more refined annotation structure having a zero annotation, we might conclude the 'irrelevance' of x in the computation of the body of the function.

7.5.2 Some remarks on lazy evaluation

There is a problem, though, when trying to apply relevance analysis to optimise call-by-need language implementations, as discussed at the beginning of the section.

Proposition 7.5.1 is valid as long as we consider strategies that fully reduce context arguments to normal form. However, both call-by-value and call-by-need strategies do never fully reduce contexts of functional type. Both strategies are thus defined in terms the weaker notion of weak-head normal form (WHNF).

Neededness analysis would find the following function 'strict' on its first argument:

$$H \stackrel{\text{def}}{=} \lambda f{:}(\phi^{\geq 1} \multimap \psi)^{\geq 1}.\lambda x{:}\phi^{\geq 1}.f x;$$

[8] Actually, what statistics have shown is that most functions are strict.

but H is clearly non-strict, because

$$H\ \Omega \to \lambda x{:}\phi^{\geq 1}.\Omega\ x$$

is a terminating reduction sequence.

The same happens if we consider "lazy pairs". The following is a valid typing assertion of pair type:

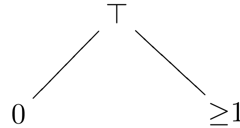
$$x : \phi^{\geq 1} \vdash \langle x, x\rangle^{\geq 1, \geq 1} : \phi^{\geq 1} \otimes \phi^{\geq 1};$$

but, once again, plugging-in Ω yields the lazy value $\langle \Omega, \Omega\rangle^{\geq 1, \geq 1}$.

We should note that Proposition 7.5.1 remains valid for contexts with arguments of ground type, also including pair types having components of ground type. We can therefore apply any strictness-based optimisations provided that we are careful not to do it for functional or pair contexts, for the reasons outlined. How this restriction may affect the performance of generated code is clearly a question that we are not able to answer.

7.5.3 Related work

Baker-Finch has considered a type system for relevance analysis, not different 'in spirit' from ours, inspired from relevance logic [4]. Actually, his system of "strictness types" is closely related to the implicational fragment of a simple version of IRL (without annotation subtyping and polymorphism). He also considers a logic where formulae are annotated using three distinct labels, corresponding to our annotations 0, ≥1 and ⊤, and ordered as shown:



His Contraction rule combines these annotations in a non-distributive fashion.

Static analysis methods that are aimed at separating neededness and non-neededness are usually referred to as "sharing and absence" static analyses. The earliest such system can perhaps be attributed to Johnsson [41]. His theory is actually more directly connected to abstract interpretation (of contexts) rather than to logic, but there are some interesting similarities. An important difference, though, is in the treatment of recursion.

Jacobs [39] studied a logic with two separate modalities, written here $!^0$ and $!^{\geq 1}$, for controlling weakening and contraction, respectively. (We would also need to consider a third modality !, allowing both weakening and contraction[9].) A type system based on Jacob's logic would have two Weakening and two Contraction rules, as follows:

$$\frac{\Gamma \vdash M : \psi}{\Gamma, !^0\phi \vdash M : \psi} \qquad \frac{\Gamma \vdash M : \psi}{\Gamma, !\phi \vdash M : \psi}$$

$$\frac{\Gamma, x_1 : !^{\geq 1}\phi, x_2 : !^{\geq 1}\phi \vdash M : \psi}{\Gamma, x : !^{\geq 1}\phi \vdash M[x/x_1, x/x_2] : \psi} \qquad \frac{\Gamma, x_1 : !\phi, x_2 : !\phi \vdash M : \psi}{\Gamma, x : !\phi \vdash M[x/x_1, x/x_2] : \psi}$$

Unsurprisingly, this rules match our definition of + for IRL.

[9] The interested reader is referred to Bierman's paper [14] for further details.

Benton [11] proposed a simple strictness analyser based on relevance logic where both the analysis and the translation are taken care of within the same typed framework. His typing judgments have the form $J \triangleright J^*$, where J is a typing judgment of the source language and J^* is a typing judgment of an intermediate language corresponding to a variant of Moggi's computational λ-calculus [46]. The subset of intutionistic relevance logic he uses is different from ours, especially tailored to match the corresponding translations into Moggi's language.

Chapter 8

Conclusions

8.1 Summary

We have introduced structural analysis as a form of static analysis for inferring usage information for higher-order typed functional programs. We have formulated our framework in terms of an annotated type system for a target (or intermediate) language, whose terms carry explicit type and usage information. All structural analyses have linear logic as starting point, so most of this thesis concerns the detailed presentation of a case study, linearity analysis, which is aimed at detecting when values are used exactly once. The property 'used exactly once' applies to those values for which no reduction strategy exists that may syntactically erase or duplicate them. Structural properties are therefore not dependent on any reduction strategy, and can effectively be used to enable a number of beneficial source language transformations, for which information about the structural behaviour of programs is needed. To illustrate this possibility, we have seen how the annotations carried by the target terms of linearity analysis could be exploited to formalise a simple inlining transformation. However, since structural analysis can only detect properties that are consistent with all reduction strategies, its range of applicability must somewhat be limited.

Because the target language carries explicit usage information and, although closely related, is different from the source language, inferring structural properties for a source term implies finding its optimal translation into the target language. For linearity analysis, this optimal translation has a parallel in linear proof theory, since it corresponds, by the Curry-Howard correspondence, to the well-studied optimal translation of proofs from intuitionistic into linear logic. Linearity analysis embodies a different characterisation of the same problem, but on the side of functional programming instead of proof theory.

We have extended our basic type theory of linearity analysis with notions of annotation subtyping and annotation polymorphism. Annotation subtyping augments the expressive power of the analysis, as it allows terms to be assigned many different types (of the same subtype family), depending on their use contexts. For first-order type signatures, subtyping suffices to derive all the types necessary by all use contexts. For higher-order type signatures, constrained annotation polymorphism is needed. We have shown that, from a pure static analysis viewpoint, annotation subtyping is subsumed by annotation polymorphism, so there seems to be no reason to use it in practice, other than the fact that it helps to reduce the number of inequations that annotation inference algorithms must consider. From a type-theoretic viewpoint, subtyping corrects a problem introduced by restricting ourselves

to a particular fragment of intuitionistic linear logic; namely, it allows typing information to be preserved across η-reductions of intuitionistic functions. The main motivation for the extension of linearity analysis with annotation polymorphism was to support languages with separately compiled modules. With annotation polymorphism, linearity analysis becomes 'compositional', in the sense that, it becomes possible to analyse a set of definitions, and a program (context) that uses these definitions, separately, without compromising the accuracy of the result. In other words, the resulting analysis provides the same static information as if all the elements had been analysed simultaneously. To do this, we did not require the full power of annotation polymorphism, even though we did propose a theory that is able to accomodate more powerful analyses.

Our strategy for modular linearity analysis was based on a restricted version of our full type theory that allowed constrained annotation polymorphism to be introduced for definitions only, which we called let-based annotation polymorphism. We have shown that the theory accomodates terms that encode the decoration spaces of all source language terms. We have supported this claim constructively, by providing an annotation inference algorithm for which we proved syntactic soundness and completeness results. We based the problem of inferring the simple decoration space of a source language term in terms of the similar problem of inferring the principal type of a term in the theory with annotation polymorphism containing only annotation parameters. We showed that a simple extension of this algorithm suffices to compute principal types of terms drawn from our let-based annotation-polymorphic theory.

Finally, we have shown that only some minor modifications to the original framework of linearity analysis are required to obtain other sorts of structural analysis, including absence, relevance (strictness or sharing), and affine analyses. We have defined a structural analysis in terms of a few properties that ensure its correctness with respect to the underlying reduction semantics. An important correctness criterion is the admissibility of the Substitution rule, which, as we have observed, is easily invalidated by many practical examples. A key property necessary to ensure correctness is the distributivity property, which is also responsible for the admissibility of the Transfer rule. Distributivity is a rather strong property, as it implies that the usage of a variable having multiple occurrences must approximate the information of any of its occurrences.

8.2 Further directions

8.2.1 A generic toolkit

The obvious next step is to generalise our prototype implementation of linearity analysis to support other annotation structures.This would allow us to experiment with other forms of structural analysis, like relevance analysis, to have a first approximate idea of its expressivity and, perhaps, overall performance.

Also, the existing type inference algorithm implements the restricted form of annotation polymorphism we introduced in Section 5.5, so it would be really interesting to extend this algorithm to implement more expressive analyses that would recover the full power of annotation polymorphism[1].

[1] This is indeed possible, although we must always keep in mind that annotation inference must remain within reasonable bounds of complexity.

8.2.2 Computational structural analysis

An interesting question is whether we could obtain more expressive analyses by considering, for instance, a linear version of Moggi's computational meta-language [46]. A source language with a given reduction strategy would give rise to a particular translation into the intermediate language, making the order of evaluation explicit in the syntax.

We might hopefully establish some interesting connections with the typing systems specifically designed with particular reduction strategies in mind, by studying how the annotations are affected by the different translations, with the aim of 'feeding' this information back into the typing rules of the source language. If this works, we could have the best of both worlds, a linear intermediate language verifying subject reduction, and a simple method to derive better analyses for specific reduction strategies.

We must not forget, though, that we would still remain in the realm of structural analysis, so we should not expect the properties obtained to be useful to enable optimisations based on the low-level details of the implementation.

8.2.3 Expressivity and comparison

Expressivity is a rather disturbing issue, as one never knows how it should be addressed on the first place. More expressive analyses generally evolve from simpler analyses, usually because someone has observed that an 'interesting' example is not correctly handled by the existing analysis, or that the analysis does not perform as it was supposed to, compared to other analyses. Examples abound in the literature. We have addressed expressivity by pointing out to some simple results involving decorations. We have shown, for instance, that annotation polymorphism is powerful enough to subsume annotation subtyping (although this may not be desirable for both theoretical and practical reasons), and proved that annotation polymorphism could be successfully used to reason about decoration spaces.

Another way to address expressivity is to compare our work against other existing analyses. This can be quite problematic in our case, as the structural notion of usage is rather different from other notions of usage, especially those that are based upon a denotational description of the source language. A typical example is strictness versus relevance. Strictness contains relevance, but relevance is closer to the intuition one has of usage, which is the property we are actually interested in. The classical counter-example $\lambda x{:}\sigma.\Omega$ of the function that is clearly strict but not relevant is rather pathological in itself; the body of the function clearly diverges, so it is really not important if our relevance analyser fails to see this. But realistic counter-examples that do argue in terms of divergence can be found and refer to the intrinsic difference existing between abstract interpretation and structural analysis, which is in the treatment of recursion and sums. At a certain level, relevance analysis may be understood as a sort of context-centered abstract interpretation (or backwards analysis, to use a relatively forgotten term), where the qualified terms of annotation polymorphism play the role of context-functions, akin to the abstract functions. We have stumbled upon some early work by Johnsson on a static 'sharing and absence' analyser [41]. Even if it was targeted for first-order languages, it has many points of convergence with structural analysis. An exact comparison would need annotation polymorphism in order to emulate the context-functions that are typical of formulations based on abstract interpretation.

Appendix A

An alternative presentation

In this appendix, we draw the attention of the reader to an equivalent presentation of NLL that does not require separate side-conditions for the structural constraints. As such, the system we introduce here is formally more pleasing and slightly more compact. This is especially true of annotation polymorphism.

A.1 The simple case

We begin by showing, in Figure A.1, the typing rules of $\mathrm{NLL}^{\sqcup}$, which corresponds to the alternative formulation of NLL.

This formulation is based on the idea that an inequation $a \sqsupseteq b$ can be rewritten as the equation $a = a \sqcup b$, so that the trivial way to force a to be at least as small as b is to substitute it, everywhere it occurs, by $a \sqcup b$.

Only the rules that have side-conditions in our presentation are modified. These appear implicitly in the conclusion in the form of 'weakened' contexts, of the form Γ^a. This notation is defined as follows:

$$(x_1^{a_1}, \ldots, x_n^{a_n})^b \stackrel{\text{def}}{=} x_1^{a_1 \sqcup b}, \ldots, x_n^{a_n \sqcup b} \tag{A.1}$$

Proposition A.1.1
$\Gamma \underset{\mathrm{NLL}}{\vdash} M : \sigma \Longleftrightarrow \Gamma \underset{\mathrm{NLL}^{\sqcup}}{\vdash} M : \sigma.$

Proof. Easy. As an example, the $\multimap_{\mathcal{E}}$ of $\mathrm{NLL}^{\sqcup}$ is admissible in NLL as the following derivation shows:

$$\dfrac{\Gamma_1 \vdash M : \sigma^a \multimap \tau \qquad \dfrac{\Gamma_2 \vdash N : \sigma}{{\Gamma_2}^a \vdash N : \sigma}\ \text{Transfer}}{\Gamma_1, {\Gamma_2}^a \vdash MN : \tau}\ \multimap_{\mathcal{E}}$$

The side-condition required by the application rule $|{\Gamma_2}^a| \sqsupseteq a$ trivially holds, for any choice of Γ_2 and a. To prove the other direction of the implication, notice that the application rule of $\mathrm{NLL}^{\sqcup}$ coincides with the application rule of NLL precisely when $|\Gamma_2| \sqsupseteq a$, in which case ${\Gamma_2}^a = \Gamma_2$. □

This presentation corresponds closely to the one originally proposed by Bierman [13], with only a few minor notational changes. At first sight, one distinctive difference between our

$$\frac{}{x : \sigma^a \vdash x : \sigma}\ \text{Identity} \qquad \frac{\Sigma(\pi) = \sigma}{- \vdash \pi : \sigma}\ \text{Primitive}$$

$$\frac{\Gamma, x : \sigma^a \vdash M : \tau}{\Gamma \vdash \lambda x{:}\sigma^a.M : \sigma^a \multimap \tau}\ {\multimap}_{\mathcal{I}} \qquad \frac{\Gamma_1 \vdash M : \sigma^a \multimap \tau \quad \Gamma_2 \vdash N : \sigma}{\Gamma_1, {\Gamma_2}^a \vdash MN : \tau}\ {\multimap}_{\mathcal{E}}$$

$$\frac{\Gamma_1 \vdash M_1 : \sigma_1 \quad \Gamma_2 \vdash M_2 : \sigma_2}{{\Gamma_1}^{a_1}, {\Gamma_2}^{a_2} \vdash \langle M_1, M_2 \rangle^{a_1,a_2} : {\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2}}\ \otimes_{\mathcal{I}}$$

$$\frac{\Gamma_1 \vdash M : {\sigma_1}^{a_1} \otimes {\sigma_2}^{a_2} \quad \Gamma_2, x_1 : {\sigma_1}^{a_1}, x_2 : {\sigma_2}^{a_2} \vdash N : \tau}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ \langle x_1, x_2 \rangle = M\ \mathsf{in}\ N : \tau}\ \otimes_{\mathcal{E}}$$

$$\frac{\Gamma_1 \vdash M : \mathsf{bool} \quad \Gamma_2 \vdash N_1 : \sigma \quad \Gamma_2 \vdash N_2 : \sigma}{\Gamma_1, \Gamma_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \sigma}\ \text{Conditional}$$

$$\frac{\Gamma, x : \sigma^\top \vdash M : \sigma}{\Gamma^\top \vdash \mathsf{fix}\ x{:}\sigma.M : \sigma}\ \text{Fixpoint}$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma, x : \sigma^\top \vdash M : \tau}\ \text{Weakening} \qquad \frac{\Gamma, x_1 : \sigma^{a_1}, x_2 : \sigma^{a_2} \vdash M : \tau}{\Gamma, x : \sigma^{a_1 + a_2} \vdash M[x/x_1, x/x_2] : \tau}\ \text{Contraction}$$

Figure A.1: $\text{NLL}^\sqcup$ typing rules

type system and Bierman's is in the formulation of the Conditional rule:

$$\frac{\Gamma_1 \vdash M : \mathsf{bool} \quad \Gamma_2' \vdash N_1 : \sigma \quad \Gamma_2'' \vdash N_2 : \sigma}{\Gamma_1, \Gamma_2' \sqcup \Gamma_2'' \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \sigma}\ \text{Conditional}^{\sqcup}$$

where $\Gamma_2' \sqcup \Gamma_2''$ stands for the join of the two contexts Γ_2' and Γ_2'', defined by

$$(x_1^{a_1}, \ldots, x_n^{a_n}) \sqcup (x_1^{b_1}, \ldots, x_n^{b_n}) \stackrel{\text{def}}{=} (x_1^{a_1 \sqcup b_1}, \ldots, x_n^{a_n \sqcup b_1}).$$

(Therefore, the join of two contexts is defined if both contexts are equal modulo the context annotations.)

It is not difficult to see that the above rule is admissible as a result of the admissibility of the Transfer rule[1]:

$$\frac{\Gamma_1 \vdash M : \mathsf{bool} \quad \dfrac{\Gamma_2' \vdash N_1 : \sigma}{\Gamma_2' \sqcup \Gamma_2'' \vdash N_1 : \sigma}\ \text{Transfer} \quad \dfrac{\Gamma_2'' \vdash N_2 : \sigma}{\Gamma_2' \sqcup \Gamma_2'' \vdash N_2 : \sigma}\ \text{Transfer}}{\Gamma_1, \Gamma_2' \sqcup \Gamma_2'' \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \sigma}\ \text{Conditional}$$

A.2 The annotation polymorphic case

For the alternative version of NLL that includes annotation polymorphism, we can also exploit the idea that an inequation of the form $t \sqsupseteq t'$ can be replaced by an annotation term of the form $t \sqcup t'$. The interesting case is when t or t' contain annotation parameters, in which case they are assumed to be universally quantified.

Universal quantification is simpler to deal with than constrained quantification, so the typing rules of $\text{NLL}^{\forall\sqcup}$ are simpler, as they can be formulated without the need of the notion of constrained set. These are shown in Figure A.2.

The typing rules of $\text{NLL}^{\forall\sqcup}$ differ from those of $\text{NLL}^{\sqcup}$ in the form of the annotation terms, which we may inductively define as follows:

$$t ::= a \mid \mathsf{p} \mid t + t \mid t \sqcup t.$$

Notice that qualified types can now be more compactly written as $\forall \mathsf{p}.\phi$, and similarly for qualified terms.

There are two rules two deal with quantification per se, $\forall_{\mathcal{I}}$ and $\forall_{\mathcal{E}}$, and we have also added a (right) equality rule, $\text{Equality}_{\mathcal{R}}$, useful to be able to reason with types having more complex annotations.

Two types, ϕ and ψ, are regarded as equal if, roughly speaking, they are structurally equivalent under all interpretations of their free parameters[2]:

$$\phi = \psi \text{ implies } \phi[\theta] \equiv_\alpha \psi[\theta], \quad \text{for all } \theta \text{ covering both } \phi \text{ and } \psi. \tag{A.2}$$

The notation Γ^t introduces annotated terms of the form $t \sqcup t'$ into type derivations:

$$(x_1 : {\phi_1}^{t_1}, \ldots, x_1 : {\phi_n}^{t_n})^{t'} \stackrel{\text{def}}{=} x_1 : {\phi_1}^{t_1 \sqcup t'}, \ldots, x_1 : {\phi_n}^{t_n \sqcup t'}. \tag{A.3}$$

[1]This would not be true of some of the type theories of occurrence analysis in the last chapter—the ones based on what we referred to as non-distributive annotation lattices. Like us, Bierman was careful enough to explicitly include the Transfer rule into his system, so in principle he could have used our own version of the conditional.

[2]Equality would be necessary to compare both versions of linearity analysis.

$$\frac{}{x : \phi^t \vdash x : \phi}\ \text{Identity} \qquad \frac{\Sigma(\pi) = \phi}{- \vdash \pi : \phi}\ \text{Primitive}$$

$$\frac{\Gamma, x : \phi^t \vdash M : \psi}{\Gamma \vdash \lambda x{:}\phi^t.M : \phi^t \multimap \psi}\ \multimap_{\mathcal{I}} \qquad \frac{\Gamma_1 \vdash M : \phi^t \multimap \psi \quad \Gamma_2 \vdash N : \phi}{\Gamma_1, {\Gamma_2}^t \vdash MN : \psi}\ \multimap_{\mathcal{E}}$$

$$\frac{\Gamma_1 \vdash M_1 : \phi_1 \quad \Gamma_2 \vdash M_2 : \phi_2}{{\Gamma_1}^{t_1}, {\Gamma_2}^{t_2} \vdash \langle M_1, M_2 \rangle^{t_1,t_2} : {\phi_1}^{t_1} \otimes {\phi_2}^{t_2}}\ \otimes_{\mathcal{I}}$$

$$\frac{\Gamma_1 \vdash M : {\phi_1}^{t_1} \otimes {\phi_2}^{t_2} \quad \Gamma_2, x_1 : {\phi_1}^{t_1}, x_2 : {\phi_2}^{t_2} \vdash N : \psi}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ \langle x_1, x_2 \rangle = M\ \mathsf{in}\ N : \psi}\ \otimes_{\mathcal{E}}$$

$$\frac{\Gamma_1 \vdash M : \mathsf{bool} \quad \Gamma_2 \vdash N_1 : \phi \quad \Gamma_2 \vdash N_2 : \phi}{\Gamma_1, \Gamma_2 \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \phi}\ \text{Conditional}$$

$$\frac{\Gamma, x : \phi^\top \vdash M : \phi}{\Gamma^\top \vdash \mathsf{fix}\ x{:}\phi.M : \phi}\ \text{Fixpoint}$$

$$\frac{\Gamma \vdash M : \phi \quad \mathsf{p} \notin FA(\Gamma)}{\Gamma \vdash \Lambda\mathsf{p}.M : \forall\mathsf{p}.\phi}\ \forall_{\mathcal{I}} \qquad \frac{\Gamma \vdash M : \forall\mathsf{p}.\phi}{\Gamma \vdash M\ t : \phi[t/\mathsf{p}]}\ \forall_{\mathcal{E}}$$

$$\frac{\Gamma \vdash M : \phi \quad \phi = \psi}{\Gamma \vdash M : \psi}\ \text{Equality}_{\mathcal{R}}$$

$$\frac{\Gamma \vdash M : \psi}{\Gamma, x : \phi^\top \vdash M : \psi}\ \text{Weakening} \qquad \frac{\Gamma, x_1 : \phi^{t_1}, x_2 : \phi^{t_2} \vdash M : \psi}{\Gamma, x : \phi^{t_1 + t_2} \vdash M[x/x_1, x/x_2] : \psi}\ \text{Contraction}$$

Figure A.2: $\text{NLL}^{\forall\sqcup}$ typing rules

The simplicity of $\mathrm{NLL}^{\forall\sqcup}$ does come at a price, though. Indeed, $\mathrm{NLL}^{\forall}$ is apparently more expressive. The notations we use for quantified types and terms are different in both theories, so we should first define how to interpret the types and terms in one theory with respect to the other theory. We shall not be formal about this, because the translations are rather cumbersome to define. We only show an example $\mathrm{NLL}^{\forall}$ typing judgment that does not have a counterpart in $\mathrm{NLL}^{\forall\sqcup}$. So, consider the following $\mathrm{NLL}^{\forall}$ type derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{q} \sqsupseteq \mathsf{p}\,;f : (\mathsf{int}^{\mathsf{p}} \multimap \mathsf{bool})^1 \vdash f : \mathsf{int}^{\mathsf{p}} \multimap \mathsf{bool}}\text{Identity} \qquad \dfrac{}{\mathsf{q} \sqsupseteq \mathsf{p}\,;x : \mathsf{int}^{\mathsf{q}} \vdash x : \mathsf{int}}\text{Identity}}{\mathsf{q} \sqsupseteq \mathsf{p}\,;f : (\mathsf{int}^{\mathsf{p}} \multimap \mathsf{bool})^1, x : \mathsf{int}^{\mathsf{q}} \vdash f\ x : \mathsf{bool}}\multimap_{\mathcal{E}}}{\mathsf{q} \sqsupseteq \mathsf{p}\,;x : \mathsf{int}^{\mathsf{q}} \vdash \lambda f{:}(\mathsf{int}^{\mathsf{p}} \multimap \mathsf{bool})^1.f\ x : (\mathsf{int}^{\mathsf{p}} \multimap \mathsf{bool})^1 \multimap \mathsf{bool}}\multimap_{\mathcal{I}}}{-\,;x : \mathsf{int}^{\mathsf{q}} \vdash \Lambda\mathsf{p}\,|\,\mathsf{q} \sqsupseteq \mathsf{p}.\lambda f{:}(\mathsf{int}^{\mathsf{p}} \multimap \mathsf{bool})^1.f\ x : \forall\mathsf{p}\,|\,\mathsf{q} \sqsupseteq \mathsf{p}.(\mathsf{int}^{\mathsf{p}} \multimap \mathsf{bool})^1 \multimap \mathsf{bool}}\forall_{\mathcal{I}}$$

Using the typing rules of $\mathrm{NLL}^{\forall\sqcup}$, the condition $\mathsf{q} \sqsupseteq \mathsf{p}$ must be replaced by an annotation $\mathsf{q} \sqcup \mathsf{p}$ on the typing declaration of x, as required by $\multimap_{\mathcal{E}}$. But we cannot apply the $\forall_{\mathcal{I}}$ rule after abstracting over f, since both p and q appear free in the typing context:

$$\dfrac{\dfrac{\dfrac{}{f : (\mathsf{int}^{\mathsf{p}} \multimap \mathsf{bool})^1 \vdash f : \mathsf{int}^{\mathsf{p}} \multimap \mathsf{bool}}\text{Identity} \qquad \dfrac{}{x : \mathsf{int}^{\mathsf{q}} \vdash x : \mathsf{int}}\text{Identity}}{f : (\mathsf{int}^{\mathsf{p}} \multimap \mathsf{bool})^1, x : \mathsf{int}^{\mathsf{q} \sqcup \mathsf{p}} \vdash f\ x : \mathsf{bool}}\multimap_{\mathcal{E}}}{x : \mathsf{int}^{\mathsf{q} \sqcup \mathsf{p}} \vdash \lambda f{:}(\mathsf{int}^{\mathsf{p}} \multimap \mathsf{bool})^1.f\ x : (\mathsf{int}^{\mathsf{p}} \multimap \mathsf{bool})^1 \multimap \mathsf{bool}}\multimap_{\mathcal{I}}$$

So, choosing the annotations to encode the context restrictions of structural analysis may not be a good idea after all[3].

[3] In the derivation shown, notice that we could instead encode the condition $\mathsf{q} \sqsupseteq \mathsf{p}$ in the type of f, as $f : \mathsf{int}^{\mathsf{p} \sqcap \mathsf{q}} \multimap \mathsf{bool}$, assuming that we enrich the annotation language appropriately. This would imply the need for another version of the $\multimap_{\mathcal{E}}$ rule, as both would be necessary, introducing meets of annotations instead of joins.

Bibliography

[1] Aiken, A., and Wimmers, E. L. Type inclusion constraints and type inference. In *Proceedings of the 7th ACM Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*. ACM Press, 1993, pp. 31–41.

[2] Alberti, F. J. An abstract machine based on linear logic and explicit substitutions. Master's thesis, School of Computer Science, University of Birmingham, December 1997.

[3] Anderson, A. R., and Belnap, N. D. *Entailment*, vol. I. Princeton University Press, 1975.

[4] Baker-Finch, C. A. Relevance and contraction: A logical basis for strictness and sharing analysis.

[5] Barber, A., and Plotkin, G. Dual intuitionistic linear logic. Tech. Rep. ECS-LFCS-96-347, Laboratory for Foundations of Computer Science, University of Edinburgh, October 1997.

[6] Barendregt, H. P. Lambda calculi with types. In *Handbook of Logic in Computer Science*, S. e. a. Abramsky, Ed., vol. 2. Clarendon Press, United Kingdom, 1992, pp. 117–309.

[7] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*, vol. 104 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994. Revised edition.

[8] Benton, N. *Strictness Analysis of Lazy Functional Programs*. PhD thesis, Computer Laboratory, December 1992.

[9] Benton, N. Strictness logic and polymorphic invariance. In *Proceedings of the Second International Symposium on Logical Foundations of Computer Science*, vol. 620 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[10] Benton, N. A mixed linear and non-linear logic: Proofs, terms and models. In *Proceedings of Computer Science Logic, Kazimierz, Poland*, vol. 933 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995, pp. 121–135.

[11] Benton, N. A Unified Approach to Strictness Analysis and Optimising Transformations. Tech. Rep. 388, Computer Laboratory, University of Cambridge, February 1996.

[12] Benton, N., Bierman, G., de Paiva, V., and Hyland, M. A term calculus for intuitionistic linear logic. In *Proceedings of the International Conference on Typed*

Lambda Calculi and Applications, Utrecht, The Netherlands, M. Bezem and J. F. Groote, Eds., vol. 664 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993, pp. 75–90.

[13] Bierman, G. M. Type systems, linearity and functional languages, December 1991. Paper given at the Second Montréal Workshop on Programming Language Theory.

[14] Bierman, G. M. Multiple modalities. Tech. Rep. 455, Computer Laboratory, University of Cambridge, 1998.

[15] Braüner, T. The girard translation extended with recursion. In *Computer Science Logic*, L. Pacholski and J. Tiuryn, Eds. Springer-Verlag, 1994, pp. 31–45.

[16] Burn, G. L. A logical framework for program analysis. In *Proceedings of the 1992 Glasgow Functional Programming Workshop* (July 1992), J. Launchbury and P. Sansom, Eds., Springer-Verlag Workshops in Computer Science series, pp. 30–42.

[17] Burn, G. L., Hankin, C. L., and Abramsky, S. The theory of strictness analysis for higher order functions. In *Programs as Data Objects*, H. Ganzinger and N. D. Jones, Eds., vol. 217 of *Lecture Notes in Computer Science*. Springer-Verlag, October 1986, pp. 42–62.

[18] Chirimar, J., Gunter, C. A., and Riecke, J. G. Proving memory management invariants for a language based on linear logic. In *ACM Conference on Lisp and Functional Programming* (April 1992), ACM Press.

[19] Coppo, M., and Dezani-Ciancaglini, M. A new type-assignment for lambda terms. *Archiv für Mathematische Logik 19* (1978), 139–156.

[20] Courtenage, S. A. *The Analysis of Resource Use in the λ-calculus by Type Inference*. PhD thesis, Department of Computer Science, London, September 1995.

[21] Courtenage, S. A., and Clack, C. D. Analysing resource use in the λ-calculus by type inference. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (1994).

[22] Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (January 1977), pp. 238–252.

[23] Curtis, P. Constrained quantification in polymorphic type analysis. Tech. Rep. CSL-90-1, Xerox Palo Alto Research Center, February 1990.

[24] Damas, L., and Milner, R. Principal type schemes for functional programs. In *Proceedings of the 9'th ACM Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, January 1982).

[25] Danos, V., Joinet, J.-B., and Schellinx, H. The structure of exponentials: uncodering the dynamics of linear logic proofs. In *Computational Logic and Proof Theory*, G. Gottlob, L. A., and M. D., Eds. Springer-Verlag, August 1993, pp. 159–171.

[26] Danos, V., Joinet, J.-B., and Schellinx, H. On the linear decoration of intuitionistic derivations. *Archive for Mathematical Logic 33* (1995), 387–412. Slightly revised and condensed version of the 1993 technical report with the same title.

[27] Davey, B. A., and Priestley, H. A. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[28] Dunn, J. M. Relevance logic and entailment. In *Handbook of Philosophical Logic: Alternatives in Classical Logic*, D. Gabbay and F. Guenthner, Eds., vol. III. Reidel, Dordrecht, 1986, ch. 3, pp. 117–225.

[29] Erik, B., and Sjaak, S. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6 (1996), 579–612.

[30] Girard, J.-Y. Linear logic. *Theoretical Computer Science 50*, 1 (1987), 1–101.

[31] Girard, J.-Y. On the unity of logic. *Annals of Pure and Applied Logic 59* (1993), 201–217.

[32] Girard, J.-Y., Scedrov, A., and Scott, P. J. Bounded linear logic: A modular approach to polynomial time computability. *Theoretical Computer Science 97* (1992), 1–66.

[33] Gustavsson, J. *Space-Safe Transformations and Usage Analysis for Call-by-Need Languages*. PhD thesis, Göteborg Unniversity, 2001.

[34] Gustavsson, J., and Svenningsson, J. Constraint abstractions. In *Proceedings of the Symposium on Programs and Data Objects II* (May 2001), vol. 2053 of *Lecture Notes in Computer Science*, Springer-Verlag.

[35] Gustavsson, J., and Svenningsson, J. A usage analysis with bounded usage polymorphism and subtyping. *Lecture Notes in Computer Science 2011* (2001).

[36] Hindley, J. R. *Basic Type Theory*, vol. 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1997.

[37] Hughes, J. Compile-time Analysis of Functional Programs. In *Research Topics in Functional Programming* (1990), D. Turner, Ed., Addison Wesley.

[38] Igarashi, A., and Kobayashi, N. Resource usage analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2002), ACM Press, pp. 331–342.

[39] Jacobs, B. Semantics of weakening and contraction. *Annals of Pure and Applied Logic 69* (1994), 73–106.

[40] Jensen, T. P. Strictness analysis in logical form. In *Functional Programming Languages and Computer Architecture* (Harvard, Massachusetts, USA, 1991), J. Hughes, Ed., vol. 523 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 352–366.

[41] Johnsson, T. Detecting when call-by-value can be used instead of call-by-need. Tech. Rep. PMG-14, Institutionen för Informations behandling, Chalmers Tekniska Högskola, Göteborg, 1981.

[42] Mackie, I. Lilac: A functional programming language based on linear logic. Master's thesis, Department of Computing, Imperial College, London, 1991.

[43] Maraist, J. Separating weakening and contraction in a linear lambda calculus. Tech. Rep. iratr-1996-25, Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation, 1996.

[44] Milner, R., Tofte, M., Harper, R., and MacQueen, D. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[45] Mogensen, T. Æ. Types for 0, 1, or many uses. In *Proceedings of the Workshop on Implementation of Functional Languages* (September 1997), pp. 157–165.

[46] Moggi, E. Computational lambda-calculus and monads. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science* (1989), pp. 14–23.

[47] Mycroft, A. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, Department of Computer Science, December 1981.

[48] Nielson, F., R., N. H., and Hankin, C. *Principles of Program Analysis*. Springer-Verlag, 1999.

[49] Palsberg, J., and Smith, S. Constrained types and their expressiveness. *ACM Transactions on Programming Languages and Systems 18*, 5 (1996), 519–527.

[50] Peyton Jones, S. L. Compiling Haskell by program transformation: a report from the trenches. In *Proceedings of the European Symposium on Programming (ESOP'96), Linkping, Sweden*, vol. 1058 of *Lecture Notes in Computer Science*. Springer-Verlag, January 1996.

[51] Peyton-Jones, S. L., and Marlow, S. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming 12*, 4&5 (July 2002), 393–433.

[52] Peyton Jones, S. L., and Santos, A. A transformation-based optimiser for Haskell. *Science of Computer Programming 32*, 1:3 (September 1998), 3–47.

[53] Pierce, B. C. *Types and Programming Languages*. MIT Press, 2002.

[54] Pitts, A. M. Operationally-based theories of program equivalence. In *Semantics and Logics of Computation*, P. Dybjer and A. M. Pitts, Eds., Publications of the Newton Institute. Cambridge University Press, 1997, pp. 241–298.

[55] Plotkin, G. D. A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, University of Aarhus, 1981.

[56] Pravato, A., and Roversi, L. $\lambda_!$ considered both as a paradigmatic language and as a meta-language. In *Fifth Italian Conference on Theoretical Computer Science* (Salerno, Italy, 1995).

[57] Roversi, L. A compiler from Curry-typed λ-terms to linear-λ-terms. In *Theoretical Computer Science: Proceedings of the Fourth Italian Conference* (L'Aquila, Italy, October 1992), World Scientific, pp. 330–344.

[58] Santos, A. *Compilation by transformation in non-strict functional languages*. PhD thesis, Department of Computer Science, University of Glasgow, September 1995.

[59] Sulzmann, M., Müller, M., and Christoph, Z. Hindley/Milner style type systems in constraint form. Tech. Rep. ACRC-99-009, University of South Australia, 1999.

[60] Tsung-Min, K., and Mishra, P. Strictness analysis: A new perspective based on type inference. In *FPCA '89, Functional Programming Languages and Computer Architecture* (London, United Kingdom, September 11–13, September 1989), ACM Press, New York, USA, pp. 260–272.

[61] Turner, D. N., and Wadler, P. Operational interpretations of linear logic. *Theoretical Computer Science 227*, 1–2 (1999), 231–248.

[62] Turner, D. N., Wadler, P., and Mossin, C. Once upon a type. In *7'th International Conference on Functional Programming and Computer Architecture* (San Diego, California, June 1995).

[63] Vaughan, R. To inline or not to inline. *Dr. Dobbs Journal* (May 2004).

[64] Wadler, P. Linear types can change the world! In *Programming Concepts and Methods* (Sea of Galilee, Israel, April 1990), M. Broy and C. Jones, Eds., North Holland.

[65] Wadler, P. Is there a use for linear logic? In *Proceedings of the Symposium on Partial Evaluations and Semantics-Based Program Manipulation* (New Haven, Connecticut, June 1991), pp. 255–273.

[66] Wansbrough, K. *Simple Polymorphic Usage Analysis*. PhD thesis, Computer Laboratory, University of Cambridge, 2002.

[67] Wansbrough, K., and Peyton-Jones, S. L. Simple usage polymorphism. In *ACM SIGPLAN Workshop on Types in Compilation* (Montreal, Canada, September 2000).

[68] Wansworth, K., and Peyton-Jones, S. L. Once upon a polymorphic type. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, January 1999).

[69] Wright, D. A. A new technique for strictness analysis. In *TAPSOFT '91*, vol. 494 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1991.

[70] Wright, D. A. Linear, strictness and usage logics. In *CATS '96* (January 1996).