HAL
open science

Static Analyses for Manipulations of Hierarchically Structured Data

Alan Schmitt

To cite this version:

Alan Schmitt. Static Analyses for Manipulations of Hierarchically Structured Data. Software Engineering [cs.SE]. Université de Grenoble, 2011. tel-00637917

HAL Id: tel-00637917

https://theses.hal.science/tel-00637917v1

Submitted on 3 Nov 2011

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.

HABILITATION À DIRIGER LES RECHERCHES

présentée par

Alan SCHMITT

pour obtenir le diplôme d'Habilitation à Diriger les Recherches
de l'Université de Grenoble

(Spécialité: Informatique)

Analyses Statiques pour Manipulations de Données Structurées Hiérarchiquement.

Date de soutenance : 23 mai 2011

Composition du jury :	Président	Joseph Sifakis
	Rapporteurs	Sandrine Blazy
		Giuseppe Castagna
		Matthew Hennessy
	Examinateurs	Florence Maraninchi
		Davide Sangiorgi

Travaux de recherche effectués au sein de l'équipe de Benjamin Pierce (Université de Pennsylvanie, Philadelphie, USA), l'équipe-projet Sardes (INRIA Grenoble – Rhône-Alpes) et l'équipe de Davide Sangiorgi (Université de Bologne, Italie).

Contents

1 Introduction **1**

I Manipulating Unordered Trees **3**

2 Messages as Records **7**
2.1 Introduction 7
2.2 Dream Overview 8
2.3 Simple Message Manipulation 9
2.4 Structural Routing 15
2.5 Semantic Routing 20
2.6 Related Work 27
2.7 Conclusion 28

3 Bidirectional Transformations On Trees **29**
3.1 Introduction 29
3.2 A Small Example 31
3.3 Semantic Foundations 32
3.4 Generic Lenses 36
3.5 Lenses for Trees 37
3.6 Conditionals 46
3.7 Derived Lenses for Lists 48
3.8 Related Work 52
3.9 Conclusions 56

II Manipulating Ordered Data **59**

4 Lenses for Text **63**
4.1 Introduction 63
4.2 Basic String Lenses 68
4.3 Dictionary Lenses 71
4.4 Quasi-Obliviousness 76
4.5 Boomerang 78
4.6 Experience 80
4.7 Related Work 80

5 Xtatic **83**
5.1 Introduction 83
5.2 A Taste of Xtatic 83
5.3 Types 85
5.4 Values 89

5.5 Pattern Matching 92
5.6 Related work 97

6 Efficient Static Analysis of XML Paths and Types 101
6.1 Introduction 101
6.2 Trees with Focus 102
6.3 The Logic 103
6.4 XPath and Regular Tree Languages 105
6.5 Satisfiability-Testing Algorithm 107
6.6 Implementation Techniques 111
6.7 Typing Applications and Experimental Results 111
6.8 Related Work 113
6.9 Conclusion 116

III Manipulating Programs 117

7 HO Core 121
7.1 Introduction 121
7.2 The Calculus 123
7.3 HO Core is Turing Complete 124
7.4 Bisimilarity in HO Core 127
7.5 Barbed Congruence and Asynchronous Equivalences 130
7.6 Axiomatization and Complexity 131
7.7 Undecidability and Static Restrictions 133
7.8 Other Extensions 135
7.9 Concluding Remarks 136

8 Localities and equivalences 139
8.1 Introduction 139
8.2 HOπ and HOπP 140
8.3 Normal bisimulation for LHOπP 147
8.4 Normal bisimulations and HOπP 150
8.5 Contextual Semantics and Howe's Method 152
8.6 Complementary Semantics for HOπ 154
8.7 Characterizing Barbed Congruence for HOπP 158
8.8 Related Work 162
8.9 Conclusion 163

9 Conclusion: Toward Certified Analyses 165

Bibliography 167

Remerciements

Je remercie très chaleureusement Sandrine Blazy, Giuseppe Castagna et Matthew Hennessy qui ont non seulement accepté de relire ce document, mais ont tout fait pour rendre leur rapport à temps lorsque l'échéance a été avancée pour raison administrative. Grâce à eux, la soutenance a pu avoir lieu au moment prévu. Je suis aussi reconnaissant envers Joseph Sifakis, qui a accepté de présider le jury, et Florence Maraninchi et Davide Sangiorgi pour leur participation en tant qu'examinateurs.

J'ai eu la chance de rencontrer lors de mon postdoc à l'université de Pennsylvanie une équipe enthousiaste et sympathique: Benjamin Pierce, Vladimir Gapeyev, Michael Levin, Stephanie Weirich, Steve Zdancewik, Geoff Washburn, Dimitrios Vytiniotis, Nate Foster. J'ai particulièrement apprécié les déjeuners PLClub, très instructifs, et les soirées Sig*, tout autant passionnantes.

Je remercie les membres de l'équipe Sardes, où je suis arrivé en janvier 2004. Pouvoir échanger sur de nombreux sujets, plus ou moins proches de ma thématique, a été très bénéfique. Je suis très reconnaissant envers Jean-Bernard Stefani, notre chef bienveillant, qui après m'avoir accueilli quelques mois durant ma thèse a su aider à mon intégration dans l'équipe. Je garderai un excellent souvenir de nos discussions académiques, mais aussi sur des thèmes plus légers tels les jeux vidéos ou la science-fiction. J'ai également beaucoup apprécié nos activités non professionnelles, comme le tournoi de Can't Stop avec Vivien, Alessio et Valerio, les parties endiablées de Race for the Galaxy avec Jean, les soirées jeux de société avec Florent, Stéphane, Benoit et Jean (encore), les matchs de Starcraft (I et II) avec Sergueï, Thomas et Jean (toujours). Je suis aussi reconnaissant envers Ludovic pour avoir su si bien jouer le rôle de gardien de l'heure, et nous prévenir sans faute pour le café, le déjeuner et la pause.

Au-delà de l'équipe Sardes, j'ai eu la chance d'échanger ou de collaborer avec de nombreux chercheurs de l'INRIA Rhône-Alpes. Je pense en particulier à Nabil Layaïda et Pierre Genevès, dont nos recherches communes constituent une partie de ce document, et à Alain Girault et Gwenaël Delaval pour de stimulantes discussions.

Ce tour d'horizon des collaborations en Rhône-Alpes ne serait être complet sans la mention des poids lourds de l'équipe Plume du LIP à l'ENS Lyon: Tom Hirschowitz, Daniel Hirschkoff, Damien Pous (devenu depuis mon co-bureau) et Romain Demangeon. Que nos échanges et collaborations à venir soient à la hauteur de ces dernières années !

J'ai eu grand plaisir à travailler avec Michaël Lienhardt, Sergueï Lenglet et Claudio Mezzina, trois étudiants en thèse que j'ai encadrés ou co-encadrés. Une partie de ce document est basée sur leur dur labeur, et je leur en suis très reconnaissant.

Je tiens également à remercier les personnes avec qui j'ai travaillé pendant mon congé sabbatique d'un an à l'université de Bologne: Davide Sangiorgi, Ivan Lanese, Jorge Pérez et Cinzia di Giusto. Leur accueil et tolérance quant à mon italien hésitant m'a beaucoup touché.

L'INRIA est un excellent environnement pour chercher sereinement et efficacement. Je voudrais ainsi remercier tous les services qui nous permettent de nous concentrer sur notre recherche, et en particulier les services généraux, les moyens informatiques, la doc et le bureau des cours et colloques.

Je ne saurais bien sûr oublier les assistantes que j'ai eu la chance de côtoyer durant ces sept années, elles sont fondamentales à la réussite d'un projet. Merci donc à Valérie, Elodie, Maud et Diane.

Enfin, je suis très reconnaissant envers mon épouse Christelle, et mes enfants Augustin, Hermine et Albertine, qui ont supporté mes absences, parfois pour plusieurs semaines d'affilé, ou mes moments "dans la lune". Qu'ils en soient infiniment remerciés.

Résumé & Introduction en français

Ce document présente une partie de mes activités de recherche réalisée depuis ma thèse, soutenue en septembre 2002. Je me concentre ainsi sur les travaux portant sur des analyses statiques de programmes manipulant des données hiérarchiquement structurées.

La notion de *donnée* est centrale en informatique. En effet, on peut considérer que l'expressivité d'un programme découle de la variété des données qu'il peut manipuler, soit en entrée, soit créées lors de son exécution. Le programme en lui-même n'est qu'un contrôle fini, guidé par des données.

Ces données sont naturellement structurées. Primitivement, elles prennent la forme de cellules sur un ruban d'une machine de Turing, ou de mots dans une mémoire. Un programmeur assemble ces structures primitives afin d'en construire de plus complexes: nombres codés sur plusieurs mots, chaînes de caractères, enregistrements ... Ces structures sont souvent hiérarchiques : prenez par exemple les *property lists* que l'on trouve dans les systèmes d'exploitation descendant de NextSTEP. Leurs données prennent la forme (récursive) de chaînes de caractères, de tableaux (contenant des données) ou de dictionnaires associants des clés à des données. Cette structure peut paraître simple, elle suffit pourtant pour représenter de manière pratique la plupart des fichiers de configuration sous Mac OS X, ainsi que des ensembles de données plus grosses comme le contenu d'une bibliothèque iTunes.

Dès que l'on impose une structure à des données, on se donne des contraintes sur les manipulations que l'on s'autorise. Par exemple, une fonction retournant le premier élément d'une liste ne peut pas être appliquée à la liste vide. De manière similaire, on ne peut pas accéder à une case d'un tableau au delà de sa dernière case, ou au contenu d'un dictionnaire pour une clé qu'il ne contient pas. Une certaine adéquation entre un programme et la structure des données qu'il manipule doit donc être présente.

Ces contraintes peuvent devenir encore plus cruciales quand les données manipulées ne sont plus statiques mais deviennent elle-mêmes des programmes. C'est ce qui se passe dans le λ-calcul : les données passées en argument aux fonction sont elles-mêmes des termes du λ-calcul, et peuvent donc s'exécuter. L'encodage de Church des entiers en est un parfait exemple : l'entier n est une fonction qui prend en argument une transformation qu'il appliquera n fois à un cas de base, lui aussi passé en argument. Cet entier n'est donc clairement pas une donnée passive. Préserver l'adéquation entre programmes et données peut ainsi être bien plus complexe dans ce cas.

Je m'intéresse dans ce document aux garanties *statiques* que l'on peut donner à des programmes manipulant des données. Le terme *statique* signifie que les garanties se font en considérant simplement le programme, avant son exécution. On peut ainsi exprimer que, quelles que soient les données en entrées, une certaine famille d'erreurs ne se produira pas, ou un programme se comportera comme un autre.

Le document comporte trois parties portant chacune sur une familles de structures de données différentes. La première partie s'intéresse aux données sous forme d'arbres non ordonnés, la deuxième porte sur les structures ordonnées, tandis que la dernière considère des programmes se manipulant eux-mêmes. Chaque partie porte sur deux ou trois familles d'analyses statiques, maintenant présentées.

Les arbres non-ordonnées peuvent être vus comme des dictionnaires ou comme des enregistrements. Dans ce cadre, lire ou supprimer un champ qui n'est pas présent est considéré comme une erreur. De plus, ajouter un champ dont le nom est déjà utilisé est aussi une erreur. Des analyses permettant de garantir que ces erreurs n'ont pas lieu existent depuis longtemps pour des langages séquentiels tel ML (Rémy, 1994a,b). Elles sont adaptés dans le chapitre 2 à un système à composants de manipulation de messages. La principale complexité de cette adaptation est la prise en compte du *routage* des messages, qui permet à plusieurs flots de messages d'être joints, manipulés ensembles, puis séparés. Les systèmes présentés proposent plusieurs formes de routage, donnant des garanties plus ou moins faciles à utiliser.

Tout en restant dans le cadre des arbres non-ordonnés, je me tourne ensuite vers un domaine assez différent: celui des *transformations bidirectionnelles*. Ces transformations peuvent être lues de deux manières complémentaire. Dans un sens, c'est un programme classique qui transforment des données d'un format C dans un format A. Dans l'autre sens, c'est un programme qui propage les modifications apportées aux données dans le format A et les incorpore aux données initiales dans le format C. Ces deux transformations ne sont bien sûr pas arbitraires : elles doivent satisfaire certaines propriétés qui montrent qu'elles sont bien inverses l'une de l'autre, et doivent aussi être en adéquation avec les données manipulées. Le chapitre 3 décrit un langage de programmation dédié permettant de construire de telles transformations, correctes par construction, manipulant des arbres non-ordonnés.

Même si ces arbres permettent d'encoder des structures ordonnées, ces encodages rendent les analyses bien plus complexes et moins précises. C'est pourquoi la deuxième partie se concentre sur des structures ordonnées. Un premier exemple porte sur le *texte structuré*. Cette structure de données est très commune: textes en format LaTeX ou Markdown, fichiers décrivant des calendrier au format iCalendar, fichiers de configuration, code source ... Le chapitre 4 applique l'approche par transformations directionnelles à ces données, toujours sous la forme d'un langage dédié permettant de bâtir des transformations correctes par construction.

Je me tourne ensuite vers une structure textuelle ordonnée très répandue: XML. Je présente dans les chapitres 5 et 6 deux approches pour garantir la bonne manipulation de telles données. La première est basé sur l'extension d'un langage existant, C♯, avec des primitives permettant la manipulation bien typée de XML. La deuxième consiste en un algorithme garantissant la satisfiabilité de requêtes XPath. Cet algorithme peut par exemple être utilisé pour s'assurer qu'une requête est équivalente à une autre, ou qu'une requête retournera toujours au moins un résultat.

La dernière partie, enfin, porte sur les programmes se manipulant eux-mêmes. Plus précisément, je m'intéresse aux *calculs de processus d'ordre supérieur*, et pour de tels calculs à la notion d'équivalence : deux programmes font-ils la même chose ? L'équivalence est une analyse bien plus précise que celles vues précédemment, puisqu'elle porte sur tous les comportements possibles d'un programme. On pourrait donc s'attendre que qu'elle soit extrêmement complexe, voire impossible, à mettre en œuvre. Je montre dans le chapitre 7 que pour un calcul minimal on peut décider l'équivalence de programme efficacement. Ce calcul n'est pour autant pas trivial : il est Turing complet.

J'étudie ensuite dans le chapitre 8 l'influence qu'ont les ajouts de certaines constructions sur la facilité de montrer l'équivalence. Plus précisément, je considère les notions de *restriction de nom*, permettant de cacher un nom à l'extérieur d'un processus, et de *passivation*, permettant de capturer un processus en cours d'exécution. Ajouter ces deux constructions conduit à un calcul de processus pour lequel prouver l'équivalence de programmes est très compliqué.

Bien que j'utilise le pronom "je" dans les paragraphes précédents, chacun des travaux décrits a été réalisé en collaboration avec de nombreuses personnes.

Durant mon postdoc à l'université de Pennsylvanie dans l'équipe de Benjamin Pierce,

j'ai tout d'abord travaillé sur Xtatic, décrit dans le chapitre 5, avec Vladimir Gapeyev, Michael Levin et Benjamin Pierce (2005b; 2005a; 2005c). J'ai ensuite contribué à Harmony (chapitre 3) avec Nate Foster, Michael Greenwald, Jonathan Moore et Benjamin Pierce (2007b). Notre collaboration a continué après mon retour en France sur le projet Boomerang (chapitre 4), réalisé avec Aaron Bohannon, Nate Foster, Benjamin Pierce et Alexandre Pilkiewicz (2008).

J'ai intégré le projet Sardes à l'INRIA Grenoble – Rhône-Alpes en janvier 2004. J'ai travaillé sur le chapitre 2 avec Michaël Lienhardt et Claudio Mezzina (deux étudiants en thèse que j'ai co-encadrés), ainsi que Jean-Bernard Stefani (2008; 2009). Le chapitre 8 représente une grande partie de la thèse de Sergueï Lenglet, que j'ai co-encadré. Jean-Bernard Stefani a également participé à ces travaux (Lenglet et al., 2009b,a).

En parallèle, je débutais une collaboration avec Nabil Layaïda et Pierre Genevès du projet WAM, toujours à l'INRIA Grenoble – Rhône-Alpes. Nos premiers résultats (Genevès et al., 2007) sont décrits dans le chapitre 6.

Enfin, j'ai effectué un séjour sabbatique à l'université de Bologne, dans l'équipe de Davide Sangiorgi. C'est à cette occasion que nous avons réalisé le travail décrit dans le chapitre 7, avec Ivan Lanese, Jorge A. Pérez et Davide Sangiorgi (2010b).

Ce document ne décrit pas mes recherches sur des domaines ne portant pas directement sur les analyses statiques pour des manipulations de données structurées. Manquent en particulier les travaux sur la conception de calculs modélisant des systèmes à composants (Bidinger et al., 2005b; Hirschkoff et al., 2005; Lienhardt et al., 2007), sur la résolution de conflits pour des données répliquées de manière optimiste (Greenwald et al., 2006), sur l'utilisation de schémas pour guider la synchronization de données (Foster et al., 2007a) et sur l'expressivité du π-calcul d'ordre supérieur polyadique (Lanese et al., 2010a).

Enfin, ce document ne décrit pas non plus mes activités d'enseignement. En 2008, j'ai donné des cours ainsi que des travaux dirigés sur les thèmes "Systèmes de Types" et "Systèmes d'Exploitation" à l'Université de Bologne. En 2010 en 2011 j'ai donnée des cours et travaux dirigés sur le thème du λ-calcul à l'Université Joseph Fourier à Grenoble. En 2010 j'ai donné un cours sur "Bisimulations et Calculus de Processus" à l'école doctorale de l'Université Joseph Fourier. Enfin, en 2011 j'ai donné un cours sur "Assistants de Preuves : de la théorie à la pratique" de nouveau à cette école doctorale.

Chapter 1

Introduction

> *"Computer: a programmable usually electronic device that can store, retrieve, and process data."*
>
> —Merriam-Webster dictionary

> *"I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships."*
>
> —Linus Torvalds

Programs manipulate data. We often think that the program is the clever part of the computer, but a program is just some finite control. As in Turing machines, it is the tape, and its infinite potential, that gives its full expressivity to a program. The data empowers the program.

Data has structure. In a computer memory, it consists of cells each containing a small number. The first job of a programmer is typically to build more complex structures that encompass several memory cells, such as records, arrays, or lists. There needs to be consistency between the representation of the data and the way it is manipulated: one cannot look up the head of an empty list, or access an array beyond its last cell.

Complex data structures are usually built upon simpler ones, in a recursive fashion. Lists may contain records, records may contain arrays of lists ... Consider for instance a pervasive data structure in the Mac OS X[1] operating system: the *property list*. Data in a property list may be a string, an array of data, or a record (called a *dictionary*) mapping keys as strings to data. Such simple data model is sufficient to conveniently store preferences for most applications. It is also use for much bigger data sets, such as the description of the contents of an iTunes library.

Data is not necessarily passive, as in the λ-calculus. One point of view is to say that the "program" is β-reduction, and that the data is the λ-term under consideration. An alternative, richer, point of view consists of taking functions in the λ-term as programs, and their arguments as data. Such a program manipulates itself and changes as it executes. Preserving the consistency between the program representation and the data it manipulates is even trickier in this setting.

The study of the adequacy between structured data and the programs that manipulate it is the core topic of this document. It is organized in three parts, from simple to

[1] and iOS, NeXTSTEP, GNUStep

complex data structures. Part I is about unordered trees, part II is about ordered data, and part III is about programs that manipulate themselves. Each part will consist of two or three case studies, focusing on a particular programming language and data model. These case studies will show how static analyses help maintaining the consistency between programs and data.

I use the "we" pronoun through the main body of this document as all of this research has been done in collaboration with many people. The work described in chapters 3, 4, and 5 was done during my postdoc at University of Pennsylvania with Benjamin Pierce, or during the collaboration we kept alive after my stay. Chapter 6 is the result of a collaboration with Pierre Genevès and Nabil Layaïda of the WAM team at INRIA Grenoble. The work of chapter 7 was done when I spent a year at University of Bologna working with Davide Sangiorgi. Finally, the two remaining chapters were done with students I supervised during their PhD in the Sardes team: Michaël Lienhardt for Chapter 2 and Sergueï Lenglet for Chapter 8.

This document does not describe some of the research done on subjects that are not directly about static analyses for the manipulation of structured data. More precisely, I do not address some work about the design of calculi for component-based programming (Bidinger et al., 2005b; Hirschkoff et al., 2005; Lienhardt et al., 2007), about conflict resolution for optimistically replicated data (Greenwald et al., 2006), about the use of schemas to drive data synchronization (Foster et al., 2007a), and about the expressiveness of polyadic higher-order π-calculus (Lanese et al., 2010a).

Part I

Manipulating Unordered Trees

"But at the moment the hobbits noted little but the eyes. These deep eyes were now surveying them, slow and solemn, but very penetrating."

—J.R.R. Tolkien, "Treebeard"

We start our journey by considering manipulations of *trees*. More precisely, we work with an extremely simple form of trees: unordered, edge-labeled trees with no repeated labels among the children of a given node. This model is a natural fit for applications where the data is unordered, such as a dictionary or an address book. A *record* can also be seen as an edge-labeled tree, where the leaves contain some special primitive values.

For these unordered trees, we first consider component-based programs that manipulate messages structured as records. We develop in Chapter 2 a type system that guarantees that record manipulation primitives never fail. In fact, we consider two type systems for two programming languages, that differ in how messages may be *routed*. This work was done in collaboration with Michaël Lienhardt, Claudio Mezzina, and Jean-Bernard Stefani (Lienhardt et al., 2008, 2009), and was the topic of Michaël Lienhardt's PhD dissertation.

We then turn to a different form of tree manipulation in Chapter 3: bidirectional tree transformations. Such transformations may be run forward and backward, changing the shape of the tree. We develop analyses that guarantee that these transformation are well-behaved, in the sense that a round trip returns to its starting point. This work was done in collaboration with Nate Foster, Michael Greenwald, Jonathan Moore, and Benjamin Pierce (Foster et al., 2007b).

Chapter 2

Messages as Records

2.1 Introduction

Our first example of static analyses for the manipulation of unordered trees is set up in the world of *components*, and more precisely *component-based communication frameworks*. Building software systems from components has many benefits compared to less modular approaches (Szyperski, 2002): easier design and development, easier adaptation, maintenance, and evolution. However, constructing a system from components can give rise to non trivial assemblage errors. Some of these errors cannot be detected by the type systems of the programming languages used for implementation, such as C++, Java, or ML. In particular, as noted by Liu et al. (1999), this is the case with communication systems built with dedicated component-based frameworks such as Appia (Miranda et al., 2001), Click (Kohler et al., 2000), Coyote (Bhatti et al., 1998), Dream (Leclercq et al., 2005), or Ensemble (van Renesse et al., 1998). These frameworks comprise many components (sometimes called *micro-protocols*), that encapsulate low-level system code. Assembling micro-protocols can give rise to subtle errors, in particular errors arising because of incompatible manipulation of protocol data units in different components. These errors are hard to catch because they may be purely the result of a faulty assemblage, and may arise even if individual components are correct.

Dealing with assemblage errors in communication systems has been approached in five main ways. First, one may use theorem proving to check the expected properties of an assemblage on a formal specification of the behavior of individual components and of the assemblage, as in Ensemble (van Renesse et al., 1998). The second approach uses an architecture description language (ADL) to specify component behaviors and assemblage constraints, typically component dependencies, and to automatically verify the assemblage consistency, as in Aster (Issarny et al., 1998), Knit (Reid et al., 2000), or Plastik (Joolia et al., 2005). The third approach relies on type systems for interaction contracts, as in the Singularity system (Fähndrich et al., 2006) or in web service workflows (Honda et al., 2008). The fourth approach uses model checking to verify the expected properties of a formally specified assemblage, as in the Vercors system (Barros et al., 2007). Finally, one may rely on property-preserving compositions, as described in Bensalem et al. (2008), where it is applied to deadlock-free assemblages.

The theorem-proving approach is comprehensive and can address arbitrary properties, but it requires theorem-proving expertise, which is not readily available for systems programmers. The ADL approach is more automatic, but it typically supports a limited set of architectural constraints, and a limited set of behavioral checks that fail to address subtler run-time errors such as data manipulation errors. The type-system approach can be made entirely automatic if type inference is decidable, but the type systems devised so far fail to deal with the data handling errors we consider in this chapter.

The model-checking approach is automatic, but may require considerable expertise in the property language used, again not necessarily available for systems programmers. The property-preserving composition approach also can be made entirely automatic, for instance using model checking techniques, but to this date does not readily apply to the data handling errors we consider.

We thus propose an extension of the ADL approach with a type analysis devised to deal with a class of data manipulation errors that occur in ill-formed communication systems assemblages. A first attempt, described in Section 2.4, results in a type system whose type inference is undecidable. We thus slightly reduce the expressive power of *routing* in a second attempt in Section 2.5. In both cases, we first define a simple process calculus to specify an operational *model* of the target component assemblage (and where program execution is abstracted by a *reduction* relation). We then define a domain-specific *type system*, that operates on programs abstracted as terms of the process calculus, and that ensures that typable assemblages do not exhibit the targeted class of errors. These type systems are based on row polymorphism (Rémy, 1994a) and process types (Yoshida and Hennessy, 2002; Maffeis, 2005).

We illustrate our approach with the handling of data manipulation errors that can occur when building incorrect assemblages using the Dream framework (Leclercq et al., 2005). Dream is interesting because it provides one of the more fine-grained frameworks for building communication systems, with constructs that generalize or subsume those in other communication frameworks such as Appia, Click, or Coyote. However, our approach is not limited to Dream only: the same calculus and type system can be applied to Appia (Miranda et al., 2001), Click (Kohler et al., 2000) and Coyote (Bhatti et al., 1998). In fact, in the setting of our second attempt (where type inference is decidable), we describe our implementation which we have used to check Dream and Click assemblages.

This chapter is organized as follows. After a brief overview of the Dream framework, we illustrate in Section 2.3 our approach with a very simple calculus without routing. We then consider structural routing in Section 2.4 and semantic routing in Section 2.5. We discuss related work in Section 2.6 and conclude the chapter in Section 2.7.

2.2 Dream Overview

To explain the assemblage verifications we target in this chapter, we use the example of the Dream framework, which we now briefly present. Dream is a component-based framework, written in Java, designed for the construction of communication systems (protocol stacks, communication subsystems of middleware for distributed execution). It is built on top of the Java implementation of the Fractal component model (Bruneton et al., 2006).

The primary data structure in Dream is called a *message*. Messages are used to implement protocol data units (in other words, the data that communication protocols exchange during their execution). Messages are exchanged between Dream components through input and output *channels*. A message is a list of *labeled chunks*, which can be any Java objects including messages. Within a component, messages can be freely manipulated. Basic operations, like removing, adding, or accessing chunks are provided. The Dream framework comprises a library of components that encapsulate functions and behaviors commonly found in communication subsystems. These include: *message queues* that are used to store messages, *transformers* that transform a message received on their single input channel and deliver the result to their single output channel, *routers* that forward messages received on their single input channel to one or several output channels, *multiplexers* that forward messages received on their input channels to their single output channel, *aggregators* that aggregate messages received on

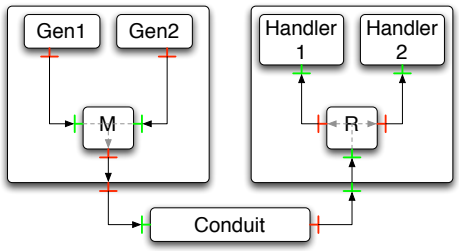


Figure 2.1: A DREAM Assemblage

one or several input channels and deliver the aggregated message on their single output channel, *deaggregators* that are dual to aggregators, and *conduits* that allow messages to be exchanged between different address spaces.

Figure 2.1[1] shows a simple assemblage of DREAM components that corresponds to two communicating sites, `Site A` sending different kinds of messages to `Site B`. The assemblage is constituted by two generator components, `Gen1` and `Gen2`, that emit different messages. These messages are then sent to a multiplexer and transferred to `Site B`. On `Site B`, router `R` forwards messages to the `Handler 1` or `Handler 2` component

Verifying the correctness of the assemblage implies verifying structural constraints to guarantee that input and output channels are properly matched, and ensuring that a component does not receive a message it is not able to handle (typically, a message with missing or unexpected chunks). In our simple example above, this could be the case if the conduit between the sites could not handle messages generated by the two components `Gen1` and `Gen2` (for instance because of a missing chunk), or if one handler could not process the messages forwarded to it. In the presence of complex assemblages, such an analysis can quickly become difficult.

2.3 Simple Message Manipulation

To whet our appetite, we will first consider a very simple family of component assemblages to manipulate messages, and the associated type system.

Message Manipulation Calculus

In this setting, the contents of messages are records associating *fields* (or names) to terms. We write $\{a_1 = M_1; \ldots; a_n = M_n\}$ for the record with fields a_1 to a_n and associated terms M_1 to M_n. As in DREAM, we call a *chunk* the pair of a field and associated term. Note that a given field occurs at most once in a record. For convenience, we write in the following $(M_1, \ldots, M_n)$ for the tuple represented as a record indexed by integers $\{1 = M_1; \ldots; n = M_n\}$.

Record manipulation consists of adding chunks, removing chunks, or accessing chunks (i.e., looking up the term associated to a given field). We write $(.\, a\ M)$ for the selection of field a in record M, $(+_a\ (M', M))$ for the addition of a chunk named a with contents M' in record M, and $(\backslash a\ M)$ for the removal of the chunk named a in record M. As we may consider other manipulations, for instance arithmetic operations on integers, we assume we have a set of constants c that include the record manipulations described above. To simplify notations, we write the chunk removal and access constants as postfix

[1] The figure is simplified: we do not detail the conduit between site A and site B.

(as in $\{a = \text{''}\texttt{foo}\text{''}, b = 1\} \setminus a$), and the chunk addition constant as infix. We also assume, for our examples, a constant `print` that takes a string as argument and prints it.

This approach, based on a simple specification language, is very simple but not very powerful, as seen below. We describe in Section 2.5 a more expressive approach based on *primitive components* that may have arbitrary behaviors.[2]

A message on *channel* e with contents M is written $\overline{e}\langle M\rangle$. A message receiver $e(x).\,B$ is identical to a receiver of value-passing CCS: behavior B is associated to the channel e, and upon reception of a message $\overline{e}\langle M\rangle$, the formal variable x is substituted by the actual message contents M in B. Behaviors B include parallel composition of behaviors $B \parallel B$, sending of messages $\overline{e}\langle M\rangle$, the inactive behavior $\mathbf{0}$, additional receivers $e(x).\,B$, and replicated behaviors $!B$.

Finally, a program D is organized as a hierarchy of component. A component $b^I_O\,[D]$ consists of a name b, a set of channel names I it claims to define, a set of channel names O it requires from the environment, and its contents. To lighten the syntax, When either the set I or O is empty, we simply leave it blank. A program may also be a simple behavior B or the parallel composition of components $D \parallel D$.

Formally, the syntax of our first calculus is as follows.

$$
\begin{array}{rcll}
D & ::= & b^I_O\,[D] \quad | \quad B \quad | \quad D \parallel D & \text{Components} \\
B & ::= & \mathbf{0} \quad | \quad \overline{e}\langle M\rangle \quad | \quad B \parallel B \quad | \quad e(x).\,B \quad | \quad !B & \text{Behaviors} \\
M & ::= & x \quad | \quad \{a_1 = M_1; \ldots; a_n = M_n\} \quad | \quad c \quad | \quad (M_1\ M_2) & \text{Message contents} \\
c & ::= & .\,a \quad | \quad +_a \quad | \quad \backslash a \quad | \quad \texttt{print} \quad | \quad \ldots & \text{Constants}
\end{array}
$$

In the following, we consider as *values* a special subset of messages that may not further reduce:

$$
v \quad ::= \quad \{a_1 = v_1; \ldots; a_n = v_n\} \quad | \quad x \quad | \quad c
$$

Examples

We start by describing some simple behaviors. Our first behavior generates some messages indefinitely: $!\overline{g}\langle\{a = 1; c = \text{''}\texttt{foo}\text{''}\}\rangle$.

Our second behavior receives such a message on channel h and prints the contents of the c field. As the `print` constant may only be used in the contents of a message, the resulting empty record is sent on another message: $!h(x).\,\overline{s}\langle\texttt{print}(x.\,c)\rangle$. This message may in turn be thrown away by the following behavior: $!s(x).\,\mathbf{0}$.

One may combine these last two behaviors in a component H to hide the extra message that has to be thrown away, exposing only the receiver on h.

$$
H^h \begin{bmatrix} !h(x).\,\overline{s}\langle\texttt{print}(x.\,c)\rangle \\ !s(x).\,\mathbf{0} \end{bmatrix}
$$

Note that we omit the parallel operator when behaviors are stacked vertically as in the example above.

To illustrate record manipulations, the following behavior decreases the `ttl` field of a message: $!e(x).\,(x \setminus \texttt{ttl}) +_{\texttt{ttl}} (x.\,\texttt{ttl} - 1)$.

It should be noted that this calculus is not able to express some behaviors, such as throwing away a message whose `ttl` has reached 0. This is the case even in the presence of a constant that is a conditional. Indeed, constants can only be applied to the contents

[2] We could also use an arbitrary language, like the λ calculus, but we prefer to focus on record manipulations for this domain-specific language.

$$\textsc{Context-D}\quad \frac{D \;\rhd\; D'}{\mathbb{E}_D[D] \;\rhd\; \mathbb{E}_D[D']} \qquad \textsc{Context-M}\quad \frac{M \;\rhd\; M'}{\mathbb{E}_M[M] \;\rhd\; \mathbb{E}_M[M']} \qquad \textsc{App}\quad \frac{\texttt{eval}(c, v) = M}{(c\; v) \;\rhd\; M}$$

$$\textsc{Com}\quad \overline{e}\langle v\rangle \parallel e(x).\, B \;\rhd\; \{v/x\}B \qquad \textsc{In}\quad \frac{e \in I}{\overline{e}\langle v\rangle \parallel b_O^I\,[D] \;\rhd\; b_O^I\,[\overline{e}\langle v\rangle \parallel D]}$$

$$\textsc{Out}\quad \frac{s \in O}{b_O^I\,[\overline{s}\langle v\rangle \parallel D] \;\rhd\; b_O^I\,[D] \parallel \overline{s}\langle v\rangle}$$

Figure 2.2: Reduction rules

of a message, thus they cannot condition the sending of a message. We could add a conditional at the level of behaviors, but as the more complex calculi we introduce later are able to deal with this, we prefer to lighten the presentation at the cost of expressivity.

A *binding* or *connector* between components may be defined as $!g(x).\, \overline{h}\langle x\rangle$. We may thus combine the examples above using such a connector to forward the message on g to the receiver on h of the second component.

$$G_g\,[!\overline{g}\langle\{a = 1; c = \text{''}\texttt{foo}\text{''}\}\rangle] \quad \parallel \quad !g(x).\, \overline{h}\langle x\rangle \quad \parallel \quad H^h \begin{bmatrix} !h(x).\, \overline{s}\langle \texttt{print}(x.\, c)\rangle \\ !s(x).\, \mathbf{0} \end{bmatrix}$$

The Semantics of Message Manipulation

The operational semantics of our calculus is defined by a reduction relation, denoted $\rhd$, defined on closed terms (terms with no free variable), modulo a structural equivalence relation on process terms and modulo evaluation contexts. Structural equivalence makes the parallel operator commutative, associative, with neutral element $\mathbf{0}$, allows the unfolding of replications, and makes irrelevant the order of fields in a record.

The reduction relation is defined as the smallest relation that verifies the rules of Figure 2.2. We assume given a partial function `eval` that gives the result of applying a constant to a value. This function is partial as record operations are not always defined. We formally define `eval` in Figure 2.3.

The two context rules mean that reduction, i.e.execution, can happen anywhere in the program. Evaluation contexts are terms with a *hole* []. We distinguish contexts which may be filled by components or behaviors, written $\mathbb{E}_D$, from contexts which may be filled by message contents, written $\mathbb{E}_M$.

$$\begin{array}{lllllllll} \mathbb{E}_M ::= [] & | & \{a_1 = \mathbb{E}_M; a_2 = M_2; \ldots; a_n = M_n\} & | & (M\; \mathbb{E}_M) & | & (\mathbb{E}_M\; M) \\ \mathbb{E}_D ::= [] & | & \overline{s}\langle \mathbb{E}_M\rangle & | & \mathbb{E}_D \parallel D & | & b_O^I\,[\mathbb{E}_D] \end{array}$$

Rule App deals with application in message contents, and relies on the definition of `eval`. Three communication rules are defined: Com allows communication between two programs in a same component; In allows a message in a component that accepts it as input; Out allows a message out of a component that declares its channel name as output. These communication rules may only be applied when the message is fully reduced to a value.

$$\textsc{Select}\quad \texttt{eval}(.\,a, \{a = M_1; a_2 = M_2; \ldots; a_n = M_n\}) = M_1$$

$$\textsc{Add}\quad \frac{\forall 0 < i \leq n,\ a_i \neq a}{\texttt{eval}(+_a, (M, \{a_1 = M_1; \ldots; a_n = M_n\})) = \{a = M; a_1 = M_1; \ldots; a_n = M_n\}}$$

$$\textsc{Remove}\quad \texttt{eval}(-_a, \{a = M_1; a_2 = M_2; \ldots; a_n = M_n\}) = \{a_2 = M_2; \ldots; a_n = M_n\}$$

$$\textsc{Print}\quad \texttt{eval}(\texttt{print}, s) = \{\}$$

Figure 2.3: Constant application

Message Errors

The errors we consider are the ones resulting from incorrect record manipulation. More precisely, an error occurs when there is an application $(v\ v')$ such that $\texttt{eval}(v, v')$ is undefined. This may be because v is not a constant, or because v' is not in the domain of this constant. Such errors thus includes accessing or removing a chunk that is not present, as well as adding a chunk whose name is already in the record.[3]

Consider the following component assemblage, identical to the previous example with a small modification: the message generator puts the string on field b instead of field a.

$$G_g\,[!\overline{g}\langle\{a = 1; b = ''\texttt{foo}''\}\rangle] \quad \| \quad !g(x).\,\overline{h}\langle x\rangle \quad \| \quad H^h \begin{bmatrix} !h(x).\,\overline{s}\langle \texttt{print}(x.\,c)\rangle \\ !s(x).\,\mathbf{0} \end{bmatrix}$$

We compute as depicted in Figure 2.4. In component H, we end up with an application of the accessor $.\,c$ to the record $\{a = 1; b = ''\texttt{foo}''\}$. As $\texttt{eval}(.\,c, \{a = 1; b = ''\texttt{foo}''\})$ is undefined, this is an error, and the computation is stuck for this particular message.

Formally, a configuration D *has an error* iff $D \equiv \mathbb{E}_D[\mathbb{E}_M[(v\ v')]]$ where $\texttt{eval}(v, v')$ is undefined. This definition is motivated, from a practical point of view, by assemblage errors that occur most when using the Dream framework. Dream components manipulate messages by adding a chunk to it, or by removing or accessing one of its chunks. Such operations may fail when applied to messages with the wrong structure, e.g., not having the field the component has to access. A Dream assemblage can typecheck correctly as a Java program (as every message has an atomic `Message` type that does not reveal its structure), but still exhibit run-time errors because of this message manipulation operations.

Typing Components

We now give the intuition for a type system that will ensure that a well-typed configuration never evolves to one that has an error. This type system is a simple extension of the type system for extensible records of Rémy (Rémy, 1994a,b). We present this type system by examples, highlighting a few typing rules. We will not formally define the full

[3] In sequential languages, an error can be detected by a computation that is not a value being stuck. As usual for concurrent languages, we need to be more precise in the definition of where the error is, since other computations may reduce in parallel of an error.

$$G_g\,[!\overline{g}\langle\{a=1;b=''\texttt{foo}''\}\rangle] \;\parallel\; !g(x).\,\overline{h}\langle x\rangle \;\parallel\; H^h\begin{bmatrix}!h(x).\,\overline{s}\langle\texttt{print}(x.\,c)\rangle\\ !s(x).\,\mathbf{0}\end{bmatrix} \equiv$$

$$G_g\begin{bmatrix}!\overline{g}\langle\{a=1;b=''\texttt{foo}''\}\rangle\\ \overline{g}\langle\{a=1;b=''\texttt{foo}''\}\rangle\end{bmatrix} \;\parallel\; !g(x).\,\overline{h}\langle x\rangle \;\parallel\; H^h\begin{bmatrix}!h(x).\,\overline{s}\langle\texttt{print}(x.\,c)\rangle\\ !s(x).\,\mathbf{0}\end{bmatrix} \rhd$$

$$G_g\,[!\overline{g}\langle\{a=1;b=''\texttt{foo}''\}\rangle] \;\parallel\; \begin{matrix}\overline{g}\langle\{a=1;b=''\texttt{foo}''\}\rangle\\ !g(x).\,\overline{h}\langle x\rangle\end{matrix} \;\parallel\; H^h\begin{bmatrix}!h(x).\,\overline{s}\langle\texttt{print}(x.\,c)\rangle\\ !s(x).\,\mathbf{0}\end{bmatrix} \rhd$$

$$G_g\,[!\overline{g}\langle\{a=1;b=''\texttt{foo}''\}\rangle] \;\parallel\; \begin{matrix}\overline{h}\langle\{a=1;b=''\texttt{foo}''\}\rangle\\ !g(x).\,\overline{h}\langle x\rangle\end{matrix} \;\parallel\; H^h\begin{bmatrix}!h(x).\,\overline{s}\langle\texttt{print}(x.\,c)\rangle\\ !s(x).\,\mathbf{0}\end{bmatrix} \rhd$$

$$G_g\,[!\overline{g}\langle\{a=1;b=''\texttt{foo}''\}\rangle] \;\parallel\; !g(x).\,\overline{h}\langle x\rangle \;\parallel\; H^h\begin{bmatrix}\overline{h}\langle\{a=1;b=''\texttt{foo}''\}\rangle\\ !h(x).\,\overline{s}\langle\texttt{print}(x.\,c)\rangle\\ !s(x).\,\mathbf{0}\end{bmatrix} \rhd$$

$$G_g\,[!\overline{g}\langle\{a=1;b=''\texttt{foo}''\}\rangle] \;\parallel\; !g(x).\,\overline{h}\langle x\rangle \;\parallel\; H^h\begin{bmatrix}\overline{s}\langle\texttt{print}(\{a=1;b=''\texttt{foo}''\}.\,c)\rangle\\ !h(x).\,\overline{s}\langle\texttt{print}(x.\,c)\rangle\\ !s(x).\,\mathbf{0}\end{bmatrix}$$

Figure 2.4: Computation ending in an error

T:Var
$$\Gamma \vdash x : \Gamma(x)$$

T:Message
$$\frac{\forall 0 < i \leq n,\ \Gamma \vdash M_i : \tau_i}{\Gamma \vdash \{a_1 = M_1; \ldots; a_n = M_n\} : \{a_1 : \mathrm{Pre}(\tau_1); \ldots; a_n : \mathrm{Pre}(\tau_n); \mathrm{Abs}\}}$$

T:App
$$\frac{\Gamma \vdash M_1 : \tau \to \tau' \qquad \Gamma \vdash M_2 : \tau}{\Gamma \vdash (M_1\ M_2) : \tau'}$$

T:Inst
$$\frac{\Gamma \vdash M : \forall\overline{\kappa}.\,\tau \qquad \mathsf{dom}(\Sigma) = \overline{\kappa}}{\Gamma \vdash M : \Sigma(\tau)}$$

Figure 2.5: Typing rules for messages

type system nor its properties, further details may be found in Bidinger et al. (2005a); Lienhardt (2009).

Message Contents The typing of message contents, more precisely of record manipulation, is based on row polymorphism. A record type has the shape $\{a_1 : \mathrm{Pre}(\tau_1); \ldots, a_n : \mathrm{Pre}(\tau_n); \mathrm{Abs}\}$ meaning the fields a_1 to a_n are present and have types τ_1 to τ_n, respectively. The final Abs indicates no other field is present. For instance, the message $\{a = 1; c = \texttt{"foo"}\}$ has type $\{a : \mathrm{Pre}(\mathrm{int}); c : \mathrm{Pre}(\mathrm{string}); \mathrm{Abs}\}$: chunks named a and c are present and respectively contain an integer and a string.

The syntax of types for message contents is as follows.

σ	$::=$	$\forall\overline{\kappa}.\,\tau$							Type Schema
τ	$::=$	$\tau \to \tau'$	$\mid$	ψ	$\mid$	α	$\mid$ int $\mid$ string		Value Type
ψ	$::=$	$a : \mathrm{Pre}(\tau); \psi$	$\mid$	Abs	$\mid$	ρ			Row Type

Type variables, written κ, include value type variables α and row type variables ρ.

$$\text{T:Channel}\quad \frac{\Gamma \vdash M : S(s)}{\Gamma \vdash \overline{s}\langle M\rangle : S} \qquad \text{T:Receiver}\quad \frac{e \in dc(S) \qquad \forall 1 \leq i \leq n,\ (\Gamma \uplus \{x : S(e)\} \vdash B_i : S)}{\Gamma \vdash e(x).\,(B_1 \parallel \cdots \parallel B_n) : S}$$

$$\text{T:Parallel}\quad \frac{\Gamma \vdash D_1 : S \qquad \Gamma \vdash D_2 : S}{\Gamma \vdash D_1 \parallel D_2 : S} \qquad \text{T:Zero}\quad \Gamma \vdash \mathbf{0} : S \qquad \text{T:Bang}\quad \frac{\Gamma \vdash B : S}{\Gamma \vdash !B : S}$$

$$\text{T:Box}\quad \frac{\Gamma \vdash D : S \qquad \forall e \in I \cup O, S(e) = S'(e)}{\Gamma \vdash b^I_O\,[D]\ : S'}$$

Figure 2.6: Typing rules for processes

Free type variables of a type $\mathsf{fv}(\forall\overline{\kappa}.\,\tau)$ are the type variables of τ which do not occur in $\overline{\kappa}$. We extend this notion to typing environments Γ. In record types, we assume that each field name occurs at most once[4]. We also assume that the order of fields is not important, i.e., we work up to reordering of fields.

Typing rules for message contents, including the application of constants, may be found in Figure 2.5. In rule T:Inst, Σ is a substitution mapping the type variables $\overline{\kappa}$ to types. We assume that this substitution is well-sorted, in the sense that row type variables ρ are substituted by rows types ψ and value type variables α by value types τ.

Constants We assume the typing of constants is present in the initial typing environment, written Γ in the typing rules of Figure 2.5. We now give the types of a few constants, to illustrate the use of row polymorphism. As these constants are functions, they will all have a functional type. Note that the type of these constants is chosen to be compatible with their reductions, i.e., such that subject reduction holds.

The type of the "access a" constant $.\,a$ is as follows.

$$\forall\alpha,\rho.\,\{a : \mathrm{Pre}(\alpha), \rho\} \to \alpha$$

This type may be read as "given an arbitrary value type α and arbitrary row type ρ, the constant $.\,a$ maps a record with a field a present and of type α (the rest of the record type being arbitrary and captured by ρ) to a value of type α."

Similarly, the type of the "delete a" constant $\backslash a$ is as follows.

$$\forall\alpha,\rho.\,\{a : \mathrm{Pre}(\alpha), \rho\} \to \{\rho\}$$

In this case, field a still has to be present, but it's the rest of the record ρ that is returned.

Finally, adding a value of type α to a field a requires that the field be absent before.

$$\forall\alpha,\rho.\,\alpha \times \{a : \mathrm{Abs}, \rho\} \to \{a : \mathrm{Pre}(\alpha), \rho\}$$

Upon typing the application of a constant to a record, one needs to instantiate the type variables α and ρ so that the types match the record's type.

[4]One typically uses a kind system to enforce this, annotating row types with the set of names that cannot occur in the row. We prefer to use this less formal approach for readability.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash .\, a : \forall \alpha, \rho.\, \{a : \mathrm{Pre}(\alpha); \rho\} \to \alpha}\,\text{T:Var}}{\Gamma \vdash .\, a : \{a : \mathrm{Pre}(\alpha); \rho\} \to \alpha}\,\text{T:Inst} \qquad \dfrac{}{\Gamma \vdash x : \{a : \mathrm{Pre}(\alpha); \rho\}}\,\text{T:Var}}{\Gamma \vdash x.\, a : \alpha}\,\text{T:App}}{\Gamma \vdash \overline{s}\langle x.\, a\rangle : S}\,\text{T:Channel}}{\emptyset \vdash e(x).\, \overline{s}\langle x.\, a\rangle : S}\,\text{T:Receive}
$$

Figure 2.7: Typing a Behavior

Behaviors, Messages, and Components Typing of these constructions is done globally. It is inspired in part by process types in the the $\lambda\pi_v$-calculus (Yoshida and Hennessy, 2002) and sequence types for the π-calculus (Maffeis, 2005). More precisely, we check that the use of messages and behaviors is consistent with a given global type S mapping channel names to the types of the values they carry. We write $dc(S)$ for the domain of this mapping, and $S(e)$ for the type associated with channel e.

In the case of messages (T:Channel), the message contents are checked to correspond to the channel's expected arguments. The typing of behaviors (T:Receiver) is slightly more complex: the continuation of a reception is checked in an extended environment where the reception variable has the type corresponding to the channel. Finally, the typing of components (T:Box) is straightforward. The global types are requested to coincide on the channels that are exported or imported, whereas there is no constraint on the other channels.

Note that our types do not check whether a channel is actually present. A channel may be required either because it is imported or exported by a component, or because a message is sent on it. Adding such verification could be done by adapting our previous work on the dynamic Join calculus (Schmitt, 2002).

Discussion

This first type system is very limited: as only constants are allowed to have polymorphic types, this implies that a given channel name may only carry one type of message, whose structure is fixed. This limitation is overcome in the extension of this calculus described in Section 2.4.

Nevertheless, one may type the behavior $e(x).\, \overline{s}\langle x.\, a\rangle$ with the process type $S = e(\{a : \mathrm{Pre}(\alpha); \rho\}), s(\alpha)$ where α and ρ are unspecified, as depicted in Figure 2.7 (in this figure we have $\Gamma = x : \{a : \mathrm{Pre}(\alpha); \rho\}$, and we assume a global environment containing typing definitions for constants). This process type says that s may carry values of some type α, and that e carries record with a field a present with the same type α.

This type system has a second limitation due to the absence of *routing*, which will be the main motivation for the extension of Section 2.4. This limitation is twofold. At the level of the calculus, if several flows of messages end up on the same channel, then there is no way to separate these flows. As a consequence one cannot share components between flows of messages. Moreover, even if such a primitive was added, the type systems require that the types of the messages be identical, which is too strong a restriction.

2.4 Structural Routing

We now enrich our calculus and our type system to allow some modularity in the use of components. More precisely, we allow flows of messages with different types to be joined, the messages to be manipulated, then the flows to be separated again.

$$A_o \left[\begin{array}{rcl} \texttt{Gen1}_{g_1}\,[!\overline{g_1}\langle\{a=1;b=1\}\rangle] & \parallel & !g_1(x).\,\overline{m_1}\langle x\rangle \\ \texttt{Gen2}_{g_2}\,[!\overline{g_2}\langle\{a=1;c=\texttt{"foo"}\}\rangle] & \parallel & !g_2(x).\,\overline{m_2}\langle x\rangle \\ \texttt{M}_{m_o}^{m_1 m_2}\,[!m_1(x).\,\overline{m_o}\langle x\rangle \parallel !m_2(x).\,\overline{m_o}\langle x\rangle] & \parallel & !m_o(x).\,\overline{t_i}\langle x\rangle \\ \texttt{TCPIP}_{t_o}^{t_i}\left[!t_i(x).\,\overline{t_o}\langle\left\{\begin{array}{c} ip=\texttt{192.168.1.42;} \\ val=x \end{array}\right\}\rangle\right] & \parallel & !t_o(x).\,\overline{o}\langle x\rangle \end{array} \right]$$

Figure 2.8: Site A

The changes to the calculus are fairly small: we simply add a new *routing* primitive that is able to separate messages according to their contents. The changes to the type system, however, are much more significant.

Calculus

As hinted above, we extend the syntax with a special feature to directly encode a router-like behavior: $\mathrm{IfPre}(a, M, s_1, s_2)$. This construct tests if a field named 'a' is present in the message 'M'. If it is, the message is sent on the 's_1' output channel, otherwise it is sent on 's_2.

Abstract syntax Most of the syntax of our calculus remains as before. We only add one behavior corresponding to the routing operator.

$$B \;::=\; \ldots \quad | \quad \mathrm{IfPre}(a, M, s_1, s_2)$$

Examples A *multiplexer* can be defined as

$$\texttt{Mult} \triangleq b_s^{e_1 e_2}\,[!e_1(x).\,\overline{s}\langle x\rangle \parallel !e_2(x).\,\overline{s}\langle x\rangle]$$

The two receivers in `Mult`, listening on e_1 and e_2, send messages they receive on output s, thus *multiplexing* them on one output channel.

A *router* on name a can be defined as

$$\texttt{Router} = b_{s_1 s_2}^{e}\,[!e(x).\,\mathrm{IfPre}(a, x, s_1, s_2)]$$

It has three ports: one input (e) and two outputs (s_1 and s_2). The routing behavior in this case is simply implemented using the `IfPre` operator. If the message received on the input contains a field labeled a, then it is sent on s_1, otherwise, it is sent on s_2.

Figure 2.8 presents an encoding of a refinement of the assemblage for `site A` of Figure 2.1, where we added a component to create a TCP/IP packet. Generators send messages on their output channel, 'g_1' for `Gen1` and 'g_2' for `Gen2`. `Gen1` sends messages with two chunk named 'a' and 'b', while `Gen2` sends messages having 'a' and 'c' chunks, and whose 'c' chunk carries a string. Bindings are encoded as in Section 2.3, using a simple forwarding behavior. For instance, $!g_1(x).\,\overline{m_1}\langle x\rangle$ connects the output channel of `Gen1` to the first input channel of the multiplexer.[5] Finally, the `TCPIP` component sends messages containing two chunks: 'ip' contains an IP address, and 'val' contains the message to transmit to site B. The output channel of this component is then forwarded on channel 'o', which is the output channel of site A.

Figure 2.9 presents an encoding of `site B`. As in Figure 2.1, this component must be read from bottom to top: the input channel of site B is 'i', which is forwarded to the

[5] A less verbose encoding would directly use the destination name directly, but it may not be known when implementing components independently.

$$B^i \left[\begin{array}{c} Handler1^{h_1} \left[\begin{array}{c} !h_1(x).\,\overline{s}\langle \mathtt{print}(x.\,c)\rangle \\ !s(x).\,\mathbf{0} \end{array} \right] \;\|\; \mathtt{Handler2}^{h_2}\,[!h_2(x).\,\mathbf{0}] \\ \mathtt{R}^r_{s_1 s_2}\,[!r(x).\,\mathrm{IfPre}(a, x.\,val, s_1, s_2)] \;\|\; !s_1(x).\,\overline{h_1}\langle x\rangle \;\|\; !s_2(x).\,\overline{h_2}\langle x\rangle \\ !i(x).\,\overline{r}\langle x\rangle \end{array} \right]$$

Figure 2.9: Site B

input of the router component '`R`'. The router extracts the '*val*' chunk from its input messages, and send the result depending on the presence of the 'a' chunk on the output channel 's_1' or 's_2'. Channel 's_1' is bound to the input interface of `Handler1`, which prints the contents of the 'c' chunk of the message. Channel s_2 si bound to the input interface of `Handler2`, which throws the message away.

Operational Semantics

The operational semantics needs to be extended in two ways: first by giving rules to the routing operator, then by taking it into accounts for evaluation contexts. As concerns the reduction, we add the following two rules.

$$\textsc{ifPre}\quad \frac{M = \{a = M_1; a_2 = M_2; \ldots; a_n = M_n\}}{\mathrm{IfPre}(a, M, s_1, s_2) \;\rhd\; \overline{s_1}\langle M\rangle}$$

$$\textsc{ifAbs}\quad \frac{M = \{a_1 = M_1; \ldots; a_n = M_n\} \qquad \forall 0 < i \leq n,\, a_i \neq a}{\mathrm{IfPre}(a, M, s_1, s_2) \;\rhd\; \overline{s_2}\langle M\rangle}$$

We extend the evaluation contexts as follows.

$$\mathbb{E}_D \;::=\; \ldots \;\mid\; \mathrm{IfPre}(a, \mathbb{E}_M, s_1, s_2)$$

Message Errors

We extend as well the notion of a message error, by requiring that the routing operator be defined only on records. We thus now have the following formal definition.

Definition 2.4.1. *A program D has a* Message Error *iff either:*

- *There exist $\mathbb{E}_D$, $\mathbb{E}_M$, v, and v' such that $D \equiv \mathbb{E}_D[\mathbb{E}_M[(v\ v')]]$ where $\mathtt{eval}(v, v')$ is undefined.*
- *There exist $\mathbb{E}_D$, v, a, s_1, and s_2 such that $D \equiv \mathbb{E}_D[\mathit{IfPre}(a, v, s_1, s_2)]$ and v is not a record.*

Example Site B of Figure 2.9 alone has no error, but the assemblage of site A and site B will: the message $\{a = 1; b = 1\}$ will be routed to `Handler1`, which cannot access to the undefined 'c' chunk.

Type system

We now turn to the modification required for our type system to detect these message errors. It is based on the same principles as in Section 2.3, namely row polymorphism for record operations and global process types. We extend this system with two specific

$$\text{T:Channel}\quad \frac{\Gamma \vdash M : \tau \qquad \tau \in S(s)}{\Gamma \vdash \overline{s}\langle M\rangle : S} \qquad \text{T:IfPre1}\quad \frac{\Gamma \vdash M : \tau \qquad \tau = \{a : \mathrm{Pre}(\dots); \dots\} \qquad \tau \in S(s_1)}{\Gamma \vdash \mathrm{IfPre}(a, M, s_1, s_2) : S}$$

$$\text{T:IfPre2}\quad \frac{\Gamma \vdash M : \tau \qquad \tau = \{a : \mathrm{Abs}; \dots\} \qquad \tau \in S(s_2)}{\Gamma \vdash \mathrm{IfPre}(a, M, s_1, s_2) : S}$$

$$\text{T:Receiver}\quad \frac{e \in dc(S) \qquad \forall \tau \in S(e), \forall 1 \leq i \leq n, (\Gamma \uplus \{x : \tau\} \vdash B_i : S)}{\Gamma \vdash e(x).\,(B_1 \parallel \cdots \parallel B_n) : S}$$

$$\text{T:Parallel}\quad \frac{\Gamma \vdash D_1 : S \qquad \Gamma \vdash D_2 : S}{\Gamma \vdash D_1 \parallel D_2 : S} \qquad \text{T:Zero}\quad \Gamma \vdash \mathbf{0} : S \qquad \text{T:Bang}\quad \frac{\Gamma \vdash B : S}{\Gamma \vdash !B : S}$$

$$\text{T:Box}\quad \frac{\Gamma \vdash D : S \qquad \forall e \in I \cup O, S(e) = S'(e)}{\Gamma \vdash b^I_O\,[D] : S'}$$

Figure 2.10: Typing rules for processes

typing rules inspired by *intensional type analysis* (Crary et al., 1998) to handle the routing procedure.

The syntax of types is identical to the previous section, with one exception. Instead of mapping channel names to message types, our process types S now map channels to *finite sets* of message types. For instance, the process type $s : (\tau_1) \cup s : (\tau_2)$ specifies that channel s may carry messages of types τ_1 and τ_2. We write $S(e)$ the set of types associated with channel e, and $dc(S)$ the set of all channels e such that $S(e) \neq \emptyset$.

The typing rules for the contents of messages of Figure 2.5 are unchanged. The main changes to the typing rules of processes, depicted in Figure 2.10, are the addition of two rules for the routing operator (T:IfPre1 and T:IfPre2), and the modifications of the T:Channel and T:receiver rules. As regards the first modification, the T:Channel rules now checks that the type carried belongs to the set of types for the channel. Conversely, the T:Receiver rule now must check the typing of the body of the receiver for each type that may be received.

Typing examples. Our multiplexer `Mult` can admit several types, depending on the messages it has in input, as in: $e_1 : (\tau) \cup e_2 : (\tau_1) \cup e_2 : (\tau_2) \cup s : (\tau) \cup s : (\tau_1) \cup s : (\tau_2)$. To check it, one uses the rules T:Parallel and T:Receiver on the processes $e_1(x).\,\overline{s}\langle x\rangle$ with x typed τ, and $e_2(x).\,\overline{s}\langle x\rangle$ with x first typed τ_1 and then typed τ_2. Note that these union types allow us to recover a form of (ad-hoc) polymorphism: channels are no longer limited to a single type.

Router. It may seem that the typing rules are too restrictive to capture the expressiveness of the 'IfPre' construct. However, consider what happens when typing a router program such as $e(x).\,\mathrm{IfPre}(a, x, s_1, s_2)$. Because of rule T:Receiver, we have to consider all the message types mapped on channel e. Combining the rules T:IfPre1, T:IfPre2 and T:Receiver, we can thus recover expected types for the routing process. For instance, one can verify that the following process type validates the router

$$
\begin{aligned}
& g_1 : (\{a : \text{Pre(int)}; b : \text{Pre(int)}; \text{Abs}\}) \\
& \cup\, g_2 : (\{a : \text{Pre(int)}; c : \text{Pre(string)}; \text{Abs}\}) \\
& \cup\, m_1 : (\{a : \text{Pre(int)}; b : \text{Pre(int)}; \text{Abs}\}) \\
& \cup\, m_2 : (\{a : \text{Pre(int)}; c : \text{Pre(string)}; \text{Abs}\}) \\
& \cup\, m_o : (\{a : \text{Pre(int)}; b : \text{Pre(int)}; \text{Abs}\}) \\
& \cup\, m_o : (\{a : \text{Pre(int)}; c : \text{Pre(string)}; \text{Abs}\}) \\
& \cup\, t_i : (\{a : \text{Pre(int)}; b : \text{Pre(int)}; \text{Abs}\}) \\
& \cup\, t_i : (\{a : \text{Pre(int)}; c : \text{Pre(string)}; \text{Abs}\}) \\
& \cup\, t_o : (\{ip : \text{Pre(IP)}; val : \text{Pre}(\{a : \text{Pre(int)}; b : \text{Pre(int)}; \text{Abs}\}); \text{Abs}\}) \\
& \cup\, t_o : (\{ip : \text{Pre(IP)}; val : \text{Pre}(\{a : \text{Pre(int)}; c : \text{Pre(string)}; \text{Abs}\}); \text{Abs}\}) \\
& \cup\, o : (\{ip : \text{Pre(IP)}; val : \text{Pre}(\{a : \text{Pre(int)}; b : \text{Pre(int)}; \text{Abs}\}); \text{Abs}\}) \\
& \cup\, o : (\{ip : \text{Pre(IP)}; val : \text{Pre}(\{a : \text{Pre(int)}; c : \text{Pre(string)}; \text{Abs}\}); \text{Abs}\})
\end{aligned}
$$

Figure 2.11: Type of site A

program above, using the rules T:IfPre1, T:IfPre2 and T:Receiver.

$$
\begin{aligned}
S \triangleq\ & e : (\{a : \text{Pre(int)}; \text{Abs}\}) \\
& \cup\, e : (\{b : \text{Pre(int)}; \text{Abs}\}) \cup e : (\{\text{Abs}\}) \\
& \cup\, s_1 : (\{a : \text{Pre(int)}; \text{Abs}\}) \\
& \cup\, s_2 : (\{b : \text{Pre(int)}; \text{Abs}\}) \cup s_2 : (\{\text{Abs}\})
\end{aligned}
$$

Simple assemblage. As previously stated, site A and site B from Figure 2.1 are typable, and indeed, we can find a type for each of these components, as presented in Figures 2.11 and 2.12. The T:Box rules allows us to hide the type of some channels, but we do not leverage this possibility and give a (quite large) global type. Consider the type of site A (Figure 2.11). The 1st line states that `Gen1` generates messages having a 'a' and a 'b' chunk, each containing an integer. The 3rd line states that these messages are transmitted to the input channels of the multiplexer, which will send them on its output interface (line 5). The 7th and 8th lines state that the messages are transmitted to the input channel of the `TCP/IP` component. The output of the `TCP/IP` component is then described line 9 and 10, and transmitted to the output of site A (line 11 and 12).

The type for site B can be read likewise, from its input channel to the handler components. Some types are left unspecified (as type variables); they may be instantiated with any type. More interestingly, types may also be duplicated, replacing type variables by fresh ones. This way, the type of a simple wire $e : (\alpha) \cup s : (\alpha)$ can be duplicated into $e : (\alpha) \cup s : (\alpha) \cup e : (\beta) \cup s : (\beta)$, then each α and β can have an independent instance. This mechanism, called "generalized substitutions" is formally described in Lienhardt (2009), where it is shown that if an assemblage has a type, then it also has types resulting from this duplication.

Nevertheless, even using duplication, site B cannot be given a type compatible with site A. Indeed, site B routes all messages with the a chunk through channel s_1, which expects messages with both the a and c chunks, the latter containing a string. There is no provision for messages containing a chunk named a yet no chunk named c. Because of this, connecting site A and site B would result in an ill-typed assemblage, and is thus forbidden by our type system.

Type System Properties

This type system is sound with respect to message errors and obeys the classic correction and subject reduction theorems, which are detailed in Lienhardt (2009).

This system still has some limitations, however, in the sense that it does not have principal types. Consider for instance the assemblage depicted in Figure 2.13. In this

$i : (\{ip : \text{Pre(IP)}; val : \text{Pre}(\{a : \text{Pre}(\eta_1); c : \text{Pre(string)}; \rho\}); \rho'\})$
$\cup i : (\{ip : \text{Pre(IP)}; val : \text{Pre}(\{a : \text{Abs}; \rho\}); \rho'\})$
$\cup r : (\{ip : \text{Pre(IP)}; val : \text{Pre}(\{a : \text{Pre}(\eta_1); c : \text{Pre(string)}; \rho\}); \rho'\})$
$\cup r : (\{ip : \text{Pre(IP)}; val : \text{Pre}(\{a : \text{Abs}; \rho\}); \rho'\})$
$\cup s_1 : (\{a : \text{Pre}(\eta_1); c : \text{Pre(string)}; \rho\})$
$\cup s_2 : (\{a : \text{Abs}; \rho\})$
$\cup h_1 : (\{a : \text{Pre}(\eta_1); c : \text{Pre(string)}; \rho\})$
$\cup s : (\{\text{Abs}\})$
$\cup h_2 : (\{a : \text{Abs}; \rho\})$

Figure 2.12: Type of site B

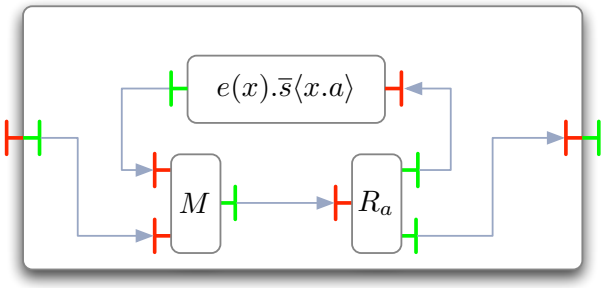


Figure 2.13: Routing with a Loop

figure, router R_a output messages with field a on its top interface and other messages on its bottom interface. Thus messages remain in the loop as long as they have a field named a, and the top component accesses the contents of the field. For instance, calling e and s the input and output of the component, if one sends $\{a = \{\}\}$ on channel i, the component returns $\{\}$ on channel s; if we send $\{a = \{a = \{b = 2\}\}; c = 3\}$, the returned message is $\{b = 2\}$. We can thus give this component the types $\emptyset$, $i : (\{a : \text{Pre}(\alpha); \rho\}) \cup s : (\alpha)$, $i : (\{a : \text{Pre}(\{a : \text{Pre}(\alpha); \rho_2\}); \rho_1\}) \cup s : (\alpha)$, ... We can even take the union of these types. However, the most general type would be an infinite union generating every chain of embedded a fields.

We have in fact shown, through an encoding of the Post Correspondance Problem (Post, 1946), that type inference is undecidable. We may go around this issue in two ways: either accept that the inference algorithm may not terminate (which would be the case with the example above), or have a semi-inference algorithm that asks the user for some input when dealing with loops. These approaches are detailed in Lienhardt (2009). Instead, we now turn to a different approach to routing that allows type inference, at the cost of some expressiveness.

2.5 Semantic Routing

The core idea behind semantic routing is the following one: marry multiplexers with routers, and route messages according to the interface from which they entered the multiplexer. This small paradigm shift has two main consequences. First, multiplexers are no longer simply two wires pointing to the same output interface, as they must somehow change the messages to indicate the interface they came from. Second, routers are now only routers: they cannot be used to inspect the contents of messages.

This small loss in expressiveness is compensated by the fact that type inference is now decidable. Moreover, we have found that this form of routing was sufficient for every practical example we have studied.

Calculus

Syntax The syntax of the calculus is given below. The first difference with the previous calculus is the replacement of explicit behaviors with a more generic notion of *primitive component*, written p below. Such primitive components include message manipulation components, as well as multiplexers and routers.

The second difference lies in the use of *routed values*, written v^δ, where δ records where the value comes from.

$$\begin{array}{rcllllll} D & ::= & p & \mid & b_O^I\,[D] & \mid & \overline{e}\langle v^\delta\rangle & \mid & D_1 \parallel D_2 & \mid & \mathbf{0} & \text{Components} \\ v & ::= & c & \mid & \{a_1 = v_1; \ldots; a_n = v_n\} & & & & & & & \text{Value} \\ \delta & ::= & \emptyset & \mid & \downarrow r; \delta & \mid & \uparrow r; \delta & & & & & \text{Tag list} \end{array}$$

An assemblage of components D is a parallel composition of components and messages. Components can be primitive or composite. A composite takes the form $b_O^I\,[D]$ as before. Messages take the form $\overline{e}\langle v^\delta\rangle$, where e is a channel name, and v^δ is a routed value. In the following we write M for a routed value when the actual value and tags do not matter, and J for a parallel composition of messages. A routed value is a record or a base value decorated with a list of routing tags. We always assume that each tag occur at most once in a list. Intuitively, a list of routing tags δ encodes a particular message flow in a component assemblage. Primitive components can act on these flows, as illustrated by the router and multiplexer primitive components described below. Although each tag is unique in a tag list, component assemblages can contain loops (e.g., through a combination of routers and multiplexers), and (as before) record fields can contain records. These two features allow the modeling of complex communication stacks, including ones featuring protocol tunneling, such as IP over IP.

The set of primitive components is a parameter of the calculus, and can be extended as required. It is assumed to contain at least the following: components `Add`, `Sub`, and `Select` provide classical basic operations on extensible records; components `Router` and `Mult` provide elementary routing and multiplexing capabilities; component `Conn` corresponds to a simple unidirectional connector.

Operational semantics The operational semantics of the calculus is defined modulo structural equivalence: the parallel operator is associative, commutative, and has $\mathbf{0}$ has neutral element, and the order of fields in a record does not matter. The reduction relation, written $D_1 \rhd D_2$, is defined as a binary relation on assemblages that satisfies the following rules.

R:Ctx
$$\frac{D \rhd D'}{\mathbb{E}[D] \rhd \mathbb{E}[D']}$$

R:In
$$\frac{e \in I}{\overline{e}\langle M\rangle \parallel c_O^I\,[D] \rhd c_O^I\,[\overline{e}\langle M\rangle \parallel D]}$$

R:Out
$$\frac{s \in O}{c_O^I\,[\overline{s}\langle M\rangle \parallel D] \rhd c_O^I\,[D] \parallel \overline{s}\langle M\rangle}$$

R:Prim
$$\frac{\texttt{eval}(p, J) = D}{J \parallel p \rhd p \parallel D}$$

As our syntax is much simpler, thanks to the use of primitive components, evaluation contexts are also simpler.

$$\mathbb{E} ::= [] \quad \mid \quad \mathbb{E} \parallel D \quad \mid \quad b_O^I\,[\mathbb{E}]$$

Most rules are similar to our previous system. Rule R:Prim is a generalization of the previous reduction for constant applications. The partial evaluation function `eval` now takes a primitive p and a parallel composition of messages J as arguments. This function is defined as follows for `Add`, `Sub`, `Select`, `Mult`, and `Router`. In the following, we assume that a is distinct from every a_i.

$$\texttt{eval}(\texttt{Add}[e_1\, e_2/s](a), \overline{e_1}\langle\{a_1 = v_1; \ldots; a_n = v_n\}^{\delta_1}\rangle \parallel \overline{e_2}\langle v^{\delta_2}\rangle) = \overline{s}\langle\{a = v; a_1 = v_1; \ldots; a_n = v_n\}^{\delta_1}\rangle$$

$$\texttt{eval}(\texttt{Sub}[e, s](a), \overline{e}\langle\{a = v; a_1 = v_1; \ldots; a_n = v_n\}^{\delta_1}\rangle) = \overline{s}\langle\{a_1 = v_1; \ldots; a_n = v_n\}^{\delta_1}\rangle$$

$$\texttt{eval}(\texttt{Select}[e/s](a), \overline{e}\langle\{a = v; a_1 = v_1; \ldots; a_n = v_n\}^{\delta_1}\rangle) = \overline{s}\langle v^{\delta_1}\rangle$$

$$\begin{aligned}
\texttt{eval}(\texttt{Mult}[e_1\, e_2/s](r), \overline{e_1}\langle v^{\delta}\rangle) &= \overline{s}\langle v^{\uparrow r;\delta}\rangle \quad \text{if } r \notin \delta \\
\texttt{eval}(\texttt{Mult}[e_1\, e_2/s](r), \overline{e_2}\langle v^{\delta}\rangle) &= \overline{s}\langle v^{\downarrow r;\delta}\rangle \quad \text{if } r \notin \delta \\
\texttt{eval}(\texttt{Router}[e/s_1\, s_2](r), v^{\delta_1;\uparrow r;\delta_2}) &= \overline{s_1}\langle v^{\delta_1;\delta_2}\rangle \\
\texttt{eval}(\texttt{Router}[e/s_1\, s_2](r), v^{\delta_1;\downarrow r;\delta_2}) &= \overline{s_2}\langle v^{\delta_1;\delta_2}\rangle
\end{aligned}$$

`Add`, `Sub`, and `Select` provide usual record manipulation. Note that `Add` uses a *join pattern* to capture two messages at the same time. `Mult` adds a tag to a routed value to signal the input channel on which it received it. `Router` checks the tags of the received routed values to send them on the appropriate channel.

Errors In our concurrent setting, where primitive components may consume several messages at the same time, the definition of error as a stuck evaluation is less immediate than before. We choose to say there is an error if a primitive component accepts messages on a given name e, yet a message is present on e that cannot be accepted by the component. Formally, a message $\overline{e}\langle M\rangle$ *cannot be processed* by a primitive component p if there are some N and J such that $\texttt{eval}(p, \overline{e}\langle N\rangle \parallel J)$ is defined, but for every J', $\texttt{eval}(p, \overline{e}\langle M\rangle \parallel J')$ is undefined. An assemblage D has an *error* if $D \equiv \mathbb{E}[\overline{e}\langle M\rangle \parallel p]$ and $\overline{e}\langle M\rangle$ cannot be processed by p.

Type System

Syntax Our type system is based on two main ideas: (i) the type of values exchanged on channels are *routed types*: rows (extensible record) or base types, decorated with routing information; (ii) the type of an assemblage is an *assemblage type*, presented as a function from its input channel types to its output channels types. The syntax of types is defined as follows.

τ	::=		Value type				
		α	variable	Ξ	::=		Routed type
	|	$\{W\}$	row			$\xi[\tau]$	variable value flow
	|	$\text{int}, \text{string}, \ldots$	base type		|	$r(\Xi_1, \Xi_2)$	tagged pair
ψ	::=		Row	Δ	::=		Interface type
		ρ	variable			$\emptyset$	empty declaration
	|	$a : \text{Pre}(\tau); \psi$	present field		|	$e : (\Xi)$	channel declaration
	|	$a : \text{Abs}; \psi$	absent field		|	$\Delta \cup \Delta$	interface union
	|	Abs	empty row				

The type of an assemblage, written σ in the following, takes the form of a type scheme $\forall \kappa_1 \ldots \kappa_n.\, \Delta_I \to \Delta_O$ where κ_i are (value, row, or flow) type variables, Δ_I collects the types of input channels in the assemblage, and Δ_O collects the types of output channels in the assemblage. We write $dc(\sigma)$ for the channel names that appear in σ. A channel type takes the form $e : (\Xi)$, where e is a channel name, and Ξ is a routed type. A routed type is either a value flow $\xi[\tau]$, where the value type τ is carried by the data flow variable ξ, or a tagged pair of the form $r(\Xi_1, \Xi_2)$, where r is a tag, and Ξ_1, Ξ_2 are routed types. Rows are defined as before.

Informally, a routed type is a binary tree where each leaf corresponds to a value type carried by a data flow, and the branch leading to it defines the routing annotation carried by the value (a given routing tag appears at most once on each branch). For instance, the type $r_1(\xi_1[\text{int}], r_2(\xi_2[\text{string}], \xi_3[\alpha]))$ consists of three branches corresponding to three different values. The second branch $r_1(_, r_2(\xi_2[\text{string}], _))$ corresponds to a flow accepting only strings tagged with at least the tags $\downarrow r_1$ and $\uparrow r_2$. This tree structure uses explicit references to data flows as they enable *type duplication*, which is a requirement to properly deal with routing and multiplexing. Type duplication allows two multiplexers in a row to type check correctly and is the main innovation of this type system (see the discussion below).

Typing Types for primitive components are given by a function Υ that maps primitive components to assemblage types. Just as the set of primitive components is a parameter of our calculus, function Υ is a parameter of our type system and needs to be defined for every primitive component to be typed. To ensure that these assemblage types correspond to the operational semantics of the primitive components, the function Υ must obey two constraints: (i) for each primitive component p, the input channel type of $\Upsilon(p)$ should only allow valid patterns; (ii) the output type of the parallel composition of a primitive component p with one of its valid input pattern J must contain the type of $\texttt{eval}(p, J)$. Formally, for every primitive component p and parallel composition of messages J such $\texttt{eval}(p, J)$ is defined, there exists an assemblage type $\Delta_1 \to \Delta_2$ such that $p \parallel J : \Delta_1 \to \Delta_2$ holds, and there exists Δ_2' with $\Delta_2' \subseteq \Delta_2$ such that $p \parallel \texttt{eval}(p, J) : \Delta_1 \to \Delta_2'$ holds. These constraints ensure that the type of a primitive component is consistent with its behavior (defined by `eval`). For instance, the types associated with the primitive components introduced before, and of a simple connector $\texttt{Conn}[e/s]$ (that forward any value received on its input channel e to its output channel s), can be defined as follows:

$$
\begin{aligned}
&\Upsilon(\texttt{Add}[e_1 e_2/s](a)) = \\
&\qquad \forall \alpha, \rho, \xi_1, \xi_2.\ e_1 : (\xi_1[\{a : \text{Abs}; \rho\}]) \cup e_2 : (\xi_2[\alpha]) \to s : (\xi_1[\{a : \text{Pre}(\alpha); \rho\}]) \\
&\Upsilon(\texttt{Select}[e/s](a)) = \forall \alpha, \rho, \xi.\ e : (\xi[\{a : \text{Pre}(\alpha); \rho\}]) \to s : (\xi[\alpha]) \\
&\Upsilon(\texttt{Router}[e/s_1\, s_2](r)) = \forall \alpha, \beta, \xi, \xi'.\ e : (r(\xi[\alpha], \xi'[\beta])) \to s_1 : (\xi[\alpha]) \cup s_2 : (\xi'[\beta]) \\
&\Upsilon(\texttt{Mult}[e_1\, e_2/s](r)) = \forall \alpha, \beta, \xi, \xi'.\ e_1 : (\xi[\alpha]) \cup e_2 : (\xi'[\beta]) \to s : (r(\xi[\alpha], \xi'[\beta])) \\
&\Upsilon(\texttt{Conn}[e/s]) = \forall \alpha, \xi.\ e : (\xi[\alpha]) \to s : (\xi[\alpha])
\end{aligned}
$$

The type system is equipped with a (classical) subtyping relation $\leq$, which we do not detail fully here. For instance, the subtyping rules for assemblage types T:Func and T:Gen, and tagged pairs T:TagPair, are given below.

$$
\text{T:Func}\ \frac{\Delta_1 \leq \Delta_1' \qquad \Delta_2 \leq \Delta_2'}{\Delta_1' \to \Delta_2 \leq \Delta_1 \to \Delta_2'}
\qquad
\text{T:Gen}\ \frac{\sigma \leq \sigma'}{\forall \kappa.\, \sigma \leq \forall \kappa.\, \sigma'}
\qquad
\text{T:TagPair}\ \frac{\Xi_1 \leq \Xi_1' \qquad \Xi_2 \leq \Xi_2'}{r(\Xi_1, \Xi_2) \leq r(\Xi_1', \Xi_2')}
$$

$$\text{T:Prim}\ \frac{\Upsilon(p) = \sigma}{p : \sigma} \qquad \text{T:Subst}\ \frac{D : \sigma}{D : \Sigma(\sigma)} \qquad \text{T:Inst}\ \frac{D : \forall\kappa.\,\sigma}{D : \sigma} \qquad \text{T:Gen}\ \frac{D : \sigma}{D : \forall\kappa.\,\sigma} \qquad \text{T:Channel}\ \frac{\emptyset \vdash M : \Xi}{\overline{e}\langle M\rangle : \emptyset \to e : (\Xi)}$$

$$\text{T:Sub}\ \frac{D : \sigma \qquad \sigma \leq \sigma'}{D : \sigma'} \qquad \text{T:Par}\ \frac{D : \Delta_1 \to \Delta_2 \qquad D' : \Delta'_1 \to \Delta'_2 \qquad \Delta_2 \precsim \Delta'_1 \qquad \Delta'_2 \precsim \Delta_1 \qquad dc(\Delta_1) \cap dc(\Delta'_1) = \emptyset}{D \parallel D' : (\Delta_1 \cup \Delta'_1) \to (\Delta_2 \cup \Delta'_2)}$$

$$\text{T:Box}\ \frac{D : \Delta_1 \to \Delta_2 \qquad \Delta'_1 \precsim \Delta_1 \qquad \Delta_2 \precsim \Delta'_2 \qquad \Delta'_2 \precsim \Delta'_1 \qquad dc(\Delta'_1) = I \wedge dc(\Delta'_2) = O}{c^I_O\,[D] : \Delta'_1 \to \Delta'_2}$$

Figure 2.14: Typing rules for assemblages

The typing rules in our type system comprise rules for assemblages and rules for routed values. Typing judgements take the form $D : \sigma$ for assemblages, $v : \tau$ for simple values, and $\mathcal{R} \vdash R : \Xi$ for routed values. The environment $\mathcal{R}$ is a set of routing tags. The typing rules make use of the $\precsim$ binary relation between channel types, which is defined as follows: given two channel types $\Delta \triangleq \bigcup_{i\in I} e_i : (\Xi_i)$ and $\Delta' \triangleq \bigcup_{j\in J} e'_j : (\Xi'_j)$, we note $\Delta \precsim \Delta'$ iff for all $i \in I, j \in J$, $e_i = e'_j$ implies $\Xi_i \leq \Xi'_j$.

Typing rules for assemblages are given below in figure 2.14. Rule T:Prim states that the type of a primitive component is given by function Υ. Rules T:Subst, T:Inst, and T:Gen are classical rules for substitution, instantiation, and generalization, respectively. Since type duplication is integrated into substitutions, because of the different forms of type variables and their associated constraints (e.g., unique occurrence of tags in routing annotations), our notion of substitution Σ in rule T:Subst is slightly more complex than usual. It mostly behaves as expected, replacing variables with terms (see the discussion below; formal details can be found in Lienhardt (2009)).

The parallel composition D_1 of two assemblages D and D' yields a function having the *capacity* of both assemblages, i.e., that accepts as input any message either D or D' accepts, and that can generate any message either D or D' can generate. Rule T:Par has three side conditions: the first two ($\Delta_2 \precsim \Delta'_1$ and $\Delta'_2 \precsim \Delta_1$) ensure that all values (Δ_2 and Δ'_2) sent on input channels for $D \parallel D'$ are indeed valid inputs for this program; the third one ($dc(\Delta_1) \cap dc(\Delta'_1) = \emptyset$) states that D and D' must have distinct input channels to avoid the possibility of *implicit routing*, i.e., of distinct components listening on the same channel, thus doing a routing operation without an explicit router to support it. Rule T:Box specifies the constraints that apply to obtain the type $\Delta'_1 \to \Delta'_2$ of a composite. The sets Δ'_1 and Δ'_2 must give a type to every channel mentioned in I and O. If a channel is mentioned in both, then the output type must be a subtype of the input type ($\Delta'_2 \precsim \Delta'_1$) as this corresponds to a loop. We also impose that the valid inputs of the component must be valid ones for the component's inner process (stated by the constraint $\Delta'_1 \precsim \Delta_1$), and that all outputs of this process must be valid output of the component (stated by the constraint $\Delta_2 \precsim \Delta'_2$).

Typing rules for routed values are given in Figure 2.15 (we have left out rules and conditions that apply to base values and base types). Rule T:Record is the standard typing rule for extensible record, using rows. The three typing rules T:Empty, T:Up, and T:Down, construct a routed type by induction on the cardinality of the routing annotation. Rule T:Empty is used when the routing annotation is empty: the routing type is in such case just a leaf representing the value's type. Rules T:Up and T:Down

$$\text{T:Record}\quad \frac{\forall 1 \leq i \leq n,\ v_i : \tau_i \qquad \forall 1 \leq i \neq j \leq n,\ a_i \neq a_j}{\{a_1 = v_1; \ldots; a_n = v_n\} : \{a_1 : \mathrm{Pre}(\tau_1); \ldots; a_n : \mathrm{Pre}(\tau_n); \mathrm{Abs}\}} \qquad \text{T:Empty}\quad \frac{v : \tau}{\mathcal{R} \vdash v^{\emptyset} : \xi[\tau]}$$

$$\text{T:Up}\quad \frac{\mathcal{R} \uplus \{r\} \vdash v^{\delta} : \Xi \qquad \mathcal{R} \uplus \{r\} \vdash \Xi_k}{\mathcal{R} \vdash v^{\uparrow r;\delta} : r(\Xi, \Xi_k)} \qquad \text{T:Down}\quad \frac{\mathcal{R} \uplus \{r\} \vdash v^{\delta} : \Xi \qquad \mathcal{R} \uplus \{r\} \vdash \Xi_k}{\mathcal{R} \vdash v^{\downarrow r;\delta} : r(\Xi_k, \Xi)}$$

Figure 2.15: Typing rules for routed values

define how we construct the routing type tree when one or more elements are present in the routing annotation. We write $\mathcal{R} \uplus \{r\}$ for the disjoint union of the two sets. The use of routing tags environments $\mathcal{R}$ in these three rules ensures the validity of the constructed routed type. Finally, kind judgements of the form $\mathcal{R} \vdash \Xi$ make sure that Ξ does not use tags from $\mathcal{R}$.

Example assemblage Assume that the generators, handlers, multiplexer, router, and conduit components in Figure 2.1 are primitive components, and their types are as given in the following table. We can type the assemblages `SiteA` and `SiteB` as indicated in the last two lines of the same table.

Component	Types
`Gen1`	$\forall \xi.\, \emptyset \to s_1 : (\xi[\tau_1])$
`Gen2`	$\forall \xi.\, \emptyset \to s_2 : (\xi[\tau_2])$
`Handler1`	$\forall \xi.\, e_1 : (\xi[\tau_3]) \to \emptyset$
`Handler2`	$\forall \xi.\, e_2 : (\xi[\tau_4]) \to \emptyset$
`M`	same type as $\texttt{Mult}[s_1\, s_2/t_A](r)$
`R`	same type as $\texttt{Router}[t_B/e_1\, e_2](r)$
`Conduit`	same type as $\texttt{Conn}[t_A/t_B]$
`SiteA`	$\forall \xi, \xi'.\, \emptyset \to t_A : (r(\xi[\tau_1], \xi'[\tau_2]))$
`SiteB`	$\forall \xi, \xi'.\, t_B : (r(\xi[\tau_3], \xi'[\tau_4])) \to \emptyset$

If we assume further that τ_3 can be transformed using subtyping and substitution into τ_1, and similarly for τ_4 into τ_2, then we can type the (closed) assemblage

$$c^{\emptyset}_{\emptyset}\,[\texttt{SiteA} \parallel \texttt{Conduit} \parallel \texttt{SiteB}]$$

with the type: $\emptyset \to \emptyset$.

Properties of the type system We show in Lienhardt (2009) the usual subject reduction and correction theorems: typing is preserved by reduction (up to subtyping and substitution), and that a typed assemblage has no error. More importantly, type inference is now decidable and has been implemented, as detailed below.

Discussion

Type duplication. In our presentation of the type system, we have glossed over several details. In particular, our notion of substitution is more complex than the usual one because of type duplication. Let us explain it by way of an example. One of the objectives of this type system is to allow flexible data flows in programs, using a routing

tree structure to type channels. Let us consider a program where a component `Rem` that remove a field a follows a multiplexer. The output type of the multiplexer is of the form $r(\xi_1[\alpha_1], \xi_2[\alpha_2])$, whereas the input type of `Rem` is of the form $\xi_3[\{a : \mathrm{Pre}(\alpha_3); \rho\}]$. The difficulty here is that we need to be able to unify these two types. With our definition of substitution, this unification is made in two steps. We first *duplicate* the type $\xi_3[\{a : \mathrm{Pre}(\alpha_3); \rho\}]$ into

$$\Xi \triangleq r(\xi_4[\{a : \mathrm{Pre}(\alpha_3); \rho\}], \xi_5[\{a : \mathrm{Pre}(\alpha_3); \rho\}])$$

One can remark that the two branches of the resulting routing tree have the same row and type variables. But because they are declared in different flows (ξ_4 and ξ_5), they can be instantiated with different terms. We then have two tree structures with the same form that we can simply unify into Ξ.

Duplication allows to instantiate a leaf in a routing tree into a whole sub-tree, while keeping the constraint of the leaf (here, the constraint being that the message must have the field 'a' defined) and allowing the variables on the fresh leaves to be instantiated independently. One can see duplication as a way to enable polymorphism without using type schemas.

Limitations. Our type system has a few limitations. We already pointed out that there can only be a single type for channel and a set of tags (union types are not supported). Also, since a routing type is a binary tree, one has to encode router and multiplexer types with more than two output or input channels by a combination of binary routers and multiplexers. Another consequence is the complexity of encoding routers that route on fields into our calculus, as is sometimes the case in the Dream framework. Typically, we encode the presence of a field a in a message with a pair of tags $\uparrow a$ (when the field a is present) and $\downarrow a$ (when a is absent from the message). This simple encoding is difficult to apply in complex assemblages involving loops with multiple routers and multiplexers. An encoding can be found in most cases, but can be tricky to define and manipulate. However, based on our experience with the Dream and Click frameworks (see below), these limitations are not show-stoppers, and we have not been hindered by them.

Type Inference and its Implementation

A key property of this type system (in contrast to the one of Section 2.4) is that type inference is decidable. We have devised and proved correct a constraint-based algorithm, along the lines of Palsberg et al. (1997) and Pottier (2003). The type inference algorithm and its proof can be found in Lienhardt (2009). The algorithm comprises a constraint generator that computes from a given program a set of constraints a type must satisfy to be valid for the input program, and a constraint solver that decides whether the generated constraint set has a solution (the program is typable) or not (the program is not typable). Technically, our type inference is based on the one defined in Pottier (2003), extended to deal with routing types, channel types, and type duplication.

We have implemented the type inference algorithm in OCaml, and used it to extend the assemblage tool chains used by the Dream and Click frameworks. In the case of Dream, we have extended the Fractal ADL toolchain described in Leclercq et al. (2007). Figure 2.16 provides an overview of this toolchain. It is organized as a component-based framework, that consists of a front-end, realized by the `Loader` component, and a back-end, with the `ASTProcessingOrganizer` and the `Scheduler` components. The back-end is responsible for the generation and execution of tasks such as code generation, code installation, code deployment, etc. The `Loader` component reads a set of input files and produces an Abstract Syntax Tree (AST). This tree provides a unified representation

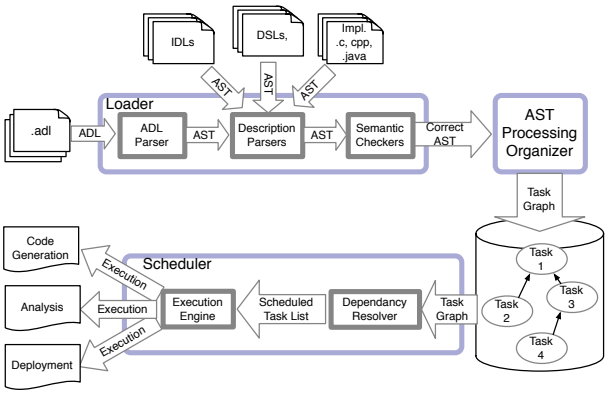


Figure 2.16: Fractal ADL toolchain

of the system architecture that can be described through a combination of description languages, such as ADL, IDL, or DSL. The `Loader` is organized essentially as a pipeline comprising parsers for the various possible input languages, and semantic analyzers. We have integrated our type analyzer as a specific semantic analyzer component in this pipeline. We have also devised an extension to the XML-based Fractal ADL to take into account our type annotations for primitive components, and added its associated parser component in the `Loader` pipeline.

In the case of Click, a C++ software framework dedicated to the component-based construction of configurable routers (Kohler et al., 2000), assemblages are specified by configuration files written in a simple scripting language (Kohler et al., 2002). We found it simpler to just document type annotations for Click in a separate, additional configuration file. This way, our type analyzer remains an entirely separate and external analysis tool for Click, and its use does not require any change to the Click toolset.

We have conducted several experiments to check the correctness of non-trivial assemblages built using both frameworks. They demonstrate that our approach is practical, requiring minimal extensions to existing assemblage toolsets, and that it can indeed be applied to different component-based frameworks, implemented in different programming languages. The following table provides an indication of the time taken to check (correct) Dream and Click assemblages. The Dream assemblage originates from the Cosmos project, which develops protocols for roaming mobile devices. The Click assemblages are examples taken from the Click website. The performance of our type analyzer appears quite reasonable, bearing in mind that the complexity of type inference in our system is non-polynomial.

Assemblage	Components	Primitive	Channels	Time (sec)
COSMOS (Dream)	439	340	662	180.428
dnsproxy (Click)	9	8	7	0.025
fromhost-tunnet (Click)	24	22	24	0.166
mazu-nat (Click)	60	56	54	4.489

2.6 Related Work

We have mentioned in the introduction previous work dealing with assemblage issues in component-based communication frameworks. Type systems checking architectural constraints or component assemblages have been the subject of various works in the past decade. For instance, the work done on the Wright language (Allen and Garlan, 1997) supports the verification of behavioral compatibility constraints in a software architec-

ture. Work on Plastik (Joolia et al., 2005) deals mostly with structural constraints, although in a dynamical setting. Work on ArchJava (Aldrich et al., 2002) uses ownership types to enforce communication integrity between components. Another work develops behavioral types for component assembly (Carrez et al., 2003), which is close to the notion of session types as developed in Yoshida and Vasconcelos (2007). None of these type systems allow to capture the errors due to incorrect message manipulation.

We know of no type system that is capable of dealing with our notion of message errors, with the complex data flows that are allowed in our calculus. Indeed, type systems such as Cardelli and Gordon (1999); Hindley (1997); Pierce and Turner (2000); Simonet and Pottier (2007) are too restrictive concerning data flow manipulation, and cannot adequately deal with *routers* and *multiplexers*. On the other hand, type systems which satisfactorily handle data flows by way of session types and process types (Maffeis, 2005; Yoshida and Hennessy, 2002; Yoshida and Vasconcelos, 2007) do not take in account structured mutable messages.

Type inference for distributed calculi has been studied for the Join-calculus (Conchon and Pottier, 2001), Mobile Ambients-like calculi (Makholm and Wells, 2005), Dπ (Lhoussaine, 2004), which have an inference algorithm, and Pict, which does not. While the reasons for type inference undecidability in Pict spawn from higher-order polymorphism and subtyping, we believe that our own undecidable inference is related with polymorphic recursion (Henglein, 1993). Indeed, undecidability in our case is caused by channels being mapped to a finite set whose cardinality is not constrained, thus allowing a form of polymorphic recursion in loops. The use of tags in semantic routing lets us avoid this problem. Finally, one can consider the *routing process* present in the calculus as a weak form of *type analysis* (Weirich, 2002) on rows.

2.7 Conclusion

As first examples of static analyses for the manipulation of unordered trees, we have presented type systems that guarantee the absence of record manipulation errors in component-based communication frameworks, such as Dream. These type systems are all based on the union of row polymorphism and process types, and differ in the way routing is handled. In the case of semantic routing, we have also presented an implementation of the analysis and its integration in the Fractal toolchain.

There are opportunities for future research in this line of work, in particular to take into account the dynamic evolution of component assemblages. This could be done in a setting where components are represented by localities in a higher-order calculus, similar to the one developed in Chapter 8

Chapter 3

Bidirectional Transformations On Trees

3.1 Introduction

While the data model we consider remains unordered trees, the setting of our second example is quite different: we leave the world of concurrent message-passing applications to move to (statically correct) bidirectional transformations.

Most of the time, we use programs in just one direction, from input to output. But sometimes, having computed an output, we need to be able to *update* this output and then "calculate backwards" to find a correspondingly updated input. The problem of writing such bidirectional transformations arises in a multitude of domains, including data converters and synchronizers, parsers and pretty printers, picklers and unpicklers, structured editors, constraint maintainers for user interfaces, and, of course, in databases, where it is known as the *view update* problem.

The naive way to write a bidirectional transformation is simply to write two separate functions in any language you like and check (by hand) that they fit together in some appropriate sense—e.g., that composing them yields the identity function. However, this approach is unsatisfying for all but the simplest examples. For one thing, verifying that the two functions fit together in this way requires intricate reasoning about their behaviors. Moreover, it creates a maintenance nightmare: both functions will embody the structure that the input and output schemas have in common, so changes to the schemas will require coordinated changes to both.

A better alternative is to design a notation in which both transformations can be described at the same time—i.e., a *bidirectional programming language*. In a bidirectional language, every expression, when read from left to right, denotes a function mapping inputs to outputs; when read from right to left, the same expression denotes a function mapping an updated output together with an original input to an appropriately updated version of the input. Not only does this eliminate code duplication; it also eliminates paper-and-pencil proofs that the two transformations fit together properly: we can design the language to guarantee it.

We address a specific instance of the view update problem that arises in a larger project called Harmony (Foster et al., 2006). Harmony is a generic framework for synchronizing tree-structured data—a tool for propagating updates between different copies of tree-shaped data structures, possibly stored in different formats. For example, Harmony can be used to synchronize the bookmark files of several different web browsers, allowing bookmarks and bookmark folders to be added, deleted, edited, and reorganized in any browser and propagated to the others. The ultimate aim of the project is to provide a platform on which a Harmony programmer can quickly assemble a high-quality synchronizer for a new type of tree-structured data stored in a standard low-level format such as XML.

Views play a key role in Harmony: to synchronize structures that may be stored in disparate concrete formats, we define a single common abstract format and a collection of *lenses* that transform each concrete format into this abstract one. For example, we can synchronize a Mozilla bookmark file with an Internet Explorer bookmark file by transforming each into an *abstract bookmark structure* and propagating changed information between these. Afterwards, we need to take the updated abstract structures and reflect the corresponding updates back into the original concrete structures. Thus, each lens must include not one but *two* functions—one for extracting an abstract view from a concrete one and another for putting an updated abstract view back into the original concrete view to yield an updated concrete view. We call these the *get* and *put* components, respectively. The intuition is that the mapping from concrete to abstract is commonly some sort of projection, so the *get* direction involves getting the abstract part out of a larger concrete structure, while the *put* direction amounts to putting a new abstract part into an old concrete structure. We show a concrete example of this process in Section 3.2.

The difficulty of the view update problem springs from a fundamental tension between *expressiveness* and *robustness*. The richer we make the set of possible transformations in the *get* direction, the more difficult it becomes to define corresponding functions in the *put* direction in such as way that each lens is both *well behaved*—its *get* and *put* behaviors fit together in a sensible way—and *total*—its *get* and *put* functions are defined on all the inputs to which they may be applied.

To reconcile this tension, a successful approach to the view update problem must be carefully designed with a particular application domain in mind. The approach described here is tuned to the kinds of projection-and-rearrangement transformations on trees and lists that we have found useful for implementing Harmony instances. It does not directly address some well-known difficulties with view update in the classical setting of relational databases—such as the difficulty of "inverting" queries involving joins. (We do hope that our work will suggest new attacks on these problems, however; a first step in this direction is described in Bohannon et al. (2006).)

A second difficulty concerns ease of use. In general, there are many ways to equip a given *get* function with a *put* function to form a well-behaved and total lens; we need some means of specifying which *put* is intended that is natural for the application domain and that does not involve onerous proof obligations or checking of side conditions. We adopt a linguistic approach to this issue, proposing a set of lens *combinators*—a small domain-specific language—in which every expression simultaneously specifies both a *get* function and the corresponding *put*. Moreover, each combinator is accompanied by a *type declaration*, designed so that the well-behavedness and (for non-recursive lenses) totality of composite lens expressions can be verified by straightforward, compositional checks. Proving totality of recursive lenses, like ordinary recursive programs, requires global reasoning that goes beyond types.

The first step in our formal development (Section 3.3) is identifying a natural mathematical space of well-behaved lenses over arbitrary data structures. There is a good deal of territory to be explored at this semantic level. First, we must phrase our basic definitions to allow the underlying functions in lenses to be partial, since there will in general be structures to which a given lens cannot sensibly be applied. The sets of structures to which we *do* intend to apply a given lens are specified by associating it with a type of the form $C \rightleftharpoons A$, where C is a set of concrete "source structures" and A is a set of abstract "target structures." Second, we define a notion of well-behavedness that captures our intuitions about how the *get* and *put* parts of a lens should behave in concert. For example, if we use the *get* part of a lens to extract an abstract view a from a concrete view c and then use the *put* part to push the very same a back into c, we should get c back. Third, we deploy standard tools from domain theory to define monotonicity and continuity for lens combinators parameterized on other lenses, estab-

lishing a foundation for defining lenses by recursion. (Recursion is needed because the trees that our lenses manipulate may in general have arbitrarily deep nested structure—e.g., when they represent directory hierarchies, bookmark folders, etc.) Finally, to allow lenses to be used to create new concrete structures rather than just updating existing ones (needed, for example, when new records are added to a database in the abstract view), we adjoin a special "missing" element to the structures manipulated by lenses and establish suitable conventions for how it is treated.

With these semantic foundations in hand, we proceed to syntax. In Section 3.4, we present a group of generic lens combinators (identities, composition, and constants), which can work with any kind of data. In Section 3.5, we focus attention on tree-structured data and present several more combinators that perform various manipulations on trees (hoisting, splitting, mapping, etc.); we also show how to assemble these primitives, along with the generic combinators from before, to yield some useful derived forms. Section 3.6 introduces another set of generic combinators implementing various sorts of bidirectional conditionals. Section 3.7 gives a more ambitious illustration of the expressiveness of these combinators by implementing a number of bidirectional list-processing transformations as derived forms, including lenses for projecting the head and tail of a list, mapping over a list, and—our most complex example—implementing a bidirectional filter lens whose *put* function performs a rather intricate "weaving" operation to recombine an updated abstract list with the concrete list elements that were filtered away by the *get*.

An extensive example derived from the Harmony bookmark synchronizer may be found in Foster et al. (2007b), along with the description of additional lenses for lists or relational data encoded as trees.

Section 3.8 surveys related work and we conclude in Section 3.9.

3.2 A Small Example

Suppose our concrete tree c is a simple address book:

$$c \quad = \quad \left\{\begin{array}{l} \texttt{Pat} \mapsto \left\{\begin{array}{l}\texttt{Phone} \mapsto \texttt{333-4444} \\ \texttt{URL} \mapsto \texttt{http://pat.com}\end{array}\right\} \\ \texttt{Chris} \mapsto \left\{\begin{array}{l}\texttt{Phone} \mapsto \texttt{888-9999} \\ \texttt{URL} \mapsto \texttt{http://chris.org}\end{array}\right\} \end{array}\right\}$$

We draw trees sideways to save space. Each set of hollow curly braces corresponds to a tree node, and each "$\texttt{X} \mapsto \ldots$" denotes a child labeled with the string $\texttt{X}$. The children of a node are unordered. To avoid clutter, when an edge leads to an empty tree, we usually omit the braces, the $\mapsto$ symbol, and the final childless node—e.g., "$\texttt{333-4444}$" above actually stands for "$\left\{\texttt{333-4444} \mapsto \{\}\right\}$." When trees are linearized in running text, we separate children with commas for easier reading.

Now, suppose that we want to edit the data from this concrete tree in a yet simpler format where each name is associated directly with a phone number.

$$a \quad = \quad \left\{\begin{array}{l}\texttt{Pat} \mapsto \texttt{333-4444} \\ \texttt{Chris} \mapsto \texttt{888-9999}\end{array}\right\}$$

Why would we want this? Perhaps because the edits are going to be generated by synchronizing this abstract tree with another replica of the same address book in which no URL information is recorded. Or perhaps there is no synchronizer involved and the edits are going to be performed by a human who is only interested in phone information and doesn't want to see URLs. Whatever the reason, we are going to make our changes to the abstract tree a, yielding a new abstract tree a' of the same form but with modified

content.[1] For example, let us change Pat's phone number, drop Chris, and add a new friend, Jo.

$$a' \quad = \quad \left\{\!\!\left\{\begin{array}{l}\texttt{Pat} \mapsto \texttt{333-4321}\\ \texttt{Jo} \mapsto \texttt{555-6666}\end{array}\right\}\!\!\right\}$$

Lastly, we want to compute a new concrete tree c' reflecting the new abstract tree a'. That is, we want the parts of c' that were kept when calculating a (e.g., Pat's phone number) to be overwritten with the corresponding information from a', while the parts of c that were filtered out (e.g., Pat's URL) have their values carried over from c.

$$c' \quad = \quad \left\{\!\!\left\{\begin{array}{l}\texttt{Pat} \mapsto \left\{\!\!\left\{\begin{array}{l}\texttt{Phone} \mapsto \texttt{333-4321}\\ \texttt{URL} \mapsto \texttt{http://pat.com}\end{array}\right\}\!\!\right\}\\ \texttt{Jo} \mapsto \left\{\!\!\left\{\begin{array}{l}\texttt{Phone} \mapsto \texttt{555-6666}\\ \texttt{URL} \mapsto \texttt{http://google.com}\end{array}\right\}\!\!\right\}\end{array}\right\}\!\!\right\}$$

We also need to "fill in" appropriate values for the parts of c' (in particular, Jo's URL) that were created in a' and for which c therefore contains no information. Here, we simply set the URL to a constant default, though in general we might want to compute it from other information.

Together, the transformations from c to a and from a' plus c to c' form a lens. Our goal is to find a set of combinators that can be assembled to describe a wide variety of lenses in a concise, natural, and mathematically coherent manner. To whet the reader's appetite, the lens expression that implements the transformations above is `map (focus Phone {|URL ↦ http://google.com|})`.

3.3 Semantic Foundations

Although many of our combinators work on trees, their semantic underpinnings can be presented in an abstract setting parameterized by the data structures (which we call "views") manipulated by lenses.[2] In this section—and in Section 3.4, where we discuss generic combinators—we simply assume some fixed set $\mathcal{V}$ of views; from Section 3.5 on, we will choose $\mathcal{V}$ to be the set of trees.

Basic Structures

When f is a partial function, we write $f(a)\downarrow$ if f is defined on argument a and $f(a) = \bot$ otherwise. We write $f(a) \sqsubseteq b$ for $f(a) = \bot \vee f(a) = b$. We write $\mathsf{dom}(f)$ for $\{s \mid f(s)\downarrow\}$, the set of arguments on which f is defined. When $S \subseteq \mathcal{V}$, we write $f(S)$ for $\{r \mid s \in S \wedge f(s)\downarrow \wedge f(s) = r\}$ and $\mathsf{ran}(f)$ for $f(\mathcal{V})$. We take function application to be strict: $f(g(x))\downarrow$ implies $g(x)\downarrow$.

Definition 3.3.1 (Lenses)**.** *A* lens *l comprises a partial function $l\nearrow$ from $\mathcal{V}$ to $\mathcal{V}$, called the get* function *of l, and a partial function $l\searrow$ from $\mathcal{V}\times\mathcal{V}$ to $\mathcal{V}$, called the put* function.

[1] Note that we are interested here in the final tree a', not the particular sequence of edit operations that was used to transform a into a'. This is important in the context of Harmony, which is designed to support synchronization of off-the-shelf applications, where in general we only have access to the current states of the replicas, rather than a trace of modifications; the tradeoffs between state-based and trace-based synchronizers are discussed in detail elsewhere (Pierce and Vouillon, 2001; Foster et al., 2007a).

[2] We use the word "view" here in a slightly different sense than some of the database papers that we cite, where a view is a *query* that maps concrete to abstract states—i.e., it is a function that, for each concrete database state, picks out a view in our sense. Also, note that we use "view" to refer uniformly to both concrete and abstract structures—when we come to programming with lenses, the distinction will be merely a matter of perspective anyway, since the output of one lens is often the input to another.

The intuition behind the notations $l\nearrow$ and $l\searrow$ is that the *get* part of a lens "lifts" an abstract view out of a concrete one, while the *put* part "pushes down" a new abstract view into an existing concrete view. We often say "put a into c (using l)" instead of "apply the *put* function (of l) to (a, c)."

Definition 3.3.2 (Well-behaved lenses)**.** *Let l be a lens and let C and A be subsets of $\mathcal{V}$. We say that l is a* well behaved *lens from C to A, written $l \in C \rightleftharpoons A$, if it maps arguments in C to results in A and vice versa*

$$\begin{array}{ll} l\nearrow(C) \subseteq A & (\textsc{Get}) \\ l\searrow(A \times C) \subseteq C & (\textsc{Put}) \end{array}$$

and its get and put functions obey the following laws:

$$\begin{array}{lll} l\searrow(l\nearrow c,\, c) \sqsubseteq c & \textit{for all } c \in C & (\textsc{GetPut}) \\ l\nearrow(l\searrow(a,\, c)) \sqsubseteq a & \textit{for all } (a, c) \in A \times C & (\textsc{PutGet}) \end{array}$$

We call C the source *and A the* target *in $C \rightleftharpoons A$. Note that a given l may be a well-behaved lens from C to A for many different Cs and As; in particular, every l is trivially a well-behaved lens from $\emptyset$ to $\emptyset$, while the everywhere-undefined lens belongs to $C \rightleftharpoons A$ for every C and A.*

Intuitively, the GetPut law states that, if we *get* some abstract view a from a concrete view c and immediately *put* a (with no modifications) into c, we must get back exactly c if both operations are defined. PutGet, on the other hand, demands that the *put* function must capture all of the information contained in the abstract view: if putting a view a into a concrete view c yields a view c', then the abstract view obtained from c' is exactly a.

An example of a lens satisfying PutGet but not GetPut is the following. Suppose $C = \texttt{string} \times \texttt{int}$ and $A = \texttt{string}$, and define l by:

$$l\nearrow(s, n) = s \qquad\qquad l\searrow(s',\, (s, n)) = (s', 0)$$

Then $l\searrow(l\nearrow(s, 1),\, (s, 1)) = (s, 0) \not\sqsubseteq (s, 1)$. Intuitively, the law fails because the *put* function has "side effects": it modifies information in the concrete view that is not reflected in the abstract view.

An example of a lens satisfying GetPut but not PutGet is the following. Let $C = \texttt{string}$ and $A = \texttt{string} \times \texttt{int}$, and define l by

$$l\nearrow s = (s, 0) \qquad\qquad l\searrow((s', n),\, s) = s'$$

PutGet fails here because some information contained in the abstract view does not get propagated to the new concrete view. For example, $l\nearrow(l\searrow((s', 1),\, s)) = l\nearrow s' = (s', 0) \not\sqsubseteq (s', 1)$.

The GetPut and PutGet laws reflect fundamental expectations about the behavior of lenses; removing either law significantly weakens the semantic foundation. We may also consider an optional third law, called PutPut:

$$l\searrow(a',\, l\searrow(a,\, c)) \sqsubseteq l\searrow(a',\, c) \quad \text{for all } a, a' \in A \text{ and } c \in C.$$

This law states that the effect of a sequence of two *put*s is (modulo definedness) just the effect of the second: the first gets completely overwritten. Alternatively, a series of changes to an abstract view may be applied either incrementally or all at once, resulting in the same final concrete view. We say that a well-behaved lens that also satisfies PutPut is *very well behaved*. Both well-behaved and very well behaved lenses correspond to familiar classes of "update translators" from the classical database literature;

see Section 3.8. The foundational development in this section is valid for both well-behaved and very well behaved lenses. However, when we come to defining our lens combinators for tree transformations, we will not require PUTPUT because some of our lens combinators—in particular, `map`, `flatten`, `merge`, and conditionals—fail to satisfy it for reasons that seem pragmatically unavoidable (see Section 3.5).

For now, a simple example of a lens that is well behaved but not very well behaved is as follows. Consider the following lens, where $C = \texttt{string} \times \texttt{int}$ and $A = \texttt{string}$. The second component of each concrete view intuitively represents a version number.

$$l\nearrow(s,n) = s \qquad\qquad l\searrow(s,\,(s',n)) = \begin{cases}(s,n) & \text{if } s = s'\\ (s,n{+}1) & \text{if } s \neq s'\end{cases}$$

The *get* function of l projects away the version number and yields just the "data part." The *put* function overwrites the data part, checks whether the new data part is the same as the old one, and, if not, increments the version number. This lens satisfies both GETPUT and PUTGET but not PUTPUT, as we have $l\searrow(s,\,l\searrow(s',\,(c,n))) = (s,n+2) \not\sqsubseteq (s,n+1) = l\searrow(s,\,(c,n))$.

Another critical property of lenses is *totality* with respect to a given source and target.

Definition 3.3.3 (Totality)**.** *A lens $l \in C \rightleftharpoons A$ is said to be* total*, written $l \in C \Longleftrightarrow A$, if $C \subseteq \mathsf{dom}(l\nearrow)$ and $A \times C \subseteq \mathsf{dom}(l\searrow)$.*

The reasons for considering both partial and total lenses instead of building totality into the definition of well-behavedness are much the same as the reasons for considering partial functions in conventional functional languages. In practice, we want lenses to be total:[3] to guarantee that Harmony synchronizers will work predictably, lenses must be defined on the whole of the domains where they are used; the *get* direction should be defined for any structure in the concrete set, and the *put* direction should be capable of putting back any possible updated version from the abstract set.[4] All of our primitive lenses are designed to be total, and all of our lens combinators map total lenses to total lenses—with the sole, but important, exception of lenses defined by recursion; as usual, recursive lenses must be constructed in the semantics as limits of chains of increasingly defined partial lenses. The soundness of the type annotations we give for our syntactic lens combinators guarantees that *every* well-typed lens expression is well-behaved, but only recursion-free expressions can be shown total by completely compositional reasoning with types; for recursive lenses, more global arguments are required.

Recursion

Since we will be interested in lenses over trees, and since trees in many application domains may have unbounded depth (e.g., a bookmark can be either a link or a folder containing a list of bookmarks), we will often want to define lenses by recursion. Our next task is to set up the necessary structure for interpreting such definitions.

[3]Indeed, well-behavedness is rather trivial in the absence of totality: for *any* function $l\nearrow$ from C to A, we can obtain a well-behaved lens by taking $l\searrow$ to be undefined on all inputs—or, slightly less trivially, to be defined only on inputs of the form $(l\nearrow c, c)$.

[4]Since we intend to use lenses to build synchronizers, the updated structures here will be results of synchronization. A fundamental property of the core synchronization algorithm in Harmony is that, if all of the updates between synchronizations occur in just one of the replicas, then the effect of synchronization will be to propagate all these changes to the other replica. This implies that the *put* function in the lens associated with the other replica must be prepared to accept any value from the abstract domain. In other settings, different notions of totality may be appropriate. For example, Hu, Mu, and Takeichi (2004) have argued that, in the context of interactive editors, a reasonable definition of totality is that $l\searrow(a,\,c)$ should be defined whenever a differs by at most one edit operation from $l\nearrow c$.

The development follows familiar lines. We introduce an information ordering on lenses and show that the set of lenses equipped with this ordering is a complete partial order (CPO). We then apply standard tools from domain theory to interpret a variety of common syntactic forms from programming languages—in particular, functional abstraction and application ("higher-order lenses") and lenses defined by single or mutual recursion.

We say that a lens l' is *more informative* than a lens l, written $l \prec l'$, if both the *get* and *put* functions of l' have domains that are at least as large as those of l and their results agree on their common domains.

A *cpo* is a partially ordered set in which every increasing chain of elements has a least upper bound in the set. If $l_0 \prec l_1 \prec \ldots \prec l_n \prec \ldots$ is an increasing chain, we write $\bigsqcup_{n\in\omega} l_n$ (often shortened to $\bigsqcup_n l_n$) for its least upper bound. A *cpo with bottom* is a cpo with an element $\perp$ that is smaller than every other element. In our setting, the bottom element $\perp_l$ is the lens whose *get* and *put* functions are everywhere undefined. It is obviously the smallest lens according to $\prec$ and is well-behaved at any lens type (it trivially satisfies all equations).

Lemma 3.3.4. *Let $l_0 \prec l_1 \prec \ldots \prec l_n \prec \ldots$ be an increasing chain of lenses. The lens l defined by*

$$\begin{array}{rl} l \searrow (a,\, c) = l_i \searrow (a,\, c) & \textit{if } l_i \searrow (a,\, c) \downarrow \textit{ for some } i \\ l \nearrow c = l_i \nearrow c & \textit{if } l_i \nearrow c \downarrow \textit{ for some } i \end{array}$$

and undefined elsewhere is a least upper bound for the chain.

Lemma 3.3.5. *Let $l_0 \prec l_1 \prec \ldots \prec l_n \prec \ldots$ be an increasing chain of lenses, and let $C_0 \subseteq C_1 \subseteq \ldots$ and $A_0 \subseteq A_1 \subseteq \ldots$ be increasing chains of subsets of $\mathcal{V}$. Then:*

1. *Well-behavedness commutes with limits:*
   $(\forall i \in \omega.\ \ l_i \in C_i \rightleftharpoons A_i)$ *implies* $\bigsqcup_n l_n \in (\bigcup_i C_i) \rightleftharpoons (\bigcup_i A_i)$.
2. *Totality commutes with limits:*
   $(\forall i \in \omega.\ \ l_i \in C_i \Longleftrightarrow A_i)$ *implies* $\bigsqcup_n l_n \in (\bigcup_i C_i) \Longleftrightarrow (\bigcup_i A_i)$.

Theorem 3.3.6. *Let $\mathcal{L}$ be the set of well-behaved lenses from C to A. Then $(\mathcal{L},\, \prec)$ is a cpo with bottom.*

We can now apply standard domain theory to interpret a variety of constructs for defining continuous lens combinators. We say that an expression e is continuous in the variable x if the function $\lambda x.\, e$ is continuous. An expression is said to be continuous in its variables, or simply continuous, if it is continuous in every variable separately. Examples of continuous expressions are variables, constants, tuples (of continuous expressions), projections (from continuous expressions), applications of continuous functions to continuous arguments, lambda abstractions (whose bodies are continuous), let bindings (of continuous expressions in continuous bodies), case constructions (of continuous expressions), and the fixed point operator itself. Tupling and projection let us define mutually recursive functions: if we want to define f as $F(f, g)$ and g as $G(f, g)$, where both F and G are continuous, we define $(f, g) = \mathit{fix}(\lambda(x, y).\, (F(x, y), G(x, y)))$.

Dealing with Creation

In practice, there will be cases where we need to apply a *put* function, but where no old concrete view is available, as we saw with `Jo`'s `URL` in Section 3.2. We deal with these cases by enriching the universe $\mathcal{V}$ of views with a special placeholder Ω, pronounced "missing," which we assume is not already in $\mathcal{V}$. (There are other, formally equivalent,

ways of handling missing concrete views. The advantages of this one are discussed in Section 3.5.) When $S \subseteq \mathcal{V}$, we write S_Ω for $S \cup \{\Omega\}$.

Intuitively, $l \searrow (a, \Omega)$ means "create a *new* concrete view from the information in the abstract view a." By convention, Ω is only used in an interesting way when it is the second argument to the *put* function: in all of the lenses defined below, we maintain the invariants that (1) $l \nearrow \Omega = \Omega$, (2) $l \searrow (\Omega, c) = \Omega$ for any c, (3) $l \nearrow c \neq \Omega$ for any $c \neq \Omega$, and (4) $l \searrow (a, c) \neq \Omega$ for any $a \neq \Omega$ and any c (including Ω). We write $C \overset{\Omega}{\rightleftharpoons} A$ for the set of well-behaved lenses from C_Ω to A_Ω obeying these conventions and $C \overset{\Omega}{\Longleftrightarrow} A$ for the set of total lenses obeying these conventions. For brevity in the lens definitions below, we always assume that $c \neq \Omega$ when defining $l \nearrow c$ and that $a \neq \Omega$ when defining $l \searrow (a, c)$, since the results in these cases are uniquely determined by these conventions. A useful consequence of these conventions is that a lens $l \in C \overset{\Omega}{\rightleftharpoons} A$ also has type $C \rightleftharpoons A$.

3.4 Generic Lenses

With these semantic foundations in hand, we are ready to move on to syntax. We begin in this section with several *generic* lens combinators (we will usually say just *lenses* from now on), whose definitions are independent of the particular choice of universe $\mathcal{V}$. Each definition is accompanied by a type declaration asserting its well-behavedness under certain conditions—e.g., "the identity lens belongs to $C \overset{\Omega}{\rightleftharpoons} C$ for any C".

Many of the lens definitions are parameterized on one or more arguments. These may be of various types: views (e.g., `const`), other lenses (e.g., composition), predicates on views (e.g., the conditional lenses in Section 3.6), or—in some of the lenses for trees in Section 3.5—edge labels, predicates on labels, etc.

One may find in Foster et al. (2007b) the proofs that the lenses we define are well behaved (i.e., that the type declaration accompanying its definition is a theorem) and total, and that lenses that take other lenses as parameters are continuous in these parameters and map total lenses to total lenses.

Identity

The simplest lens is the identity. It copies the concrete view in the *get* direction and the abstract view in the *put* direction.

$$\begin{array}{rcl} \texttt{id} \nearrow c & = & c \\ \texttt{id} \searrow (a, c) & = & a \\ \hline \multicolumn{3}{l}{\forall C \subseteq \mathcal{V}. \quad \texttt{id} \in C \overset{\Omega}{\rightleftharpoons} C} \end{array}$$

Composition

The lens composition combinator $l; k$ places l and k in sequence.

$$\begin{array}{rcl} l; k \nearrow c & = & k \nearrow (l \nearrow c) \\ l; k \searrow (a, c) & = & l \searrow (k \searrow (a, l \nearrow c), c) \\ \hline \multicolumn{3}{l}{\forall A, B, C \subseteq \mathcal{V}.\ \forall l \in C \overset{\Omega}{\rightleftharpoons} B.\ \forall k \in B \overset{\Omega}{\rightleftharpoons} A. \qquad l; k \in C \overset{\Omega}{\rightleftharpoons} A} \end{array}$$

The *get* direction applies the *get* function of l to yield a first abstract view, on which the *get* function of k is applied. In the other direction, the two *put* functions are applied in turn: first, the *put* function of k is used to put a into the concrete view that the *get* of k was applied to, i.e., $l \nearrow c$; the result is then put into c using the *put* function of l. (If the concrete view c is Ω, then, $l \nearrow c$ will also be Ω by our conventions on the treatment

of Ω, so the effect of $l; k \searrow (a, \Omega)$ is to use k to put a into Ω and then l to put the result into Ω.)

Constant

Another simple combinator is `const` v d, which transforms any view into the constant view v in the *get* direction. In the *put* direction, `const` simply restores the old concrete view if one is available; if the concrete view is Ω, it returns a default view d.

$$\begin{array}{rcll} \texttt{const}\ v\ d \nearrow c & = & v & \\ \texttt{const}\ v\ d \searrow (a, c) & = & c & \text{if } c \neq \Omega \\ & & d & \text{if } c = \Omega \\ \hline \multicolumn{4}{c}{\forall C \subseteq \mathcal{V}.\ \forall v \in \mathcal{V}.\ \forall d \in C. \quad \texttt{const}\ v\ d \in C \overset{\Omega}{\rightleftharpoons} \{v\}} \end{array}$$

Note that the type declaration demands that the *put* direction only be applied to the abstract argument v.

We will define a few more generic lenses in Section 3.6; for now, though, let us turn to some lens combinators that work on tree-structured data, so that we can ground our definitions in specific examples.

3.5 Lenses for Trees

To keep the definitions of our lens primitives as straightforward as possible, we work with an extremely simple form of trees: unordered, edge-labeled trees with no repeated labels among the children of a given node. This model is a natural fit for applications where the data is unordered, such as the keyed address books described in Section 3.2. Unfortunately, unordered trees do not have all the structure we need for other applications; in particular, we need to deal with ordered data such as lists and XML documents via an encoding (see Foster et al. (2007b)). A more direct treatment of ordered structures will be addressed in Chapter 4, but, in the context of the Harmony system, where we are interested in both ordered and unordered data, the choice of a simpler foundation seems to have been a good one: the increase in complexity of lens *programs* that must manipulate ordered data in encoded form is more than made up by the reduction in the complexity of the definitions of lens *primitives* due to the simpler data model.

Notation

From this point on, we choose the universe $\mathcal{V}$ to be the set $\mathcal{T}$ of finite, unordered, edge-labeled trees with labels drawn from some infinite set $\mathcal{N}$ of *names*—e.g., character strings—and with the children of a given node all labeled with distinct names. Trees of this form (often extended with labels on internal nodes as well as on children) are sometimes called *deterministic* trees or *feature trees* (e.g., Niehren and Podelski (1993)). The variables a, c, d, and t range over $\mathcal{T}$; by convention, we use a for trees that are thought of as abstract and c or d for concrete trees.

A tree is essentially a finite partial function from names to trees. It is more convenient, though, to adopt a slightly different perspective: we consider a tree $t \in \mathcal{T}$ to be a *total* function from $\mathcal{N}$ to $\mathcal{T}_\Omega$ that yields Ω on all but a finite number of names. We write $\mathsf{dom}(t)$ for the domain of t—i.e., the set of the names for which it returns something other than Ω—and $t(n)$ for the subtree associated to name n in t, or Ω if $n \notin \mathsf{dom}(t)$.

Tree values are written using hollow curly braces. The empty tree is written $\{\!\}$. (Note that $\{\!\}$, a node with no children, is different from Ω.) We often describe trees by comprehension, writing $\{\!\!\{ n \mapsto F(n) \mid n \in N \}\!\!\}$, where F is some function from $\mathcal{N}$ to $\mathcal{T}_\Omega$ and $N \subseteq \mathcal{N}$ is some set of names. When t and t' have disjoint domains, we write $t \cdot t'$

or $\{\!|t\ t'|\!\}$ (the latter especially in multi-line displays) for the tree mapping n to $t(n)$ for $n \in \mathsf{dom}(t)$, to $t'(n)$ for $n \in \mathsf{dom}(t')$, and to Ω otherwise.

When $p \subseteq \mathcal{N}$ is a set of names, we write $\overline{p}$ for $\mathcal{N}\backslash p$, the complement of p. We write $t|_p$ for the restriction of t to children with names taken from p—i.e., the tree $\{\!|n \mapsto t(n) \mid n \in p \cap \mathsf{dom}(t)|\!\}$—and $t\backslash_p$ for $\{\!|n \mapsto t(n) \mid n \in \mathsf{dom}(t)\backslash p|\!\}$. When p is just a singleton set $\{n\}$, we drop the set braces and write just $t|_n$ and $t\backslash_n$ instead of $t|_{\{n\}}$ and $t\backslash_{\{n\}}$. To shorten some of the lens definitions, we adopt the conventions that $\mathsf{dom}(\Omega) = \emptyset$ and that $\Omega|_p = \Omega\backslash_p = \Omega$ for any p.

For writing down types,[5] we extend these tree notations to sets of trees. If $T \subseteq \mathcal{T}$ and $n \in \mathcal{N}$, then $\{\!|n \mapsto T|\!\}$ denotes the set of singleton trees $\{\{\!|n \mapsto t|\!\} \mid t \in T\}$. If $T \subseteq \mathcal{T}$ and $N \subseteq \mathcal{N}$, then $\{\!|N \mapsto T|\!\}$ denotes the set of trees $\{t \mid \mathsf{dom}(t) = N \text{ and } \forall n \in N.\ t(n) \in T\}$ and $\left\{\!\!\left|N \overset{?}{\mapsto} T\right|\!\!\right\}$ denotes the set of trees $\{t \mid \mathsf{dom}(t) \subseteq N \text{ and } \forall n \in N.\ t(n) \in T_\Omega\}$. We write $T_1 \cdot T_2$ for $\{t_1 \cdot t_2 \mid t_1 \in T_1, t_2 \in T_2\}$ and $T(n)$ for $\{t(n) \mid t \in T\} \setminus \{\Omega\}$. If $T \subseteq \mathcal{T}$, then $\mathsf{doms}(T) = \{\mathsf{dom}(t) \mid t \in T\}$. Note that $\mathsf{doms}(T)$ is a set of sets of names, while $\mathsf{dom}(t)$ is a set of names.

A *value* is a tree of the special form $\{\!|k \mapsto \{\!|\}|\!\}$, often written just k. For instance, the phone number $\{\!|\texttt{333-4444} \mapsto \{\!|\}|\!\}$ in the example of Section 3.2 is a value. We write *Val* for the type whose denotation is the set of all values.

Hoisting and Plunging

Let's warm up with some combinators that perform simple structural transformations on trees. The lens `hoist` n is used to shorten a tree by removing an edge at the top. In the *get* direction, it expects a tree that has exactly one child, named n. It returns this child, removing the edge n. In the *put* direction, the value of the old concrete tree is ignored and a new one is created, with a single edge n pointing to the given abstract tree. (Later we will meet a derived form, `hoist_nonunique`, that works on bushier trees.)

$$\begin{array}{rcl} \texttt{hoist}\ n \nearrow c & = & c(n) \\ \texttt{hoist}\ n \searrow (a,\, c) & = & \{\!|n \mapsto a|\!\} \\ \hline \multicolumn{3}{l}{\forall C \subseteq \mathcal{T}.\ \forall n \in \mathcal{N}.\quad \texttt{hoist}\ n \in \{\!|n \mapsto C|\!\} \overset{\Omega}{\rightleftharpoons} C} \end{array}$$

Conversely, the `plunge` lens is used to deepen a tree by adding an edge at the top. In the *get* direction, a new tree is created, with a single edge n pointing to the given concrete tree. In the *put* direction, the value of the old concrete tree is ignored and the abstract tree is required to have exactly one subtree, labeled n, which becomes the result of the `plunge`.

$$\begin{array}{rcl} \texttt{plunge}\ n \nearrow c & = & \{\!|n \mapsto c|\!\} \\ \texttt{plunge}\ n \searrow (a,\, c) & = & a(n) \\ \hline \multicolumn{3}{l}{\forall C \subseteq \mathcal{T}.\ \forall n \in \mathcal{N}.\quad \texttt{plunge}\ n \in C \overset{\Omega}{\rightleftharpoons} \{\!|n \mapsto C|\!\}} \end{array}$$

Forking

The lens combinator `xfork` applies different lenses to different parts of a tree. More precisely, it splits the tree into two parts according to the names of its immediate

[5] Note that, although we are defining a syntax for lens expressions, the types used to classify these expressions are semantic—they are just sets of lenses or views. We are not proposing an algebra of types or an algorithm for mechanically checking membership of lens expressions in type expressions.

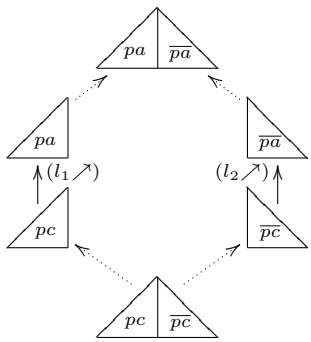


Figure 3.1: The *get* direction of `xfork`

children, applies a different lens to each, and concatenates the results. Formally, `xfork` takes as arguments two sets of names and two lenses. The *get* direction of `xfork` pc pa l_1 l_2 can be visualized as in Figure 3.1 (the concrete tree is at the bottom). The triangles labeled pc denote trees whose immediate children have labels in pc; dotted arrows represent splitting or concatenating trees. The result of applying $l_1 \nearrow$ to $c|_{pc}$ (the tree formed by dropping the immediate children of c whose names are not in pc) must be a tree whose top-level labels are in the set pa; similarly, the result of applying $l_2 \nearrow$ to $c\backslash_{pc}$ must be in $\overline{pa}$. That is, the lens l_1 may change the names of immediate children of the tree it is given, but it must map the part of the tree with immediate children belonging to pc to a tree with children belonging to pa. Likewise, l_2 must map the part of the tree with immediate children belonging to $\overline{pc}$ to a tree with children in $\overline{pa}$. Conversely, in the *put* direction, l_1 must map from pa to pc and l_2 from $\overline{pa}$ to $\overline{pc}$. Here is the full definition:

$$\begin{array}{rcl} \texttt{xfork}\ pc\ pa\ l_1\ l_2 \nearrow c & = & (l_1 \nearrow c|_{pc}) \cdot (l_2 \nearrow c\backslash_{pc}) \\ \texttt{xfork}\ pc\ pa\ l_1\ l_2 \searrow (a,\ c) & = & (l_1 \searrow (a|_{pa},\ c|_{pc})) \cdot (l_2 \searrow (a\backslash_{pa},\ c\backslash_{pc})) \end{array}$$

$$\begin{array}{l} \forall pc, pa \subseteq \mathcal{N}.\ \forall C_1 \subseteq \mathcal{T}|_{pc}.\ \forall A_1 \subseteq \mathcal{T}|_{pa}.\ \forall C_2 \subseteq \mathcal{T}\backslash_{pc}.\ \forall A_2 \subseteq \mathcal{T}\backslash_{pa}. \\ \forall l_1 \in C_1 \overset{\Omega}{\rightleftharpoons} A_1.\ \forall l_2 \in C_2 \overset{\Omega}{\rightleftharpoons} A_2. \\ \quad \texttt{xfork}\ pc\ pa\ l_1\ l_2 \in (C_1 \cdot C_2) \overset{\Omega}{\rightleftharpoons} (A_1 \cdot A_2) \end{array}$$

We rely here on our convention that $\Omega|_p = \Omega\backslash_p = \Omega$ to avoid explicitly splitting out the Ω case in the *put* direction.

We have now defined enough basic lenses to implement several useful derived forms for manipulating trees.

In many uses of `xfork`, the sets of names specifying where to split the concrete tree and where to split the abstract tree are identical. We can define a simpler `fork` as:

$$\texttt{fork}\ p\ l_1\ l_2 \quad = \quad \texttt{xfork}\ p\ p\ l_1\ l_2$$

$$\begin{array}{l} \forall p \subseteq \mathcal{N}.\ \forall C_1, A_1 \subseteq \mathcal{T}|_p.\ \forall C_2, A_2 \subseteq \mathcal{T}\backslash_p.\ \forall l_1 \in C_1 \overset{\Omega}{\rightleftharpoons} A_1.\ \forall l_2 \in C_2 \overset{\Omega}{\rightleftharpoons} A_2. \\ \quad \texttt{fork}\ p\ l_1\ l_2 \in (C_1 \cdot C_2) \overset{\Omega}{\rightleftharpoons} (A_1 \cdot A_2) \end{array}$$

We can use `fork` to define a lens that discards all of the children of a tree whose names do not belong to some set p:

$$\texttt{filter}\ p\ d \quad = \quad \texttt{fork}\ p\ \texttt{id}\ (\texttt{const}\ \{\}\ d)$$

$$\begin{array}{l} \forall C \subseteq \mathcal{T}.\ \forall p \subseteq \mathcal{N}.\ \forall d \in C\backslash_p. \\ \quad \texttt{filter}\ p\ d \in (C|_p \cdot C\backslash_p) \overset{\Omega}{\rightleftharpoons} C|_p \end{array}$$

In the *get* direction, this lens takes a concrete tree, keeps the children with names in p (using `id`), and throws away the rest (using `const` $\{\!\}$ d). The tree d is used when putting an abstract tree back into a missing concrete tree, providing a default for information that does not appear in the abstract tree but is required in the concrete tree. The type of `filter` follows directly from the types of the three primitive lenses used to define it: `const` $\{\!\}$ d, with type $C\backslash_p \overset{\Omega}{=\!=} \{\{\!\}\}$, the lens `id`, with type $C|_p \overset{\Omega}{=\!=} C|_p$, and `fork` (with the observation that $C|_p = C|_p \cdot \{\!\}$).

Let us see how `filter` behaves in an example. Let c be the following concrete tree $\{\!\texttt{name} \mapsto \texttt{Pat},\ \texttt{phone} \mapsto \texttt{333-4444}\!\}$, and lens $l = \texttt{filter}\ \{\texttt{name}\}\ \{\!\}$. We calculate $l \nearrow c$, underlining the next term to be simplifed at each step.

$$
\begin{aligned}
l \nearrow c &= \underline{(\texttt{fork}\ \{\texttt{name}\}\ \texttt{id}\ (\texttt{const}\{\!\}\ d)) \nearrow \{\!\texttt{name} \mapsto \texttt{Pat},\ \texttt{phone} \mapsto \texttt{333-444}\!\}} \\
&\qquad \text{by the definition of } l \\
&= \underline{\texttt{id} \nearrow \{\!\texttt{name} \mapsto \texttt{Pat}\!\}} \cdot \underline{(\texttt{const}\ \{\!\}\ d) \nearrow \{\!\texttt{phone} \mapsto \texttt{333-4444}\!\}} \\
&\qquad \text{by the definition of } \texttt{fork} \text{ and splitting } c \text{ using } \{\texttt{name}\} \\
&= \{\!\texttt{name} \mapsto \texttt{Pat}\!\} \cdot \{\!\} \quad = \quad \{\!\texttt{name} \mapsto \texttt{Pat}\!\} = a \\
&\qquad \text{by the definitions of } \texttt{id} \text{ and } \texttt{const}
\end{aligned}
$$

Now suppose that we update this tree, a, to $\{\!\texttt{name} \mapsto \texttt{Patty}\!\}$. Let us calculate the result of putting back a into c. To save space, we write k for $(\texttt{const}\ \{\!\}\ \{\!\})$.

$$
\begin{aligned}
& l \searrow (a,\, c) \\
= \ & \underline{(\texttt{fork}\ \{\texttt{name}\}\ \texttt{id}\ k) \searrow \big(\{\!\texttt{name} \mapsto \texttt{Pat}\!\},\ \{\!\texttt{name} \mapsto \texttt{Pat},\ \texttt{phone} \mapsto \texttt{333-4444}\!\}\big)} \\
& \qquad \text{by the definition of } l \\
= \ & \underline{\texttt{id} \searrow \big(\{\!\texttt{name} \mapsto \texttt{Patty}\!\},\ \{\!\texttt{name} \mapsto \texttt{Pat}\!\}\big)} \cdot \underline{k \searrow \big(\{\!\},\ \{\!\texttt{phone} \mapsto \texttt{333-4444}\!\}\big)} \\
& \qquad \text{by the definition of } \texttt{fork} \text{ and splitting } a \text{ and } c \text{ using } \{\texttt{name}\} \\
= \ & \{\!\texttt{name} \mapsto \texttt{Patty},\ \texttt{phone} \mapsto \texttt{333-4444}\!\} \\
& \qquad \text{by the definition of } \texttt{id} \text{ and } \texttt{const}
\end{aligned}
$$

Note that the *put* function restores the filtered part of the concrete tree and propagates the change made to the abstract tree. In the case of creation—i.e., if we put back an abstract tree using Ω— then the default argument to `const` is concatenated to the abstract tree to form the result, since there is no filtered part of the concrete tree to restore.

Another way to thin a tree is to explicitly specify a child that should be removed if it exists:

$$
\boxed{\begin{array}{c}
\texttt{prune}\ n\ d \quad = \quad \texttt{fork}\ \{n\}\ \big(\texttt{const}\ \{\!\}\ \{\!n \mapsto d\!\}\big)\ \texttt{id} \\
\hline
\forall C \subseteq \mathcal{T}.\ \forall n \in \mathcal{N}.\ \forall d \in C(n). \\
\texttt{prune}\ n\ d \in (C|_n \cdot C\backslash_n) \overset{\Omega}{=\!=} C\backslash_n
\end{array}}
$$

This lens is similar to `filter`, except that (1) the name given is the child to be removed rather than a set of children to keep, and (2) the default tree is the one to go under n if the concrete tree is Ω.

Conversely, we can grow a tree in the *get* direction by explicitly adding a child. The type annotation disallows changes in the newly added tree, so it can be dropped in the *put*.

$$
\boxed{\begin{array}{c}
\texttt{add}\ n\ t \quad = \quad \texttt{xfork}\ \{\}\ \{n\}\ (\texttt{const}\ t\ \{\!\};\ \texttt{plunge}\ n)\ \texttt{id} \\
\hline
\forall n \in \mathcal{N}.\ \forall C \subseteq \mathcal{T}\backslash_n.\ \forall t \in \mathcal{T}. \\
\texttt{add}\ n\ t \in C \overset{\Omega}{=\!=} \{\!n \mapsto \{t\}\!\} \cdot C
\end{array}}
$$

Let us explore the behavior of `add` through an example. Let $c = \{\!|\mathtt{a} \mapsto \{\!|\}\!|\}$ and $l = \mathtt{add\ b}\,\{\!|\mathtt{x} \mapsto \{\!|\}\!|\}$. To save space, write k for $\mathtt{const}\ \{\!|\mathtt{x} \mapsto \{\!|\}\!|\}\ \{\!|\}\!|$ and p for $\mathtt{plunge\ b}$. We calculate $l \nearrow c$ directly, underlining the term to be simplifed at each step.

$$
\begin{aligned}
l \nearrow c &= \underline{(\mathtt{xfork}\ \{\}\ \{\mathtt{b}\}\ (k;\ p)\ \mathtt{id}) \nearrow c} \\
&\quad \text{by the definition of } l \\
&= \underline{(k;\ p) \nearrow \{\!|\}\!|} \cdot \underline{\mathtt{id} \nearrow \{\!|\mathtt{a} \mapsto \{\!|\}\!|\}} \\
&\quad \text{by the definition of } \mathtt{xfork} \text{ and splitting } c \text{ using } \{\} \\
&= p \nearrow (\underline{k \nearrow \{\!|\}\!|}) \cdot \{\!|\mathtt{a} \mapsto \{\!|\}\!|\} \\
&\quad \text{by the definitions of the composition and } \mathtt{id} \\
&= \left(\underline{p \nearrow \{\!|\mathtt{x} \mapsto \{\!|\}\!|\}}\right) \cdot \{\!|\mathtt{a} \mapsto \{\!|\}\!|\} \\
&\quad \text{by the definition of } k \\
&= \left\{\!|\mathtt{a} \mapsto \{\!|\}\!|,\ \mathtt{b} \mapsto \{\!|\mathtt{x} \mapsto \{\!|\}\!|\}\!|\right\} \\
&\quad \text{by the definition of } p
\end{aligned}
$$

Now suppose we modify this tree by renaming the child `a` to `c`, obtaining the tree $a = \{\!|\mathtt{c} \mapsto \{\!|\}\!|,\ \mathtt{b} \mapsto \{\!|\mathtt{x} \mapsto \{\!|\}\!|\}\!|\}$. The result of the *put* function, $l \searrow (a,\ c)$, is calculated as follows:

$$
\begin{aligned}
l \searrow (a,\ c) &= \underline{(\mathtt{xfork}\ \{\}\ \{\mathtt{b}\}\ (k;\ p)\ \mathtt{id}) \searrow (a,\ c)} \\
&\quad \text{by the definition of } l \\
&= \left((k;\ p) \searrow \left(\left\{\!|\mathtt{b} \mapsto \{\!|\mathtt{x} \mapsto \{\!|\}\!|\}\!|\right\},\ \{\!|\}\!|\right)\right) \cdot \left(\underline{\mathtt{id} \searrow (\{\!|\mathtt{c} \mapsto \{\!|\}\!|\},\ \{\!|\mathtt{a} \mapsto \{\!|\}\!|\})}\right) \\
&\quad \text{by the definition of } \mathtt{xfork}, \text{ splitting } a \text{ using } \{\mathtt{b}\} \text{ and } c \text{ using } \{\} \\
&= \left(\underline{(k;\ p) \searrow \left(\left\{\!|\mathtt{b} \mapsto \{\!|\mathtt{x} \mapsto \{\!|\}\!|\}\!|\right\},\ \{\!|\}\!|\right)}\right) \cdot \{\!|\mathtt{c} \mapsto \{\!|\}\!|\} \\
&\quad \text{by the definition of } \mathtt{id} \\
&= \left(k \searrow \left(\underline{p \searrow \left(\left\{\!|\mathtt{b} \mapsto \{\!|\mathtt{x} \mapsto \{\!|\}\!|\}\!|\right\},\ k \nearrow \{\!|\}\!|\right)},\ \{\!|\}\!|\right)\right) \cdot \{\!|\mathtt{c} \mapsto \{\!|\}\!|\} \\
&\quad \text{by the definition of composition} \\
&= \left(\underline{k \searrow (\{\!|\mathtt{x} \mapsto \{\!|\}\!|\},\ \{\!|\}\!|)}\right) \cdot \{\!|\mathtt{c} \mapsto \{\!|\}\!|\} \\
&\quad \text{by the definition of } p \\
&= \{\!|\}\!| \cdot \{\!|\mathtt{c} \mapsto \{\!|\}\!|\} \quad = \quad \{\!|\mathtt{c} \mapsto \{\!|\}\!|\} \\
&\quad \text{by the definition of } k
\end{aligned}
$$

Another derived lens focuses attention on a single child n:

$$
\boxed{\begin{array}{c}
\mathtt{focus}\ n\ d \quad = \quad (\mathtt{filter}\ \{n\}\ d);\ (\mathtt{hoist}\ n) \\
\hline
\forall n{\in}\mathcal{N}.\ \forall C{\subseteq}\mathcal{T}\backslash_n.\ \forall d{\in}C.\ \forall D{\subseteq}\mathcal{T}. \\
\quad \mathtt{focus}\ n\ d \in (C \cdot \{\!|n \mapsto D|\!\}) \overset{\Omega}{\rightleftharpoons} D
\end{array}}
$$

In the *get* direction, `focus` filters away all other children, then removes the edge n and yields n's subtree. As usual, the default tree is only used in the case of creation, where it is the default for children that have been filtered away. The type of `focus` follows from the types of the lenses from which it is defined, observing that $\mathtt{filter}\ \{n\}\ d \in (C \cdot \{\!|\ n \mapsto D\ |\!\}) \overset{\Omega}{\rightleftharpoons} \{\!|\ n \mapsto D\ |\!\}$ and that $\mathtt{hoist\ n} \in \{\!|\ n \mapsto D\ |\!\} \overset{\Omega}{\rightleftharpoons} D$.

The `hoist` primitive defined earlier requires that the name being hoisted be the unique child of the concrete tree. It is often useful to relax this requirement, hoisting one child out of many. This generalized version of `hoist` is annotated with the set p of possible names of the grandchildren that will become children after the hoist, which must be disjoint from the names of the existing children.

$$\frac{\texttt{hoist_nonunique}\ n\ p \quad = \quad \texttt{xfork}\ \{n\}\ p\ (\texttt{hoist}\ n)\ \texttt{id}}{\begin{array}{l}\forall n{\in}\mathcal{N}.\ \forall p{\subseteq}\mathcal{N}.\ \forall D{\subseteq}\mathcal{T}\backslash_{\{n\}\cup p}.\ \forall C{\subseteq}\mathcal{T}|_p. \\ \quad \texttt{hoist_nonunique}\ n\ p \in (\{\!|n \mapsto C|\!\} \cdot D) \stackrel{\Omega}{=\!=} (C \cdot D)\end{array}}$$

A last derived lens renames a single child.

$$\frac{\texttt{rename}\ m\ n \quad = \quad \texttt{xfork}\ \{m\}\ \{n\}\ (\texttt{hoist}\ m;\ \texttt{plunge}\ n)\ \texttt{id}}{\begin{array}{l}\forall m,n{\in}\mathcal{N}.\ \forall C{\subseteq}\mathcal{T}.\ \forall D{\subseteq}\mathcal{T}\backslash_{\{m,n\}}. \\ \quad \texttt{rename}\ m\ n \in (\{\!|m \mapsto C|\!\} \cdot D) \stackrel{\Omega}{=\!=} (\{\!|n \mapsto C|\!\} \cdot D)\end{array}}$$

In the *get* direction, `rename` splits the concrete tree in two. The first tree has a single child m (which is guaranteed to exist by the type annotation) and is hoisted up, removing the edge named m, and then plunged under n. The rest of the original tree is passed through the `id` lens. Similarly, the *put* direction splits the abstract view into a tree with a single child `n`, and the rest of the tree. The tree under `n` is put back using the lens (`hoist` m; `plunge` n), which first removes the edge named n and then plunges the resulting tree under m. Note that the type annotation on `rename` demands that the concrete view have a child named m and that the abstract view have a child named n. In Section 3.6 we will see how to wrap this lens in a conditional to obtain a lens with a more flexible type.

Mapping

So far, all of our lens combinators do things near the root of the trees they are given. Of course, we also want to be able to perform transformations in the interior of trees. The `map` combinator is our fundamental means of doing this. When combined with recursion, it also allows us to iterate over structures of arbitrary depth.

The `map` combinator is parameterized on a single lens l. In the *get* direction, `map` applies $l{\nearrow}$ to each subtree of the root and combines the results together into a new tree. (Later in this section, we will define a more general combinator, called `wmap`, that can apply a different lens to each subtree. Defining `map` first lightens the notational burden in the explanations of several fine points about the behavior and typing of both combinators.) For example, the lens `map` l has the following behavior in the *get* direction when applied to a tree with three children:

$$\left\{\begin{vmatrix} n_1 \mapsto t_1 \\ n_2 \mapsto t_2 \\ n_3 \mapsto t_3 \end{vmatrix}\right\} \quad \text{becomes} \quad \left\{\begin{vmatrix} n_1 \mapsto l{\nearrow}t_1 \\ n_2 \mapsto l{\nearrow}t_2 \\ n_3 \mapsto l{\nearrow}t_3 \end{vmatrix}\right\}$$

The *put* direction of `map` is more interesting. In the simple case where a and c have equal domains, its behavior is straightforward: it uses $l{\searrow}$ to combine concrete and abstract subtrees with identical names and assembles the results into a new concrete tree, c':

$$(\texttt{map}\ l){\searrow}\left(\left\{\begin{vmatrix} n_1 \mapsto t_1 \\ n_2 \mapsto t_2 \\ n_3 \mapsto t_3 \end{vmatrix}\right\}, \left\{\begin{vmatrix} n_1 \mapsto t'_1 \\ n_2 \mapsto t'_2 \\ n_3 \mapsto t'_3 \end{vmatrix}\right\}\right) = \left\{\begin{vmatrix} n_1 \mapsto l{\searrow}(t_1,\ t'_1) \\ n_2 \mapsto l{\searrow}(t_2,\ t'_2) \\ n_3 \mapsto l{\searrow}(t_3,\ t'_3) \end{vmatrix}\right\}$$

In general, however, the abstract tree a in the *put* direction need not have the same domain as c (i.e., the edits that produced the new abstract view may have involved adding and deleting children); the behavior of `map` in this case is a little more involved. Observe, first, that the domain of c' is determined by the domain of the abstract argument to *put*. Since we aim at building total lenses, we may suppose that $(\texttt{map}\ l){\nearrow}((\texttt{map}\ l){\searrow}(a,\ c))$

is defined, in which case $\mathsf{dom}((\texttt{map}\ l)\nearrow((\texttt{map}\ l)\searrow(a,\,c))) = \mathsf{dom}(a)$ by rule PUTGET, and $\mathsf{dom}((\texttt{map}\ l)\searrow(a,\,c)) = \mathsf{dom}(a)$ as $(\texttt{map}\ l)\nearrow$ does not change the domain of the tree. This means we can simply drop children that occur in $\mathsf{dom}(c)$ but not in $\mathsf{dom}(a)$. Children bearing names that occur both in $\mathsf{dom}(a)$ and $\mathsf{dom}(c)$ are dealt with as described above. This leaves the children that only appear in $\mathsf{dom}(a)$, which need to be passed through l so that they can be included in c'; to do this, we need some concrete argument to pass to $l\searrow$. There is no corresponding child in c, so instead these abstract trees are put into the missing tree Ω—indeed, this case is precisely why we introduced Ω. Formally, the behavior of `map` is defined as follows. (It relies on the convention that $c(n) = \Omega$ if $n \notin \mathsf{dom}(c)$; the type declaration also involves some new notation, explained below.)

$$\begin{array}{rcl} \texttt{map}\ l\nearrow c & = & \{\!\!\{ n \mapsto l\nearrow c(n) \mid n \in \mathsf{dom}(c) \}\!\!\} \\ \texttt{map}\ l\searrow(a,\,c) & = & \{\!\!\{ n \mapsto l\searrow(a(n),\,c(n)) \mid n \in \mathsf{dom}(a) \}\!\!\} \\ \hline \multicolumn{3}{l}{\forall C, A \subseteq \mathcal{T} \textit{ with } C = C^{\circlearrowleft},\ A = A^{\circlearrowleft},\ \mathsf{doms}(C) = \mathsf{doms}(A).} \\ \multicolumn{3}{l}{\forall l \in (\bigcap_{n\in\mathcal{N}}.\ C(n) \overset{\Omega}{\rightleftharpoons} A(n)).} \\ \multicolumn{3}{l}{\quad \texttt{map}\ l \in C \overset{\Omega}{\rightleftharpoons} A} \end{array}$$

Because of the way that it takes the tree apart, transforms the pieces, and reassembles them, the typing of `map` is a little subtle. For example, in the *get* direction, `map` does not modify the names of the immediate children of the concrete tree, and in the *put* direction, the names of the abstract tree are left unchanged; we might therefore expect a simple typing rule stating that, if $l \in (\bigcap_{n\in\mathcal{N}} C(n) \overset{\Omega}{\rightleftharpoons} A(n))$—i.e., if l is a well-behaved lens from the concrete subtree type $C(n)$ to the abstract subtree type $A(n)$ for each child n—then $\texttt{map}\ l \in C \overset{\Omega}{\rightleftharpoons} A$. Unfortunately, for arbitrary C and A, the `map` lens is not guaranteed to be well-behaved at this type. In particular, if $\mathsf{doms}(C)$, the set of domains of trees in C, is not equal to $\mathsf{doms}(A)$, then the *put* function can produce a tree that is not in C, as the following example shows. Consider the sets of trees

$$C = \{\, \{\!\!\{ \mathtt{x} \mapsto \mathtt{m} \}\!\!\},\ \{\!\!\{ \mathtt{y} \mapsto \mathtt{n} \}\!\!\} \,\} \qquad A = C \cup \{\, \{\!\!\{ \mathtt{x} \mapsto \mathtt{m},\ \mathtt{y} \mapsto \mathtt{n} \}\!\!\} \,\}$$

and observe that with trees

$$a = \{\!\!\{ \mathtt{x} \mapsto \mathtt{m},\ \mathtt{y} \mapsto \mathtt{n} \}\!\!\} \qquad c = \{\!\!\{ \mathtt{x} \mapsto \mathtt{m} \}\!\!\}$$

we have $\texttt{map}\ \texttt{id}\searrow(a,\,c) = a$, a tree that is not in C. This shows that the type of `map` must include the requirement that $\mathsf{doms}(C) = \mathsf{doms}(A)$. (Recall that, for any type T, the set $\mathsf{doms}(T)$ is a set of sets of names.)

A related problem arises when the sets of trees A and C have dependencies between the names of children and the trees that may appear under those names. Again, one might naively expect that, if l has type $C(m) \overset{\Omega}{\rightleftharpoons} A(m)$ for each name m, then $\texttt{map}\ l$ would have type $C \overset{\Omega}{\rightleftharpoons} A$. Consider, however, the set

$$A = \{\{\!\!\{ \mathtt{x} \mapsto \mathtt{m},\ \mathtt{y} \mapsto \mathtt{p} \}\!\!\},\ \{\!\!\{ \mathtt{x} \mapsto \mathtt{n},\ \mathtt{y} \mapsto \mathtt{q} \}\!\!\}\},$$

in which the value `m` only appears under `x` when `p` appears under `y`, and the set

$$C = \{\{\!\!\{ \mathtt{x} \mapsto \mathtt{m},\ \mathtt{y} \mapsto \mathtt{p} \}\!\!\},\ \{\!\!\{ \mathtt{x} \mapsto \mathtt{m},\ \mathtt{y} \mapsto \mathtt{q} \}\!\!\},\ \{\!\!\{ \mathtt{x} \mapsto \mathtt{n},\ \mathtt{y} \mapsto \mathtt{p} \}\!\!\},\ \{\!\!\{ \mathtt{x} \mapsto \mathtt{n},\ \mathtt{y} \mapsto \mathtt{q} \}\!\!\}\},$$

where both `m` and `n` appear with both `p` and `q`. When we consider just the projections of C and A at specific names, we obtain the same sets of subtrees: $C(\mathtt{x}) = A(\mathtt{x}) = \{\{\!\!\{ \mathtt{m} \}\!\!\}, \{\!\!\{ \mathtt{n} \}\!\!\}\}$ and $C(\mathtt{y}) = A(\mathtt{y}) = \{\{\!\!\{ \mathtt{p} \}\!\!\}, \{\!\!\{ \mathtt{q} \}\!\!\}\}$. The lens `id` has type $C(\mathtt{x}) \overset{\Omega}{\rightleftharpoons} A(\mathtt{x})$ and

$C(\mathtt{y}) \overset{\Omega}{\rightleftharpoons} A(\mathtt{y})$ (and $C(z) = \emptyset \overset{\Omega}{\rightleftharpoons} \emptyset = A(z)$ for all other names z). But it is clearly not the case that $\mathtt{map\ id} \in C \overset{\Omega}{\rightleftharpoons} A$.

To avoid this error, but still give a type for `map` that is precise enough to derive interesting types for lenses defined in terms of `map`, we require that the source and target sets in the type of `map` be closed under the "shuffling" of their children. Formally, if T is a set of trees, then the set of *shufflings* of T, denoted $T^{\circlearrowleft}$, is

$$T^{\circlearrowleft} = \bigcup_{D \in \mathsf{doms}(T)} \{\!\!\{\, n \mapsto T(n) \ \mid\ n \in D \,\}\!\!\}$$

where $\{\!\!\{\, n \mapsto T(n) \ \mid\ n \in D \,\}\!\!\}$ is the set of trees with domain D whose children under n are taken from the set $T(n)$. We say that T is *shuffle closed* iff $T = T^{\circlearrowleft}$. In the example above, $A^{\circlearrowleft} = C^{\circlearrowleft} = C$—i.e., C is shuffle closed, but A is not.

Alternatively, every shuffle-closed set T can be identified with a set of set of names D and a function f from names to types, such that $t \in T$ iff $\mathsf{dom}(t) \in D$ and $t(n) \in f(n)$ for every name $n \in \mathsf{dom}(t)$. Formally, the shuffle closed set T is defined as follows:

$$T = \bigcup_{d \in D} \{\!\!\{\, n \mapsto f(n) \ \mid\ n \in d \,\}\!\!\}$$

In the situations where `map` is used, shuffle closure is typically easy to check. For example, the restriction on tree grammars embodied by W3C Schema implies shuffle closure (informally, the restriction on W3C Schema is analogous to imposing shuffle closure on the schemas along every path, not just at the root). Additionally, any set of trees whose elements each have singleton domains is shuffle closed. Also, for every set of trees T, the encoding introduced in 3.7 of lists with elements in T is shuffle closed, which justifies using `map` (with recursion) to implement operations on lists. Furthermore, types of the form $\{\!\!\{\, n \mapsto T \ \mid\ n \in \mathcal{N} \,\}\!\!\}$ with infinite domain but with the same structure under each edge, which are heavily used in database examples (where the top-level names are keys and the structures under them are records) are shuffle closed.

Another point to note about `map` is that it does not obey the PutPut law. Consider a lens l and $(a, c) \in \mathsf{dom}(l\searrow)$ such that $l\searrow(a,\, c) \neq l\searrow(a,\, \Omega)$. We have

$$\begin{aligned} & (\mathtt{map}\ l)\searrow\big(\{\!\!\{\mathtt{n} \mapsto a\}\!\!\}\,,\ ((\mathtt{map}\ l)\searrow(\{\!\!\{\}\!\!\},\ \{\!\!\{\mathtt{n} \mapsto c\}\!\!\}))\big) \\ = \ & (\mathtt{map}\ l)\searrow\big(\{\!\!\{\mathtt{n} \mapsto a\}\!\!\}\,,\ \{\!\!\{\}\!\!\}\big) \\ = \ & \{\!\!\{\mathtt{n} \mapsto l\searrow(a,\, \Omega)\}\!\!\} \end{aligned}$$

whereas

$$\{\!\!\{\mathtt{n} \mapsto l\searrow(a,\, c)\}\!\!\} \quad = \quad (\mathtt{map}\ l)\searrow\big(\{\!\!\{\mathtt{n} \mapsto a\}\!\!\}\,,\ \{\!\!\{\mathtt{n} \mapsto c\}\!\!\}\big).$$

Intuitively, there is a difference between, on the one hand, modifying a child n and, on the other, removing it and then adding it back: in the first case, any information in the concrete view that is "projected away" in the abstract view will be carried along to the new concrete view; in the second, such information will be replaced with default values. This difference seems pragmatically reasonable, so we prefer to keep `map` and lose PutPut.[6]

A final point of interest is the relation between `map` and the missing tree Ω. The *put* function of most lens combinators only results in a *put* into the missing tree if the

[6] Alternatively, we could use a refinement of the type system to track when PutPut does hold, annotating some of the lens combinators with extra type information recording the fact that they are oblivious (i.e., ignore their concrete argument), and then give `map` two types: the one we gave here plus another saying "when `map` is applied to an oblivious lens, the result is very well behaved."

combinator itself is called on Ω. In the case of `map` l, calling its *put* function on some a and c where c is not the missing tree may result in the application of the *put* of l to Ω if a has some children that are not in c. In an earlier variant of `map`, we dealt with missing children by providing a default concrete child tree, which would be used when no actual concrete tree was available. However, we discovered that, in practice, it is often difficult to find a single default concrete tree that fits all possible abstract trees, particularly because of `xfork` (where different lenses are applied to different parts of the tree) and recursion (where the depth of a tree is unknown). We tried parameterizing this default concrete tree by the abstract tree and the lens, but noticed that most primitive lenses ignore the concrete tree when defining the *put* function, as enough information is available in the abstract tree. The natural choice for a concrete tree parameterized by a and l was thus $l \searrow (a, \Omega)$, for some special tree Ω. The only lens for which the *put* function needs to be defined on Ω is `const`, as it is the only lens that discards information. This led us to the present design, where only the `const` lens (along with other lenses defined from it, such as `focus`) expects a default tree d. This approach is much more convenient to program with than the others we tried, since one only provides defaults at the exact points where information is discarded.

We now define a more general form of `map` that is parameterized on a total function from names to lenses rather than on a single lens.

$$\begin{array}{rcl} \texttt{wmap}\ m \nearrow c & = & \{\!\!\{ n \mapsto m(n) \nearrow c(n) \mid n \in \mathsf{dom}(c) \}\!\!\} \\ \texttt{wmap}\ m \searrow (a,\, c) & = & \{\!\!\{ n \mapsto m(n) \searrow (a(n),\, c(n)) \mid n \in \mathsf{dom}(a) \}\!\!\} \end{array}$$

$$\begin{array}{l} \forall C, A \subseteq \mathcal{T} \textit{ with } C = C^{\circlearrowleft},\ A = A^{\circlearrowleft},\ \mathsf{doms}(C) = \mathsf{doms}(A). \\ \forall m \in (\Pi n \in \mathcal{N}.\ C(n) \overset{\Omega}{\rightleftharpoons} A(n)). \\ \quad \texttt{wmap}\ m \in C \overset{\Omega}{\rightleftharpoons} A \end{array}$$

In the type annotation, we use the dependent type notation $m \in \Pi n.\ C(n) \overset{\Omega}{\rightleftharpoons} A(n)$ to mean that m is a total function mapping each name n to a well-behaved lens from $C(n)$ to $A(n)$. Although m is a total function, we will often describe it by giving its behavior on a finite set of names and adopting the convention that it maps every other name to `id`. For example, the lens `wmap` $\{x \mapsto$ `plunge a`$\}$ maps `plunge a` over trees under x and `id` over the subtrees of every other child. We can also easily define `map` as a derived form: `map` $l =$ `wmap` $(\lambda n \in \mathcal{N}.\ l)$.

Merging

It sometimes happens that a concrete representation requires equality between two distinct subtrees. The following `merge` lens is one way to preserve this invariant when the abstract view is updated. In the *get* direction, `merge` takes a tree with two equal branches and deletes one of them. In the *put* direction, `merge` copies the updated value of the remaining branch to *both* branches in the concrete view.

$$\begin{array}{rcl} \texttt{merge}\ m\ n \nearrow c & = & c \backslash_n \\ \texttt{merge}\ m\ n \searrow (a,\, c) & = & \begin{cases} a \cdot \{\!\!\{ n \mapsto a(m) \}\!\!\} & \text{if } c(m) = c(n) \\ a \cdot \{\!\!\{ n \mapsto c(n) \}\!\!\} & \text{if } c(m) \neq c(n) \end{cases} \end{array}$$

$$\begin{array}{l} \forall m, n \in \mathcal{N}.\ \forall C \subseteq \mathcal{T} \backslash_{\{m,n\}}.\ \forall D \subseteq \mathcal{T}. \\ \quad \texttt{merge}\ m\ n \in (C \cdot \{\!\!\{ m \mapsto D_\Omega,\ n \mapsto D_\Omega \}\!\!\}) \overset{\Omega}{\rightleftharpoons} (C \cdot \{\!\!\{ m \mapsto D_\Omega \}\!\!\}) \end{array}$$

There is some freedom in the type of `merge`. On one hand, we can give it a precise type that expresses the intended equality constraint in the concrete view; the lens is

well-behaved and total at that type. Alternatively, we can give it a more permissive type (as we do) by ignoring the equality constraint—even if the two original branches are unequal, `merge` is still defined and well-behavedness is preserved. This is possible because the old concrete view is an argument to the *put* function, and can be tested to see whether the two branches were equal or not in c. If not, then the value in a does not overwrite the value in the deleted branch, allowing `merge` to obey PutGet.

The `merge` lens turns out to be quite useful in our synchronization framework. For example, our bookmark synchronizer must deal with the fact that the XML representation of Apple Safari bookmark files includes the URL data for every link twice. By merging the appropriate children, we record this dependency and ensure that updates to the URL fields are consistently propagated to both locations.

3.6 Conditionals

Conditional lens combinators, which can be used to selectively apply one lens or another to a view, are necessary for writing many interesting derived lenses. Whereas `xfork` and its variants split their input trees into two parts, send each part through a separate lens, and recombine the results, a conditional lens performs some test and sends the whole tree(s) through one or the other of its sub-lenses.

The requirement that makes conditionals tricky is totality: we want to be able to take a concrete view, put it through a conditional lens to obtain some abstract view, and then take *any* other abstract view of suitable type and push it back down. But this will only work if either (1) we somehow ensure that the abstract view is guaranteed to be sent to the same sub-lens on the way down as we took on the way up, or else (2) the two sub-lenses are constrained to behave coherently. Since we want reasoning about well-behavedness and totality to be compositional in the absence of recursion (i.e., we want the well-behavedness and totality of composite lenses to follow just from the well-behavedness and totality of their sub-lenses, not from special facts about the behavior of the sub-lenses), the second is unacceptable.

Interestingly, once we adopt the first approach, we can give a *complete* characterization of all possible conditional lenses: we argue in Foster et al. (2007b) that every binary conditional operator that yields well-behaved and total lenses is an instance of the general `cond` combinator presented below. Since this general `cond` is a little complex, however, we start by discussing two particularly useful special cases.

Concrete Conditional

Our first conditional, `ccond`, is parameterized on a predicate C_1 on views and two lenses, l_1 and l_2. In the *get* direction, it tests the concrete view c and applies the *get* of l_1 if c satisfies the predicate and l_2 otherwise. In the *put* direction, `ccond` again examines the concrete view, and applies the *put* of l_1 if it satisfies the predicate and the *put* of l_2 otherwise. This is arguably the simplest possible way to define a conditional: it fixes all of its decisions in the *get* direction, so the only constraint on l_1 and l_2 is that they have the same target. (Since we are interested in using `ccond` to define total lenses, this condition can actually be rather hard to achieve in practice.)

$$\texttt{ccond}\ C_1\ l_1\ l_2 \nearrow c \quad = \quad \begin{cases} l_1 \nearrow c & \text{if } c \in C_1 \\ l_2 \nearrow c & \text{if } c \notin C_1 \end{cases}$$

$$\texttt{ccond}\ C_1\ l_1\ l_2 \searrow (a,\, c) \quad = \quad \begin{cases} l_1 \searrow (a,\, c) & \text{if } c \in C_1 \\ l_2 \searrow (a,\, c) & \text{if } c \notin C_1 \end{cases}$$

$$\frac{\forall C, C_1, A \subseteq \mathcal{V}.\ \forall l_1 \in C \cap C_1 \overset{\Omega}{\rightleftharpoons} A.\ \forall l_2 \in C \setminus C_1 \overset{\Omega}{\rightleftharpoons} A.}{\texttt{ccond}\ C_1\ l_1\ l_2 \in C \overset{\Omega}{\rightleftharpoons} A}$$

Abstract Conditional

A quite different way of defining a conditional lens is to make it ignore its *concrete* argument in the *put* direction, basing its decision whether to use $l_1\searrow$ or $l_2\searrow$ entirely on its abstract argument.

$$\begin{array}{rcl}
\texttt{acond}\ C_1\ A_1\ l_1\ l_2{\nearrow}c & = & \begin{cases} l_1{\nearrow}c & \text{if } c \in C_1 \\ l_2{\nearrow}c & \text{if } c \notin C_1 \end{cases} \\
\texttt{acond}\ C_1\ A_1\ l_1\ l_2{\searrow}(a,\,c) & = & \begin{cases} l_1{\searrow}(a,\,c) & \text{if } a \in A_1 \wedge c \in C_1 \\ l_1{\searrow}(a,\,\Omega) & \text{if } a \in A_1 \wedge c \notin C_1 \\ l_2{\searrow}(a,\,c) & \text{if } a \notin A_1 \wedge c \notin C_1 \\ l_2{\searrow}(a,\,\Omega) & \text{if } a \notin A_1 \wedge c \in C_1 \end{cases}
\end{array}$$

$$\forall C, A, C_1, A_1 \subseteq \mathcal{V}.\ \forall l_1 \in C \cap C_1 \overset{\Omega}{\rightleftharpoons} A \cap A_1.\ \forall l_2 \in (C \backslash C_1) \overset{\Omega}{\rightleftharpoons} (A \backslash A_1).$$
$$\texttt{acond}\ C_1\ A_1\ l_1\ l_2 \in C \overset{\Omega}{\rightleftharpoons} A$$

In Section 3.5, we defined the lens `rename` m n, whose type demands that each concrete tree have a child named m and that every abstract tree have a child named n. Using this conditional, we can write a more permissive lens that renames a child if it is present and otherwise behaves like the identity.

$$\begin{array}{rcl}
\texttt{rename_if_present}\ m\ n & = & \texttt{acond} \\
& & (\{\!|\ m \mapsto \mathcal{T}\ |\!\} \cdot \mathcal{T}\backslash_{\{m,n\}})\ (\{\!|\ n \mapsto \mathcal{T}\ |\!\} \cdot \mathcal{T}\backslash_{\{m,n\}}) \\
& & (\texttt{rename}\ m\ n) \\
& & \texttt{id}
\end{array}$$

$$\forall n, m \in \mathcal{N}.\ \forall C \subseteq \mathcal{T}.\ \forall D, E \subseteq (\mathcal{T}\backslash_{\{m,n\}}).$$
$$\texttt{rename_if_present}\ m\ n \in (\{\!|m \mapsto C|\!\} \cdot D) \cup E \overset{\Omega}{\rightleftharpoons} (\{\!|n \mapsto C|\!\} \cdot D) \cup E$$

General Conditional

The general conditional, `cond`, is essentially obtained by combining the behaviors of `ccond` and `acond`. The concrete conditional requires that the targets of the two lenses be identical, while the abstract conditional requires that they be disjoint. Here, we let them overlap arbitrarily, behaving like `ccond` in the region where they do overlap (i.e., for arguments (a, c) to *put* where a is in the intersection of the targets) and like `acond` in the regions where the abstract argument to *put* belongs to just one of the targets. To this we can add one additional observation: that the use of Ω in the definition of `acond` is actually arbitrary. All that is required is that, when we use the *put* of l_1, the concrete argument should come from $(C_1)_\Omega$, so that l_1 is guaranteed to do something reasonable with it. These considerations lead us to the following definition.

$$\begin{array}{rcl}
\texttt{cond}\ C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2{\nearrow}c & = & \begin{cases} l_1{\nearrow}c & \text{if } c \in C_1 \\ l_2{\nearrow}c & \text{if } c \notin C_1 \end{cases} \\
\texttt{cond}\ C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2{\searrow}(a,\,c) & = & \\
& & \begin{cases} l_1{\searrow}(a,\,c) & \text{if } a \in A_1 \cap A_2 \wedge c \in C_1 \\ l_2{\searrow}(a,\,c) & \text{if } a \in A_1 \cap A_2 \wedge c \notin C_1 \\ l_1{\searrow}(a,\,c) & \text{if } a \in A_1 \backslash A_2 \wedge c \in (C_1)_\Omega \\ l_1{\searrow}(a, f_{21}(c)) & \text{if } a \in A_1 \backslash A_2 \wedge c \notin (C_1)_\Omega \\ l_2{\searrow}(a,\,c) & \text{if } a \in A_2 \backslash A_1 \wedge c \notin C_1 \\ l_2{\searrow}(a, f_{12}(c)) & \text{if } a \in A_2 \backslash A_1 \wedge c \in C_1 \end{cases}
\end{array}$$

$$\forall C, C_1, A_1, A_2 \subseteq \mathcal{V}.\ \forall l_1 \in (C \cap C_1) \overset{\Omega}{\rightleftharpoons} A_1.\ \forall l_2 \in (C \backslash C_1) \overset{\Omega}{\rightleftharpoons} A_2.$$
$$\forall f_{21} \in (C \backslash C_1) \to (C \cap C_1)_\Omega.\ \forall f_{12} \in (C \cap C_1) \to (C \backslash C_1)_\Omega.$$
$$\texttt{cond}\ C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2 \in C \overset{\Omega}{\rightleftharpoons} (A_1 \cup A_2)$$

When a is in the targets of both l_1 and l_2, `cond`$\searrow$ chooses between them based solely on c (as does `ccond`, whose targets always overlap). If a lies in the range of only l_1 or l_2, then `cond`'s choice of lens for *put* is predetermined (as with `acond`, whose targets are disjoint). Once $l\searrow$ is chosen to be either $l_1\searrow$ or $l_2\searrow$, if the old value of c is not in $\mathsf{ran}(l\searrow)_\Omega$, then we apply a "fixup function," f_{21} or f_{12}, to c to choose a new value from $\mathsf{ran}(l\searrow)_\Omega$. Ω is one possible result of the fixup functions, but in general we can compute a more interesting value, as we will see in the `list_filter` lens, defined in Section 3.7.

3.7 Derived Lenses for Lists

XML and many other concrete data formats make heavy use of ordered lists. We describe in this section how we can represent lists as trees, using a standard cons-cell encoding, and introduce some derived lenses to manipulate them. We begin with very simple lenses for projecting the head and tail of a list. We then define recursive lenses implementing some more complex operations on lists: mapping and filtering. In Foster et al. (2007b), we also show how to derive a list-reversing lens that takes a list encoded as a tree and yields the same list in reverse order (in both directions, ignoring its concrete argument in the *put* direction) and a "grouping" lens that, in the *get* direction, takes a list whose elements alternate between elements of D and elements of E and returns a list of pairs of Ds and Es—e.g., it maps `[d1 e1 d2 e2 d3 e3]` to `[[d1 e1] [d2 e2] [d3 e3]]`.

Encoding

Definition 3.7.1. *A tree t is said to be a* list *iff either it is empty or it has exactly two children, one named* `*h` *and another named* `*t`*, and t(`*t`) is also a list. We use the lighter notation* `[`$t_1 \ldots t_n$`]` *for the tree*

$$\left\{\begin{array}{l} \texttt{*h} \mapsto t_1 \\ \texttt{*t} \mapsto \left\{\begin{array}{l} \texttt{*h} \mapsto t_2 \\ \texttt{*t} \mapsto \left\{\ldots \mapsto \left\{\begin{array}{l} \texttt{*h} \mapsto t_n \\ \texttt{*t} \mapsto \{\!\{\}\!\} \end{array}\right\}\right\}\right\} \end{array}\right\}\right\}.$$

In types, we write `[]` *for the set $\{\{\!\{\}\!\}\}$ containing only the empty list, $C :: D$ for the set $\{\!\{\texttt{*h} \mapsto C,\ \texttt{*t} \mapsto D\}\!\}$ of "cons-cell trees" whose head belongs to C and whose tail belongs to D, and* `[C]` *for the set of lists with elements in C—i.e., the smallest set of trees satisfying* `[C]` $= \texttt{[]} \cup (C :: \texttt{[C]})$*. Given two list values, l_1 and l_2, the set of lists denoted by the* interleaving *$l_1 \& l_2$ consists of all the lists formed by interleaving the elements of l_1 with the elements of l_2 in an arbitrary fashion. For example,* `[a, b]&[c]` *is the set $\{$*`[a, b, c]`*,* `[a, c, b]`*,* `[c, a, b]`*$\}$. We lift the interleaving operator to list types in the obvious way: the interleaving of two list types,* `[B]` *and* `[C]`*, is the union of all the interleavings of lists belonging to* `[B]` *with lists belonging to* `[C]`*.*

Head and Tail Projections

Our first list lenses extract the head or tail of a cons cell.

$$\frac{\texttt{hd}\ d \quad = \quad \texttt{focus *h}\ \{\!\{\texttt{*t} \mapsto d\}\!\}}{\forall C, D \subseteq \mathcal{T}.\ \forall d \in D. \quad \texttt{hd}\ d \in (C :: D) \overset{\Omega}{\rightleftharpoons} C}$$

$$\frac{\texttt{tl}\ d \quad = \quad \texttt{focus *t}\ \{\!\{\texttt{*h} \mapsto d\}\!\}}{\forall C, D \subseteq \mathcal{T}.\ \forall d \in C. \quad \texttt{tl}\ d \in (C :: D) \overset{\Omega}{\rightleftharpoons} D}$$

The lens `hd` expects a default tree, which it uses in the *put* direction as the tail of the created tree when the concrete tree is missing; in the *get* direction, it returns the tree under `*h`. The lens `tl` works analogously. Note that the types of these lenses apply to both homogeneous lists (the type of `hd` implies $\forall C{\subseteq}\mathcal{T}.\ \forall d{\in}\texttt{[}C\texttt{]}.\ \texttt{hd}\ d \in \texttt{[}C\texttt{]} \overset{\Omega}{\Longleftrightarrow} C$) as well as cons cells whose head and tail have unrelated types. The types of `hd` and `tl` follow from the type of `focus`.

List Map

The `list_map` lens applies a lens l to each element of a list:

$$\frac{\texttt{list_map}\ l \quad = \quad \texttt{wmap}\ \{\texttt{*h} \mapsto l,\ \texttt{*t} \mapsto \texttt{list_map}\ l\}}{\forall C, A{\subseteq}\mathcal{T}.\ \forall l \in C \overset{\Omega}{\rightleftharpoons} A. \qquad \texttt{list_map}\ l \in \texttt{[}C\texttt{]} \overset{\Omega}{\rightleftharpoons} \texttt{[}A\texttt{]}}$$

The *get* direction applies l to the subtree under `*h` and recurses on the subtree under `*t`. The *put* direction uses $l{\searrow}$ on corresponding pairs of elements from the abstract and concrete lists. The result has the same length as the abstract list; if the concrete list is longer, the extra tail is thrown away. If it is shorter, each extra element of the abstract list is *put* into Ω.

Since `list_map` is our first recursive lens, it is worth noting how recursive calls are made in each direction. The *get* function of the `wmap` lens simply applies l to the head and `list_map` l to the tail until it reaches a tree with no children. Similarly, in the *put* direction, `wmap` applies l to the head of the abstract tree and either the head of the concrete tree (if it is present) or Ω, and it applies `list_map` l to the tail of the abstract tree and the tail of the concrete tree (if it is present) or Ω. In both directions, the recursive calls continue until the entire tree—concrete (for the *get*) or abstract (for the *put*)—has been traversed. (The recursion is controlled by the abstract argument in the *put* direction because the `map` combinator uses the children of the abstract tree to determine how many times to call its argument lens.)

Filter

Our most interesting derived list processing lens, `list_filter`, is parameterized on two sets of views, D and E, which we assume to be disjoint and non-empty. In the *get* direction, it takes a list whose elements belong to either D or E and projects away those that belong to E, leaving an abstract list containing only Ds; in the *put* direction, it restores the projected-away Es from the concrete list. Its definition utilizes our most complex lens combinators—`wmap` and two forms of conditionals—and recursion, yielding a lens that is well-behaved and total on lists of arbitrary length.

In the *get* direction, the desired behavior of `list_filter` D E (for brevity, let us call it l) is clear. In the *put* direction, things are more interesting because there are many ways that we could restore projected elements from the concrete list. The lens laws impose some constraints on the behavior of $l{\searrow}$. The GetPut law forces the *put* function to restore each of the filtered elements when the abstract list is put into the original concrete list. For example (letting `d` and `e` be elements of D and E) we must have $l{\searrow}$(`[d]`, `[e d]`) = `[e d]`. The PutGet law forces the *put* function to include every element of the abstract list in the resulting concrete list in the same order, and these elements must be the only Ds in the result; there is, however, no restriction on the Es when the abstract tree is not the filtered concrete tree.

In the general case, where the abstract list a is different from the filtered concrete list $l{\nearrow}c$, there is some freedom in how $l{\searrow}$ behaves. First, it may selectively restore only some of the elements of E from the concrete list (or indeed, even less intuitively, it might

add some new elements of E that it somehow makes up). Second, it may interleave the restored Es with the Ds from the abstract list in any order, as long as the order of the Ds is preserved from a. From these possibilities, the behavior that seems most natural to us is to overwrite elements of D in c with elements of D from a, element-wise, until either c or a runs out of elements of D. If c runs out first, then $l\searrow$ appends the rest of the elements of a at the end of c. If a runs out first, then $l\searrow$ restores the remaining Es from the end of c and discards any remaining Ds in c (as it must to satisfy PutGet).

These choices lead us to the following specification for a single step of the *put* part of a recursively defined lens implementing l. If the abstract list a is empty, then we restore all the Es from c. If c is empty and a is not empty, then we return a. If a and c are both cons cells whose heads are in D, then we return a cons cell whose head is the head of a and whose tail is the result obtained by recursing on the tails of both a and c. Otherwise (i.e., c has type $E::(\texttt{[}D\texttt{]}\&\texttt{[}E\texttt{]})$) we restore the head of c and recurse on a and the tail of c. Translating this into lens combinators leads to the definition below of a recursive lens `inner_filter`, which filters lists containing at least one D, and a top-level lens `list_filter` that handles arbitrary lists of Ds and Es.

```
inner_filter D E  =
    ccond (E::((D::[D])&[E]))
      (tl any_E; inner_filter D E)
      (wmap {*h ↦ id,
             *t ↦ (cond [E] [] (D::[D]) fltr_E (λc. c++[any_D])
                      (const [] [])
                      (inner_filter D E))})

 list_filter D E  =
    cond [E] [] (D::[D]) fltr_E (λc. c++[any_D])
        (const [] [])
        (inner_filter D E)}
```

$$\forall D, E \subseteq \mathcal{T}.\ \textit{with}\ D \cap E = \emptyset\ \textit{and}\ D \neq \emptyset\ \textit{and}\ E \neq \emptyset.$$
$$\texttt{inner_filter}\ D\ E \in (D::\texttt{[}D\texttt{]})\&\texttt{[}E\texttt{]} \overset{\Omega}{\rightleftharpoons} (D::\texttt{[}D\texttt{]})$$
$$\texttt{list_filter}\ D\ E \in \texttt{[}D\texttt{]}\&\texttt{[}E\texttt{]} \overset{\Omega}{\rightleftharpoons} \texttt{[}D\texttt{]}$$

The "choice operator" any_D denotes an arbitrary element of the (non-empty) set D.[7] The function $fltr_E$ is the usual list-filtering *function*, which for present purposes we simply assume has been defined as a primitive. (In our actual implementation, we use `list_filter`$\nearrow$; but for expository purposes, and to simplify the totality proofs, we avoid this extra bit of recursiveness.) Finally, the function $\lambda c.\ c\texttt{++}\texttt{[}any_D\texttt{]}$ appends some arbitrary element of D to the right-hand end of a list c. These "fixup functions" are applied in the *put* direction by the `cond` lens.

The behavior of the *get* function of `list_filter` can be described as follows. If $c \in \texttt{[}E\texttt{]}$, then the outermost `cond` selects the `const [] []` lens, which produces `[]`. Otherwise the `cond` selects `inner_filter`, which uses a `ccond` instance to test if the head of the list is in E. If this test succeeds, it strips away the head using `tl` and recurses; if not, it retains the head and filters the tail using `wmap`.

In the *put* direction, if $a = \texttt{[]}$ then the outermost `cond` lens selects the `const[] []` lens, with c as the concrete argument if $c \in \texttt{[}E\texttt{]}$ and $(fltr_E\ c)$ otherwise. This has the effect of restoring all of the Es from c. Otherwise, if $a \neq \texttt{[]}$ then the `cond` instance selects the *put* of the `inner_filter` lens, using c as the concrete argument if c contains

[7] We are dealing with countable sets of finite trees here, so this construct poses no metaphysical conundrums; alternatively, but less readably, we could just as well pass `list_filter` an extra argument $d \in D$.

at least one D, and $(\lambda c.\ c\texttt{++}[\mathit{any}_D])\ c$, which appends a dummy value of type D to the tail of c, if not. The dummy value, any_D, is required because `inner_filter` expects a concrete argument that contains at least one D. Intuitively, the dummy value marks the point where the head of a should be placed.

To illustrate how all this works, let us step through some examples in detail. In each example, the concrete type is $[D]\&[E]$ and the abstract type is $[D]$. We will write $\mathtt{d}_i$ and $\mathtt{e}_j$ to stand for elements of D and E respectively. To shorten the presentation, we will write l for `list_filter` D E (i.e., for the `cond` lens that it is defined as) and i for `inner_filter` D E. In the first example, the abstract tree a is $[\mathtt{d}_1]$, and the concrete tree c is $[\mathtt{e}_1\ \mathtt{d}_2\ \mathtt{e}_2]$. At each step, we underline the term that is simplified in the next step.

$$
\begin{aligned}
& \underline{l \searrow (a,\, c)} = \underline{i \searrow (a,\, c)} \\
& \qquad \text{by the definition of } \texttt{cond}\text{, as } a \in (D :: [D]) \text{ and } c \in ([D]\&[E]) \setminus [E] \\
= \ & \underline{(\texttt{tl}\ \mathit{any}_E;\ i) \searrow (a,\, c)} \\
& \qquad \text{by the definition of } \texttt{ccond}\text{, as } c \in E :: ((D :: [D])\&[E]) \\
= \ & (\texttt{tl}\ \mathit{any}_E) \searrow \Big(i \searrow \Big(a,\, \underline{(\texttt{tl}\ \mathit{any}_E) \nearrow c}\Big),\, c\Big) \\
& \qquad \text{by the definition of composition} \\
= \ & (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\underline{i \searrow (a,\, [\mathtt{d}_2\ \mathtt{e}_2])},\, c\Big) \\
& \qquad \text{reducing } (\texttt{tl}\ \mathit{any}_E) \nearrow c \\
= \ & (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\underline{\texttt{wmap}\ \{\texttt{*h} \mapsto \texttt{id}, \texttt{*t} \mapsto l\} \searrow (a,\, [\mathtt{d}_2\ \mathtt{e}_2])},\, c\Big) \\
& \qquad \text{by the definition of } \texttt{ccond}\text{, as } [\mathtt{d}_2\ \mathtt{e}_2] \notin E :: ((D :: [D])\&[E]) \\
= \ & (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\mathtt{d}_1 :: (\underline{l \searrow ([],\, [\mathtt{e}_2])}),\, c\Big) \\
& \qquad \text{by the definition of } \texttt{wmap} \text{ with } \texttt{id} \searrow (\mathtt{d}_1,\, \mathtt{d}_2) = \mathtt{d}_1 \\
= \ & (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\mathtt{d}_1 :: (\underline{(\texttt{const}\ []\ []) \searrow ([],\, [\mathtt{e}_2])}),\, c\Big) \\
& \qquad \text{by the definition of } \texttt{cond}\text{, as } [] \in [] \text{ and } [\mathtt{e}_2] \in [E] \\
= \ & \underline{(\texttt{tl}\ \mathit{any}_E) \searrow (\mathtt{d}_1 :: [\mathtt{e}_2],\, c)} \\
& \qquad \text{by the definition of } \texttt{const} \\
= \ & [\mathtt{e}_1\ \mathtt{d}_1\ \mathtt{e}_2] \quad \text{by the definition of } \texttt{tl}.
\end{aligned}
$$

Our next two examples illustrate how the "fixup functions" supplied to the `cond` lens are used. The first function, fltr_E, is used when the abstract list is empty and the concrete list is not in $[E]$. Let $a = []$ and $c = [\mathtt{d}_1\ \mathtt{e}_1]$.

$$
\begin{aligned}
& \underline{l \searrow (a,\, c)} = (\texttt{const}\ []\ []) \searrow \Big([],\, \underline{\mathit{fltr}_E\, [\mathtt{d}_1\ \mathtt{e}_1]}\Big) \\
& \qquad \text{by the definition of } \texttt{cond}\text{, as } a = [] \text{ but } c \notin [E] \\
= \ & \underline{(\texttt{const}\ []\ []) \searrow ([],\, [\mathtt{e}_1])} \\
& \qquad \text{by the definition of } \mathit{fltr}_E \\
= \ & [\mathtt{e}_1] \quad \text{by definition of } \texttt{const}.
\end{aligned}
$$

The other fixup function, $(\lambda c.\ c\texttt{++}[\mathit{any}_D])$, inserts a dummy D element when `list_filter` is called with a non-empty abstract list and a concrete list whose elements are all in E. Let $a = [\mathtt{d}_1]$ and $c = [\mathtt{e}_1]$ and assume that $\mathit{any}_D = \mathtt{d}_0$.

$$
\begin{aligned}
& \underline{l \searrow (a,\, c)} = i \searrow \Big(a,\, \underline{(\lambda c.\ c\texttt{++}[\mathit{any}_D])\ c}\Big) \\
& \qquad \text{by the definition of } \texttt{cond}\text{, as } a \in (D :: [D]) \text{ and } c \in [E] \\
= \ & \underline{i \searrow (a,\, [\mathtt{e}_1\ \mathtt{d}_0])} \\
& \qquad \text{by the definition of } (\lambda c.\ c\texttt{++}[\mathit{any}_D]) \\
= \ & \underline{(\texttt{tl}\ \mathit{any}_E;\ i) \searrow (a,\, [\mathtt{e}_1\ \mathtt{d}_0])} \\
& \qquad \text{by the definition of } \texttt{ccond}\text{, as } [\mathtt{e}_1\ \mathtt{d}_0] \in E :: ((D :: [D])D\&[E])
\end{aligned}
$$

$$
\begin{aligned}
&= (\texttt{tl}\ \mathit{any}_E) \searrow \Big(i \searrow \Big(a,\ \underline{(\texttt{tl}\ \mathit{any}_E) \nearrow [\texttt{e}_1\ \texttt{d}_0]}\Big),\ [\texttt{e}_1\ \texttt{d}_0]\Big) \\
&\qquad \text{by the definition of composition} \\
&= (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\underline{i \searrow (a,\ [\texttt{d}_0])},\ [\texttt{e}_1\ \texttt{d}_0]\Big) \\
&\qquad \text{reducing } (\texttt{tl}\ \mathit{any}_E) \nearrow [\texttt{e}_1\ \texttt{d}_0] \\
&= (\texttt{tl}\ \mathit{any}_E) \\
&\qquad \searrow \Big(\underline{\texttt{wmap}\ \{\texttt{*h} \mapsto \texttt{id}, \texttt{*t} \mapsto l\} \searrow (a,\ [\texttt{d}_0])},\ [\texttt{e}_1\ \texttt{d}_0]\Big) \\
&\qquad \text{by the definition of } \texttt{ccond}, \text{ as } [\texttt{d}_0] \notin E :: ((D :: [D]) \& [E]) \\
&= (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\texttt{d}_1 :: (\underline{l \searrow ([],\ [])}),\ [\texttt{e}_1\ \texttt{d}_0]\Big) \\
&\qquad \text{by the definition of } \texttt{wmap} \text{ with } \texttt{id} \searrow (\texttt{d}_1,\ \texttt{d}_0) = \texttt{d}_1 \\
&= (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\texttt{d}_1 :: (\underline{(\texttt{const}\ []\ []) \searrow ([],\ [])}),\ [\texttt{e}_1\ \texttt{d}_0]\Big) \\
&\qquad \text{by the definition of } \texttt{cond}, \text{ as } [] \in [] \text{ and } [] \in [E] \\
&= \underline{(\texttt{tl}\ \mathit{any}_E) \searrow (\texttt{d}_1 :: [],\ [\texttt{e}_1\ \texttt{d}_0])} \\
&\qquad \text{by the definition of } \texttt{const} \\
&= [\texttt{e}_1\ \texttt{d}_1] \quad \text{by the definition of } \texttt{tl}.
\end{aligned}
$$

3.8 Related Work

Our lens combinators evolved in the setting of the Harmony data synchronizer. The overall architecture of Harmony and the role of lenses in building synchronizers for various forms of data are described elsewhere (Foster et al., 2007a; Pierce et al., 2003), along with a detailed discussion of related work on synchronization.

Our foundational structures—lenses and their laws—are not new: closely related structures have been studied for decades in the database community. However, our treatment of these structures is arguably simpler (transforming states rather than "update functions") and more refined (treating well-behavedness as a form of type assertion). Our formulation is also novel in addressing the issues of totality, offering programmers a static guarantee that lenses cannot fail at run time, and of continuity, supporting a rich variety of surface language structures including definition by recursion.

The idea of defining programming languages for constructing bidirectional transformations of various sorts has also been explored previously in diverse communities. We appear to be the first to take totality as a primary goal (while connecting the language with a formal semantic foundation, choosing primitives that can be combined into composite lenses whose totality is guaranteed by construction), and the first to emphasize types—i.e., compositional reasoning about well-behavedness and totality—as an organizing design principle.

Foundations of View Update The foundations of view update translation were studied intensively by database researchers in the late '70s and '80s. This thread of work is closely related to our semantics of lenses in Section 3.3.

Dayal and Bernstein (1982) gave a seminal formal account of "correct update translation." Their notion of "exactly performing an update" corresponds, intuitively, to our PutGet law. Their "absence of side effects" corresponds to our GetPut and PutPut laws. Their requirement of preservation of semantic consistency corresponds to the partiality of our *put* functions.

Bancilhon and Spyratos (1981) developed an elegant semantic characterization of update translation, introducing the notion of *complement* of a view, which must include at least all information missing from the view. When a complement is fixed, there

exists at most one update of the database that reflects a given update on the view while leaving the complement unmodified—i.e., that "translates updates under a constant complement." The constant complement approach has influenced numerous later works in the area, including more recent papers by Lechtenbörger (2003) and Hegner (2004).

Gottlob, Paolini, and Zicari (1988) offered a more refined theory based on a syntactic translation of view updates. They present a general framework and identify two special cases, one being formally equivalent to Bancilhon and Spyratos's constant complement translators and another—more permissive and which they advocate on pragmatic grounds—called "dynamic views".

Our notion of lenses adopts the same, more permissive, attitude towards reasonable behavior of update translation. Indeed, modulo some technical refinements, we have sketched that the correspondence is tight: the set of all well-behaved lenses is isomorphic to the set of dynamic views in the sense of Gottlob, Paolini, and Zicari. Moreover, the set of very well-behaved lenses is isomorphic to the set of translators under constant complement in the sense of Bancilhon and Spyratos.[8]

In the literature on programming languages, laws similar to our lens laws (but somewhat simpler, dealing only with total *get* and *put* functions) appear in Oles' category of "state shapes" (Oles, 1985) and in Hofmann and Pierce's work on "positive subtyping" (Hofmann and Pierce, 1995).

Languages for Bidirectional Transformations At the level of syntax, different forms of bidirectional programming have been explored across a surprisingly diverse range of communities, including programming languages, databases, program transformation, constraint-based user interfaces, and quantum computing. One useful way of classifying these languages is by the "shape" of the semantic space in which their transformations live. We identify three major classes. *Bidirectional languages*, including ours, form lenses by pairing a *get* function of type $C \to A$ with a *put* function of type $A \times C \to C$. In general, the *get* function can project away some information from the concrete view, which must then be restored by the *put* function. In *bijective languages*, the *put* function has the simpler type $A \to C$, being given no concrete argument to refer to. To avoid loss of information, the *get* and *put* functions must form a (perhaps partial) bijection between C and A. *Reversible languages* go a step further, demanding only that the work performed by any function to produce a given output can be undone by applying the function "in reverse" working backwards from this output to produce the original input. Here, there is no separate *put* function at all: instead, the *get* function itself is constructed so that each step can be run in reverse.

In the first class, the work that is fundamentally most similar to ours is Meertens's formal treatment of *constraint maintainers* for constraint-based user interfaces (Meertens, 1998). Meertens's semantic setting is actually even more general: he takes *get* and *put*

[8]To be precise, we need an additional condition regarding partiality. The frameworks of Bacilhon and Spyratos and of Gottlob, Paolini, and Zicari are both formulated in terms of translating *update functions* on A into update functions on C, i.e., their *put* functions have type $(A \longrightarrow A) \longrightarrow (C \longrightarrow C)$, while our lenses translate abstract *states* into update functions on C, i.e., our *put* functions have type (isomorphic to) $A \longrightarrow (C \longrightarrow C)$. Moreover, in both of these frameworks, "update translators" (the analog of our *put* functions) are defined only over some particular chosen set U of abstract update functions, not over all functions from A to A, and these update functions may be composed. Update translators return *total* functions from C to C. Our *put* functions, on the other hand, are slightly more general as they are defined over all abstract states and return *partial* functions from C to C. Finally, the *get* functions of lenses are allowed to be partial, whereas the corresponding functions (called *views*) in the other two frameworks are assumed to be total. In order to make the correspondences tight, our sets of well-behaved and very well behaved lenses need to be restricted to subsets that are also total in a suitable sense and the set of dynamic views should already include every abstract update functions that are needed and not rely on composition. A related observation is that, if we restrict both *get* and *put* to be total functions (i.e., *put* must be total with respect to *all* abstract update functions), then our lens laws (including PutPut) characterize the set C as isomorphic to $A \times B$ for some B.

to be *relations*, not just functions, and his constraint maintainers are symmetric: *get* relates pairs from $C \times A$ to elements of A and *put* relates pairs in $A \times C$ to elements of C; the idea is that a constraint maintainer forms a connection between two graphical objects on the screen so that, whenever one of the objects is changed by the user, the change can be propagated by the maintainer to the other object such that some desired relationship between the objects is always maintained. Taking the special case where the *get* relation is actually a function (which is important for Meertens because this is the case where composition [in the sense of our ; combinator] is guaranteed to preserve well-behavedness), yields essentially our very well behaved lenses. Meertens proposes a variety of combinators for building constraint maintainers, most of which have analogs among our lenses, but does not directly deal with definition by recursion; also, some of his combinators do not support compositional reasoning about well-behavedness. He considers constraint maintainers for structured data such as lists, as we do for trees, but here adopts a rather different point of view from ours, focusing on constraint maintainers that work with structures not directly but in terms of the "edit scripts" that might have produced them. In the terminology of synchronization, he switches from a state-based to an operation-based treatment at this point.

More recent work of Mu, Hu, and Takeichi on "injective languages" for view-update-based structure editors (2004) adopts a similar perspective. Although their transformations obey our GetPut law, their notion of well-behaved transformations is informed by different goals than ours, leading to a weaker form of the PutGet law. A primary concern is using the view-to-view transformations to simultaneously restore invariants *within* the source view as well as update the concrete view. For example, an abstract view may maintain two lists where the name field of each element in one list must match the name field in the corresponding element in the other list. If an element is added to the first list, then not only must the change be propagated to the concrete view, it must also add a new element to the second list in the abstract view. It is easy to see that PutGet cannot hold if the abstract view, itself, is—in this sense—modified by the *put*. Similarly, they assume that edits to the abstract view mark all modified fields as "updated." These marks are removed when the *put* lens computes the modifications to the concrete view—another change to the abstract view that must violate PutGet. Consequently, to support invariant preservation within the abstract view, and to support edit lists, their transformations only obey a much weaker variant of PutGet.

Another paper by Hu, Mu, and Takeichi (2004) applies a bidirectional programming language closely related to ours to the design of "programmable editors" for structured documents. As in Mu et al. (2004), they support preservation of local invariants in the *put* direction. Here, instead of annotating the abstract view with modification marks, they assume that a *put* or a *get* occurs after *every* modification to either view. They use this "only one update" assumption to choose the correct inverse for the lens that copied data in the *get* direction—because only one branch can have been modified at any given time. Consequently, they can *put* the data from the modified branch and overwrite the unmodified branch. Here, too, the notion of well-behavedness needs to be weakened.

The TRIP2 system (e.g., Matsuoka et al. (1992)) uses bidirectional transformations specified as collections of Prolog rules as a means of implementing direct-manipulation interfaces for application data structures. The *get* and *put* components of these mappings are written separately by the user.

Languages for Bijective Transformations An active thread of work in the program transformation community concerns *program inversion* and *inverse computation*—see, for example, Abramov and Glück (2000; 2002) and many other papers cited there. Program inversion (Dijkstra, 1979) derives the inverse program from the forward program. Inverse computation (McCarthy, 1956) computes a possible input of a program

from a particular output. One approach to inverse computation is to design languages that produce easily invertible expressions—for example, languages that can only express injective functions, where every program is trivially invertible.

In the database community, Abiteboul, Cluet, and Milo (1997) defined a declarative language of *correspondences* between parts of trees in a data forest. In turn, these correspondence rules can be used to translate one tree format into another through non-deterministic Prolog-like computation. This process assumes an isomorphism between the two data formats. The same authors (Abiteboul et al., 1998) later defined a system for bidirectional transformations based around the concept of *structuring schemas* (parse grammars annotated with semantic information). Thus their *get* functions involved parsing, whereas their *put*s consisted of unparsing. Again, to avoid ambiguous abstract updates, they restricted themselves to *lossless* grammars that define an isomorphism between concrete and abstract views.

Ohori and Tajima (1994) developed a statically-typed polymorphic record calculus for defining views on object-oriented databases. They specifically restricted which fields of a view are updatable, allowing only those with a ground (simple) type to be updated, whereas our lenses can accommodate structural updates as well.

A related idea from the functional programming community, called *views* (Wadler, 1987), extends algebraic pattern matching to abstract data types using programmer-supplied *in* and *out* operators.

Update Translation for Tree Views There have been many proposals for query languages for trees (e.g., XQuery and its forerunners, UnQL, StruQL, and Lorel), but these either do not consider the view update problem at all or else handle update only in situations where the abstract and concrete views are isomorphic.

For example, Braganholo, Heuser, and Vittori (2001), and Braganholo, Davidson, and Heuser (2003) studied the problem of updating relational databases "presented as XML." Their solution requires a 1:1 mapping between XML view elements and objects in the database, to make updates unambiguous. Tatarinov, Ives, Halevy, and Weld (2001) described a mechanism for translating updates on XML structures that are stored in an underlying relational database. In this setting there is again an isomorphism between the concrete relational database and the abstract XML view, so updates are unambiguous—rather, the problem is choosing the most efficient way of translating a given XML update into a sequence of relational operations.

The view update problem has also been studied in the context of object-oriented databases. School, Laasch, and Tresch (1991) restrict the notion of views to queries that preserve object identity. The view update problem is greatly simplified in this setting, as the objects contained in the view are the objects of the database, and an update on the view is directly an update on objects of the database.

Update Translation for Relational Views Research on view update translation in the database literature has tended to focus on taking an existing language for defining *get* functions (e.g., relational algebra) and then considering how to infer corresponding *put* functions, either automatically or with some user assistance. By contrast, we have designed a new language in which the definitions of *get* and *put* go hand-in-hand. Our approach also goes beyond classical work in the relational setting by directly transforming and updating tree-structured data, rather than flat relations. (Of course, trees can be encoded as relations, but it is not clear how our tree-manipulation primitives could be expressed using the recursion-free relational languages considered in previous work in this area.)

Recent work by Bohannon, Pierce, and Vaughan (2006) extends the framework presented here to obtain lenses that operate natively on relational data. Their lenses are

based on the primitives of classical relational algebra, with additional annotations that specify the desired "update policy" in the *put* direction. They develop a type system, using record predicates and functional dependencies, to aid compositional reasoning about well-behavedness. The chapter on view update in Date's textbook (2003) articulates a similar perspective on translating relational updates.

Masunaga (1984) described an automated algorithm for translating updates on views defined by relational algebra. The core idea was to annotate where the "semantic ambiguities" arise, indicating they must be resolved either with knowledge of underlying database semantic constraints or by interactions with the user.

Keller (1985) catalogued all possible strategies for handling updates to a select-project-join view and showed that these are exactly the set of translations that satisfy a small set of intuitive criteria. Building on this foundation, Barsalou, Siambela, Keller, and Wiederhold (1991) described a scheme for interactively constructing update translators for object-based views of relational databases.

Medeiros and Tompa (1985) presented a design tool for exploring the effects of choosing a view update policy. This tool shows the update translation for update requests supplied by the user; by considering all possible valid concrete states, the tool predicts whether the desired update would in fact be reflected back into the view after applying the translated update to the concrete database.

Atzeni and Torlone (1997; 1996) described a tool for translating views and observed that if one can translate any concrete view to and from a *meta-model* (shared abstract view), one then gets bidirectional transformations between any pair of concrete views. They limited themselves to mappings where the concrete and abstract views are isomorphic.

Complexity bounds have also been studied for various versions of the view update inference problem. In one of the earliest, Cosmadakis (1983) and Cosmadakis and Papadimitriou (1984) considered the view update problem for a single relation, where the view is a projection of the underlying relation, and showed that there are polynomial time algorithms for determining whether insertions, deletions, and tuple replacements to a projection view are translatable into concrete updates. More recently, Buneman, Khanna, and Tan (2002) established a variety of intractability results for the problem of inferring "minimal" view updates in the relational setting for query languages that include both join and either project or union.

3.9 Conclusions

We have worked to design a collection of combinators that fit together in a sensible way and that are easy to program with and reason about. Starting with lens laws that define "reasonable behavior," adding type annotations, and proving that each of our lenses is total, has imposed strong constraints on our design of new lenses—constraints that, paradoxically, make the design process easier. In the early stages of the Harmony project, working in an under-constrained design space, we found it extremely difficult to converge on a useful set of primitive lenses. Later, when we understood how to impose the framework of type declarations and the demand for compositional reasoning, we experienced a huge increase in manageability. The types helped not just in finding programming errors in derived lenses, but in exposing design mistakes in the primitives at an early stage.

Our interest in bidirectional tree transformations arose in the context of the Harmony data synchronization framework. Besides the bookmark synchronizer described in Foster et al. (2007b), we have developed prototype synchronizers for calendars, address books, and structured text. Building implementations continues to provide valuable stress-testing for both our combinators and their formal foundations. It also gives us confidence

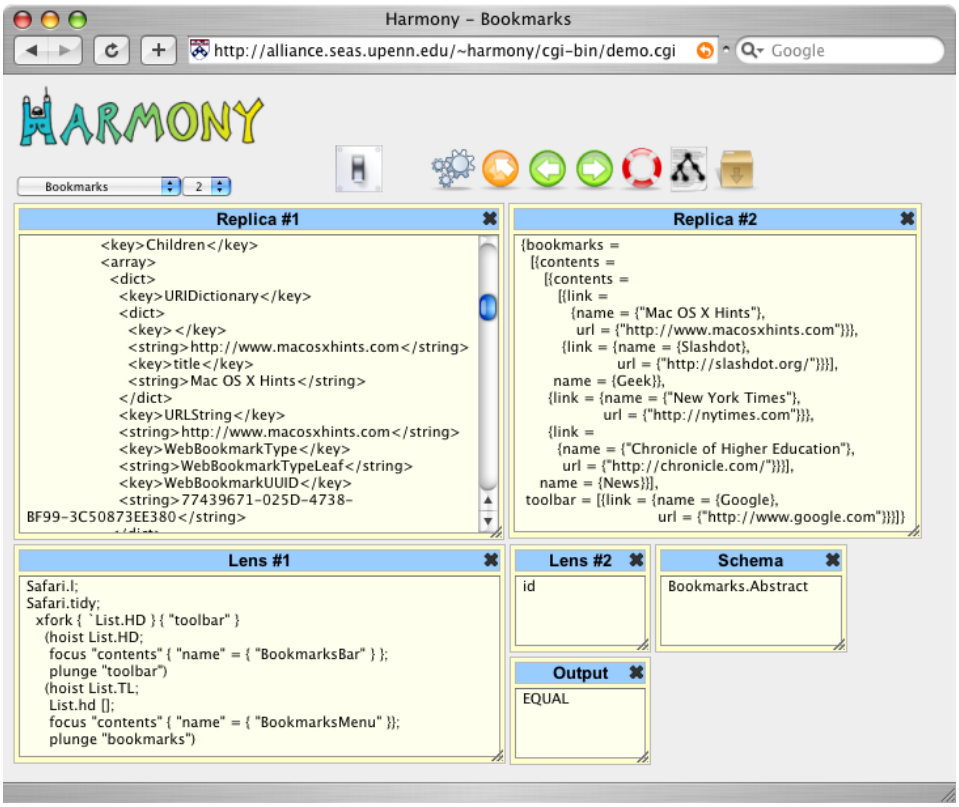


Figure 3.2: Web demo of Safari Bookmark lens

that our lenses are practically useful.

The source code for each of these prototypes, along with our lens compiler and synchronization engine, can be found on the Harmony web page.[9] We have also made the system available as an online web demo (a screenshot from the Safari component of our bookmarks portion of this demo is shown in Figure 3.2).

Naturally, the progress we have made on lens combinators raises a host of further challenges. The most urgent of these is automated typechecking. At present, it is the lens programmers' responsibility to check the well-behavedness of the lenses that they write. Our compiler has the ability to perform simple run-time checking and some debugging using probe points and to track stack frames. These simple dynamic techniques have proven helpful in developing and debugging small-to-medium sized lens programs, but we would like to be able to reason statically that a given program is type correct. Fortunately, the types of the primitive combinators have been designed so that these checks are both local and essentially mechanical. The obvious next step is to reformulate the type declarations as a type *algebra* and find a mechanical procedure for statically checking (or, more ambitiously, inferring) types.

In the semantic framework of lens types we have developed, the key properties tracked by the types are well-behavedness and totality. However, there are other properties of lenses that one might want to track in a type system including very well behavedness, obliviousness, adherence to the conventions about Ω, etc. Moreover, there is a natural subsumption relation between these different lens types: e.g., every oblivious lens is very well behaved. Once basic mechanized type checking for lenses is in place, the natural

[9] `https://alliance.seas.upenn.edu/~harmony/old/index.html`

next step is to stratify the type system to facilitate reasoning about these more refined properties of lenses.

A number of other interesting questions are related to static analysis of lenses. For instance, can we characterize the complexity of programs built from these combinators? Is there an algebraic theory of lens combinators that would underpin optimization of lens expressions in the same way that the relational algebra and its algebraic theory are used to optimize relational database queries? (For example, the combinators we have described here have the property that `map` l_1; `map` l_2 = `map` $(l_1; l_2)$ for all l_1 and l_2, but the latter should run substantially faster.)

This algebraic theory will play a crucial role in any serious effort to compile lens programs efficiently. Our current prototype performs a straightforward translation from a concrete syntax similar to the one used in this chapter to a combinator library written in OCaml. This is fast enough for experimenting with lens programming and for small demos (our calendar lenses can process a few thousands of appointments in under a minute), but we would like to apply the Harmony system to applications such as synchronization of biological databases that will require much higher throughput.

Another area for further investigation is the design of additional combinators. While we have found the ones we have described here to be expressive enough to code a large number of examples—both intricate structural manipulations such as the list transformations in Section 3.7 and more prosaic application transformations such as the ones needed by the bookmark synchronizer—there are some areas where we would like more general forms of the lenses we have (e.g., a more flexible form of `xfork`, where the splitting and recombining of trees is not based on top-level names, but involves deeper structure), lenses expressing more global transformations on trees (including analogs of database operations such as `join`), or lenses addressing completely different sorts of transformations (e.g., none of our combinators do any significant processing on edge labels, which might include string processing, arithmetic, etc.). Higher-level combinators embodying more global transformations on trees—perhaps modeled on a familiar tree transformation notation such as XSLT—are another interesting possibility. There is also the question of bidirectional transformations for ordered data, which we address in Chapter 4

Finally, we would also like to investigate recursion combinators that are less powerful than *fix*, but that come equipped with simpler principles for reasoning about totality. We already have one such combinator: `map` iterates over the *width* of the tree. However, we think it should be possible to go further; e.g., one could define lenses by structural recursion on trees.

More generally, what are the limits of bidirectional programming? How expressive are the combinators we have defined here? Do they cover any known or succinctly characterizable classes of computations (in the sense that the set of *get* parts of the total lenses built from these combinators coincide with this class)? We have put considerable energy into these questions, but at the moment we can only report that they are challenging! One reason for this is that questions about expressiveness tend to have trivial answers when phrased semantically. For example, it is not hard to show that *any* surjective *get* function can be equipped with a *put* function—indeed, typically many—to form a total lens. Indeed, if the concrete domain C is recursively enumerable, then this *put* function is even computable. The real problems are thus syntactic—how to conveniently pick out a *put* function that does what is wanted for a given situation.

In restricted cases, it may be possible to build lenses in simpler ways than by explicit programming—e.g., by generating them automatically from schemas for concrete and abstract views, or by inference from a set of pairs of inputs and desired outputs ("programming by example"). Such a facility might be used to do part of the work for a programmer wanting to add synchronization support for a new application (where the abstract schema is already known, for example), leaving just a few spots to fill in.

Part II

Manipulating Ordered Data

"The art of progress is to preserve order amid change and to preserve change amid order."

—A N Whitehead

We now turn to the manipulation of *ordered* data structures. We first continue our study of bidirectional transformations in Chapter 4, considering structured text as our data model. This work was done with Aaron Bohannon, Nate Foster, Benjamin Pierce, and Alexandre Pilkiewicz (Bohannon et al., 2008).

In the next two chapters, we enrich the object of study by addressing the manipulation of XML, a very common form of ordered data. Such manipulations are usually done in one of two ways: using a map or giving directions. One may specify the context (the map) in which the data of interest occurs, and capture this data, in a way similar to the *pattern matching* paradigm of ML. Alternatively, one may ignore the context and simply give directions, or *path*, to reach the data.

We describe in Chapter 5 an extension of the C# programming languages with primitives that provide statically checked pattern matching against XML data. In this chapter, we concentrate mostly on the design of the language and the major pitfalls we encountered. This work was done with Vladimir Gapeyev, Michael Levin, and Benjamin Pierce (Gapeyev et al., 2005b,c).

Finally, we develop in Chapter 6 an algorithm to check the satisfiability of an XPath request when run against some XML schema. We prove that this algorithm is correct, complete, and has a good theoretical and practical complexity. This last chapter is joint work with Pierre Genevès and Nabil Layaïda (Genevès et al., 2007).

Chapter 4

Lenses for Text

4.1 Introduction

In this chapter we address the special challenges that arise when *ordered* data is manipulated in bidirectional languages. Our goals are both foundational and pragmatic. Foundationally, we explore the correct treatment of ordered data, embodied abstractly as a new semantic law stipulating that the *put* function must align pieces of the concrete and abstract structures in a reasonable way, even when the update involves a reordering. Pragmatically, we investigate lenses on ordered data by developing a new language based around notions of chunks, keys, and dictionaries. To ground our investigation, we work within the tractable arena of string transformations. Strings already expose many fundamental issues concerning ordering, allowing us to grapple with these issues without the complications of a richer data model.

While primary focus is on exposing fundamental issues, we have also tried to design our experimental language, Boomerang, to be useful in practice. There is a lot of string data in the world—textual database formats (iCalendar, vCard, BibTeX, CSV), structured documents (LaTeX, Wiki, Markdown, Textile), scientific data (SwissProt, Genebank, FASTA), and simple XML formats (RSS, AJAX data) and microformats (JSON, YAML) whose schemas are non-recursive—and it is often convenient to manipulate this data directly, without first mapping it to more structured representations. Since most programmers are already familiar with regular languages, we hope that a bidirectional language for strings built around regular operations (i.e., finite state transducers) will have broad appeal.

Our contributions can be summarized as follows.

1. We develop a set of *string lens combinators* with intuitive semantics and typing rules that ensure the lens laws, all based on familiar regular operators (union, concatenation, and Kleene-star).

2. We address a serious issue in bidirectional manipulation of ordered data—the need for lenses to be able to match up chunks of data in the concrete and abstract structures by key rather than by position, which we call *resourcefulness*—by adding two more combinators and interpreting all the combinators in an enriched semantic space of *dictionary lenses*.

3. We formalize a condition called *quasi-obliviousness* and use it to study properties of dictionary lenses. Some previously studied properties of basic lenses also have neat characterizations using this condition.

4. We sketch the design and implementation of *Boomerang*, a full-blown *bidirectional programming language* based on dictionary lenses, and describe some programs we

have built for transforming real-world data structures such as SwissProt.

Lenses

The language described in this chapter is an extension of the one in Chapter 3—called *basic lenses* here.[1]

Formally, a basic lens l mapping between a set of inputs C ("concrete structures") and a set of outputs A ("abstract structures") comprises three functions

$$\begin{aligned} l.\,get &\in C \to A \\ l.\,put &\in A \to C \to C \\ l.\,create &\in A \to C \end{aligned}$$

obeying the following laws for every $c \in C$ and $a \in A$:

$$l.\,put\ (l.\,get\ c)\ c = c \qquad \text{(GetPut)}$$

$$l.\,get\ (l.\,put\ a\ c) = a \qquad \text{(PutGet)}$$

$$l.\,get\ (l.\,create\ a) = a \qquad \text{(CreateGet)}$$

The set of basic lenses from C to A is written $C \Longleftrightarrow A$.

String Lenses

To give a first taste of the use of lenses with strings, let us consider a simple example where the concrete structures are newline-separated records, each with three comma-separated fields representing the name, dates, and nationality of a classical composer

```
"Jean Sibelius, 1865-1957, Finnish
 Aaron Copland, 1910-1990, American
 Benjamin Britten, 1913-1976, English"
```

and the abstract structures include just names and nationalities:

```
"Jean Sibelius, Finnish
 Aaron Copland, American
 Benjamin Britten, English"
```

Here is a string lens that implements this transformation:

```
let ALPHA = [A-Za-z ]+
let YEARS = [0-9]{4} . "-" . [0-9]{4}
let comp = copy ALPHA . copy ", "
         . del YEARS . del ", "
         . copy ALPHA

let comps = copy "" | comp . (copy "\n" . comp)*
```

[1] The minor differences are as follow. First we handle situations where an element of C must be created from an element of A using a *create* function instead of enriching C with the special element Ω and using *put*. Second, as we are not considering lenses defined by recursion, we take the components of lenses to be total functions rather than defining lenses with *partial* components and establishing totality later. Finally, we take the behavioral laws as part of the fundamental definition of basic lenses, rather than first defining bare structures of appropriate type and then adding the laws—i.e., basic lenses correspond to *well-behaved, total lenses*.

The first two lines define ordinary regular expressions for alphabetical data and year ranges. We use standard POSIX notation for character sets (`[A-Za-z ]` and `[0-9]`) and repetition (`+` and `{4}`).

The lens that processes a single composer is `comp`; lists of composers are processed by `comps`. In the *get* direction, these lenses can be read as ordinary string transducers, written in regular expression style: `copy ALPHA` matches `ALPHA` in the concrete structure and copies it to the abstract structure, and `copy ", "` matches and copies a literal comma-space, while `del YEARS` matches `YEARS` in the concrete structure but adds nothing to the abstract structure. The union (`|`), concatenation (`.`), and iteration (`*`) operators work as expected. Thus, the *get* component of `comp` matches an entry for a single composer, consisting of a substring matching the regular expression `ALPHA`, followed by a comma and a space (all of which are copied to the output), followed by a string matching `YEARS` and another comma and space (which are not copied) and a final `ALPHA`. The *get* of `comps` matches either a completely empty concrete structure (which it copies to the output) or a newline-separated concatenation of entries, each of which is processed by `comp`.

The *put* component of `comps` restores the dates positionally: the name and nationality from the nth line in the abstract structure are combined with the years from the nth line in the concrete structure, using a default year range to handle cases where the abstract structure has more lines than the concrete one. We will see precisely how all this works in Section 4.2; for now, the important point is that the *put* component of `comps` operates by splitting both of its arguments into lines and invoking the *put* component of `comp` on the first line from the abstract structure together with the first line from the concrete structure, then the second line from the abstract structure together with the second line from the concrete structure, etc. For some updates—e.g., when entries have been edited and perhaps added at the end of the abstract structure but the order of lines has not changed—this policy does a good job. For example, if the update to the abstract structure replaces Britten's nationality with "British" and adds an entry for Tansman, the *put* function combines the new abstract structure

```
"Jean Sibelius, Finnish
 Aaron Copland, American
 Benjamin Britten, British
 Alexandre Tansman, Polish"
```

with the original concrete structure and yields

```
"Jean Sibelius, 1865-1957, Finnish
 Aaron Copland, 1910-1990, American
 Benjamin Britten, 1913-1976, British
 Alexandre Tansman, 0000-0000, Polish"
```

(the default year range `0000-0000` is generated by the `del` lens in `comp` from the regular expression `YEARS`).

Problems with Order

On other examples, however, the behavior of this *put* function is highly unsatisfactory. If the update to the abstract string breaks the positional association between lines in the concrete and abstract strings, the output will be mangled—e.g., when the update to the abstract string is a reordering, combining

```
"Jean Sibelius, Finnish
 Benjamin Britten, English
 Aaron Copland, American"
```

with the original concrete structure yields an output

```
"Jean Sibelius, 1865-1957, Finnish
 Benjamin Britten, 1910-1990, English
 Aaron Copland, 1913-1976, American"
```

where the year data has been taken from the entry for Copland and inserted into into the entry for Britten, and vice versa.

This is a serious problem, and a pervasive one: it is triggered whenever a lens whose *get* function discards information is iterated over an ordered list and the update to the abstract list breaks the association between elements in the concrete and abstract lists. It is a show-stopper for many of the applications we want to write.

What we want is for the *put* to align the entries in the concrete and abstract strings by matching up lines with identical name components. On the inputs above, this *put* function would produce

```
"Jean Sibelius, 1865-1957, Finnish
 Benjamin Britten, 1913-1976, English
 Aaron Copland, 1910-1990, American"
```

but neither basic lenses nor any other existing bidirectional language provides the means to achieve this effect.

Dictionary Lenses

Our solution is to enrich lenses with a simple mechanism for tracking *provenance* (Cui and Widom, 2003; Buneman et al., 2001). The idea is that the programmer should identify *chunks* of the concrete string and a *key* for each chunk. These induce an association between chunks and pieces of the abstract string, and this association can be used by *put* during the translation of updates to find the chunk corresponding to each piece of the abstract, even if the abstract pieces have been reordered. Operationally, we retool all our *put* functions to use this association by parsing the whole concrete string into a dictionary, where each concrete chunk is stored under its key, and then making this dictionary, rather than the string itself, available to the *put* function. We call these enriched structures *dictionary lenses*.

Here is a dictionary lens that gives us the desired behavior for the composers example:

```
let comp = key ALPHA . copy ", "
         . del (YEARS . ", ")
         . copy ALPHA

let comps = "" | <comp> . ("\n" . <comp>)*
```

The first change from the earlier version of this program is that the two occurrences of `comp` in `comps` are marked with angle brackets, indicating that these subexpressions are the reorderable chunks of information. The corresponding substring of the concrete structure at each occurrence (which is passed to the *put* of `comp`) is obtained not positionally but by matching keys. The second change is that the first `copy` at the beginning of `comp` has been replaced by the special primitive `key`. The lens `key ALPHA` has the same copying behavior as `copy ALPHA`, but it additionally specifies that the matched substring is to be used as the key of the chunk in which it appears—i.e., in this case, that the key of each composer's entry is their name. This choice means that we can both reorder the entries in the abstract structure and edit their nationalities, since the correspondence between chunks in the concrete and abstract structures is based just on names. We do not actually demand that the key of each chunk be unique—i.e., these "keys" are not required to be keys in the strict database sense. If several pieces of the abstract structure have the same key, they are matched by position.

Quasi-Obliviousness

For the composers example, the behavior of the new dictionary lens is clearly preferable to that of the original basic lens: its *put* function has the effect of translating a reordering on the abstract string into a corresponding reordering on the concrete string, whereas the *put* function of the original lens works by position and produces a mangled result. We would like a characterization of this difference—i.e., a way of expressing the intuition that the second lens does something good, while the first does not.

To this end, we define a semantic space of lenses called *quasi-oblivious lenses*. Let l be a lens in $C \Longleftrightarrow A$ and let $\sim$ be an equivalence relation on C. We say that l is a quasi-oblivious lens with respect to $\sim$ if its *put* function ignores differences between equivalent concrete arguments.

We are primarily interested in lenses that are quasi-oblivious with respect to an equivalence relating concrete strings up to reorderings of chunks. It should be clear that a dictionary lens that operates on dictionaries in which the relative order of concrete lines is forgotten will be quasi-oblivious with respect to such an equivalence, while the analogous basic lens, which operates on lines positionally, is not. Using the above condition on *put*, we can derive intuitive properties for many such quasi-oblivious lenses—e.g., for the dictionary lens for composers above, we can show that updates to the abstract list consisting of reorderings are translated by the *put* as corresponding reorderings on the concrete list.

Lenses that are quasi-oblivious with respect to equivalences other than reordering are also interesting. Indeed, we can characterize some important special cases of basic lenses (*oblivious* and *very well behaved* lenses) in terms of quasi-obliviousness.

Boomerang

Our theoretical development focuses on a small set of basic combinators. Of course, writing large programs entirely in terms of such low-level primitives would be tedious; we don't do this. Instead, we have implemented a full-blown programming language, called Boomerang, in which the combinators are embedded in a functional language, *à la* Algol-60. That is, a Boomerang program is a functional program over the base type "lens"; to apply it to string data, we first evaluate the functional program to produce a lens, and then apply this lens to the strings. This functional infrastructure can be used to abstract out common patterns as generic bidirectional libraries (e.g., for processing XML structures) that make higher-level programming quite convenient.

Boomerang also comes equipped with a type checker that infers lens types and checks the conditions needed to ensure that a dictionary lens satisfies the lens laws. The domain and codomain types for dictionary lenses are regular languages, so the analysis performed by this checker is very precise—a huge aid in writing correct lens programs.

Using Boomerang, we have developed several large lenses for processing a variety of data including vCard, CSV, and XML address books, BibTeX and RIS bibliographic databases, LaTeX documents, iTunes libraries, and databases of protein sequences represented in the ASCII SwissProt format and XML.

Outline

Section 4.2 introduces notation and formally defines the basic string lenses used in the first example above. Syntax, semantics, and typing rules for dictionary lenses are given in Section 4.3. Section 4.4 defines the refined semantic space of quasi-oblivious lenses. Sections 4.5 and 4.6 describe Boomerang and our experiences building lenses for real-world data formats. Section 4.7 discusses related work. Proofs may be found in an accompanying technical report (Bohannon et al., 2007).

4.2 Basic String Lenses

Before presenting dictionary lenses, let us warm up by formalizing the language for basic lenses from the first example in the introduction. Let Σ be a fixed alphabet (e.g., ASCII). A language is a subset of Σ^*. Metavariables u, v, w range over strings in Σ^*, and ϵ denotes the empty string. The concatenation of two strings u and v is written $u{\cdot}v$; concatenation is lifted to languages L_1 and L_2 in the obvious way: $L_1{\cdot}L_2 = \{u{\cdot}v \mid u \in L_1 \text{ and } v \in L_2\}$. We write L^* to denote the iteration of L: i.e., $L^* = \bigcup_{n=0}^{\infty} L^n$ where L^n denotes the n-fold concatenation of L with itself

$$L^0 = \{\epsilon\} \text{ and } L^1 = L, L^2 = L{\cdot}L, \ldots$$

The typing rules for some lenses require that for every string belonging to the concatenation of two languages, there be a unique way of splitting that string into two substrings belonging to the concatenated languages. Two languages L_1 and L_2 are unambiguously concatenable, written $L_1{\cdot}^! L_2$, if for every u_1, v_1 in L_1 and u_2, v_2 in L_2 if $u_1{\cdot}u_2 = v_1{\cdot}v_2$ then $u_1 = v_1$ and $u_2 = v_2$. Similarly, a language L is unambiguously iterable, written $L^{!*}$, if for every $u_1, \ldots, u_m, v_1, \ldots, v_n, \in L$, if $u_1{\cdot}\cdots{\cdot}u_m = v_1{\cdot}\cdots{\cdot}v_n$ then $m = n$ and $u_i = v_i$ for every i from 1 to n.

Regular expressions are generated by the grammar

$$\mathcal{R} \ ::= \ u \ \mid \ \mathcal{R}{\cdot}\mathcal{R} \ \mid \ \mathcal{R}|\mathcal{R} \ \mid \ \mathcal{R}^*$$

where u ranges over arbitrary strings (including ϵ). The notation $[\![E]\!]$ denotes the (non-empty) language described by $E \in \mathcal{R}$. The function $\mathit{choose}(E)$ picks an arbitrary element from $[\![E]\!]$.

With that notation in hand, we now define five combinators for building basic string lenses over regular languages. Recall that we write $l \in C \Longleftrightarrow A$ when l is a basic lens mapping between strings in C and A. Each basic lens expects to be applied to arguments in its domain/codomain—it is nonsensical to do otherwise. In our implementation, we perform a membership test on every string before supplying it to a lens. (We do this check just once, at the top level: internally, the typing rules guarantee that every sublens is provided with well-typed inputs.)

The simplest primitive, *copy* E, copies every string belonging to (the language denoted by) E from the concrete structure to the abstract structure, and conversely in the *put* direction. The components of *copy* are precisely defined in the box below. The second primitive lens, *const* E u v maps every string belonging to E to a constant string u. Its *put* function restores its concrete argument. It also takes as an argument a default v belonging to E, which is used by *create* when no concrete argument is available. Note that *const* satisfies PutGet because its codomain is a singleton set.

The inference rules should be read as the statements of lemmas that each combinator is a basic lens at the given type.

$$\frac{E \in \mathcal{R}}{\mathit{copy}\ E \in [\![E]\!] \Longleftrightarrow [\![E]\!]}$$

$$\begin{array}{lcl} \mathit{get}\ c & = & c \\ \mathit{put}\ a\ c & = & a \\ \mathit{create}\ a & = & a \end{array}$$

$$\frac{E \in \mathcal{R} \qquad u \in \Sigma^* \qquad v \in [\![E]\!]}{\mathit{const}\ E\ u\ v \in [\![E]\!] \Longleftrightarrow \{u\}}$$

$$\begin{array}{lcl} \mathit{get}\ c & = & u \\ \mathit{put}\ a\ c & = & c \\ \mathit{create}\ a & = & v \end{array}$$

Several lenses can be expressed as syntactic sugar using *const*:

$$\begin{array}{lcl} E \leftrightarrow u & \in & [\![E]\!] \Longleftrightarrow \{u\} \\ E \leftrightarrow u & = & \mathit{const}\ E\ u\ (\mathit{choose}(E)) \end{array}$$

$$\begin{array}{rcl} \mathit{del}\ E & \in & [\![E]\!] \Longleftrightarrow \{\epsilon\} \\ \mathit{del}\ E & = & E \leftrightarrow \epsilon \\ \mathit{ins}\ u & \in & \{\epsilon\} \Longleftrightarrow \{u\} \\ \mathit{ins}\ u & = & \epsilon \leftrightarrow u \end{array}$$

They behave as follows: $E \leftrightarrow u$ is like *const*, but uses an arbitrary element of E for *create*; the *get* function of *del* E deletes a concrete string belonging to E and restores the deleted string in the *put* direction; *ins* u inserts a fixed string u in the *get* direction and deletes u in the opposite direction.

The next three combinators build bigger lenses from smaller ones using regular operators. Concatenation is simplest:

$$\frac{C_1 \cdot^! C_2 \qquad A_1 \cdot^! A_2 \qquad l_1 \in C_1 \Longleftrightarrow A_1 \qquad l_2 \in C_2 \Longleftrightarrow A_2}{l_1 \cdot l_2 \in C_1 \cdot C_2 \Longleftrightarrow A_1 \cdot A_2}$$

$$\begin{array}{lcl} \mathit{get}\ (c_1 \cdot c_2) & = & (l_1.\, \mathit{get}\ c_1) \cdot (l_2.\, \mathit{get}\ c_2) \\ \mathit{put}\ (a_1 \cdot a_2)\ (c_1 \cdot c_2) & = & (l_1.\, \mathit{put}\ a_1\ c_1) \cdot (l_2.\, \mathit{put}\ a_2\ c_2) \\ \mathit{create}\ (a_1 \cdot a_2) & = & (l_1.\, \mathit{create}\ a_1) \cdot (l_2.\, \mathit{create}\ a_2) \end{array}$$

The notation $c_1 \cdot c_2$ used in the definition of concatenation assumes that c_1 and c_2 are members of C_1 and C_2 respectively; we use this convention silently from now on.

The typing rule for concatenation requires that the concrete domains and the abstract codomains each be unambiguously concatenable. This condition is essential for ensuring that the components of the lens are well-defined functions and that the whole lens is well behaved. As an example of what would go wrong without these conditions, consider the (ill-typed) lens l_{ambig}, defined as $(\mathtt{a} \leftrightarrow \mathtt{a} \mid \mathtt{aa} \leftrightarrow \mathtt{aa}) \cdot (\mathtt{a} \leftrightarrow \mathtt{b} \mid \mathtt{aa} \leftrightarrow \mathtt{b})$ (we assume "$\leftrightarrow$" binds tighter than "$|$", which is defined formally below). The *get* component is not well defined since, according to the above specification, $l_{ambig}.\, \mathit{get}\ \mathtt{aaa} = \mathtt{ab}$ if we split $\mathtt{aaa}$ into $\mathtt{a} \cdot \mathtt{aa}$ and $l_{ambig}.\, \mathit{get}\ \mathtt{aaa} = \mathtt{aab}$ if we split it into $\mathtt{aa} \cdot \mathtt{a}$. This issue could be side-stepped using a fixed policy for choosing among multiple parses (e.g., with a shortest match policy, $l_{ambig}.\, \mathit{get}\ \mathtt{aaa} = \mathtt{ab}$). However, doing so would not always give us a lens that satisfies the lens laws; intuitively, just because one direction uses a given match policy does not mean that the string it produces will split the same way using the same policy in the other direction. Consider l_{bogus} defined as $k \cdot k$ where k is $(\mathtt{a} \leftrightarrow \mathtt{bb} \mid \mathtt{aa} \leftrightarrow \mathtt{a} \mid \mathtt{b} \leftrightarrow \mathtt{b} \mid \mathtt{ba} \leftrightarrow \mathtt{ba})$. Then using the shortest match policy we have $l_{bogus}.\, \mathit{get}\ \mathtt{aaa}$ equals $(k.\, \mathit{get}\ \mathtt{a}) \cdot (k.\, \mathit{get}\ \mathtt{aa})$, which is $\mathtt{bba}$, but $l_{bogus}.\, \mathit{put}\ \mathtt{bba}\ \mathtt{aaa}$ equals $(k.\, \mathit{put}\ \mathtt{b}\ \mathtt{a}) \cdot (k.\, \mathit{put}\ \mathtt{ba}\ \mathtt{aa})$, which is $\mathtt{bba}$. That is, the GetPut law fails. For these reasons, we require that each pair of C_1 and C_2 and A_1 and A_2 be unambiguously concatenable.

The Kleene-star combinator is similar:

$$\frac{l \in C \Longleftrightarrow A \qquad C^{!*} \qquad A^{!*}}{l^* \in C^* \Longleftrightarrow A^*}$$

$$\begin{array}{lcl} \mathit{get}\ (c_1 \cdots c_n) & = & (l.\, \mathit{get}\ c_1) \cdots (l.\, \mathit{get}\ c_n) \\ \mathit{put}\ (a_1 \cdots a_n)\ (c_1 \cdots c_m) & = & c'_1 \cdots c'_n \\ \quad \text{where } c'_i = \begin{cases} l.\, \mathit{put}\ a_i\ c_i & i \in \{1, \ldots, \min(m,n)\} \\ l.\, \mathit{create}\ a_i & i \in \{m+1, \ldots, n\} \end{cases} & & \\ \mathit{create}\ (a_1 \cdots a_n) & = & (l.\, \mathit{create}\ a_1) \cdots (l.\, \mathit{create}\ a_n) \end{array}$$

Note that the *put* component of l^* calls the *put* of l on elements of A and C having the same index in their respective lists. This is the root of the undesirable behavior in the

example in the introduction.[2] Also note that it must handle cases where the number of As is not equal to the number of Cs. Since the number of As produced by the *get* of l^* equals the number of Cs in its argument, the result of the *put* function must have exactly as many Cs as there are As in its abstract string—otherwise, PUTGET would not hold. When there are more Cs than As, the lens simply ignores the extra Cs. When there are more As, it must put them back into the concrete domain, but it has no concrete argument to use. It uses $l.\,create$ to process these extra pieces.

The final combinator forms the union of two lenses:

$$\frac{C_1 \cap C_2 = \emptyset \qquad l_1 \in C_1 \Longleftrightarrow A_1 \qquad l_2 \in C_2 \Longleftrightarrow A_2}{l_1 \mid l_2 \in C_1 \cup C_2 \Longleftrightarrow A_1 \cup A_2}$$

$$get\ c = \begin{cases} l_1.\,get\ c & \text{if } c \in C_1 \\ l_2.\,get\ c & \text{if } c \in C_2 \end{cases}$$

$$put\ a\ c = \begin{cases} l_1.\,put\ a\ c & \text{if } c \in C_1 \wedge a \in A_1 \\ l_2.\,put\ a\ c & \text{if } c \in C_2 \wedge a \in A_2 \\ l_1.\,create\ a & \text{if } c \in C_2 \wedge a \in A_1 \setminus A_2 \\ l_2.\,create\ a & \text{if } c \in C_1 \wedge a \in A_2 \setminus A_1 \end{cases}$$

$$create\ a = \begin{cases} l_1.\,create\ a & \text{if } a \in A_1 \\ l_2.\,create\ a & \text{if } a \in A_2 \setminus A_1 \end{cases}$$

The typing rule forces C_1 and C_2 to be disjoint. Like the ambiguity condition in the rule for concatenation, this condition is essential to ensure that the lens is well defined. The abstract codomains, however, may overlap. In the *put* direction, when the abstract argument belongs to $A_1 \cap A_2$, the union lens uses the concrete argument to select a branch. In the *create* function, since no concrete argument is available, it just uses l_1. (This choice is arbitrary, but is not a limitation: to use l_2 by default, the programmer writes $l_2 \mid l_1$. It does mean, though, that union is not commutative.)

In some situations, the *put* function is invoked with an original concrete view c belonging to the concrete domain of the lens on one side of the union (say l_1) and an updated abstract view a belonging to the abstract codomain of the opposite lens (l_2). Although c is not an element of C_2, it may still have information that can be represented in C_2. If such information was thrown away when computing the abstract view, then we would like for the *put* function to reintegrate it, in some manner, with a. In Chapter 3, we used a notion of "fixup" functions—from C_2 to C_1 and vice versa—for extracting the common information and mapping it into a concrete argument of appropriate type. Semantically fixup functions are exactly what is needed—one can show that the conditional lens formulated using them is most general. But syntactically they are very unattractive, because the programmer has to write down two additional functions on the concrete domains! We believe that dictionary lenses may offer a better alternative because, in their basic form, they have the ability to transfer information from one side of a union to another via the dictionary without requiring explicit fixup functions (see Section 4.3). Thus, we refrain from introducing fixup functions here and make the simple, but arguably sub-optimal, choice of discarding the concrete argument in these cases.

[2] We cannot, however, repair the problem just by fixing Kleene-star; the same issues come up with concatenation.

4.3 Dictionary Lenses

Now that we've seen basic string lenses, let us define lenses that deal more flexibly with ordering. We will accomplish this by adding two new primitives, *match* (written with angle brackets in the concrete syntax of Boomerang) and *key*, and interpreting these new primitives and the primitives defined in the previous section in a refined space called *dictionary lenses*.

The main difference between basic and dictionary lenses is that their *put* components operate on different types of structures—strings and dictionaries respectively. Dictionary lenses also include two new components: *parse*, which is used to build a dictionary out of a concrete string, and *key*, which is used to compute the key of data that goes into a dictionary.

In a dictionary lens, the work of the basic lens *put* function is divided into two phases. In the first, the concrete string is given to *parse*, which splits it into two parts: a collection of concrete chunks organized as a dictionary and a *skeleton* representing the parts of the string outside of the chunks. In the second, the *put* function uses the abstract string, the skeleton, and the dictionary to build an updated concrete string (and a new dictionary). These phases occur in strict sequence: given a dictionary lens l, an abstract string a, and a concrete string c, we first *parse* c, which yields a skeleton s and dictionary d; these are then passed, together with a, to l's *put* function, which walks recursively over the structure of s and a, threading d through l's sublenses and pulling chunks out as needed.

To streamline the exposition, we start with the definition of dictionary lenses, which relies on several notions we have not seen yet—skeleton sets S, the set of keys K, dictionary type specifications L, dictionary types $D(L)$, and an infix operation $+\!\!+$ that appends dictionary values. These are defined below.

A dictionary lens from C to A with skeleton type S and dictionary type specification L has components

$$\begin{aligned} l.\,get &\in C \to A \\ l.\,parse &\in C \to S \times D(L) \\ l.\,key &\in A \to K \\ l.\,create &\in A \to D(L) \to C \times D(L) \\ l.\,put &\in A \to S \times D(L) \to C \times D(L) \end{aligned}$$

obeying the following behavioral laws:[3]

$$\frac{s, d' = l.\,parse\ c \qquad d \in D(L)}{l.\,put\ (l.\,get\ c)\ (s, (d' +\!\!+ d)) = c, d} \tag{GetPut}$$

$$\frac{c, d' = l.\,put\ a\ (s, d)}{l.\,get\ c = a} \tag{PutGet}$$

$$\frac{c, d' = l.\,create\ a\ d}{l.\,get\ c = a} \tag{CreateGet}$$

We write $C \overset{S,L}{\Longleftrightarrow} A$ for the set of dictionary lenses with this type.

Both *create* and *put* consume dictionaries. We thread dictionaries through calls to these functions in a functional style that simulates a global, mutable dictionary, and remove entries as they are used, so that the next lookup of the same key finds the (positionally) next chunk from the concrete string. The *put* function takes a skeleton argument, whereas the *create* function does not. The skeleton, when available,

[3] In law GetPut, the extra dictionary d shows that all and only the chunks originating from c are used by the *put* function.

represents the original concrete string with the chunks removed and provides enough information to reconstruct the original concrete string in cases where GetPut requires it.

To see how the components of a dictionary lens fit together, let us see how to build a basic lens $\bar{l}$ from a dictionary lens l:

$$\begin{array}{lcl} \bar{l}.\,get\ c & = & l.\,get\ c \\ \bar{l}.\,put\ a\ c & = & \pi_1(l.\,put\ a\ (l.\,parse\ c)) \\ \bar{l}.\,create\ a & = & \pi_1(l.\,create\ a\ \{\}) \end{array}$$

This definition embodies the strict phase separation between *parse* and *put* discussed above. It is easy to show that the dictionary lens laws guarantee the original laws for basic lenses built this way.

Theorem 4.3.1. *If* $l \in C \overset{S,L}{\Longleftrightarrow} A$ *then* $\bar{l} \in C \Longleftrightarrow A$.

We now give the definitions deferred previously. We write $h :: t$ for the list with head h and tail t, $\mathsf{List}(X)$ for the set of lists of X and, and $l_1 \mathbin{@} l_2$ for the concatenation of lists l_1 and l_2. We write $X \times Y$ for the set of pairs $\{(x, y) \mid x \in X \text{ and } y \in Y\}$. We take the set of skeletons $\mathcal{S}$ to be the smallest set closed under these operations that includes plain strings and a distinguished atom $\square$, which is used to mark the locations of chunks in skeletons. Formally, $\mathcal{S} = \bigcup_{n=0}^{\infty} S_n$, where $S_0 = \Sigma^* \cup \{\square\}$ and $S_{i+1} = S_i \cup (S_i \times S_i) \cup \mathsf{List}(S_i)$. We define K, the set of keys, to be just Σ^*.

As chunks may be nested within chunks (by nesting the *match* combinator), the type of dictionaries is recursive. A dictionary is a total function from keys to lists of pairs, each consisting of a skeleton and another dictionary. Formally, the set of dictionaries is defined recursively on the structure of a list of sets of skeletons $L \in \mathsf{List}(\mathcal{P}(\mathcal{S}))$ specifying the skeletons that may appear at each level, as follows:

$$\begin{array}{rcl} D([]) & = & K \to \{[]\} \\ D(S :: L) & = & K \to \mathsf{List}(S \times D(L)) \end{array}$$

We write $\{\}$ for the dictionary that maps every k to the empty list. Let d be a dictionary, k a key, and v a skeleton-dictionary pair list of appropriate type. The update of a dictionary, written $d[k \leftarrow v]$, is defined as

$$d[k \leftarrow v](k') = \begin{cases} d(k') & \text{if } k' \neq k \\ v & \text{if } k' = k \end{cases}$$

We write $\{k_1 \mapsto v_1, \ldots, k_n \mapsto v_n\}$ for $\{\}[k_1 \leftarrow v_1] \cdots [k_n \leftarrow v_n]$. The concatenation of two dictionaries d_1 and d_2, written $d_1 \mathbin{+\!\!+} d_2$, is defined using list concatenation as follows: $(d_1 \mathbin{+\!\!+} d_2)(k) = d_1(k) \mathbin{@} d_2(k)$. Dictionaries are accessed using a partial function *lookup* that takes a key k and a dictionary d as arguments. When it finds a matching value, *lookup* returns the value found and the dictionary that remains after deleting that value.

$$\mathit{lookup}(k, d) = \begin{cases} e, d[k \leftarrow l] & \text{if } d(k) = e :: l \\ \text{undefined} & \text{otherwise} \end{cases}$$

We now reinterpret each combinator from the previous section as a dictionary lens and give the definitions of the new combinators *key* and *match*. The *key* combinator is nearly identical to *copy*, except that the *key* component of *copy* is a constant function (returning ϵ), while the *key* component of *key* returns the abstract string.

$$\frac{E \in \mathcal{R} \qquad L \in \mathsf{List}(\mathcal{P}(\mathcal{S}))}{\begin{array}{c} \mathit{copy}\ E{\in}[\![E]\!] \overset{[\![E]\!],L}{\Longleftrightarrow} [\![E]\!] \\ \\ \begin{array}{lcl} \mathit{get}\ c & = & c \\ \mathit{parse}\ c & = & c, \{\} \\ \mathit{key}\ a & = & \epsilon \\ \mathit{create}\ a\ d & = & a, d \\ \mathit{put}\ a\ (s,d) & = & a, d \end{array} \end{array}}$$

$$\frac{E \in \mathcal{R} \qquad L \in \mathsf{List}(\mathcal{P}(\mathcal{S}))}{\begin{array}{c} \mathit{key}\ E{\in}[\![E]\!] \overset{[\![E]\!],L}{\Longleftrightarrow} [\![E]\!] \\ \\ \begin{array}{lcl} \mathit{get}\ c & = & c \\ \mathit{parse}\ c & = & c, \{\} \\ \mathit{key}\ a & = & a \\ \mathit{create}\ a\ d & = & a, d \\ \mathit{put}\ a\ (s,d) & = & a, d \end{array} \end{array}}$$

The refined definition of *const* is also straightforward.

$$\frac{E \in \mathcal{R} \qquad u \in \Sigma^* \qquad v \in [\![E]\!] \qquad L \in \mathsf{List}(\mathcal{P}(\mathcal{S}))}{\begin{array}{c} \mathit{const}\ E\ u\ v{\in}[\![E]\!] \overset{[\![E]\!],L}{\Longleftrightarrow} \{u\} \\ \\ \begin{array}{lcl} \mathit{get}\ c & = & u \\ \mathit{parse}\ c & = & c, \{\} \\ \mathit{key}\ a & = & \epsilon \\ \mathit{create}\ u\ d & = & v, d \\ \mathit{put}\ u\ (s,d) & = & s, d \end{array} \end{array}}$$

Concatenation is similar to string lenses, but *create* and *put* thread the dictionary through the corresponding sublens functions.

$$\frac{\begin{array}{c} l_1{\in}C_1 \overset{S_1,L}{\Longleftrightarrow} A_1 \quad C_1{\cdot}^{!}C_2 \\ l_2{\in}C_2 \overset{S_2,L}{\Longleftrightarrow} A_2 \quad A_1{\cdot}^{!}A_2 \end{array}}{l_1{\cdot}l_2{\in}C_1{\cdot}C_2 \overset{S_1\times S_2,L}{\Longleftrightarrow} A_1{\cdot}A_2}$$

$$\begin{array}{lcl} \mathit{get}\ c_1{\cdot}c_2 & = & (l_1.\,\mathit{get}\ c_1){\cdot}(l_2.\,\mathit{get}\ c_2) \\ \mathit{parse}\ c_1{\cdot}c_2 & = & (s_1, s_2), d_1 \mathbin{+\!\!+} d_2 \\ \quad \text{where } s_i, d_i = l_i.\,\mathit{parse}\ c_i & & \\ \mathit{key}\ a_1{\cdot}a_2 & = & l_1.\,\mathit{key}\ a_1 \cdot l_2.\,\mathit{key}\ a_2 \\ \mathit{create}\ a_1{\cdot}a_2\ d_1 & = & c_1{\cdot}c_2, d_3 \\ \quad \text{where } c_i, d_{i+1} = l_i.\,\mathit{create}\ a_i\ d_i & & \\ \mathit{put}\ a_1{\cdot}a_2\ ((s_1,s_2),d_1) & = & c_1{\cdot}c_2, d_3 \\ \quad \text{where } c_i, d_{i+1} = l_i.\,\mathit{put}\ a_i\ (s_i, d_i) & & \end{array}$$

Lens concatenation is associative, modulo coercion to basic lenses: even though the skeleton structure of a lens differentiates $(l_1{\cdot}l_2){\cdot}l_3$ from $l_1{\cdot}(l_2{\cdot}l_3)$, we have $\overline{(l_1{\cdot}l_2){\cdot}l_3} = \overline{l_1{\cdot}(l_2{\cdot}l_3)}$. We implicitly associate lens concatenation (and the corresponding set-theoretic operations) to the left.

To illustrate the definitions we have seen so far, consider the following dictionary lens:

$$\begin{array}{ll} & l_\$ = \mathit{key}\ \mathtt{x}^* \cdot \mathit{del}\ \mathtt{y}^* \cdot \mathit{copy}\ (\mathtt{z}^*{\cdot}\$) \\ \text{with} & l_\$ \in \mathtt{x}^*{\cdot}\mathtt{y}^*{\cdot}\mathtt{z}^*{\cdot}\$ \overset{S,[]}{\Longleftrightarrow} \mathtt{x}^*{\cdot}\mathtt{z}^*{\cdot}\$ \\ \text{and} & S = \mathtt{x}^* \times \mathtt{y}^* \times \mathtt{z}^*{\cdot}\$. \end{array}$$

The *parse* function of $l_\$$ breaks apart a string according to the structure of the concrete domain:

$$l_\$.\,\mathit{parse}\ \mathtt{xxyz\$} = (\mathtt{xx}, \mathtt{y}, \mathtt{z\$}), \{\} \mathbin{+\!\!+} \{\} \mathbin{+\!\!+} \{\}$$

(The dictionary is empty because none of the sublenses use the *match* operator.) The *key* function returns the `xx...x` prefix of an abstract string. The other components of this lens induce the same functions as in the basic lens semantics.

The iteration combinator is analogous to the concatenation operator. Its *parse* function builds a concatenated dictionary and its *put* and *create* functions thread their dictionary argument (from left to right) through the corresponding sublens functions.

$$\frac{l{\in}C \overset{S,L}{\Longleftrightarrow} A \qquad C^{!*} \qquad A^{!*}}{l^*{\in}C^* \overset{\mathsf{List}(S),L}{\Longleftrightarrow} A^*}$$

$$\begin{array}{ll}
\mathit{get}\ c_1 \cdots c_n & = (l_1.\,\mathit{get}\ c_1) \cdots (l.\,\mathit{get}\ c_n) \\
\mathit{parse}\ c_1 \cdots c_n & = [s_1, \ldots, s_n], d_1 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} d_n \\
\quad \text{where } s_i, d_i = l.\,\mathit{parse}\ c_i & \\
\mathit{key}\ a_1 \cdots a_n & = l.\,\mathit{key}\ a_1 \cdots l.\,\mathit{key}\ a_n \\
\mathit{create}\ a_1 \cdots a_n\ d_1 & = (c_1 \cdots c_n), d_{n+1} \\
\quad \text{where } c_i, d_{i+1} = l.\,\mathit{create}\ a_i\ d_i & \\
\mathit{put}\ a_1 \cdots a_n\ ([s_1, \ldots, s_m], d_1) & = (c_1 \cdots c_n), d_{n+1} \\
\quad \text{where } c_i, d_{i+1} = & \begin{cases} l.\,\mathit{put}\ a_i\ (s_i, d_i) \\ \quad i \in \{1, \ldots, \min(m,n)\} \\ l.\,\mathit{create}\ a_i\ d_i \\ \quad i \in \{m+1, \ldots, n\} \end{cases}
\end{array}$$

The most interesting new combinator is *match*. Its *get* component passes off control to the sublens l. The *put* component matches up its abstract argument with a corresponding item in the dictionary and supplies both to the *put* function of l.

$$\frac{l{\in}C \overset{S,L}{\Longleftrightarrow} A}{\langle l\rangle{\in}C \overset{\{\Box\},S::L}{\Longleftrightarrow} A}$$

$$\begin{array}{lcl}
\mathit{get}\ c & = & l.\,\mathit{get}\ c \\
\mathit{parse}\ c & = & \Box, \{l.\,\mathit{key}\ (l.\,\mathit{get}\ c) \mapsto [l.\,\mathit{parse}\ c]\} \\
\mathit{key}\ a & = & l.\,\mathit{key}\ a \\
\mathit{create}\ a\ d & = & \begin{cases} \pi_1(l.\,\mathit{put}\ a(s_a, d_a)), d' \\ \quad \text{if } (s_a, d_a), d' = \mathit{lookup}(l.\,\mathit{key}\ a, d) \\ \pi_1(l.\,\mathit{create}\ a\ \{\}), d \\ \quad \text{if } \mathit{lookup}(l.\,\mathit{key}\ a, d) \text{ undefined} \end{cases} \\
\mathit{put}\ a\ (\Box, d) & = & \begin{cases} \pi_1(l.\,\mathit{put}\ a\ (s_a, d_a)), d' \\ \quad \text{if } (s_a, d_a), d' = \mathit{lookup}(l.\,\mathit{key}\ a, d) \\ \pi_1(l.\,\mathit{create}\ a\ \{\}), d \\ \quad \text{if } \mathit{lookup}(l.\,\mathit{key}\ a, d) \text{ undefined} \end{cases}
\end{array}$$

To illustrate the operation of *match*, consider the lens $\langle l_{\$}\rangle^*$. It has the same *get* behavior as $l_{\$}{}^*$, but its *put* function restores the `y`s to each chunk using the association induced by keys rather than by position. Let us calculate the result produced by the following application of the derived *put* function:

$$\overline{\langle l_{\$}\rangle^*}.\,\mathit{put}\ \texttt{xxzzz\$x\$ xyz\$xxyyzz\$}$$

Here, the update to the abstract string has swapped the order of the chunks and changed the number of `z`s in each chunk. The *parse* function produces a dictionary structure

that associates (the parse of) each chunk to its key:

$$\begin{aligned}&\langle l_\$\rangle^*.\ \mathit{parse}\ \texttt{xyz\$xxyyzz\$}\\ &\quad = [\square,\square],\left\{\begin{array}{l}\texttt{x}\mapsto[((\texttt{x},\texttt{y},\texttt{z\$}),\{\})],\\ \texttt{xx}\mapsto[((\texttt{xx},\texttt{yy},\texttt{zz\$}),\{\})]\end{array}\right\}\end{aligned}$$

Each step invokes the *put* of the match lens, which locates a concrete chunk from the dictionary and invokes the *put* of $l_\$$. The final result illustrates the "resourcefulness" of this lens:

$$\overline{\langle l_\$\rangle^*}.\ \mathit{put}\ \texttt{xxzzz\$x\$}\ \texttt{xyz\$xxyyzz\$} = \texttt{xxyyzzz\$xy\$}$$

By contrast, the *put* component of the basic lens $l_\${}^*$ is not resourceful—it restores the ys to each chunk by position:

$$l_\${}^*.\ \mathit{put}\ \texttt{xxzzz\$x\$}\ \texttt{xyz\$xxyyzz\$} = \texttt{xxyzzz\$xyy\$}$$

The final operator forms the union of two dictionary lenses:

$$\frac{\begin{array}{c}l_1{\in}C_1 \overset{S_1,L}{\Longleftrightarrow} A_1\\ l_2{\in}C_2 \overset{S_2,L}{\Longleftrightarrow} A_2\\ C_1\cap C_2=\emptyset \quad S_1\cap S_2=\emptyset\end{array}}{l_1 \mid l_2{\in}C_1\cup C_2 \overset{S_1\cup S_2,L}{\Longleftrightarrow} A_1\cup A_2}$$

$$\begin{array}{ll}\mathit{get}\ c & = \begin{cases} l_1.\ \mathit{get}\ c & \text{if } c\in C_1\\ l_2.\ \mathit{get}\ c & \text{if } c\in C_2\end{cases}\\ \mathit{parse}\ c & = \begin{cases} l_1.\ \mathit{parse}\ c & \text{if } c\in C_1\\ l_2.\ \mathit{parse}\ c & \text{if } c\in C_2\end{cases}\\ \mathit{key}\ a & = \begin{cases} l_1.\ \mathit{key}\ a & \text{if } a\in A_1\\ l_2.\ \mathit{key}\ a & \text{if } a\in A_2\backslash A_1\end{cases}\\ \mathit{create}\ a\ d & = \begin{cases} l_1.\ \mathit{create}\ a\ d & \text{if } a\in A_1\\ l_2.\ \mathit{create}\ a\ d & \text{if } a\in A_2\backslash A_2\end{cases}\\ \mathit{put}\ a\ (s,d) & = \begin{cases} l_1.\ \mathit{put}\ a\ (s,d) & \text{if } a,s\in A_1{\times}S_1\\ l_2.\ \mathit{put}\ a\ (s,d) & \text{if } a,s\in A_2{\times}S_2\\ l_1.\ \mathit{create}\ a\ d & \text{if } a,s\in (A_1\backslash A_2){\times}S_2\\ l_2.\ \mathit{create}\ a\ d & \text{if } a,s\in (A_2\backslash A_1){\times}S_1\end{cases}\end{array}$$

This definition is analogous to the union operator for basic string lenses. Because the *put* function takes a skeleton and dictionary rather than a concrete string (as the basic lens *put* does), the last two cases select a branch using the skeleton value. The typing rule ensures that skeleton domains are disjoint so that this choice is well-defined. The union combinator is associative, but not commutative (for the same reason that the basic lens is not).

One interesting difference from the basic lens is that the *create* function takes a dictionary argument, which can be used to transfer information from one branch to the other. The following example illustrates why this is useful. Define $l_{\$\$} = \langle l_\$\rangle \mid \langle l_\$\rangle{\cdot}\langle l_\$\rangle$. The typing rules give us the following facts:

$$\begin{array}{c} l_{\$\$} \in E_C \mid E_C{\cdot}E_C \overset{\{\square,(\square,\square)\},[S]}{\Longleftrightarrow} E_A \mid E_A{\cdot}E_A,\\ \text{where}\quad E_C = \texttt{x}^*{\cdot}\texttt{y}^*{\cdot}\texttt{z}^*{\cdot}\texttt{\$} \qquad E_A = \texttt{x}^*{\cdot}\texttt{z}^*{\cdot}\texttt{\$}\\ S = \texttt{x}^*\times\texttt{y}^*\times\texttt{z}^*{\cdot}\texttt{\$}.\end{array}$$

Now consider $c_1, c_2 \in E_C$ and $a_1, a_2 \in E_A$, where $a_i = l_\$.\ \mathit{get}\ c_i$. We have $l_{\$\$}.\ \mathit{get}\ c_1{\cdot}c_2 = a_1{\cdot}a_2$. A natural way for the *put* function to reflect an update of $a_1{\cdot}a_2$ to a_2 on the

concrete string would be to produce c_2 as the result. However, since the update involves crossing from one branch of the union to the other, the basic lens version cannot achieve this behavior—crossing branches always triggers a *create* with defaults. For example, with $c_1 = \texttt{xyz\$}$, $c_2 = \texttt{xxyyzz\$}$, $a_1 = \texttt{xz\$}$, and $a_2 = \texttt{xxzz\$}$, we have

$$(\overline{l_{\$}} \mid \overline{l_{\$}} \cdot \overline{l_{\$}}).\, put\ \texttt{xxzz\$}\ \texttt{xyz\$xxyyzz\$} = \texttt{xxzz\$}.$$

The dictionary lens version, however, is capable of carrying information from the concrete string via its dictionary, even when the update changes which branch is selected. On the same example, we have

$$\overline{l_{\$\$}}.\, put\ \texttt{xxzz\$}\ \texttt{xyz\$xxyyzz\$} = \texttt{xxyyzz\$},$$

as we might have hoped.

4.4 Quasi-Obliviousness

As the examples above demonstrate, dictionary lenses can be written to work well in situations where the updates to abstract strings involve reordering. In particular, the dictionary lens version of the composers lens in the introduction behaves well with respect to reordering, while the original basic lens version does not. In this section, we develop a refinement of the semantic space of basic lenses that makes such comparisons precise. We first define a space of *quasi-oblivious* lenses and show how it can be used to derive intuitive properties of lenses operating on ordered data. We then show how it can be used more generally to succinctly characterize two important special cases of basic lenses—oblivious and very well behaved lenses.

Quasi-obliviousness is an extensional property of lenses—i.e., a property of the way they transform entire abstract and concrete structures. When discussing it, there is no need to mention internal structures like skeletons and dictionaries. We therefore return in this section to the simpler vocabulary of basic lenses, keeping in mind that a dictionary lens l can be converted into a basic lens $\bar{l}$ as described in Section 4.3.

Let l be a basic lens from C to A and let $\sim$ be an equivalence relation on C. Then l is *quasi-oblivious* with respect to $\sim$ if it obeys the following law for every $c, c' \in C$ and $a \in A$:

$$\frac{c \sim c'}{l.\, put\ a\ c = l.\, put\ a\ c'} \qquad \text{(EquivPut)}$$

Note that the law has equality rather than $\sim$ in the conclusion; this is because the *put* must propagate all of the information contained in a to satisfy PutGet. In particular, the order of chunks in the result of the *put* is forced by their order in a.

Like the basic lens laws, EquivPut is a simple condition that guides the design of lenses by specifying what effect they must have in specific cases where the correct behavior is clear. One way to understand its effect is to notice how it extends the range of situations to which the GetPut law applies—GetPut only constrains the behavior of the *put* on the unique abstract string generated from a concrete string by *get*; with EquivPut, it must have the same behavior on the entire equivalence class.

Here is an example demonstrating how EquivPut and GetPut can be used together to derive an useful property of the *put* component of a lens, without any additional knowledge of how *put* operates. Let C and A be regular languages and suppose that we can identify the *chunks* of every string in C and the *key* of each chunk. For example, in the composers lens, the chunks are the lines and the keys are the names of the composers. These chunks and keys induce an equivalence relation on C where two strings c and c' are equivalent if they can be obtained from each other by a *key-respecting reordering* of

chunks—i.e., by repeatedly swapping chunks such that the relative ordering of chunks with the same key is preserved. Write $\sim$ for this equivalence relation. Now let l be a quasi-oblivious lens with respect to $\sim$ and suppose that the notions of chunks and keys on C are carried by the *get* function to A in a natural way, and that every key-respecting reordering on A can be generated by applying the *get* function to a correspondingly reordered string in C. (This is the case with our dictionary lens in the composer example: chunks in the abstract codomain are lines, the composer names are preserved by the *get* function, and the order of the abstract lines after a *get* is the same as the order of the lines in the concrete structure.) Consider an arbitrary concrete string c, an abstract string $a = get\ c$, and an updated abstract string a' that is obtained by reordering chunks in a. Let us calculate the result of applying *put* to a' and c. By the above hypothesis, since a' was obtained by reordering the chunks in a, it is equal to the *get* of c' for some c' obtained from c by the corresponding reordering of chunks. By the GetPut law, applying the *put* function to a' and c' is c'; by EquivPut, applying the *put* function to a' and c also yields c'. Thus, quasi-obliviousness lets us derive an intuitive result: the *put* function translates reorderings of chunks in the abstract string as corresponding reorderings on the concrete string.

The EquivPut law is useful both as a constraint on the design of lens primitives (in particular, dictionary lenses are designed with this principle in mind, for an equivalence based on reordering chunks) and as a guide for developing intuitions. Quasi-obliviousness does not, however, provide a complete specification of the correct handling of ordering in bidirectional languages. For example, it does not say what happens when the update to the abstract string deletes chunks or edits the value of a key. To capture such cases, one could formulate a condition stipulating that the *put* function must align chunks by key. Specifying this condition, however, requires talking about the sublens that operates on the chunks, which implies a syntactic representation of lenses analogous to dictionary lenses. We thus prefer to only consider the extensional law, even though it provides guarantees in fewer situations.

By design, each dictionary lens is quasi-oblivious with respect to an equivalence relation that can be read off from its syntax. The equivalence identifies strings up to key-respecting reorderings of chunks, where chunks are denoted by the occurrences of angle brackets, and keys by the sections each chunk marked using the `key` combinator. To see that every dictionary lens is quasi-oblivious with respect to this equivalence, observe that *parse* maps strings that are equivalent in this sense to identical skeletons and dictionaries, and recall that the *put* function for a dictionary lens (when viewed as a basic lens) wraps an invocation of *parse*, and of *put*, which operates on this skeleton and dictionary directly. It follows that *put* behaves the same on equivalent concrete strings.

Returning to the composers example, we can see that why the basic lens is bad and the dictionary lens is good: the equivalence the programmer had in mind for *both* versions was the one that can be read off from the second one—every line is a chunk, and the relative order of lines with different names should not affect how dates are restored by the *put* function. The first version of the lens, which operates positionally, is not quasi-oblivious with respect to this equivalence.

So far, we have focused on equivalence relations which are key-respecting reorderings of chunks. More generally, we can consider arbitrary equivalences on C. In the rest of this section, we investigate some properties of this more general view of quasi-oblivious lenses.

For a given basic lens l, there are, in general, many equivalence relations $\sim$ such that l is an quasi-oblivious lens with respect to $\sim$. We write $\mathcal{C}l(\sim)$ for the set of equivalence classes (i.e., subsets of the concrete domain) of $\sim$. Every lens l is trivially quasi-oblivious with respect to equality, the finest equivalence relation on C, and the relation $\sim_{max}$, defined as the coarsest equivalence for which l satisfies EquivPut ($c \sim_{max} c'$ iff

$\forall a.\ put\ a\ c = put\ a\ c')$. Between equality and the coarsest equivalence, there is a lattice of equivalence relations.

Given an equivalence relation, every concrete element c may be characterized by the data preserved in the abstract codomain and the rest of the data shared by every other view of the equivalence class containing c. That is, given $C_i \in \mathcal{C}l(\sim)$ and an abstract view a, there is at most one view c such that $c \in C_i$ and $l.\,get\ c = a$. Conversely, if two different concrete views map to the same a, then they must belong to different equivalence classes.

In Chapter 3, two special classes of lenses are discussed. A lens $l \in C \Longleftrightarrow A$ is called *oblivious* if its *put* function ignores its concrete argument completely. A lens l is *very well behaved* if the effect of two *put*s in sequence just has the effect of the second—i.e., if $l.\,put\ a\ (l.\,put\ a'\ c) = l.\,put\ a\ c$ for every a, a', and c. (Very well behavedness is a strong condition and imposing it on all lenses would prevent writing many useful transformations. For example, note that neither variant of the composers lens is very well behaved: if we remove a composer and add the same composer back immediately after, the birth and death dates will be the default ones instead of the original ones. This may be seen as unfortunate, but the alternative is disallowing deletions!)

Interestingly, both of these conditions can be formulated in terms of $\sim_{max}$. A lens l is oblivious iff the coarsest relation $\sim_{max}$ satisfying EquivPut is the total relation on C. Moreover, l is very well behaved iff $\forall C_i \in \mathcal{C}l(\sim_{max}).\ l.\,get\ C_i = A$. This condition puts the abstract codomain in a bijection with each equivalence class of $\sim$ and forces the operation of the *put* function to use the information in the abstract and concrete structures as follows: the concrete structure identifies an equivalence class C_i; the information contained in the abstract structure determines an element of C_i. This turns out also to be equivalent to the classical notion of view update translation under "constant complement" (Bancilhon and Spyratos, 1981).

4.5 Boomerang

Programming with combinators alone is low-level and tedious. To make lens programing more convenient, we have implemented a high-level programming language called *Boomerang* on top of our core primitives.

Boomerang's architecture is simple: dictionary lens combinators are embedded in a simply typed functional language (we use the syntactic conventions of OCaml) built over the base types `string`, `regexp`, and `lens`. The language has all of the usual constructs: functions and `let`-definitions,[4] as well as constants for using dictionary lenses with the interface of a basic lens (as described in Section 4.3):

```
get : lens -> string -> string
put : lens -> string -> string -> string
create : lens -> string -> string
```

Evaluation in Boomerang is logically divided into two levels, in the style of Algol 60. At the first level, expressions are evaluated using the standard strategy of a call-by-value λ-calculus. This, in turn, may trigger the assembly (and type checking!) of a new dictionary lens value. The run-time representation of a dictionary lens value is a record of functions (representing the *get*, *parse*, *key*, *create*, and *put* components) and several finite-state automata (representing the concrete, abstract, skeleton, and dictionary components of the type); when a lens is built, the type checker checks the conditions mentioned in the typing rules using operations on these automata.

[4] Although it is semantically straightforward to define lenses by recursion (see chapter 3), Boomerang does not support recursive definitions as it would then be possible to define lenses with context-free types.

Using libraries and user-defined functions, it is possible to assemble large combinator programs quite rapidly. For example, the following user-defined function encapsulates the low-level details of escaping characters in XML. It takes a regular expression `excl` of excluded characters, and yields a lens mapping between raw and escaped PCDATA characters:

```
let xml_esc (excl:regexp) =
  copy ([^&<>\n] - excl)
  | ">" <-> "&gt;"
  | "<" <-> "&lt;"
  | "&" <-> "&amp;"
```

(When `xml_esc` is applied, the value passed for `excl` typically contains the "separators" of fields in the format; these are used by the type checker, e.g., to verify unambiguous iteration.)

Similarly, the next two functions handle the details of processing atomic values and entire fields in BibTeX and RIS-formatted bibliographic databases. They are defined in a context where `ws`, `quot_str`, `brac_str`, and `bare_str` are bound to the lenses used to process whitespace, quoted strings, strings enclosed in curly braces, and bare strings respectively.

```
let val (ld:string) (r:regexp) (rd:string) =
  del (ws . "=" . ws . ld) .
  copy r .
  del (rd . ws . "," . ws . "\n")

let field (bibtex:string) (ris:string) =
  let quot_val = val "\""  quot_str "\"" in
  let brac_val = val "{" brac_str "}" in
  let bare_val = val "" bare_str "" in
  let any_val = quot_val | brac_val | bare_val in
  ws . bibtex <-> ris . any_val . ins "\n"
```

The `val` function is used to tidy BibTeX values; when it is applied to a left delimiter string `ld`, a regular expression describing the value `r`, and a right delimiter string `rd`, it produces a dictionary lens that strips out the "`=`" character, whitespace, and delimiters. The `field` function takes as arguments strings representing the name of a BibTeX field (e.g. `title`) and the corresponding RIS field (`T1`) and produces a dictionary lens that maps between entire key-value pairs in each format.

The most significant challenges in implementing Boomerang come from the heavy use of regular expressions in its type system. Since the types of dictionary lenses involve regular languages, Boomerang's type checker needs to be able to decide equivalence, inclusion, and emptiness of regular languages, which are all standard. However, standard automata libraries do not provide operations for deciding unambiguous concatenation and iteration, so we implemented a custom automata library for Boomerang. Our library uses well-known techniques to optimize the representation of transition relations, and to recognize several fast paths in automata constructions. Even with these optimizations, as several operations use product constructions, the memory requirements can be significant. In our experience, performance is good enough for examples of realistic size, but we plan to investigate further optimizations in the future.

Because the type analysis performed by the dictionary lens type checker is so precise, many subtle errors—overlapping unions, ambiguous concatenations, etc.—are detected early. Boomerang supports explicit programmer annotations of dictionary lens types, written in the usual way as `let e : (C <-> A)`. It also has mechanisms for printing out inferred types and generating counterexamples when type checking fails. We have

found all these features incredibly helpful for writing, testing, and debugging large lens programs.[5]

4.6 Experience

We have built Boomerang lenses for a variety of real-world formats, including an address book lens that maps between vCard, CSV, and XML; a lens that maps BibTeX and RIS bibliographic databases; and lenses for calculating simple ASCII views of LaTeX documents and iTunes libraries represented in XML as Apple property lists. Our largest Boomerang program converts between protein sequence databases represented in ASCII using the SwissProt format and XML documents conforming to the UniProtKB schema. For example, the following snippet of a SwissProt entry

```
OS   Solanum melongena (Eggplant) (Aubergine).
OC   Eukaryota; Viridiplantae.
OX   NCBI_TaxID=4111;
```

is mapped to a corresponding UniProtKB XML value:

```
<name type="scientific">Solanum melongena</name>
<name type="common">Eggplant</name>
<name type="synonym">Aubergine</name>
<dbReference type="NCBI Taxonomy" key="1" id="4111"/>
<lineage>
  <taxon>Eukaryota</taxon>
  <taxon>Viridiplantae</taxon>
</lineage>
```

Like many textual database formats, SwissProt databases are lists of entries consisting of tagged lines; our lens follows this structure. Entries are processed by the *match* combinator as distinct chunks, so that the information discarded by the *get* (e.g., metadata about each entry's creation date) can be restored correctly when updates involve reorderings. The identifier line provides a natural key. Other lines are processed using lenses specifically written for their data (of course, we factor out common code when possible). Most of these consist of simple filtering and reformatting, and are therefore straightforward to write as dictionary lens combinators.

Interestingly, as we were developing this lens, the Boomerang type checker uncovered a subtle ambiguity in one of the lines that stems from the use of both "`,`" and "`;`" as separators. Some implicit conventions not mentioned in the specification avoid this ambiguity in practice (and we were able to revise our code to reflect these conventions). The precision of Boomerang's type system makes it a very effective tool for debugging specifications!

4.7 Related Work

Basic lenses were the starting point for this work. Chapter 3 includes an extensive survey of the connections between basic lenses and the view update problem in the database literature.

Meertens's formal treatment of *constraint maintainers* for user interfaces (1998, Section 5.3) recognizes the problem we are dealing with in this chapter when operating on lists, and proposes a solution for the special case of "small updates" specified by edit operations, using a network of constraints between list entries. The idea of using

[5] And small ones! All the lenses and examples typeset in a typewriter font in this chapter were checked and run within the Boomerang system.

constraints between concrete and abstract structures is related to our use of keys in dictionary lenses, but handling updates by translating edit operations represents a significant departure from the approach used in lenses, where "updates" are not given as operations, but by the updated value itself. The treatment of ordering for lists and trees in the bidirectional languages X and Inv (Hu et al., 2004; Mu et al., 2004), comes closest to handling the sorts of "resourceful updating" situations that motivate this work. Their approach is based on Meertens's ideas. As in his proposal, updates to lists in X are performed using edit operations. But rather than maintaining a correspondence between elements of concrete and abstract lists, the semantics of the edit operation is a function yielding a tagged value indicating which modification was performed by the edit. The structure editor described in Hu et al. (2004) based on X does handle single *insert* and *delete* operations correctly by propagating these modification tags locally in lists. However, the *move* operation is implemented as a *delete* followed by an *insert*. This means that the association between the location of the moved element in the concrete and abstract lists is not maintained, and so moved data is populated with default values at the point of insertion; e.g., our composers example is not handled correctly.

There is a large body of work on bidirectional languages for situations in which round-trips are intended to be bijective modulo "ignorable information" (such as whitespace). XSugar (Brabrand et al., 2007) is a bidirectional language that targets the special case when one structure is an XML document and the other is a string. Transformations are specified using pairs of intertwined grammars. A similar bidirectional language, biXid (Kawanaka and Hosoya, 2006), operates just on XML data. The PADS system (Fisher and Gruber, 2005) makes it possible to generate a data type, parser, and pretty printer for an ad-hoc data formats from a single, declarative description. PADS comes with a rich collection of primitives for handling a wide variety of data including characters, strings, fixed-with integers, floating point values, separated lists, etc. Kennedy's combinators (2004) describe pickler and unpicklers. Benton (2005) and Ramsey (2003) both describe systems for mapping between run-time values in a host language and values manipulated by an embedded interpreter. In all of these systems, as round-trips are intended to be essentially bijective, the problems with reordering that our dictionary lenses are designed to solve do not come up.

JT (Ennals and Gay, 2007) synchronizes programs written in different high level languages, such as C and Jekyll, an extension of C with features from ML. JT relies on a notion of distance to decide how to propagate modifications, allowing the detection of non local edits such as the swap of two functions. The synchronization seems to work well in many cases but there is no claim that the semantics of the synchronized programs are the same.

Our lens combinators are based on finite-state transducers, which were first formulated as multitape automata by Scott and Rabin (1959). Languages based on finite-state automata have been developed, largely in the area of natural language processing; the collection edited by Roche and Schabes gives a survey (1996). Mechanized checking for string processing languages that, like Boomerang, have type systems based on regular automata have also been studied (Tabuchi et al., 2002).

Chapter 5

Xtatic

5.1 Introduction

XTATIC aims at adding native support for statically typed XML processing to a mainstream object-oriented language. Its guiding design principles are *simplicity*—the extension should be lightweight and easy for programmers to understand; *integration*—it should fit cleanly and inter-operate smoothly with the host language at all levels (data model, type system, control structures, run-time system, VM, libraries); and *flexibility*—its mechanisms for manipulating and typing XML should support a full spectrum of processing styles, from dynamic investigation of documents of unknown or partially known types to fully checked processing of documents for which complete type information is known, and should be robust in the face of program and schema evolution.

XTATIC adds to C♯ two critical mechanisms, both adapted from XDUCE (Hosoya and Pierce, 2000, 2003): *regular types* for XML (Hosoya et al., 2005b) and *regular patterns* for accessing XML data in the style of "grep for trees" (Hosoya and Pierce, 2001). We have used these features to build a number of small and medium-sized demos.

The goal of this chapter is to draw together our experience with the language design as a whole, identifying the issues that seem, in the light of hindsight, most crucial and evaluating the choices we have made in addressing them. In particular, we evaluate the tradeoffs between simplicity and power in the choice of underlying tree grammar formalisms, between structural and nominal treatments of subtyping, between various approaches to static typing for attributes, and between regular patterns and XPATH-style paths.

After a small example in Section 5.2, showing the key features of the language—regular types and regular pattern matching—we discuss in detail the most important issues in the design of the type system (Section 5.3), the run-time values used to represent XML data (Section 5.4), and the mechanisms for pattern matching (Section 5.5). Section 5.6 surveys related language designs.

5.2 A Taste of Xtatic

Consider a document fragment containing a sequence of two entries from an address book, given side-by side in XML and XTATIC concrete syntax in Figure 5.1

XTATIC's notation for this document is close to XML, the only differences being the outer double brackets used to segregate the world of XML values and types from the regular syntax of C♯, and the backquotes, which distinguish PCDATA (XML textual data) from arbitrary XTATIC expressions yielding XML elements. These concrete syntax choices are discussed further at the end of Section 5.6.

A possible type for the above value is a list of persons, each containing a name, an

```
<person>
  <name>Haruo Hosoya</name>
  <email>hahasoya</email>
</person>
<person>
  <name>Jerome Vouillon</name>
  <tel>123</tel>
</person>
```

```
[[ <person>
     <name>`Haruo Hosoya`</name>
     <email>`hahasoya`</email>
   </person>
   <person>
     <name>`Jerome Vouillon`</name>
     <tel>`123`</tel>
   </person> ]]
```

Figure 5.1: XML and XTATIC concrete syntax for an address book

optional phone number, and a list of emails:

```
<person> <name>pcdata</> <tel>pcdata</>? <email>pcdata</>* </person>*
```

The type constructor "?" marks optional components, and "*" repeated sub-sequences. XTATIC also includes the type constructor "|" for non-disjoint unions of types. The shorthand `</>` is a closing bracket matching an arbitrarily named opening bracket. Every regular type in XTATIC denotes a set of sequences. Concatenation of sequences (and sequence types) is written either as simple juxtaposition or (for readability) with a comma. The constructors "*" and "?" bind more strongly than ",", which is stronger than "|". The type `pcdata` describes sequences of characters.

Types can be given names that may be referred to in other types. For example, in the presence of these definitions

```
regtype Name  [[ <name>pcdata</> ]]
regtype Tel   [[ <tel>pcdata</>  ]]
regtype Email [[ <email>pcdata</> ]]
regtype TPers [[ <person> Name Tel </> ]]
regtype APers [[ <person> Name Tel? Email* </> ]]
```

our simple address book can be given the type `APers*`.

Between XML types, subtyping is exactly regular tree language inclusion. For example, every value of type `TPers` can also be described by the type `APers`, so we have `TPers <: APers`. Between object types, subtyping follows the standard rules of C♯. (We will have more to say about subtyping in the following section.)

Types and subtyping are also the foundation of *regular pattern matching*, which generalizes both the `switch` statement of C♯ and the algebraic pattern matching popularized by functional languages such as ML. For instance, the following method extracts a sequence of type `TPers` from a sequence of type `APers`, removing persons that do not have a phone number, and eliding emails.

```
static [[ TPers* ]] addrbook ([[ APers* ]] ps) {
  [[ TPers* ]] res = [[ ]];
  bool cont = true;
  while (cont) {
    match (ps) {
      case [[ <person> Name n, Tel t, any </>, APers* rest ]]:
        res = [[ res, <person> n, t </> ]];
        ps = rest;
      case [[ APers, APers* rest ]]:
        ps = rest;
      case [[ ]]:
        cont = false;  } }
  return res; }
```

5.3 Types

Many proposals for XML processing extensions in mainstream languages adopt a *data binding* approach, in which XML types are approximated—usually somewhat awkwardly—in terms of the type structures already available in the host language. Others, notably C_ω (Meijer and Schulte, 2003; Meijer et al., 2003; Bierman et al., 2005), generalize the host language's object and sequence types so that types for XML become a special case. XTATIC steers a middle path, embodying a "mostly orthogonal" integration of XML types and object types that offers maximum flexibility and expressiveness for the former with minimal impact on the latter.

Regular Tree Types

The tree data model of XML gives rise to a natural notion of *regular tree grammars* generalizing familiar regular expressions on strings. Murata, Lee, and Mani (2001) identify four increasingly expressive classes of regular tree grammars:

- *Local tree grammars* adopt the restriction that, wherever a given tag occurs in a tree grammar, its *content model* (the sequence of types of its subtrees) must always be the same. This class corresponds roughly to DTDs. It is sufficiently expressive for many of the simpler uses of XML and has been used for static analysis in the programming language XACT (Kirkegaard et al., 2003).

- *Single-type tree grammars* enforce a weaker restriction, requiring that multiple occurrences of a tag *as children of the same parent node* must have identical content models. The W3C's XML Schema (Fallside and Walmsley, 2004) and the type systems of XQuery and XJ (Harren et al., 2005) are based on this class.

- *Restrained-competition tree grammars* relax the restriction yet further, allowing two trees with equal tags under the same parent to have different content models, as long as the earlier part of the content unambiguously determines which is expected at each point. (For example, `<a> <b>S</> <b>T</> </>` belongs to this class, but `<a> (<b>S</> | <b>T</>) </>` does not.) We are not aware of any programming languages based on this variety of grammars.

- General *regular tree grammars* allow arbitrary combinations of elements with different content models. This class forms the basis of the RELAXNG schema standard (Clark and Murata, 2001) and of the type systems of a number of programming languages, including XDUCE (Hosoya and Pierce, 2001), CDUCE (Benzaken et al., 2003), and XTATIC.

The advantage of staying low in this hierarchy is simplicity and efficiency of implementation: as expressiveness increases, so do the complexity of language membership (validation) and subtyping algorithms. The first two classes can be served by simple adaptations of ordinary *word automata*, while the latter two classes require more complicated *tree automata*. Fortunately, experience with RELAXNG, XDUCE, CDUCE, and XTATIC shows that the algorithms available for unrestricted regular tree grammars, though more complex, remain implementable and practical in usage. Moreover, the more powerful grammar classes have some significant advantages of their own—in particular, better closure properties. While all of the classes are closed under intersection (which is useful for inferring types of pattern variables; see Section 5.5), only unrestricted regular tree grammars are closed under union (useful for conditionals and `match` expressions), difference (which improves the precision of type inference, as explained in 5.5), and concatenation.

A	::=		XTATIC *type*	T	::=		*regular type*
	|	C	class type		|	()	empty sequence
	|	[[T]]	regular type		|	<(A)>T</>	tree
					|	T,T	concatenation
d	::=		*declaration*		|	T|T	alternative
	|	regtype X [[T]]			|	T*	repetition
					|	X	type name

Figure 5.2: Syntax of regular object types

On balance, we feel that both single-type tree grammars and general regular tree grammars offer reasonable foundations for new programming language designs. Local tree grammars are too limited, and restrained competition grammars are harder to understand without being much more tractable than full regular tree grammars. In designing XTATIC, we have chosen unrestricted tree grammars for their power and simplicity, accepting the additional implementation burden that they impose.

Regular Object Types

As we saw in Section 5.2, XTATIC's regular tree types and sequence values are integrated seamlessly into the ordinary C♯ world: wherever a C♯ type or value is expected, a regular type or a sequence value can appear, enclosed in `[[ ... ]]` brackets. We could stop there and disallow integration in the opposite direction, requiring that the contents of the `[[ ... ]]` brackets be pure XML not containing general C♯ values. It turns out, however, that it is easy to allow C♯ values inside XML sequences (in fact, permitting this is easier than preventing it). A side benefit of this generality is that we can use regular patterns over trees of objects to emulate the elegant and concise "datatypes and algebraic pattern matching" programming idioms found in modern functional languages such as ML.

At the beginning of the XTATIC design, we set out to allow C♯ objects to appear directly as members of XML sequence values. However, attempts to formalize this scheme encountered a number of problems. To see why, notice that it is essential for sequence values to be of some C♯ object type so that they can be used with standard generic C♯ libraries such as `Stack` and `Hashtable`. To satisfy this requirement, we designate a special class `Seq` extending `object` to denote all sequence values. Semantically, `Seq` is equivalent to the regular type `[[any]]`. Now, let `o1` and `o2` be two `Seq` objects and consider the sequence value `[[o1, o2]]`. Does it denote a sequence of size two (`o1` followed by `o2`) or does it denote the *concatenation* of the sequences denoted by `o1` and `o2`? Sorting these issues out consistently (with the aim of reducing the programmer's surprise) is not straightforward.

To avoid such puzzles, we have adopted a simpler integration strategy, allowing C♯ values only as *labels* of sequence elements. Now, assuming that o_1 and o_2 are `Seq` objects, `[[<(`o_1`)/> <(`o_2`)/>]]` is a sequence containing two childless elements labeled by o_1 and o_2; whereas `[[`o_1`, `o_2`]]` is a sequence containing the concatenation of the elements of o_1 and o_2. The latter expression is ill typed if either of the objects is not of class `Seq`.

Figure 5.2 shows the syntax of regular object types. Any object can appear as the label of a sequence element, hence the type `[[any]]`—the type of arbitrary sequence values—can be defined recursively as

```
regtype any  [[<(object)>any</>*]].
```

A consequence of allowing arbitrary objects to appear as labels is the existence of sequence values that do not correspond to XML documents. For instance, the sequence

value `<(1)/>` is not XML, since XML elements cannot be labeled by integers. To characterize the set of XML sequences, we introduce a special C♯ class `Tag` that describes precisely the set of XML element tags. Conceptually, every XML tag corresponds to a distinct subclass of `Tag`. For example, the XML fragment `<author/>` corresponds to the sequence value `<(new` $\texttt{Tag}_{\texttt{author}}$`())/>` and is classified by the type `<(`$\texttt{Tag}_{\texttt{author}}$`)/>`.

With this in mind, the type of all XML fragments and the type of purely textual XML fragments can be defined by the regular types `xml` and `pcdata`:

```
regtype xml    [[( pcdata | <(Tag)>xml</> )*]]
regtype pcdata [[<(char)/>*]]
```

As illustrated in Section 5.2, the concrete syntax of XTATIC provides lightweight XML-like notation for proper XML sequences, including textual data, and for regular types describing them. These pure XML values and types are treated specially in the implementation. For instance, subclasses of `Tag` are not created physically at run time, but rather a more compact and efficient representation is used; see Gapeyev et al. (2005c) for details.

Structural vs. Nominal

The prominence of the W3C SCHEMA notation shapes the design space for XML processing languages in an important way, forcing a stark choice between structural and nominal treatments of XML data.

The structural vs. nominal distinction is usually discussed in terms of the subtype relation: in the nominal case, the typechecker deals principally with *names* of types and subtyping is allowed only when a subsumption relation between two names has been explicitly declared by the programmer, whereas, in the structural case, type names are ignored and subsumption relations between types are determined automatically. This difference at the level of types is not especially deep in itself: nominal typing simplifies the typechecker implementation a little and prevents accidental confusions between structurally similar types, but neither of these is a very big deal. Where the rubber really meets the road is at the level of *values*. In a nominal language, each run-time value is marked with a type name, which is then also used as the type of this value during static type-checking. Typically, constructs are provided in the programming language that define and/or rely on an ordering on the type names (e.g., subclass definitions in conventional object-oriented languages), and this ordering is lifted to named types in the form of a subsumption relation.

For a language with XTATIC's goals, treating XML data in a nominal style—i.e., marking each tree node with a type name—is an attractive option. For one thing, it makes it easy to construct efficient type-checking algorithms, since there is never any need to examine the "right hand sides" of defined types to determine whether one is a subtype of another. Similarly, the type names can be put to good use for efficiently checking that a value belongs to a type at run time. This can be used in the implementation of many useful language features, including XDUCE-style regular pattern matching, downcasting, re-validation of XML trees for which type information has been lost (e.g., by storing and later retrieving them from a generic collection), and—obviously—XPATH 2.0's primitive for matching a node if it is marked with a particular type name. Another argument for nominal subtyping is that the industry standard "type system" for XML data, W3C XML Schema, is (mostly) nominal, offering various mechanisms for declaring subtypes explicitly. XQUERY, whose type system (Siméon and Wadler, 2003) is closely based on W3C XML Schema, has adopted the nominal approach for these reasons. A final argument for using nominal subtyping in XTATIC is that this would yield a pleasing similarity to C♯'s nominal treatment of objects and their types.

Nonetheless, we have chosen in XTATIC to treat XML trees and their types structurally. Doing so makes the implementation of the typechecker, runtime system, and pattern match compiler somewhat harder, but yields some significant benefits. Most importantly, it avoids what has been known as the "*the* Schema Fallacy"—i.e., the presumption that a given XML document or document fragment will always be thought of as belonging to one specific type. If types are thought of as being only *descriptions* of data, rather than being embedded in the data itself, then it makes perfect sense to think of the same data structure as satisfying multiple descriptions at different points in a program, each specifying just what is needed for the task at hand. This ability to adopt multiple views of the same data can play a critical role in software reusability. For instance, instead of writing a method that extracts the `<name>` elements from all the entries in an address book, one may write a generic method that extracts `<name>` elements from *any* sequence of data items, each containing at least a `<name>` element. This generic method can be given a type (namely `<(Tag)/>any, Name, any</>*`) that precisely describes the requirement on its input, and the validity of passing it an address book can be checked automatically, with no need for a "re-validation" step—i.e., no unsafe runtime typecast.

Inferring subtyping automatically can also help sidestep some well-known software engineering traps. For example, if we have two existing data structures with similar schemas, it may be useful to write a program that can work over both of them—i.e., that accepts a common supertype as input; but doing this in a nominal setting may in general involve adding supertype declarations to both existing schemas; this is annoying at best, and at worst may not even be possible—if, for example, they are controlled by another organization. Similarly, avoiding the requirement of writing explicit subtype declarations removes a potential source of friction as programs and schemas evolve.

Attributes

Static typing for XML attributes is an area where the design of XTATIC falls short: despite several attempts, we have not yet been able to find a treatment that satisfies all of our requirements. For the moment, we have therefore adopted a simple *un*typed scheme for building values with attributes and for pattern matching against attributes. We briefly describe the difficulties we have encountered and sketch our current stopgap solution.

A type system for attributes should support a way of specifying both closed (exactly these) and open (these and perhaps others) sets of attribute/value pairs, should support optional and required attributes, and should provide boolean operations such as union and intersection of attribute types—while maintaining the constraint that an element cannot have more than one attribute with a given name.

Hosoya and Murata's work on *attribute-element constraints* (2003) describes a powerful attribute type system that offers all of these features and also allows the programmer to express dependencies between element and attribute children of a parent element (e.g., it can be used to express constraints like "this element must have either a `date` sub-element or else `month` and `year` attributes"). Alternatively, one may begin from a conventional record type system and extend it with the features listed above. We have experimented with this approach and found that it does not seem to lead to anything much simpler than Hosoya and Murata's solution: because of arbitrary unions, the subtyping algorithms in both approaches turn out to be surprisingly similar.

Unfortunately, despite the descriptive power of these attribute type systems, there is no obvious way to adapt them to support sufficiently flexible regular pattern matching over attributes. For example, patterns based on attribute-element constraints (Hosoya and Murata, 2003) could be used to select all the attributes of an element with names belonging to a given set—but only provided it can be statically verified that the names

of the remaining attributes do not belong to the set. In particular, it is not possible to traverse the attribute set of an element one-by-one, using a wildcard name pattern to select the next attribute. This ability is important for implementing generic traversal tasks over documents of unknown types (e.g., uppercasing the values of all attributes).

XTATIC takes a simple dynamic approach: attributes are part of values and can be written in patterns, but they are ignored in types; as far as the type system is concerned, any element may have any collection of attributes. XTATIC attribute patterns provide a way of extracting values of specified attributes as well as the remaining attribute/value pairs. The latter are transformed into a sequence `<attr`$_1$`>val`$_1$`</> <attr`$_2$`>val`$_2$`</>` ... `<attr`$_n$`>val`$_n$`</>` that can be dynamically examined using conventional element pattern matching. Sequence values of this form can also be used to supply attributes of a newly created element. To ensure well-formedness, the run-time system must perform a dynamic test verifying that the given sequence does not contain repeating element names.

Overloading

Naturally XTATIC extends C♯ method overloading to support regular types in method signatures. For example, if a class contains two methods `void f([[Person*]] x)` and `void f([[Person+]] x)`, then a call `f(x)` is resolved to one or the other depending on the static type of `x` according to the standard overloading resolution rules modified to use XTATIC subtyping instead of C♯ subclassing.

Such extension of overloading requires some additional work on the part of the compiler. Since XTATIC is compiled into C♯ homogeneously—every regular type is translated into a single C♯ type `Seq`, and every sequence value is translated into an object of this class—it is possible that XTATIC methods with different signatures will be compiled into C♯ methods with the same signature, resulting in an illegal C♯ program. For instance, both of the above signatures map to `void f(Seq x)`.

We resolve this problem by generating new names for all methods that have arguments of types more precise than `[[any]]`. The purpose of renaming is to encode the regular types information from the original XTATIC method signature into the name of the compiled C♯ method in such a way that C♯ overloading resolution on renamed methods behaves the same as would XTATIC overloading resolution. In particular, we ensure that a method that overrides or hides a method from a base class receives the same mangled name as the base method. To make this arrangement work with independent typechecking of separate assemblies, our compiler generates for each assembly an auxiliary structure containing, along with the mangled names, the original method signatures with regular types.This structure is used when typechecking an XTATIC program that references the assembly.

A C♯ program compiled against an XTATIC library is expected to refer only to non-mangled method names—that is, method names with regular type arguments at most as precise as `[[any]]`. Consequently, if an XTATIC library wants to export a method operating on values of a regular type that is more precise than `[[any]]` to pure C♯ code, it can provide a wrapper method accepting values of type `[[any]]`, explicitly casting them to the appropriate regular type and then calling the original method. This approach ensures safety by eliminating the possibility of unchecked invocation of XTATIC methods with ill-typed parameters from arbitrary C♯ code.

5.4 Values

This section describes and motivates XTATIC's design of sequence values. We justify our decision to use immutable values and discuss how this choice interacts with various XML inspection styles and C♯ programming idioms. We also cover two enhancements to

the basic model of values that facilitate efficient downcasting and on-demand translation from legacy XML representations such as DOM.

Immutability

A fundamental design goal of XTATIC is ensuring type-safe manipulation of XML data. This implies that either XML structures be immutable, or that updatable parts of XML structure be marked with "ref types" to prevent subtyping. For the sake of simplicity we have chosen to pursue the first approach and make every XML value immutable. Other languages, such as XJ (Harren et al., 2005), relax the static safety requirement and do dynamic checks when mutation of XML data occurs.

Immutability, in turn, demands that we choose a representation that supports a great deal of sharing in order to retain acceptable memory performance. This sharing prevents the use of doubly-linked trees for a representation of XML values, in particular back-pointers in the style of DOM, thus hindering such value inspection mechanisms as XPATH's backward axes.[1]

There is some inherent tension between the design choice of immutability and the natural imperative programming style of C♯, which can lead to disappointing performance if care is not taken in the implementation. We now discuss some examples illustrating potential dangers and explain how they are avoided.

A common idiom of imperative XML programming is to use in-place modification when making small changes to existing documents. For example, to modify the author of a book from "John" to "Bob" one could do something like this:

```
[[book]] doc = LoadXml("file.xml");
XmlNode n = doc.FindFirstNode("book/chapter/author = 'John'");
n.text = "Bob";
```

In XTATIC, one needs to capture the context where the change occurs, and recreate it:

```
[[book]] doc = LoadXml("file.xml");
match (doc) {
  case [[<book>any c,
                <chapter>any a, <author>`John`</>, any b</chapter>,
              any d</book>]]:
    return [[<book>c,
                <chapter>a, <author>`Bob`</>, b</chapter>,
                d</book>]];
}
```

Programmers used to imperative languages need just to learn that this code is not as expensive as it could appear: because of sharing, only the path from the root of the value to the point where it is modified needs to be copied: XML values `c`, `a`, `b`, and `d` are reused. On the other hand, it can be tedious to write this recreation code. In XACT, essentially the same computation over their immutable data model is expressed by code that is very similar to its imperative conterpart: the programmer creates a *template*, which is a value with a hole in place of "John", and then fills it with the new value "Bob".

Another imperative programming idiom consists of creating a sequence of XML values by repeatedly appending XML elements to an accumulator within a `while` loop. For instance, one may create a list of persons the following way:

```
[[ Person* ]] p = [[ ]];
```

[1] The XACT design (Kirkegaard et al., 2004) shows that it is actually possible to implement backward-axis traversals in some situations by maintaining a separate "context" data structure representing the path from the root of the value to the current position.

```
while (some_condition) {
  p = [[ p, <person><Name>create_name ()</></> ]];
}
```

An efficient way of implementing such a construction when mutation is available is by keeping a pointer to the last element of the output sequence and updating its contents whenever a new element must be appended. In functional programming, an efficient idiom for the same task first builds a reverse sequence by prepending elements, and then reverses it.

In order to retain the intuitive concatenation order as shown in the code above while avoiding turning linear algorithms into quadratic ones, XTATIC uses carefully designed lazy data structures and algorithms that delay creation of concrete values until they are inspected elsewhere, see Gapeyev et al. (2005c) for more details.

Fast Downcasting

Part of the appeal of XTATIC is that it allows programmers to use familiar C♯ libraries to store and manipulate XML values; in particular, XML values can be stored in generic collections such as `Hashtable` and `Stack`. However, values extracted from such containers have type `object`, and generally need to be cast down to the intended type. In pure C♯ this operation incurs only a constant time overhead, but in XTATIC, downcasting to a regular type may involve an expensive structural traversal of the entire value (cf. Section 5.3). One way to avoid this overhead is to *stamp* a sequence value with a representation of its type (upon putting the value into a collection) and perform a run-time stamp comparison rather than full re-validation during downcasting (upon receiving the value from the collection). Our design places this stamping under programmer's control.

We extend the source language with a stamping construct, written `<[[T]]>e` ("stamp `e` with regular type `T`"). An expression of this form is well typed if `e` has static type `T`; in this case, the result type of the whole expression is `object`.[2] Casting a stamped value can then be done in constant time as long as the type used in the cast expression is syntactically identical to the type used in the stamping expression. Casting to any other type, however, must fall back to the general pattern-matching algorithm, which dynamically re-validates the value.

In our design, the burden of type stamping is placed on the programmer. We have experimented with alternative designs in which stamping is performed silently—either by adding a stamp whenever a sequence value is upcast to type `object` or by including a type stamp in every sequence object. However, we have not found a design in which the performance costs of stamping and stamp checking are acceptably predictable.

Legacy Representations

XTATIC modules are expected to be used in applications built in other .NET languages that may also use the extensive .NET libraries. The latter already contain support for XML, collected in the `System.Xml` namespace. It is essential for XTATIC's XML manipulation facilities to interoperate smoothly with native .NET XML representations and, conversely, XTATIC XML data to be accessible from XTATIC-agnostic C♯ code. We have explored the former direction of this two-sided interoperability problem by designing support for DOM, one popular XML representation available in .NET.

A straightforward solution for accessing DOM from XTATIC would be to first translate any DOM data of interest into our representation in its entirety and then proceed to working with XTATIC's native representation. This is wasteful, however, if an XTATIC

[2] Giving stamped values type `object` ensures that, at run-time, such values will never appear as part of other sequences. This makes implementation of sequence operations simpler and more efficient.

program ends up accessing only a small portion of the document. A better idea is to perform the translation from DOM lazily, namely at the time when fragments of the DOM value are accessed by XTATIC code. XTATIC provides the method `[[any]] ImportDOM(XmlNode x)` that takes a raw DOM node and wraps it into a datastructure that is visible to a program as an XTATIC sequence value. Matching a wrapped value against a regular pattern results in copying its matched fragments into the native (non-wrapped) XTATIC representation thus abandoning aliasing with its mutable DOM predecessor. Consider the following example:

```
match (ImportDOM(...)) {
  case [[ <book>any</book> b, any rest ]]: ...;
  case [[ any ]]: ...;
}
```

In the right-hand side of the first clause, the type for `b` is `<book>any</>`. If `b` was still represented by the original DOM fragment, this type assumption could be violated by a program that modified the element's tag (via DOM interface) from `<book>` to, say, `<journal>` after the pattern match. Replicating the fragments of DOM that have been pattern-matched by XTATIC is thus necessary for type safety. The fragments that have not been inspected yet, however, can stay in their original DOM representation. For example, modifying the DOM fragment corresponding to `rest` cannot violate any static assumptions since the variable has type `[[any]]`, which accurately describes the corresponding DOM fragment regardless of whether any modification happens. The same applies to the fragment matched by `any` in `b`. Even though the original DOM structure may be destructively updated at any time, the only modifications that are visible to XTATIC are those that happen before pattern matching inspects the fragments in question.

5.5 Pattern Matching

Regular patterns are both powerful and pleasingly simple to formalize: they are just regular types decorated with binders. Our experience using XTATIC has shown them to be very convenient for a broad range of tasks. For some jobs, however, regular patterns can be rather inconvenient and another style of value inspection—embodied in the popular XML XPATH standard (XPath 1.0)—works better. To accommodate paths-based processing in XTATIC, we may use a desugaring technique that converts an important fragment of XPATH ("downward axis" paths) into equivalent regular patterns, thus giving us the best of both worlds. We first sketch this approach.

We next describe type inference for XTATIC's regular patterns, concentrating in particular on the issue of *precision* of the inference algorithm—an important point where XTATIC differs from XDUCE. We conclude the section by addressing the issue of *schema evolution*, i.e., the question of how robust programs are when the type of the data they manipulate changes.

Regular Patterns and Paths

The syntax of regular patterns (Figure 5.3) mimics the syntax of regular object types, additionally providing constructs for binding sequence values and objects within element labels. For example, the pattern `[[<a/>*, <b/>* x, <c/>*]]` matches sequences composed of zero or more `a`-elements followed by zero or more `b`-elements followed by zero or more `c`-elements. The middle sub-sequence containing all the `b`-elements is extracted and bound to `x`. For an example of binding within labels consider the pattern `[[<(B x)/>]]`. It matches singleton sequences whose element is labeled by an object `o` of class `B` and binds `o` to `x`.

`Q`	::=		XTATIC pattern
	|	`C`	class pattern
	|	`[[P]]`	regular pattern
	|	`Q x`	C♯ var binding
`d`	::=		pattern declaration
	|	`regpat X [[P]]`	
`P`	::=		regular pattern
	|	`X`	type name
	|	`<(Q)> P </>`	tree
	|	`()`	empty sequence
	|	`P,P`	concatenation
	|	`P|P`	alternative
	|	`T*`	type repetition
	|	`P x`	regular var binding

Figure 5.3: Syntax of regular patterns

Regular patterns are especially convenient for "horizontal" inspection of XML sequences. Consider an HTML table that contains two sets of rows with a distinctive separator row between them—i.e., the table's contents are of type `row*, separator, row*`, where `row` and `separator` are defined as follows:

```
regtype row [[ <tr> <td>pcdata</> <td>pcdata, <a>pcdata</></>,
                    any </> ]]
regtype separator [[ <tr> <td>pcdata</> <td>pcdata</>, any </> ]]
```

Observe that there is only a slight difference between a row and a separator—the former has a hyperlink in its second cell while the latter does not. Let `table` contain an HTML table whose contents satisfies the above type. Using regular patterns, we can extract the two sets of rows in one line of XTATIC (this statement is desugared into a `match` expression with one clause whose right hand side is the rest of the program):

```
[[<table> row* x, separator, row* y </>]] = table;
```

By contrast, XPATH's *paths* are inspired by a file-system-like style of hierarchical navigation. We sketch here a fragment of XPATH (the "downward axes" fragment) that can be supported by XTATIC data model and includes the most frequently used XPATH features. Here is a typical XPATH expression: `table/tr/td`. It finds all the `table` elements at the top level of the current document, locates their `tr` subelements, and extracts all of their `td` children. In addition to parent-to-child navigation, XPATH provides a simple way of reaching arbitrary descendants of the current element. For instance, the query `table//a` finds all hyperlinks occurring somewhere inside top-level tables.

XPATH *predicates* allow the programmer to filter potential solutions against conditions that can be expressed as paths. The XPATH query `table/tr[td//a]` locates all the rows of the top-level tables such that some of their cells have a hyperlink descendant. The predicate, delimited by brackets, is an additional condition on the rows that are returned by the query.

As these examples suggest, XPATH-style paths are particularly suitable for "vertical" inspection of XML, providing a natural mechanism for specifying parent/child and parent/descendant constraints on input values. The two styles of value inspection—regular

patterns and paths—are complementary: a natural pattern matching task is cumbersome to accomplish by paths and vice versa. We now show how the subset of XPath sketched here can be desugared into regular patterns.

Implementing Paths Using Regular Patterns

The main difference between the semantics of paths and regular patterns is that a path query returns multiple answers while a regular pattern is used to match a value and return at most one set of bindings associating variables with fragments of the input. For further clarification, consider the pattern `[[any x, <b/>, any]]` that matches sequences containing a `b` element and extracts the preceding prefix into `x`. We say that this pattern is *ambiguous*: given a sequence with multiple `b` elements, there are multiple possible bindings for `x`. Nevertheless, when used in a `match` statement in Xtatic, the above pattern will only compute at most one binding. (The particular binding that is computed is left unspecified in the implementation of Xtatic.)

To get us closer to paths semantics, we propose an additional pattern matching construct `iterate` that, rather than choosing one way of matching an ambiguous pattern, explores all possible matchings and computes all possible variable bindings. Here is an example of `iterate` with the ambiguous pattern mentioned above:

```
iterate ([[<a/><b/><a/><b/>]]) matching [[any x, <b/>, any]] {
  System.Console.WriteLine(x);
}
```

An `iterate` statement consists of an expression that evaluates to the input value, a pattern, and a body that is executed for every possible match of the value against the pattern. The above fragment will print `<a/>` and `<a/><b/><a/>`.

A variant of `iterate` statement uses path queries instead of ambiguous regular patterns. For instance, the following statement prints all the `td` elements that are children of the `tr` elements that are children of the top-level `table` elements found in the document fragment stored in `table`:

```
iterate (table) matching x at path [[table/tr/td]] {
  System.Console.WriteLine(x);
}
```

Notice that since a path by itself does not define a results-bound variable that can be referred to in the body of `iterate`, this variant of the statement uses special syntax to introduce the variable separately (variable `x` in the example). Such a variable is bound to the results of the path query (`td` elements in the example).

The downward-axes subset of XPath supported by Xtatic is described by the following grammar where `p`, `s`, `q`, `a`, `L`, and `l` range over path queries, query steps, predicates, axes, label tests, and label names respectively, and where `*` is a label wildcard:

```
p ::= s | s/p
s ::= a::L | s[q]
L ::= l | *
q ::= p | q and q | q or q
a ::= self | child | descendant | descendant-or-self
```

In the previous examples, we used an abbreviated notation for the axes `child` and `descendant-or-self`: `a/b` stands for `child::a/child::b` and `a//b` stands for the expression `child::a/descendant-or-self::*/child::b` in the above syntax.

A semantics-preserving translation from this fragment of XPath into ambiguous regular patterns is defined in Gapeyev and Pierce (2004). Consider, for example, the path

query `table/tr//a`. It is converted into `<(Tag)> any, <table> X </>, any </>` assuming the following pattern declarations (notice how `x` is bound to the `a` elements that are addressed by the right-most step of the query):

```
regpat X [[ any, <tr> any, Y, any </>, any ]]
regpat Y [[ ((<a> any </>) x) | Z ]]
regpat Z [[ <(Tag)> any, Y, any </> ]]
```

This translation is correct in the sense that the nodes returned by the path query are precisely the values bound to `x` in all possible matchings of the input value against the regular pattern obtained as the result of translation. CQL (Benzaken et al., 2005) proposes an alternative approach to simulating downward paths in a language with regular patterns.

The full XPath standard is a complex language that contains many features beyond the forward-only paths covered above. XPath supports backward axes that allow the programmer to navigate from children to parents, and from right siblings to left siblings. Xtatic's lightweight immutable data model does not provide a natural basis for supporting backward-axis navigation as explained in Section 5.4. In Chapter 6, we introduce a satisfiability checking algorithm which can be used to typecheck an XPath query (potentially containing backward axes) against a regular type.

Type Inference

The goal of type inference is to assign types to the variables appearing in regular patterns. Consider the following example taken from a program that processes and converts BibTeX files into HTML:

```
regtype entry [[ ... ]] // complex type
regtype doc [[<doc> entry* </>]]

void do_xml ([[doc]] doc) {
  match (doc) {
    case [[<doc>any items</doc>]]:
      match (items) {
        case [[anyone item, any rest]]: ...
        ... } } }
```

If the type checker only used the annotations provided by the programmer, it would operate with the pretty limited knowledge that variables `items` and `rest` may contain any arbitrary sequences and variable `item` may contain an arbitrary singleton sequence. This would probably be insufficient to type-check the right hand side of the first `match` clause. However, if the type checker takes into consideration the type of the input value, it can then be much more exact and infer that `items` and `rest` are of type `entry*` and `item` is of type `entry`.

So how exact can the type checker be? In fact, type inference can be be fully exact. We say that a type inference algorithm is *precise* if given an input value of type `T` and a variable `x` occurring in a pattern `p`, it infers type `S` for `x` iff for every value `v′` of `S`, there is a value `v` of `T` such that matching `v` against `p` results in binding `x` to `v′`. In other words, the inferred type denotes *precisely* the values that may be bound to the variable. Xtatic's parent XDuce (Hosoya and Pierce, 2001) features precise type inference. So does Xtatic's sibling CDuce (Benzaken et al., 2003).

Because regular object types are not closed under boolean operations—union, intersection, and difference—type inference in Xtatic is *not* precise. The core of the problem is in element labels which can be arbitrary C♯ types. Suppose we want to compute the following difference: `<(D)/> \ <(C)/>`. This is equivalent to `<(D\C)/>`, but if `C` is a

subclass of D, in general, there is no C♯ type that corresponds to D\C. Because difference is not always defined on regular object types, XTATIC's type inference does not compute type difference at all and, therefore, does not take into account the order of clauses in a match expression, losing some precision as a result. Consider this example:

```
[[<a/>+]] id([[<a/>+]] arg) {
  match (arg) {
    case [[<a/>]]: return([[<a/>]]);
    case [[<a/>, any rest]]: return [[<a/>, id(rest)]];
  } }
```

Intuitively, the recursive call to id cannot go wrong since rest can only be bound to values of type <a/>+. The type checker, however, can only conclude this by taking the difference between the input type <a/>+ and the first pattern <a/> and then analyzing the second pattern with respect to the resulting type. If it considers the second clause independently of the first, it can only infer that the type of rest is <a/>* which is insufficient for type-checking the recursive call.

The fact that object types are not closed under union is also a problem when we have binding inside labels. Consider the pattern <(C x)/> | <(D x)/> that binds x to objects of classes C or D. Again, in general, there is no C♯ type that corresponds to C ∪ D. XTATIC uses the smallest common superclass of C and D as the inferred type of x, hence straying even further from full precision.

Finally, intersection is problematic as well. XTATIC can correctly compute C ∩ D if one is a superclass (superinterface) of the other or if C and D are unrelated classes. (In the former case, the result is the smaller type; in the latter, the result is empty.) If C and D are unrelated interfaces, however, there is no C♯ type that corresponds to their intersection.

Nevertheless, as the opening example of this section illustrates, type inference in XTATIC is still very useful. Examples such as this are common in practice—they are characterized by variables annotated with "wildcard" types such as any and anyone. The type inferred for these variables is usually precise. More generally, type inference in XTATIC is precise as long as patterns of a match statement are mutually disjoint and all of the occurrences of each variable inside labels have the same type annotation.

As this discussion shows, precise type inference can enable more convenient programming idioms; moreover, it is more difficult to explain to programmers the behavior of pattern matching mechanisms that do not enjoy precise type inference. Extending the C♯ type system with boolean operations that allows taking the union, difference, and intersection of classes and interfaces would be a most welcomed extension of XTATIC.

Patterns and Schema Evolution

We close the discussion of patterns with some remarks about schema evolution—in particular, the robustness of programs with regular patterns and programs with paths with respect to changing data formats. How well does the type system help a programmer in locating potential pitfalls?

For illustration, consider a simple instance of schema evolution in which a new optional element is added to an existing schema. (This real-life example is taken from an XTATIC program that generates the online Caml Weekly News.[3])

```
Old = [[ <cwn><cwn_title>pcdata</> <cwn_content>Content</> </cwn> ]]
New = [[ <cwn><cwn_title>pcdata</> <cwn_url>Url</>?
              <cwn_content>Content</> </cwn>]]
```

[3] http://alan.petitepomme.net/cwn/

Programs using path-like navigation are pretty robust with respect to such format changes: most path queries that work for values of type `Old` would also work for values of type `New`. (One exception involves paths that use order-sensitive "sibling" axes.) However, this flexibility has a significant down-side: since such schema evolutions are transparent for most path-based programs, the path-based type system does not automatically indicate where the program should be modified to take newly introduced elements into account.

Regular patterns, on the other hand, can be chosen by the programmer to be either "loose", like paths, or more detailed, hence stricter, resulting in errors flagged by the type checker when a potentially sensitive format change occurs. The following program is an example of the former:

```
match (input) {
  case [[<cwn>any, <cwn_title>pcdata x</>, any</cwn>]]: ...  }
```

The piece of data deemed important by this code is the contents of the `cwn_title` element; hence this program should work when the above schema change occurs, since the fragment is well-typed both for `Old` and `New` types of `input`.

Conversely, a pattern may be more precise in the specification of the context:

```
match (input) {
  case [[<cwn><cwn_title>pcdata x</>, <cwn_content>any</></cwn>]]: ...}
```

Changing the type of `input` from `Old` to `New` now triggers typing errors, indicating that some cases of the input type (namely the potential presence of a `cwn_url` element) are not covered by the matching clauses. The type checker now guides the programmer to the places in the program where changes must be made to reflect changes in the schema. We have found this mode of "edit definition and type-check to find uses" extremely helpful when programming in XTATIC. (Of course, the mode is familiar from other richly typed languages, but is particularly effective here because of the precision of XTATIC's types.)

5.6 Related work

XDUCE (Hosoya and Pierce, 2001, 2003; Hosoya et al., 2005b) was the first language featuring XML trees as built-in values, a type system based on regular types for statically type-checking computations involving XML, and a powerful form of pattern matching based on regular patterns. XTATIC is a direct descendant of XDUCE.

Another XDUCE descendant that is close to XTATIC in several respects is the CDUCE language of Benzaken, Castagna, and Frisch (2003). Like XTATIC, CDUCE is based on XDUCE-style regular types and emphasizes a declarative style of recursive tree transformation based on algebraic pattern matching. In other respects, however, the design of CDUCE is quite different: its type system includes several features (such as intersection and function types) not present in XTATIC, it is not object-oriented, and it is not integrated with an existing language. XTATIC, by contrast, has taken a more conservative approach in its type system, instead emphasizing smooth compatibility with an existing mainstream object-oriented language. Two significant differences are the object-oriented flavor of our representations and our approach to various interoperability issues such as cross-language calls and compatibility with legacy XML representations. A recent experiment by Frisch shows how many of the features of CDUCE can be added to a functional host language in a style reminiscent of XTATIC [see `http://www.cduce.org/ocaml.html`]. CQL (Benzaken et al., 2005) is an XML query language built on top of CDUCE. Compared to XQUERY, it emphasizes use of regular patterns over paths.

Another close cousin of XTATIC is Meijer, Schulte, and Bierman's C_ω language (previously called XEN) (Meijer et al., 2003; Meijer and Schulte, 2003; Bierman et al., 2005), an extension of $C^\sharp$ that integrates support for objects, relations, and XML. Some aspects of the C_ω language design are much more ambitious than XTATIC: in particular, the extensions to its type system (`sequence` and `choice` type constructors) are more tightly intertwined with the core object model—indeed, XML itself is simply a syntax for serialized object instances. In other respects, C_ω is more conservative than XTATIC: for example, its `choice` constructor is not a true least upper bound, and the subtype relation is defined by a conventional, semantically incomplete, collection of inference rules, while XTATIC's is given by a more straightforward (and, for the implementation, more demanding) "subtype = subset" construction.

XACT (Kirkegaard et al., 2004; Christensen et al., 2004) extends JAVA with XML processing, proposing another somewhat different programming idiom: the creation of XML values is done using XML *templates*, which are immutable first-class structures representing XML with named *gaps* that may be filled to obtain ordinary XML trees. XACT also features a static type system that guarantees that, at a given point in the program, a template statically satisfies a given DTD. XACT's implementation, developed independently and in parallel with XTATIC but driven by similar needs (supporting efficient sharing, etc.) and targeting a similar (object-oriented) run-time environment, has strong similarities to ours; in particular, lazy data structures are used to support efficient gap plugging.

XJ (Harren et al., 2005) is another extension of JAVA for native XML processing that emphasizes fidelity to the XML Schema and XPATH standards, for instance by having nominal subtyping (as opposed to the structural subtyping of the languages mentioned above). XJ is also one of the few XML processing languages that allow imperative modification of XML data. This feature, however, significantly weakens the safety guarantees offered by static typing: the updated tree must be re-validated dynamically, raising an exception if its new type fails to match static expectations. In keeping with its emphasis on standards and its imperative nature, XJ uses DOM for its run-time representation of XML data.

XOBE (Kempa and Linnemann, 2003) is a source to source compiler for an extension of JAVA. From a language design point of view, it is very similar to XTATIC, integrating XML with JAVA, taking a declarative style of tree processing, and providing a rich type system and subtyping relation based on regular expression types. The run-time representation, like XJ, relies on DOM. In contrast to XTATIC, XOBE uses XPATH rather than patterns to inspect XML values, and it has no provision for placing Java objects anywhere inside XML values.

SCALA, a general-purpose experimental web services language that compiles into JAVA bytecode, also has XML support (Emir et al., 2006).

Work also continues on XDUCE itself, including fully typed treatment of attributes *à la* RELAXNG (Hosoya and Murata, 2003) and parametric polymorphism (Hosoya et al., 2005a).

A survey paper by Møller and Schwartzbach (2005) offers an excellent overview of recent work on static typechecking for XML transformation languages, with detailed comparisons between a number of representative languages, including XDUCE and XACT.

In this review of related work, we have concentrated on comparisons with integrated language designs, in which XML processing features are combined with a general-purpose host language. By contrast, XSLT (Clark, 1999) is a widely used stand-alone XML processing language, which is also accessible via library APIs in most modern general-purpose languages, including JAVA and $C^\sharp$. A similar invocation mechanism can be used for XQuery. There are two well-known shortcomings of this API approach. First, no static checks of XML processing code are possible, since a host application generates it at run time and passes it to the API as a string. Second, the results of

processing are accessible to the host application only in a low-level form—usually as DOM structures or as XML data bindings. XTATIC addresses both of these problems by embedding an XML processing language into a host general-purpose language and by extending the host's data model to incorporate XML data. We will see in Chapter 6 an alternative approach for static type checking of XML processing based the satisfiability of XPATH queries.

One final point of comparison with XQUERY concerns concrete syntax for XML values. Readers familiar with XQUERY may have noted that we take "computed content" as the default and explicitly mark constant strings in XML values, while XQUERY takes constant pcdata as the default and indicates computed content by wrapping it in curly braces—e.g., `<a>3</a>` vs. `<a>{1 + 2}</a>`. At first glance, XQUERY's design is attractive, since it allows XML values to be cut and pasted into XQuery programs verbatim. However, it turns out not to work well in XTATIC, where we need concrete syntax not only for values but also for types. (XQuery does not: complex type expressions are supposed to be written down in separate files, in the completely different syntax of W3C Schema.) Making the opposite choice allows us to write `<a>B</>`, rather than `<a>{B}</>`, for the type of `<a>` elements whose content is described by the type named `B`.

Chapter 6

Efficient Static Analysis of XML Paths and Types

6.1 Introduction

The work described in this chapter is motivated by the need for efficient type checkers for XML-based programming languages where XML types and XPath queries are used as first class language constructs. In such settings, XPath decision problems in the presence of XML types such as DTDs or XML Schemas arise naturally. Examples of such decision problems include emptiness (whether an expression ever selects nodes), containment (whether the results of an expression are always included in the results of another one), overlap (whether two expressions select common nodes), and coverage (whether nodes selected by an expression are always contained in the union of the results selected by several other expressions).

XPath decision problems are not trivial in that they need to be checked on a possibly infinite quantification over trees. Another difficulty arises from the combination of upward and downward navigation on trees with recursion (Vardi, 1998).

The most basic decision problem for XPath is the test for emptiness of an expression (Benedikt et al., 2005). This test is important for optimization of host languages implementations. For instance, if one can decide at compile time that a query result is empty then subsequent bound computations can be ignored. Another basic decision problem is the XPath equivalence problem: whether or not two queries always return the same result. It is important for reformulation and optimization of an expression (Genevès and Vion-Dury, 2004), which aim at enforcing operational properties while preserving semantic equivalence (Levin and Pierce, 2005). The most essential problem for type-checking is XPath containment. It is required for the control-flow analysis of XSLT (Møller et al., 2005), for checking integrity constraints (Fallside and Walmsley, 2004), and for XML security (Fan et al., 2004).

The complexity of XPath decision problems heavily depends on the language features. Previous works (Schwentick, 2004; Benedikt et al., 2005) showed that including general comparisons of data values from an infinite domain may lead to undecidability. Therefore, we focus on a fragment of XPath that covers all features except counting (Dal Zilio et al., 2004) and data values.

In our approach to solve XPath decision problems, two issues need to be addressed. First, we identify the most appropriate logic with sufficient expressiveness to capture both regular tree types and our XPath fragment. Second, we solve efficiently the satisfiability problem which allows to test if a given formula of the logic admits a satisfying finite tree.

The essence of our results lives in a sub-logic of the alternation free modal μ-calculus

(AFMC) with converse, without greatest fixpoint, with some syntactic restrictions on formulas, and whose models are finite trees. We prove that XPath expressions and regular tree type formulas conform to these syntactic restrictions. Boolean closure is the key property for solving the containment problem (a logical implication). In order to obtain closure under negation, we prove that the least and greatest fixpoint operators collapse in a single fixpoint operator. Surprisingly, the translations of XML regular tree types and the large XPath fragment that we consider do not increase complexity since they are linear in the size of the corresponding formulas in the logic. The combination of these ingredients lead to our main result: a satisfiability algorithm for a logic for finite trees whose time complexity is a simple exponential of the size of a formula.

The decision procedure has been implemented in a system for solving XML decision problems such as XPath emptiness, containment, overlap, and coverage, both with or without XML type constraints. The system can be used as a component of static analyzers for programming languages manipulating XPath expressions and XML type annotations for both input and output.

The chapter is organized as follows. We present our data model, trees with focus, in Section 6.2. We then introduce our logic in Section 6.3. We present XPath and sketch its translation in our logic in Section 6.4. Our satisfiability algorithm is introduced and proven correct in Section 6.5, and details of the implementation are discussed in Section 6.6. Applications for type checking and some experimental results are described in Section 6.7. We study related work in Section 6.8 and conclude in Section 6.9.

6.2 Trees with Focus

In order to represent XML trees that are easy to navigate, we use *focused trees*, inspired by Huet's Zipper data structure (1997). Focused trees not only describe a tree but also its context: its previous siblings and its parent, including its parent context recursively. Exploring such a structure has the advantage to preserve all the information, which is quite useful when considering languages such as XPath that allow forward and backward axes of navigation.

Formally, we assume an alphabet Σ of labels, ranged over by σ. The syntax of our data model is as follows.

$$
\begin{array}{rcll}
t & ::= & \sigma[tl] & \text{tree} \\
tl & ::= & & \text{list of trees} \\
 & & \epsilon & \quad\text{empty list} \\
 & | & t :: tl & \quad\text{cons cell} \\
c & ::= & & \text{context} \\
 & & (tl, \mathit{Top}, tl) & \text{root of the tree} \\
 & | & (tl, c[\sigma], tl) & \text{context node} \\
f & ::= & (t, c) & \text{focused tree}
\end{array}
$$

A focused tree (t, c) is a pair consisting of a tree t and its context c. The context $(tl, c[\sigma], tl)$ comprises three components: a list of trees at the left of the current tree in reverse order (the first element of the list is the tree immediately to the left of the current tree), the context above the tree, and a list of trees at the right of the current tree. The context above the tree may be *Top* if the current tree is at the root, otherwise it is of the form $c[\sigma]$ where σ is the label of the enclosing element and c is the context in which the enclosing element occurs.

In order to deal with decision problems such as containment, we need to represent in a focused tree the place where the evaluation of a request was started. To this end, we use a *start mark*, often simply called "mark" in the following. We thus consider focused trees where a single tree or a single context node is marked, as in $\sigma^{\circledS}[tl]$ or $(tl, c[\sigma^{\circledS}], tl)$.

$\mathcal{L}_\mu \ni \phi, \psi$	$::=$		formula
		$\top$	true
	$\mid$	$\sigma \quad \mid \quad \neg\sigma$	atomic proposition (negated)
	$\mid$	$\textcircled{S} \quad \mid \quad \neg\textcircled{S}$	start proposition (negated)
	$\mid$	X	variable
	$\mid$	$\phi \vee \psi$	disjunction
	$\mid$	$\phi \wedge \psi$	conjunction
	$\mid$	$\langle a \rangle\, \phi \quad \mid \quad \neg \langle a \rangle \top$	existential (negated)
	$\mid$	$\mu \overline{X_i = \phi_i}$ in ψ	least n-ary fixpoint
	$\mid$	$\nu \overline{X_i = \phi_i}$ in ψ	greatest n-ary fixpoint

Figure 6.1: Logic formulas

When the presence of the mark is unknown, we write it as $\sigma^\circ[tl]$. We write $\mathcal{F}$ for the set of finite focused trees containing a single mark. The *name* of a focused tree is defined as $\mathtt{nm}(\sigma^\circ[tl], c) = \sigma$.

We now describe how to navigate focused trees, in binary style. There are four directions, or *modalities*, that can be followed: for a focused tree f, $f\,\langle\downarrow\rangle$ changes the focus to the first child of the current tree, $f\,\langle\rightarrow\rangle$ changes the focus to the next sibling of the current tree, $f\,\langle\uparrow\rangle$ changes the focus to the parent of the tree *if the current tree is a leftmost sibling*, and $f\,\langle\leftarrow\rangle$ changes the focus to the previous sibling.

Formally, we have:

$$
\begin{aligned}
(\sigma^\circ[t :: tl], c)\,\langle\downarrow\rangle &= (t, (\epsilon, c[\sigma^\circ], tl)) \\
(t, (tl_l, c[\sigma^\circ], t' :: tl_r))\,\langle\rightarrow\rangle &= (t', (t :: tl_l, c[\sigma^\circ], tl_r)) \\
(t, (\epsilon, c[\sigma^\circ], tl))\,\langle\uparrow\rangle &= (\sigma^\circ[t :: tl], c) \\
(t', (t :: tl_l, c[\sigma^\circ], tl_r))\,\langle\leftarrow\rangle &= (t, (tl_l, c[\sigma^\circ], t' :: tl_r))
\end{aligned}
$$

When the focused tree does not have the required shape, these operations are not defined.

6.3 The Logic

We introduce in this section the logic to which we translate XPath expressions and XML regular tree types. This logic is a sub-logic of the alternation free modal μ-calculus with converse. We also introduce a restriction on the formulas we consider and give an interpretation of formulas as sets of finite focused trees. We finally show that this restriction and this interpretation make the greatest and smallest fixpoint collapse, yielding a logic that is closed under negation.

Formulas

In the following, we use an overline bar to denote tuples. For instance, we write $\overline{X_i = \phi_i}$ for $(X_1 = \phi_1; X_2 = \phi_2; \ldots; X_n = \phi_n)$. Tuples of variables, such as $\overline{X_i}$, are often identified to sets.

In the following definitions, $a \in \{\downarrow, \rightarrow, \uparrow, \leftarrow\}$ are *programs*. Atomic propositions σ correspond to labels from Σ. We assume that $\overline{a}$ denotes the opposite direction from a ($\overleftarrow{\rightarrow} = \rightarrow$, etc). Formulas defined in Figure 6.1 include the truth predicate, atomic propositions (denoting the name of the tree in focus), start propositions (denoting the presence of the start mark), disjunction and conjunction of formulas, formulas under an existential (denoting the existence of a subtree satisfying the sub-formula), and least

$$
\begin{aligned}
[\![\top]\!]_V &= \mathcal{F} & [\![\sigma]\!]_V &= \{f \mid \mathtt{nm}(f) = \sigma\} \\
[\![X]\!]_V &= V(X) & [\![\neg\sigma]\!]_V &= \{f \mid \mathtt{nm}(f) \neq \sigma\} \\
[\![\phi \vee \psi]\!]_V &= [\![\phi]\!]_V \cup [\![\psi]\!]_V & [\![\textcircled{S}]\!]_V &= \{f \mid f = (\sigma^{\textcircled{S}}[tl], c)\} \\
[\![\phi \wedge \psi]\!]_V &= [\![\phi]\!]_V \cap [\![\psi]\!]_V & [\![\neg\textcircled{S}]\!]_V &= \{f \mid f = (\sigma[tl], c)\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\langle a\rangle\, \phi]\!]_V &= \{f\,\langle \overline{a}\rangle \mid f \in [\![\phi]\!]_V \wedge f\,\langle \overline{a}\rangle \text{ is defined}\} \\
[\![\neg\,\langle a\rangle\, \top]\!]_V &= \{f \mid f\,\langle a\rangle \text{ is undefined}\} \\
[\![\mu\overline{X_i = \phi_i} \text{ in } \psi]\!]_V &= \text{let } \overline{T_i} = \left(\bigcap \left\{\overline{T_i \subseteq \mathcal{F}} \mid \overline{[\![\phi_i]\!]_{V[\overline{T_i/X_i}]} \subseteq T_i}\right\}\right)_i \text{ in } [\![\psi]\!]_{V[\overline{T_i/X_i}]} \\
[\![\nu\overline{X_i = \phi_i} \text{ in } \psi]\!]_V &= \text{let } \overline{T_i} = \left(\bigcup \left\{\overline{T_i \subseteq \mathcal{F}} \mid \overline{T_i \subseteq [\![\phi_i]\!]_{V[\overline{T_i/X_i}]}}\right\}\right)_i \text{ in } [\![\psi]\!]_{V[\overline{T_i/X_i}]}
\end{aligned}
$$

Figure 6.2: Interpretation of formulas

and greatest n-ary fixpoints. We chose to include a n-ary version of fixpoints because regular types are often defined as a set of mutually recursive definitions, making their translation in our logic more direct and succinct. We write "$\mu X.\, \phi$" for "$\mu\overline{X = \phi}$ in ϕ".

Model

We define in Figure 6.2 an interpretation of our formulas as subsets of $\mathcal{F}$, the set of finite focused trees with a single start mark. The interpretation of the n-ary fixpoints first compute the smallest or largest interpretation for each ϕ_i, bind the resulting sets T_i to the variables X_i, then returns the interpretation of ψ.

To illustrate the interpretation of fixpoints, take the formulas $\phi = \mu X.\, \langle\downarrow\rangle\, X \vee \langle\uparrow\rangle\, X$ and $\psi = \nu X.\, \langle\downarrow\rangle\, X \vee \langle\uparrow\rangle\, X$, which expand to $\mu\overline{X = \langle\downarrow\rangle\, X \vee \langle\uparrow\rangle\, X}$ in $\langle\downarrow\rangle\, X \vee \langle\uparrow\rangle\, X$ and $\nu\overline{X = \langle\downarrow\rangle\, X \vee \langle\uparrow\rangle\, X}$ in $\langle\downarrow\rangle\, X \vee \langle\uparrow\rangle\, X$. One easily shows that $[\![\phi]\!] = \emptyset$ (there is no base case, thus the smallest fixpoint is the empty one), and that $[\![\psi]\!]$ is the set of every focused tree with at least two nodes, one being the parent of the other.

Cycle-Free Formulas

As just shown, the smallest and greatest fixpoints do not coincide. To make them collapse, we require formulas to be *cycle-free*. Intuitively, cycle-free formulas do not have a sequence of a modality and its inverse under a fixpoint. To make this notion precise, we first define the *unfolding* of a formula, which basically amount to building the set of every finite formula resulting from the unfolding of fixpoints.

Definition 6.3.1 (Unfolding of a formula)**.** *The* unfolding *of a formula ϕ is the set $unf(\phi)$ inductively defined as*

$$
\begin{aligned}
unf(\phi) &= \{\phi\} \textit{ for } \phi = \top, \sigma, \neg\sigma, \textcircled{S}, \neg\textcircled{S}, X, \neg\,\langle a\rangle\, \top \\
unf(\phi \vee \psi) &= \{\phi' \vee \psi' \mid \phi' \in unf(\phi), \psi' \in unf(\psi)\} \\
unf(\phi \wedge \psi) &= \{\phi' \wedge \psi' \mid \phi' \in unf(\phi), \psi' \in unf(\psi)\} \\
unf(\langle a\rangle\, \phi) &= \{\langle a\rangle\, \phi' \mid \phi' \in unf(\phi)\} \\
unf(\mu\overline{X_i = \phi_i} \textit{ in } \psi) &= unf(\psi\{\{\mu\overline{X_i = \phi_i} \textit{ in } \phi_i/X_i\}\}) \cup \{\mu\overline{X_i = \phi_i} \textit{ in } \psi\} \\
unf(\nu\overline{X_i = \phi_i} \textit{ in } \psi) &= unf(\psi\{\{\nu\overline{X_i = \phi_i} \textit{ in } \phi_i/X_i\}\}) \cup \{\nu\overline{X_i = \phi_i} \textit{ in } \psi\}
\end{aligned}
$$

Given a formula ϕ, the set of its paths $\mathcal{P}(\phi)$ is the set of sequential chains of modalities contained in the formula. Writing ϵ for the empty path, we have the following.

$$\begin{aligned}
\mathcal{P}(\langle a\rangle\,\phi) &= \{\langle a\rangle\, p \mid p \in \mathcal{P}(\phi)\}\\
\mathcal{P}(\phi \vee \psi) &= \mathcal{P}(\phi) \cup \mathcal{P}(\psi)\\
\mathcal{P}(\phi \wedge \psi) &= \mathcal{P}(\phi) \cup \mathcal{P}(\psi)\\
\mathcal{P}(\phi) &= \epsilon \qquad \text{otherwise}
\end{aligned}$$

A *modality cycle* in a path is a sub-sequence of the form $\langle a\rangle\,\langle\overline{a}\rangle$. We new define *cycle-free formulas* as formulas for which there is a bound in the number of modality cycles of their paths, independent on the unfolding.

Definition 6.3.2 (Cycle-free formula)**.** *A formula ϕ is cycle-free iff there exists an integer n such that for any unfolding $\psi \in unf(\phi)$, for any path $p \in \mathcal{P}(\psi)$, the number of modality cycles in p is strictly smaller than n.*

For instance, the formula "$\overline{\mu X = \langle 1\rangle\,(\top \vee \langle\overline{1}\rangle\, X)}$ in X" is not cycle free: for any integer n, there is an unfolding of the formula such that a path with n modality cycles exists. Similarly, the formulas ϕ and ψ above are also not cycle free. On the other hand, the formula "$\overline{\mu X = \langle 1\rangle\,(X \vee Y),\ Y = \langle\overline{1}\rangle\,(Y \vee \top)}$ in X" is cycle free: there is at most one modality cycle for each path, independently of the number of unfoldings.

Cycle-free formulas enjoy the following crucial property: given any tree in the model of a formula ϕ, there is a finite unfolding (i.e., a formula $\psi \in unf(\phi)$ where we replace the remaining fixpoints with a "false" formula, which we write $unf_f(\phi)$) such that the tree belongs to its model.

Lemma 6.3.3. *Let ϕ a cycle-free formula, then we have the following.*

$$[\![\phi]\!]_V = \bigcup_{\psi \in unf_f(\phi)} [\![\psi]\!]_V$$

This lemma states that for every tree, a fixpoint will be used only a finite number of times, thus the greatest and smallest fixpoints collapse. Additional details may be found in Genevès et al. (2007).

In the rest of the chapter, we thus only consider least fixpoints. An important consequence of the collapse of the fixpoints is that the logic restricted in this way is closed under negation using De Morgan's dualities, extended to eventualities and fixpoints as follows:

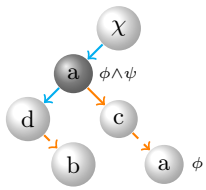
$$\begin{aligned}
\neg\,\langle a\rangle\,\phi &= \neg\,\langle a\rangle\,\top \vee \langle a\rangle\,\neg\phi\\
\neg\mu\overline{X_i = \phi_i} \text{ in } \psi &= \overline{\mu X_i = \neg\phi_i\{\overline{\{X_i/\neg X_i\}}\}} \text{ in } \neg\psi\{\overline{\{X_i/\neg X_i\}}\}
\end{aligned}$$

6.4 XPath and Regular Tree Languages

XPath (XPath 1.0) is a powerful language for navigating in XML documents and selecting sets of nodes matching a predicate. In their simplest form, XPath expressions look like "directory navigation paths". For example, the XPath expression

/child::book/child::chapter/child::section

navigates from the root of a document (designated by the leading "/") through the top-level "book" node to its "chapter" child nodes and on to its child nodes named "section". The result of the evaluation of the entire expression is the set of all the "section" nodes



Translated Query: child::a[child::b]

$$\underbrace{a \wedge (\mu X.\, \langle\uparrow\rangle\, (\chi \wedge \textcircled{s}) \vee \langle\leftarrow\rangle\, X)}_{\phi} \wedge \underbrace{\langle\downarrow\rangle\, \mu Y.\, b \vee \langle\rightarrow\rangle\, Y}_{\psi}$$

Figure 6.3: XPath Translation Example.

that can be reached in this manner. The situation becomes more interesting when combined with XPath's capability of searching along "axes" other than "child". For instance, one may use the "preceding-sibling" axis for navigating backward through nodes of the same parent, or the "ancestor" axis for navigating upward recursively. Furthermore, at each step in the navigation the selected nodes can be filtered using qualifiers: boolean expression between brackets that can test the existence or absence of paths.

We consider a large XPath fragment covering all major features of the XPath 1.0 recommendation (XPath 1.0) except counting and comparisons between data values. We give a detailed translation of this fragment into our logic in Genevès et al. (2007), where we also show it is correct and linear in the size of the expression. We illustrate it here through an example.

Figure 6.3 depicts the translation of the XPath expression "child::a[child::b]". This expression selects all "a" child nodes of a given context which have at least one "b" child. The translated $\mathcal{L}_\mu$ formula holds for "a" nodes which are selected by the expression. The first part of the translated formula, ϕ, corresponds to the step "child::a" which selects candidates "a" nodes. Since it corresponds to the selected node, the formula navigates backward from it to ensure that the starting node was the parent. The second part, ψ, navigates downward in the subtrees of these candidate nodes to verify that they have at least one immediate "b" child. Every translation has this structure: navigate backward to check that the "path" expression that selects the node has been followed, and navigate forward to test the (optional) predicate from the selected node.

Note that without converse programs we would have been unable to differentiate selected nodes from nodes whose existence is tested: we must state properties on both the ancestors and the descendants of the selected node. Equipping the $\mathcal{L}_\mu$ logic with both forward and converse programs is therefore crucial for supporting XPath.[1] Logics without converse programs may only be used for solving XPath emptiness but cannot be used for solving other decision problems such as containment efficiently.

Embedding Regular Tree Languages

Several formalisms exist for describing types of XML documents (e.g. DTD, XML Schema, Relax NG). In this chapter we embed regular tree types into $\mathcal{L}_\mu$. Regular tree types gather most of the schemas occuring in practice (Murata et al., 2005). We rely on a straightforward isomorphism between unranked regular tree types and binary regular tree types (Hosoya et al., 2005b). Assuming a countably infinite set of type variables

[1] It is possible to eliminate upward navigation at the XPath level but it is well known that such XPath rewriting techniques cause exponential blow-ups of expression sizes (Olteanu et al., 2002).

ranged over by X, binary regular tree type expressions are defined as follows:

$$
\begin{array}{rcll}
\mathcal{L}_{BT} \ni T & ::= & & \text{tree type expression} \\
& & \emptyset & \text{empty set} \\
& | & \epsilon & \text{leaf} \\
& | & T_1 \mid T_2 & \text{union} \\
& | & \sigma(X_1, X_2) & \text{label} \\
& | & \text{let } \overline{X_i.\, T_i} \text{ in } T & \text{binder}
\end{array}
$$

We refer the reader to (Hosoya et al., 2005b) for the denotational semantics of regular tree languages, and directly introduce their translation into $\mathcal{L}_\mu$:

$$
\begin{aligned}
[\![\cdot]\!] &: \mathcal{L}_{BT} \to \mathcal{L}_\mu \\
[\![T]\!] &= \sigma \wedge \neg\sigma \quad \text{for} \quad T = \emptyset, \epsilon \\
[\![T_1 \mid T_2]\!] &= [\![T_1]\!] \vee [\![T_2]\!] \\
[\![\sigma(X_1, X_2)]\!] &= \sigma \wedge \mathit{succ}_\downarrow(X_1) \wedge \mathit{succ}_\rightarrow(X_2) \\
[\![\text{let } \overline{X_i.\, T_i} \text{ in } T]\!] &= \mu \overline{X_i = [\![T_i]\!]} \text{ in } [\![T]\!]
\end{aligned}
$$

where we use the formula $\sigma \wedge \neg\sigma$ as "false", and the function $\mathit{succ}.(\cdot)$ takes care of setting the type frontier:

$$
\mathit{succ}_\alpha(X) \quad = \quad \begin{cases} \neg \langle\alpha\rangle \top & \textit{if } X \textit{ is bound to } \epsilon \\ \neg \langle\alpha\rangle \top \vee \langle\alpha\rangle X & \textit{if } \mathit{nullable}(X) \\ \langle\alpha\rangle X & \textit{if not } \mathit{nullable}(X) \end{cases}
$$

according to the predicate $\mathit{nullable}(X)$ which indicates whether the type $T \neq \epsilon$ bound to X contains the empty tree.

Note that the translation of a regular tree type uses only downward modalities since it describes the allowed subtrees at a given context. No additional restriction is imposed on the context from which the type definition starts. In particular, navigation is allowed in the upward direction so that we can support type constraints for which we have only partial knowledge in a given direction. However, when we know the position of the root, conditions similar to those of absolute paths are added in the form of additional formulas describing the position that need to be satisfied. This is particularly useful when a regular type is used by an XPath expression that starts its navigation at the root ($/p$) since the path will not go above the root of the type (by adding the restriction $\mu Z. \neg \langle\uparrow\rangle \top \vee \langle\leftarrow\rangle Z$).

On the other hand, if the type is compared with another type (typically to check inclusion of the result of an XPath expression in this type), then there is no restriction as to where the root of the type is (our translation does not impose the chosen node to be at the root). This is particularly useful since an XPath expression usually returns a set of nodes deep in the tree which we may compare to this partially defined type.

6.5 Satisfiability-Testing Algorithm

In this section we present our algorithm, show that it is sound and complete, and prove a time complexity boundary. To check a formula ϕ, our algorithm builds satisfiable formulas out of some subformulas (and their negation) of ϕ, then checks whether ϕ was produced. We first describe how to extract the subformulas from ϕ.

Preliminary Definitions

We define the *Fisher-Ladner closure* $\mathrm{cl}(\psi)$ of a formula ψ as the set of all subformulas of ψ where fixpoint formulas are additionally unwound once. Specifically, we define the relation $\rightarrow_e \subseteq \mathcal{L}_\mu \times \mathcal{L}_\mu$ as the least relation that satisfies the following:

- $\phi_1 \wedge \phi_2 \rightarrow_e \phi_1$, $\phi_1 \wedge \phi_2 \rightarrow_e \phi_2$
- $\phi_1 \vee \phi_2 \rightarrow_e \phi_1$, $\phi_1 \vee \phi_2 \rightarrow_e \phi_2$
- $\langle a \rangle\, \phi' \rightarrow_e \phi'$
- $\mu\overline{X_i = \phi_i}$ in $\psi \rightarrow_e \psi\{\overline{\{\mu\overline{X_i = \phi_i} \text{ in } X_i/X_i\}}}$

The closure $\mathrm{cl}(\psi)$ is the smallest set S that contains ψ and is closed under the relation $\rightarrow_e$, i.e. if $\phi_1 \in S$ and $\phi_1 \rightarrow_e \phi_2$ then $\phi_2 \in S$.

We call $\Sigma(\psi)$ the set of atomic propositions σ used in ψ along with another name, σ_x, that does not occur in ψ to represent atomic propositions not occurring in ψ.

We define $\mathrm{cl}^*(\psi) = \mathrm{cl}(\psi) \cup \{\neg\phi \mid \phi \in \mathrm{cl}(\psi)\}$. Every formula $\phi \in \mathrm{cl}^*(\psi)$ can be seen as a Boolean combination of formulas of a set called the Lean of ψ, inspired from Pan et al. (2006). We note this set $\mathrm{Lean}(\psi)$ and define it as follows:

$$\mathrm{Lean}(\psi) \;=\; \{\langle a \rangle\, \top \mid a \in \{\downarrow, \rightarrow, \uparrow, \leftarrow\}\} \cup \Sigma(\psi) \cup \{Ⓢ\} \cup \{\langle a \rangle\, \phi \mid \langle a \rangle\, \phi \in \mathrm{cl}(\psi)\}$$

A *ψ-type* (or simply a "*type*") (Hintikka set in the temporal logic literature) is a set $t \subseteq \mathrm{Lean}(\psi)$ such that:

- $\forall\, \langle a \rangle\, \phi \in \mathrm{Lean}(\psi), \langle a \rangle\, \phi \in t \Rightarrow \langle a \rangle\, \top \in t$ (modal consistency);
- $\langle \uparrow \rangle\, \top \notin t \vee \langle \leftarrow \rangle\, \top \notin t$ (a tree node cannot be both a first child and a second child);
- exactly one atomic proposition $\sigma \in t$ (XML labeling); we use the function $\sigma(t)$ to return the atomic proposition of a type t;
- Ⓢ may belong to t.

We call $\mathrm{Types}(\psi)$ the set of ψ-types. For a ψ-type t, the *complement* of t is the set $\mathrm{Lean}(\psi) \setminus t$. A type determines a truth assignment of every formula in $\mathrm{cl}^*(\psi)$, which we write $\dot{\in}$. We define a compatibility relation between types to state that two types are related according to a modality.

Definition 6.5.1 (Compatibility relation)**.** *Two types t and t' are* compatible *under $a \in \{\downarrow, \rightarrow\}$, written $\Delta_a(t, t')$, iff*

$$\forall\, \langle a \rangle\, \phi \in \mathit{Lean}(\psi), \langle a \rangle\, \phi \in t \Leftrightarrow \phi \,\dot{\in}\, t'$$
$$\forall\, \langle \overline{a} \rangle\, \phi \in \mathit{Lean}(\psi), \langle \overline{a} \rangle\, \phi \in t' \Leftrightarrow \phi \,\dot{\in}\, t$$

The Algorithm

The algorithm works on sets of triples of the form $(t, w_\downarrow, w_\rightarrow)$ where t is a type, and $w_\downarrow$ and $w_\rightarrow$ are sets of types which represent every witness for t according to relations $\Delta_\downarrow(t, \cdot)$ and $\Delta_\rightarrow(t, \cdot)$.

The algorithm proceeds in a bottom-up approach, repeatedly adding new triples until a satisfying model is found (i.e., a triple whose first component is a type implying the formula), or until no more triple can be added. Each iteration of the algorithm builds types representing deeper trees (in the $\downarrow$ and $\rightarrow$ direction) with pending backward modalities that will be fulfilled at later iterations. Types with no backward modalities are satisfiable, and if such a type implies the formula being tested, then it is satisfiable.

$$
\begin{aligned}
\mathtt{Upd}(X) = X \cup & \left\{ (t, \mathtt{w}_{\downarrow}(t, X^{\times}), \mathtt{w}_{\rightarrow}(t, X^{\times})) \mid \begin{array}{l} \circledS \notin t \subseteq \mathrm{Types}(\psi) \\ \wedge \langle\downarrow\rangle \top \in t \Rightarrow \mathtt{w}_{\downarrow}(t, X^{\times}) \neq \emptyset \\ \wedge \langle\rightarrow\rangle \top \in t \Rightarrow \mathtt{w}_{\rightarrow}(t, X^{\times}) \neq \emptyset \end{array} \right\} \\
\cup & \left\{ (t, \mathtt{w}_{\downarrow}(t, X^{\times}), \mathtt{w}_{\rightarrow}(t, X^{\times}))^{\circledS} \mid \begin{array}{l} \circledS \in t \subseteq \mathrm{Types}(\psi) \\ \wedge \langle\downarrow\rangle \top \in t \Rightarrow \mathtt{w}_{\downarrow}(t, X^{\times}) \neq \emptyset \\ \wedge \langle\rightarrow\rangle \top \in t \Rightarrow \mathtt{w}_{\rightarrow}(t, X^{\times}) \neq \emptyset \end{array} \right\} \\
\cup & \left\{ (t, \mathtt{w}_{\downarrow}(t, X^{\circledS}), \mathtt{w}_{\rightarrow}(t, X^{\times}))^{\circledS} \mid \begin{array}{l} \circledS \notin t \subseteq \mathrm{Types}(\psi) \\ \wedge \langle\downarrow\rangle \top \in t \Rightarrow \mathtt{w}_{\downarrow}(t, X^{\circledS}) \neq \emptyset \\ \wedge \langle\rightarrow\rangle \top \in t \Rightarrow \mathtt{w}_{\rightarrow}(t, X^{\times}) \neq \emptyset \end{array} \right\} \\
\cup & \left\{ (t, \mathtt{w}_{\downarrow}(t, X^{\times}), \mathtt{w}_{\rightarrow}(t, X^{\circledS}))^{\circledS} \mid \begin{array}{l} \circledS \notin t \subseteq \mathrm{Types}(\psi) \\ \wedge \langle\downarrow\rangle \top \in t \Rightarrow \mathtt{w}_{\downarrow}(t, X^{\times}) \neq \emptyset \\ \wedge \langle\rightarrow\rangle \top \in t \Rightarrow \mathtt{w}_{\rightarrow}(t, X^{\circledS}) \neq \emptyset \end{array} \right\}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{w}_a(t, X) &= \{\mathtt{type}(x) \mid x \in X \wedge \langle\overline{a}\rangle \top \in \mathtt{type}(x) \wedge \Delta_a(t, \mathtt{type}(x))\} \\
\mathtt{type}((t, w_{\downarrow}, w_{\rightarrow})) &= t \\
\mathtt{FinalCheck}(\psi, X) &= \exists x \in X, \mathtt{dsat}(x, \psi) \wedge \forall a \in \{\uparrow, \leftarrow\}, \langle a\rangle \top \notin \mathtt{type}(x) \\
\mathtt{dsat}((t, w_{\downarrow}, w_{\rightarrow}), \psi) &= \psi \mathrel{\dot{\in}} t \vee \exists x', \mathtt{dsat}(x', \psi) \wedge (x' \in w_{\downarrow} \vee x' \in w_{\rightarrow}) \\
X^{\circledS} &= \left\{x \in X \mid x = (_, _, _)^{\circledS}\right\} \\
X^{\times} &= \{x \in X \mid x = (_, _, _)\}
\end{aligned}
$$

Figure 6.4: Operations used by the Algorithm.

The main iteration is as follows:

$X \leftarrow \emptyset$
repeat
  $X' \leftarrow X$
  $X \leftarrow \mathtt{Upd}(X')$
  if $\mathtt{FinalCheck}(\psi, X)$ **then**
    return "ψ is satisfiable"
until $X = X'$
return "ψ is unsatisfiable"

where $X \subseteq \mathrm{Types}(\psi) \times \wp(\mathrm{Types}(\psi)) \times \wp(\mathrm{Types}(\psi))$ and the update operation $\mathtt{Upd}(\cdot)$ and success check operation $\mathtt{FinalCheck}(\cdot, \cdot)$ are defined on Figure 6.4. The update operation requires four almost identical cases to ensure that the optional mark remains *unique*. The first case corresponds to the absence of the mark, the second case to the presence of the mark at the top level, the third case to the presence of the mark deeper in the first child, and the last case to the presence of the mark deeper in the second child.

At each step of the algorithm, $\mathtt{FinalCheck}(\cdot, \cdot)$ verifies whether the tested formula is implied by newly added types without pending (unproved) backward modalities, so that the algorithm may terminate as soon as a satisfying tree is found.

We note X^i the set of triples and T^i the set of types after i iterations: $T^i = \left\{\mathtt{type}(x) \mid x \in X^i\right\}$. Note that T^{i+1} is the set of types for which at least one witness belongs to T^i.

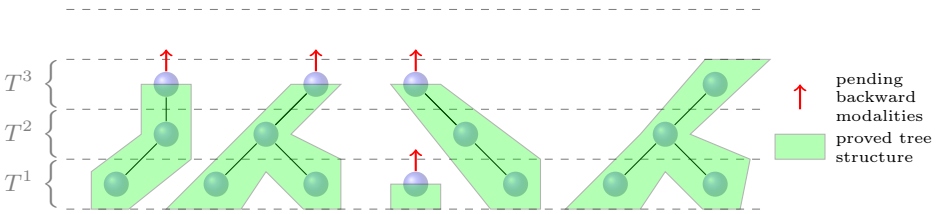
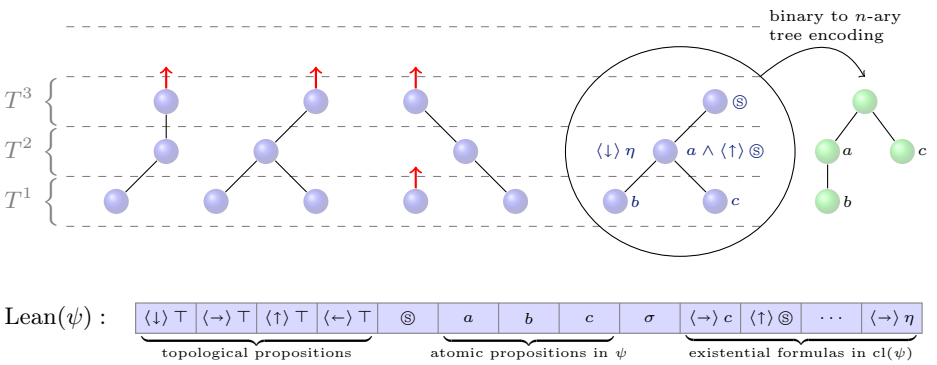


Figure 6.5: Algorithm's principle: progressive bottom-up reasoning.



$$\psi = \phi_1 \wedge \neg\phi_2$$
$$\phi_1 = E^{\rightarrow}[\![e_1]\!]_\top = a \wedge (\mu Y.\,\langle\rightarrow\rangle\,(c \wedge \theta) \vee \langle\rightarrow\rangle\,Y) \wedge \langle\downarrow\rangle\,\eta$$
$$\phi_2 = E^{\rightarrow}[\![e_2]\!]_\top = c \wedge \underbrace{\mu X.\,\langle\uparrow\rangle\,Ⓢ \vee \langle\leftarrow\rangle\,X}_{\theta} \wedge \langle\downarrow\rangle\,\underbrace{\mu Z.\,b \vee \langle\rightarrow\rangle\,Z}_{\eta}$$

$$e_1 = \text{child::}c/\text{preceding-sibling::}a[\text{child::}b]$$
$$e_2 = \text{child::}c[\text{child::}b]$$

Figure 6.6: Run of the algorithm for a sample XPath containment problem: $e_1 \overset{?}{\subseteq} e_2$.

Example Run of the Algorithm

In a sense, the algorithm performs a kind of progressive bottom-up reasoning while ensuring partial (forward) satisfiability of subformulas, as illustrated by Figure 6.5.

More specifically, Figure 6.6 illustrates a run of the algorithm for checking whether the XPath query child::c/preceding-sibling::$a[b]$ is contained in the XPath query child::$c[b]$. These expressions are first compiled into the logic. For the second translation, the final focus of the tree is a node named c. As we reach it going through a "child" step, formula θ ensures that the parent node is the starting node. Finally, from the final focus of the tree, formula η tests that a child named b is present. As concerns the first formula, the final focus of the tree is on a node named a. We get there by a "preceding-sibling" step from a c node, hence we need to make sure that there is a following sibling named c (this is the recursion on Y). Once this c node has been found, it must be made sure that c was reached by a "child" step from the start of the query, using the same formula

θ as before. Finally, going back to the final focus in the tree, we need to check there is a child named b, using again the formula η. Note that the start mark is crucial in this containment case: it ensures that when both formulas are combined, the XPath expressions start from the same context.

From the formulas ϕ_1 and ϕ_2 corresponding to each XPath query, we build a containment formula $\psi = \phi_1 \wedge \neg\phi_2$ (the negated implication). If this formula is unsatisfiable, then the first XPath expression is contained in the second one. $\text{Lean}(\psi)$ is then computed, and the fixpoint computation starts: the set of types T^1 contains all possible leaves. Each type added in T^i ($i \geq 2$) requires at least one witness type found in T^{i-1} (else it would have been added at some previous step $j < i$). In this example, a satisfying binary tree of depth 3 is found, therefore the algorithm stops just after computing T^3. The first XPath query is not contained in the second one: a counter-example tree is provided to the user (see Figure 6.6).

Correctness and Complexity

In this section we state the correctness of the satisfiability testing algorithm, and show that its time complexity is $2^{O(|\text{Lean}(\psi)|)}$. More details may be found in Genevès et al. (2007).

Theorem 6.5.2 (Correctness)**.** *The algorithm decides satisfiability of $\mathcal{L}_\mu$ formulas over finite focused trees.*

This result depends on the following two lemmas.

Lemma 6.5.3 (Soundness)**.** *Let T be the result set of the algorithm. For any type $t \in T$ and any ϕ such that $\phi \dot{\in} t$, then $[\![\phi]\!]_\emptyset \neq \emptyset$.*

Lemma 6.5.4 (Completeness)**.** *For a cycle-free closed formula $\phi \in \mathcal{L}_\mu$, if $[\![\phi]\!]_\emptyset \neq \emptyset$ then the algorithm terminates with a set of triples X such that* `FinalCheck`(ϕ, X).

We now present one of the main contributions of this chapter: the complexity of our algorithm is $2^{O(n)}$ where n is the formula size. It is well-known that $\text{cl}(\psi)$ is a finite set and its size is linear with respect to the size of ψ (i.e., the number of operators and propositional variables appearing in ψ) (Kozen, 1983). Therefore $|\text{Lean}(\psi)|$ is also trivially linear with respect to the size of ψ.[2]

Theorem 6.5.5 (Complexity)**.** *For $\psi \in \mathcal{L}_\mu$ the satisfiability problem $[\![\psi]\!]_\emptyset \neq \emptyset$ is decidable in time $2^{O(n)}$ where $n = |Lean(\psi)|$.*

6.6 Implementation Techniques

Our implementation relies on a symbolic representation and manipulation of sets of ψ-types using Binary Decision Diagrams (BDDs) (Bryant, 1986). BDDs provide a canonical representation of Boolean functions. Experience has shown that this representation is very compact for very large Boolean functions. Their effectiveness is notably well known in the area of formal verification of systems (Edmund M. Clarke et al., 1999). Details can be found in Genevès et al. (2007)

6.7 Typing Applications and Experimental Results

In this section we describe applications of our satisfiability algorithm, in the form of static XPath decisions problems, and provide some experimental results.

[2]The acute reader may notice that for large formulas, $|\text{Lean}(\psi)|$ is usually smaller than the size of ψ since disjunctions, conjunctions, and negations are not members of $\text{Lean}(\psi)$.

e_1 $/a[.//b[c/*//d]/b[c//d]/b[c/d]]$
e_2 $/a[.//b[c/*//d]/b[c/d]]$

e_3 a/b//c/foll-sibling::d/e
e_4 a/b//d[prec-sibling::c]/e
e_5 a/c/following::d/e
e_6 a/b[//c]/following::d/e ∩ a/d[preceding::c]/e

e_7 *//switch[ancestor::head]//seq//audio[prec-sibling::video]

e_8 descendant::a[ancestor::a]
e_9 /descendant::*
e_{10} html/(head | body)
e_{11} html/head/descendant::*
e_{12} html/body/descendant::*

Figure 6.7: XPath Expressions Used in Experiments.

DTD	Symbols	Binary Type Variables
SMIL 1.0	19	11
XHTML 1.0 Strict	77	325

Table 6.1: Types Used in Experiments.

Typing Applications

For XPath expressions $e_1, \ldots, e_n$, we can formulate several decision problems in the presence of XML type expressions $T_1, \ldots, T_n$, where $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]}$ is the translation of XPath expression e_1 constrained to type T_1.

- XPath containment: $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge \neg E^{\rightarrow}[\![e_2]\!]_{[\![T_2]\!]}$ (if the formula is unsatisfiable then all nodes selected by e_1 under type constraint T_1 are selected by e_2 under type constraint T_2)
- XPath emptiness: $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]}$
- XPath overlap: $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge E^{\rightarrow}[\![e_2]\!]_{[\![T_2]\!]}$
- XPath coverage: $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge \bigwedge_{2 \leq i \leq n} \neg E^{\rightarrow}[\![e_i]\!]_{[\![T_i]\!]}$

Two problems are of special interest for XML type checking:

- Static type checking of an annotated XPath query:
  $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge \neg[\![T_2]\!]$ (if the formula is unsatisfiable then all nodes selected by e_1 under type constraint T_1 are included in the type T_2.)
- XPath equivalence under type constraints:
  $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge \neg E^{\rightarrow}[\![e_2]\!]_{[\![T_2]\!]}$ and $\neg E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge E^{\rightarrow}[\![e_2]\!]_{[\![T_2]\!]}$ (This test can be used to check that the nodes selected after a modification of a type T_1 by T_2 and an XPath expression e_1 by e_2 are the same, typically when an input type changes and the corresponding XPath query has to change as well.)

Experimental Results

As no third-party implementation we know of addresses reverse axes and recursion, we simply provide evidence that our approach is efficient. We carried out extensive

XPath Decision Problem	XML Type	Time (ms)
$e_1 \subseteq e_2$ and $e_2 \not\subseteq e_1$	none	353
$e_4 \subseteq e_3$ and $e_4 \subseteq e_3$	none	45
$e_6 \subseteq e_5$ and $e_5 \not\subseteq e_6$	none	41
e_7 is satisfiable	SMIL 1.0	157
e_8 is satisfiable	XHTML 1.0	2630
$e_9 \subseteq (e_{10} \cup e_{11} \cup e_{12})$	XHTML 1.0	2872

Table 6.2: Some Decision Problems and Corresponding Results.

tests,[3] and present here only a representative sample that includes the most complex language features such as recursive forward and backward axes, intersection, large and very recursive types with a reasonable alphabet size. The tests use XPath expressions shown on Figure 6.7 (where "//" is used as a shorthand for "/desc-or-self::*/") and XML types shown on Table 6.1. Table 6.2 presents some decision problems and corresponding performance results. Times reported in milliseconds correspond to the running time of the satisfiability solver without the (negligible) time spent for parsing and translating into $\mathcal{L}_\mu$.

The first XPath containment instance was first formulated in Miklau and Suciu (2004) as an example for which the proposed tree pattern homomorphism technique is incomplete. The e_8 example shows that the official XHTML DTD does not syntactically prohibit the nesting of anchors. For the XHTML case, we observe that the time needed is more important, but it remains practically relevant, especially for static analysis operations performed only at compile-time.

Online Implementation

The system has been implemented as a Java/JSP web application and interaction with the system is offered through a web user interface in a web browser. The tool, depicted in Figure 6.8 is available online from `http://wam.inrialpes.fr/xml`.

The user can either enter a formula through area (1) or select from pre-loaded analysis tasks offered in area (4). The level of details displayed by the solver can be adjusted in area (2) and makes it possible to inspect logical translations and statistics on problem size and the different operation costs. The results of the analysis are displayed in area (3) together with XML counter-examples.

6.8 Related Work

We address related work beyond the one described in Section 5.6,

The XPath containment problem has attracted a lot of research attention in the database community. The focus was given to the study of the impact of different XPath features on the containment complexity (see Schwentick (2004) for an overview). Specifically, Neven and Schwentick (2003) proves an EXPTIME upper-bound (in the presence of DTDs) of queries containing the "child" and "descendant" axes, and union of paths. The complexity of XPath satisfiability in the presence of DTDs also is extensively studied in Benedikt et al. (2005). From these results, we know that XPath containment with or without type constraints ranges from EXPTIME to undecidable.

Most formalisms used in the context of XML are related to one of the two logics used for unranked trees: First Order logic (FO), and Monadic Second Order logic (MSO). FO

[3]Experiments have been conducted with a JAVA implementation running on a Pentium 4, 3 Ghz, with 512Mb of RAM with Windows XP.

webinterface

http://wam.inrialpes.fr/web-solver/webinterface.html

XML Reasoning Solver Project

Home | Demo | Publications | Team

Enter your formula below:

(1)

```
select("descendant::switch[ancestor::head]/descendant::seq
/descendant::audio[preceding-sibling::video]")
```

Select from these examples:

- XPath Satisfiability #1
- XPath Satisfiability #2
- XPath Containment #2
- XPath Equivalence
- Sample mu-formula with data-values
- XHTML Type Evolution
- MathML Query Evolution

Look at the user manual

(4)

(2)

Advanced Options | Check Satisfiability | Execution completed.

- XML Attributes
- Show Lean
- Show Formula
- Formula Statistics

(3)

```
Formula parsed and compiled [total time: 1 ms].

Computing Relevant Closure...
Computed Relevant Closure [0 ms].
Computed Lean [0 ms].
Lean size is 23. It contains 16 eventualities and 7 symbols.
Computing Fixpoint.....[2 ms].

Formula is satisfiable [total time: 4 ms].
A satisfying finite binary tree model is [2 ms]:
head(switch(seq(video(#, audio), #), #), #)
In XML syntax:
<head xmlns:solver="http://wam.inrialpes.fr/xml" solver:context="true">
 <switch>
  <seq>
   <video/>
   <audio solver:target="true"/>
  </seq>
 </switch>
</head>
```

This online demo is a 100% Java implementation of the solver that runs inside a Tomcat servlet. It is based on a thread-safe re-implemention of a BDD package (JavaBDD). However, the performance of this package is very slow compared to what can be achieved with an off-line solver implementation with native BDDs. Ask us if you are interested in the high-speed off-line version of the solver.

Figure 6.8: Screenshot of the Solver Interface.

and relatives are frequently used for query languages since they nicely capture their navigational features (Barceló and Libkin, 2005). For query languages, Computational Tree Logic (CTL) (Clarke and Emerson, 1981) (which is equivalent to FO over tree structures) has been proposed (Miklau and Suciu, 2004; Marx, 2004a; Barceló and Libkin, 2005). In a attempt to reach more expressive power, the work found in Afanasiev et al. (2005) proposes a variant of Propositional Dynamic Logic (PDL) (Fischer and Ladner, 1979) with an EXPTIME complexity. MSO, specifically the weak monadic second-order logic of two successors (WS2S) (Thatcher and Wright, 1968; Doner, 1970), is one of the most expressive decidable logic used when both regular types and queries (Barceló and Libkin, 2005) are under consideration. WS2S satisfiability is known to be non-elementary. A drawback of the WS2S decision procedure is that it requires the full construction and complementation of tree automata.

Some temporal and fixpoint logics closely related to MSO have been introduced and allow to avoid explicit automata construction. The propositional modal μ-calculus introduced in Kozen (1983) has been shown to be as expressive as nondeterministic tree automata (Emerson and Jutla, 1991). Since it is trivially closed under negation, it constitutes a good alternative for studying MSO-related problems. Moreover, it has been extended with converse programs in Vardi (1998). The best known complexity for the resulting logic is obtained through reduction to the emptiness problem of alternating tree automaton which is in $2^{O(n^4 \cdot \log\ n)}$, where n corresponds to the length of a formula (Grädel et al., 2002). Unfortunately the logic lacks the finite model property. From Kupferman and Vardi (1999), we know that WS2S is exactly as expressive as the alternation-free fragment (AFMC) of the propositional modal μ-calculus. Furthermore, the AFMC subsumes all early logics such as CTL (Clarke and Emerson, 1981) and PDL (Fischer and Ladner, 1979) (see Barceló and Libkin (2005) for a complete survey on tree logics). In Marx (2004b), the author considers XPath equivalence under DTDs (local tree types) for which satisfiability is shown to be in EXPTIME.

The goal of the research presented so far is focused on establishing new theoretical properties and complexity bounds. Our research differs in that we seek precise complexity bounds, efficient implementation techniques, and concrete design that may be directly applied to the type checking of XPath queries under regular tree types.

In this line of research, some experimental results based on WS2S, through the Mona tool (Klarlund et al., 2001), have recently been reported for XPath containment (Genevès and Layaïda, 2007) and even for query evaluation (Inaba and Hosoya, 2006). However, for static analysis purposes, the explosiveness of the approach is very difficult to control due to the non-elementary complexity. Closer to our contribution, the recent work found in Tanabe et al. (2005) provides a decision procedure for the AFMC with converse whose time complexity is $2^{O(n \cdot \log\ n)}$. However, models of the logic are Kripke structures (infinite graphs). Enforcing the finite tree model property can be done at the syntactic level, as illustrated in the XML setting in Genevès and Layaïda (2006). Nevertheless, the drawback of this approach is that the AFMC decision procedure requires expensive cycle-detection for rejecting infinite derivation paths for least fixpoint formulas. Furthermore, there is a fundamental difference between this approach and the algorithm presented in this chapter. The algorithm used in Genevès and Layaïda (2006) actually computes a greatest fixpoint: it starts from all possible (graph) nodes and progressively removes all inconsistent nodes until a fixpoint is reached. Finally, if the fixpoint contains a satisfying (tree) structure then the formula is satisfiable. As a consequence, unlike the algorithm presented in this article, (1) the algorithm must always explore all nodes, and (2) it cannot terminate until full completion of the fixpoint computation (otherwise inconsistencies may remain). The present work shows how this can be avoided for finite trees. As a consequence, the resulting performance is much more attractive. In an earlier work on XML type checking, a logic for finite trees was presented (Tozawa, 2004), but the logic is not closed under negation.

In Colazzo et al. (2006), a technique is presented for statically ensuring correctness of paths. The approach only deals with emptiness of XPath expressions without reverse axes, whereas our approach solves the more general problem of containment, including reverse axes.

Recently, alternative approaches based on tree automata have caught up on our complexity (Calvanese et al., 2008; Libkin and Sirangelo, 2008; Calvanese et al., 2009), even if, to the best of our knowledge, none of these approaches have been implemented so far. More specifically, these alternative approaches propose new automata techniques for trees that are simpler when compared to sophisticated infinite-tree automata-theoretic techniques. The work of Calvanese et al. (2009) underlines the fact that implementations are difficult to obtain due to the use of Safra's determinization construction and parity games. They add that it is practically infeasible to apply the symbolic approach in the the infinite-tree setting and that well-formed trees and path expressions together with the closure of a node expression are behind such results. These arguments based on automata techniques are very similar to our earlier work operating directly on finite trees.

We remain convinced that our logical approach is superior, for the following reasons. First, using the modal logic natively in the finite case throughout this chapter keeps more uniformity between the proofs and the satisfiability algorithm. From there we obtain not only an implementation but an efficient one. In addition, bottom-up construction and cycle-freeness come naturally and show exactly why the whole approach is powerful. Second, automata-based approaches require using 2ATA (mainly introduced on infinite trees if we consider Vardi's work) which are quite involved. First, they make implementations out of reach due to complex determinization and parity games (see Calvanese et al. (2008) for the full details). Then a conversion to NTA (Non-deterministic tree automata) is needed for testing non-emptiness (this is explosive for large sets). In this case one still needs to enforce finiteness to test emptiness. Since it does not simplify the implementation, the only utility we see here is to shorten some proofs. In fact, neither of the papers Libkin and Sirangelo (2008); Calvanese et al. (2008) provide an implementation, and Calvanese et al. (2008) even remarks that a naive implementation of their technique would result in a blow-up in complexity, requiring the use of techniques very similar to what we have done.

6.9 Conclusion

The main result of this chapter is a sound and complete algorithm for the satisfiability of decision problems involving regular tree types and XPath queries with a tighter $2^{O(n)}$ complexity in the length of a formula. Our approach is based on a sub-logic of the alternation-free modal μ-calculus with converse for finite trees.

Our proof method reveals deep connections between this logic and XPath decision problems. First, the translations of XML regular tree types and a large XPath fragment are cycle-free and linear in the size of the corresponding formulas in the logic. Second, on finite trees, since both operators are equivalent, the logic with a single fixpoint operator is closed under negation. This allows to address key XPath decision problems such as containment. The current solver can also support conditional XPath proposed in Marx (2004b).

Finally, there are a number of interesting directions for further research that build on ideas developed here: extending XPath to restricted data values comparisons that preserves this complexity, for instance data values on a finite domain, and integrating related work on counting (Dal Zilio et al., 2004) to our logic. We also plan on continuing to improve the performance of our implementation.

Part III

Manipulating Programs

"There are objects so peculiar they were not to be believed."

—Tim Burton's Nightmare before Christmas

For the final leg of our journey, we consider a drastic change in the shape of the data manipulated. Instead of data being purely static and the object of manipulations, we now turn to the manipulation of *programs*. More precisely, we consider *higher-order* languages, languages that manipulate their own expressions, in a concurrent setting. While we studied type systems for such languages (Schmitt and Stefani, 2003), we concentrate in the following chapters on the question of *observational equivalence*.

Higher-order process calculi are calculi in which processes (more generally, values containing processes) can be communicated. Higher-order process calculi have been put forward in the early 1990s, with CHOCS (Thomsen, 1989) and Plain CHOCS (Thomsen, 1993), the Higher-Order π-calculus (Sangiorgi, 1992) (HOπ in the following), and others. The basic operators are usually those of CCS: parallel composition, input and output prefix, and restriction. Replication and recursion are often omitted as they can be encoded. However, the possibility of exchanging processes has strong consequences on semantics: in most higher-order process calculi, labeled transition systems must deal with higher-order substitutions and scope extrusion, and ordinary definitions of bisimulation and behavioral equivalences become unsatisfactory as they are over-discriminating. Higher-order, or process-passing, concurrency is often presented as an alternative paradigm to the first order, or name-passing, concurrency of the π-calculus for the description of mobile systems. Higher-order calculi are inspired by, and are formally closer to, the λ-calculus, whose basic computational step – β-reduction – involves term instantiation. As in the λ-calculus, a computational step in higher-order calculi results in the instantiation of a variable with a term, which is then copied as many times as there are occurrences of the variable, resulting in potentially larger terms. We study in the following chapters variants of HOπ where we vary the set of features they offer, mainly name restriction and passivation (the capture and manipulation of running processes).

The static analysis we focus on is *equivalence*. Equivalence is a very precise relation: given a notion of observations, there is no context (i.e., larger system) that may distinguish between two equivalent programs. Unfortunately, such precise notions are difficult to establish, as one needs to consider every context possible. We thus consider several techniques, variants of *bisimulations*, that characterize observational equivalence while being much simpler to prove. The best we can hope for is an algorithm to test for equivalence. We show in Chapter 7 that such an algorithm exists for a minimal higher order calculus called HO Core (which is HOπ without restriction). We also show that this calculus is not trivial, in the sense that it is Turing complete. This work was done in collaboration with Ivan Lanese, Jorge A. Pérez, and Davide Sangiorgi (Lanese et al., 2010b). We then turn to the question of adding passivation, then restriction, to HO Core. In Chapter 8, we show that HO Core with passivation still offers a tractable way of proving equivalence, through the use of *normal bisimulations*. Unfortunately, this is not the case when we add both passivation and name restriction. We then describe *context bisimulations* for HOπP (HOπ with passivation), and design a new proof technique to show it characterized barbed congruence in the weak case. This work was done with Sergueï Lenglet and Jean-Bernard Stefani (Lenglet et al., 2009b,a) and is the subject of Sergueï Lenglet's PhD dissertation (Lenglet, 2010).

Chapter 7

HO Core

7.1 Introduction

The goal of this chapter is to shed light on the expressiveness of higher-order process calculi, and related questions of decidability and of behavioral equivalence.

We consider a core calculus of Higher-Order processes (briefly HO Core), whose grammar is:

$$P ::= a(x).\, P \;\mid\; \overline{a}\langle P\rangle \;\mid\; P \parallel P \;\mid\; x \;\mid\; \mathbf{0}$$

An input prefixed process $a(x).\, P$ can receive on name (or channel) a a process that will be substituted in the place of x in the body P; an output message $\overline{a}\langle P\rangle$ can send P on a; parallel composition allows processes to interact. We can view the calculus as a kind of concurrent λ-calculus, where $a(x).\, P$ is a function, with formal parameter x and body P, located at a; and $\overline{a}\langle P\rangle$ is the argument for a function located at a. HO Core is *minimal*, in that only the operators strictly necessary to obtain higher-order communications are retained. For instance, continuations following output messages have been left out. More importantly, HO Core has no restriction operator. Thus all channels are global, and dynamic creation of new channels is impossible. This makes the absence of recursion/replication also relevant, as known encodings of fixed-point combinators in higher-order process calculi exploit the restriction operator to avoid harmful interferences (notably for nested recursion).

Even though HO Core is minimal, it remains non-trivial: in Section 7.3, we show that it is Turing complete, therefore its termination problem is undecidable, by exhibiting a deterministic encoding of Minsky machines (Minsky, 1967). The cornerstone of the encoding, counters that may be tested for zero, consist of nested higher-order outputs. Each register is made of two mutually recursive behaviors capable of spawning processes incrementing and decrementing its counter.

We then turn to the question of definability and decidability of bisimilarity. The definition of a satisfactory notion of bisimilarity is a hard problem for a higher-order process language, and the "term-copying" feature inherited from the λ-calculus can make the proof that bisimilarity is a congruence difficult. In ordinary bisimilarity, as in CCS, two processes are bisimilar if any action by one of them can be matched by an equal action from the other in such a way that the resulting derivatives are again bisimilar. The two matching actions must be syntactically *identical*. This condition is unacceptable in higher-order concurrency; for instance it breaks fundamental algebraic laws such as the commutativity of parallel composition. Alternative proposals of labeled bisimilarity for higher-order processes have been put forward. In *higher-order bisimilarity* (Thomsen, 1990; Sangiorgi, 1992), one requires bisimilarity, rather than identity, of the processes emitted in a higher-order output action. This weakening is natural for higher-order calculi and the bisimulation checks involved are simple. However, higher-order bisim-

ilarity is often over-discriminating as a behavioral equivalence (Sangiorgi, 1992), and basic properties, such as congruence, may be very hard to establish. *Context bisimilarity* (Sangiorgi, 1992; Jeffrey and Rathke, 2005) avoids the separation between the argument and the continuation of an output action, this continuation being either syntactically present or consisting of other processes running in parallel. To this end, it explicitly takes into account the context in which the emitted process is supposed to go. Context bisimilarity yields more satisfactory process equalities and coincides with contextual equivalence (i.e., barbed congruence). A drawback of this approach is the universal quantification over contexts in the clause for output actions, which can hinder its use in practice to check equivalences. *Normal bisimilarity* (Sangiorgi, 1992; Jeffrey and Rathke, 2005; Cao, 2006) is a simplification of context bisimilarity without universal quantifications in the output clause. The input clause is simpler too: normal bisimilarity can indeed be viewed as a form of *open bisimilarity* (Sangiorgi, 1994), where the formal parameter of an input is not substituted in the input clause, and free variables of terms are observable during the bisimulation game. However, the definition of the bisimilarity may depend on the operators in the calculus, and the correspondence with context bisimilarity may be hard to prove.

In Sections 7.4 and 7.5 we show that HO Core has a unique reasonable relation of strong bisimilarity: all the above forms (higher-order bisimilarity, context bisimilarity, normal bisimilarity, barbed congruence) coincide, and they also coincide with their asynchronous versions. Furthermore, we show that such a bisimilarity relation is decidable.

In the concurrency literature, there are examples of formalisms which are not Turing complete and where nevertheless (strong) bisimilarity is undecidable (e.g., Petri nets (Jančar, 1995) or lossy channel systems (Schnoebelen, 2001)). We are not aware however of examples of the opposite situation: formalisms that, as HO Core, are Turing complete but at the same time maintain decidability of bisimilarity. The situation in HO Core may indeed seem surprising, if not even contradictory: one is able to tell whether two processes are bisimilar, but in general one cannot tell whether the processes will terminate or even whether the sets of their τ-derivatives (the processes obtained via reductions) are finite or not. The crux to obtaining decidability is a further characterization of bisimilarity in HO Core, as a form of open bisimilarity, called IO bisimilarity, in which τ-transitions are ignored.

For an upper bound to the complexity of the bisimilarity problem, we can adapt Dovier et al.'s algorithm (2004) to infer that bisimilarity is decidable in time which is linear in the size of the (open and higher-order) transition system underlying IO bisimilarity. In general however, this transition system is exponential with respect to the size of the root process. We show in Section 7.6 that bisimilarity in HO Core can actually be decided in time that is polynomial with respect to the size of the initial pair of processes. We obtain this through an axiomatization of bisimilarity, where we adapt to a higher-order setting both Moller and Milner's unique decomposition of processes (1993) and Hirschkoff and Pous' axioms for a fragment of (finite) CCS (2008).

The decidability result for bisimilarity breaks down with the addition of restriction, as full recursion can then be faithfully encoded (the resulting calculus subsumes, e.g., CCS without relabeling). This however requires the ability of generating unboundedly many new names (for instance, when a process that contains restrictions is communicated and copied several times). In Section 7.7, we consider the addition of static restrictions to HO Core. Intuitively, this means allowing restrictions only as the outermost constructs, so that processes take the form $\boldsymbol{\nu} a_1 \ldots \boldsymbol{\nu} a_n\, P$ where the inner process P is restriction-free. Via an encoding of the Post correspondence problem (Post, 1946), we show that the addition of four static restrictions is sufficient to produce undecidability. We do not know what happens with fewer restrictions.

In the final part of the chapter we examine the impact of some extensions to HO Core on our decidability results (Section 7.8) and give some concluding remarks (Section 7.9).

Missing details and proofs may be found in Lanese et al. (2010b).

7.2 The Calculus

We now introduce HO Core, the core of calculi for higher-order concurrency such as CHOCS (Thomsen, 1989), Plain CHOCS (Thomsen, 1993), and Higher-Order π-calculus (Sangiorgi, 1992, 1996a,b). We use a, b, c to range over names (also called channels), and x, y, z to range over variables; the sets of names and variables are disjoint.

$$
\begin{array}{rcll}
P, Q & ::= & \overline{a}\langle P\rangle & \text{output} \\
 & \mid & a(x).\, P & \text{input prefix} \\
 & \mid & x & \text{process variable} \\
 & \mid & P \parallel Q & \text{parallel composition} \\
 & \mid & \mathbf{0} & \text{nil}
\end{array}
$$

An input $a(x).\, P$ binds the free occurrences of x in P; this is the only binder in HO Core. We write $\mathsf{fv}(P)$ for the set of free variables in P, and $\mathsf{bv}(P)$ for the bound variables. We identify processes up to a renaming of bound variables. A process is *closed* if it does not have free variables. In a statement, a name is *fresh* if it is not among the names of the objects (processes, actions, etc.) of the statement. As usual, the scope of an input $a(x).\, P$ extends as far to the right as possible. For instance, $a(x).\, P \parallel Q$ stands for $a(x).\, (P \parallel Q)$. We abbreviate the input $a(x).\, P$, with $x \notin \mathsf{fv}(P)$, as $a.\, P$; the output $\overline{a}\langle\mathbf{0}\rangle$ as $\overline{a}$; and the composition $P_1 \parallel \ldots \parallel P_k$ as $\prod_{i=1}^{k} P_i$. Similarly, we write $\prod_1^n P$ as an abbreviation for the parallel composition of n copies of P. Further, $P\{\widetilde{Q}/\widetilde{x}\}$ denotes the simultaneous substitution of variables $\widetilde{x}$ with processes $\widetilde{Q}$ in P (we assume members of $\widetilde{x}$ are distinct).

We now describe the Labeled Transition System, which is defined on open processes. There are three forms of transitions: internal transitions $P \xrightarrow{\tau} P'$; input transitions $P \xrightarrow{a(x)} P'$, meaning that P can receive at a a process that will replace x in the continuation P'; and output transitions $P \xrightarrow{\overline{a}\langle P'\rangle} P''$ meaning that P emits P' at a, and in doing so evolves to P''. We use α to denote a generic label of a transition. The notion of free variables extends to labels as expected: $\mathsf{fv}(\overline{a}\langle P\rangle) = \mathsf{fv}(P)$. For bound variables in labels, we have $\mathsf{bv}(a(x)) = \{x\}$ and $\mathsf{bv}(\overline{a}\langle P\rangle) = \emptyset$.

$$
a(x).\, P \xrightarrow{a(x)} P \ \textsc{Inp} \qquad \overline{a}\langle P\rangle \xrightarrow{\overline{a}\langle P\rangle} \mathbf{0} \ \textsc{Out} \qquad \frac{P_1 \xrightarrow{\alpha} P_1' \qquad \mathsf{bv}(\alpha) \cap \mathsf{fv}(P_2) = \emptyset}{P_1 \parallel P_2 \xrightarrow{\alpha} P_1' \parallel P_2} \ \textsc{Act1}
$$

$$
\frac{P_1 \xrightarrow{\overline{a}\langle P\rangle} P_1' \qquad P_2 \xrightarrow{a(x)} P_2'}{P_1 \parallel P_2 \xrightarrow{\tau} P_1' \parallel P_2'\{P/x\}} \ \textsc{Tau1}
$$

(We have omitted Act2 and Tau2, the symmetric counterparts of the last two rules.)

Definition 7.2.1. *The* structural congruence *relation is the smallest congruence generated by the following laws:*
$P \parallel \mathbf{0} \equiv P, \quad P_1 \parallel P_2 \equiv P_2 \parallel P_1, \quad P_1 \parallel (P_2 \parallel P_3) \equiv (P_1 \parallel P_2) \parallel P_3.$

Reductions $P \longrightarrow P'$ are defined as $P \equiv \xrightarrow{\tau} \equiv P'$.

$$\text{M-Inc}\ \frac{i : \texttt{INC}(r_j) \qquad m'_j = m_j + 1 \qquad m'_{1-j} = m_{1-j}}{(i, m_0, m_1) \longrightarrow_{\mathrm{M}} (i+1, m'_0, m'_1)}$$

$$\text{M-Dec}\ \frac{i : \texttt{DECJ}(r_j, k) \qquad m_j \neq 0 \qquad m'_j = m_j - 1 \qquad m'_{1-j} = m_{1-j}}{(i, m_0, m_1) \longrightarrow_{\mathrm{M}} (i+1, m'_0, m'_1)}$$

$$\text{M-Jmp}\ \frac{i : \texttt{DECJ}(r_j, k) \quad m_j = 0}{(i, m_0, m_1) \longrightarrow_{\mathrm{M}} (k, m_0, m_1)}$$

Table 7.1: Reduction of Minsky machines

7.3 HO Core is Turing Complete

We present in this section a deterministic encoding of Minsky machines (Minsky, 1967) into HO Core. The encoding shows that HO Core is Turing complete and, as the encoding preserves termination, it also shows that termination in HO Core is undecidable.

Minsky machines

A Minsky machine is a model composed of a set of sequential, labeled instructions, and two registers. Minsky machines have been shown to be a Turing complete model (see Minsky, 1967, Chapters 11 and 14), hence termination is undecidable for Minsky machines. Registers r_j ($j \in \{0, 1\}$) can hold arbitrarily large natural numbers. Instructions $(1 : I_1), \ldots, (n : I_n)$ can be of two kinds: $\texttt{INC}(r_j)$, which adds 1 to register r_j and proceeds to the next instruction; $\texttt{DECJ}(r_j, k)$, which jumps to instruction k if r_j is zero, otherwise it decreases register r_j by 1 and proceeds to the next instruction.

A Minsky machine includes a program counter p indicating the label of the instruction being executed. In its initial state, the machine has both registers set to 0 and the program counter p set to the first instruction. The Minsky machine stops whenever the program counter is set to a non-existent instruction, i.e., $p > n$.

A *configuration* of a Minsky machine is a tuple (i, m_0, m_1); it consists of the current program counter and the values of the registers. Formally, the reduction relation over configurations of a Minsky machine, denoted $\longrightarrow_{\mathrm{M}}$, is defined in Table 7.1.

Encoding Minsky machines into HO Core

The encoding of a Minsky machine into HO Core is denoted as $[\![\cdot]\!]_{\mathsf{M}}$. In order to simplify the presentation of the encoding, we introduce two useful notations that represent limited forms of guarded choice and guarded replication. Then we show how to count and test for zero in HO Core and present the main encoding, depicted in Table 7.2.

Guarded choice We introduce here a notation for a simple form of guarded choice to choose between different behaviors. Assume, for instance, that a_i should trigger P_i, for $i \in \{1, 2\}$. This is written as $a_1.\, P_1 + a_2.\, P_2$, whereas the choice of the behavior P_i is written as $\widehat{a_i}$.

The notation can be seen as a shortcut for HO Core terms as follows:

Definition 7.3.1. *Let $\sigma = \{(a_1, a_2) \mid a_1 \neq a_2\}$ be a fixed set of pairs of distinct names. The notation for* guarded choice *can be defined as follows:*

$$
\begin{array}{lcl}
\textsc{Instructions}\ (i : I_i) & & \\
[\![(i : \texttt{INC}(r_j))]\!]_{\mathsf{M}} & = & !p_i.\,(\widehat{inc_j} \parallel ack.\,\overline{p_{i+1}}) \\
[\![(i : \texttt{DECJ}(r_j, k))]\!]_{\mathsf{M}} & = & !p_i.\,(\widehat{dec_j} \parallel ack.\,(z_j.\,\overline{p_k} + n_j.\,\overline{p_{i+1}})) \\
\\
\textsc{Registers}\ r_j & & \\
[\![r_j = 0]\!]_{\mathsf{M}} & = & (inc_j.\,\overline{r_j^{\mathsf{S}}}\langle (\!|\ 0\ |\!)_j\rangle \ +\ dec_j.\,(\overline{r_j^0} \parallel \widehat{z_j})) \parallel \mathsf{REG}_j \\
[\![r_j = m]\!]_{\mathsf{M}} & = & (inc_j.\,\overline{r_j^{\mathsf{S}}}\langle (\!|\ m\ |\!)_j\rangle \ +\ dec_j.\,(\!|\ m-1\ |\!)_j) \parallel \mathsf{REG}_j \\
\text{where:} & & \\
\mathsf{REG}_j & = & !r_j^0.\,(\overline{ack} \parallel inc_j.\,\overline{r_j^{\mathsf{S}}}\langle (\!|\ 0\ |\!)_j\rangle \ +\ dec_j.\,(\overline{r_j^0} \parallel \widehat{z_j})) \parallel \\
 & & !r_j^{\mathsf{S}}(Y).\,(\overline{ack} \parallel inc_j.\,\overline{r_j^{\mathsf{S}}}\langle \overline{r_j^{\mathsf{S}}}\langle Y\rangle \parallel \widehat{n_j}\rangle \ +\ dec_j.\,Y) \\
 & & (\!|\ k\ |\!)_j = \begin{cases} \overline{r_j^0} \parallel \widehat{n_j} & \text{if } k = 0 \\ \overline{r_j^{\mathsf{S}}}\langle (\!|\ k-1\ |\!)_j\rangle \parallel \widehat{n_j} & \text{if } k > 0. \end{cases}
\end{array}
$$

Table 7.2: Encoding of Minsky machines

$$
\begin{aligned}
a_1.\,P_1 + a_2.\,P_2 &\triangleq \overline{a_1}\langle P_1\rangle \parallel \overline{a_2}\langle P_2\rangle \\
\widehat{a_1} &\triangleq a_2(x_2).\,a_1(x_1).\,x_1 \\
\widehat{a_2} &\triangleq a_1(x_1).\,a_2(x_2).\,x_2
\end{aligned}
$$

where, in all cases, $(a_1, a_2) \in \sigma$.

We consider only *binary* guarded choice as it is sufficient to encode Minsky machines. This way, given a pair $(a_1, a_2) \in \sigma$ and the process $a_1.\,P_1 + a_2.\,P_2$, the trigger $\widehat{a_i}$ (with $i \in \{1, 2\}$) consumes both P_i's and spawns the one chosen, i.e., $(a_1.\,P_1 + a_2.\,P_2) \parallel \widehat{a_i} \xrightarrow{\tau}\xrightarrow{\tau} P_i$. This notation has the expected behavior as long as there is at most one message at a guard ($\widehat{a_1}$ or $\widehat{a_2}$ in the previous example) enabled at any given time, and as long as concurrently running guarded choices use distinct names.

Input-guarded replication We follow the standard definition of replication in higher-order process calculi, adapting it to input-guarded replication so as to make sure that diverging behaviors are not introduced. As there is no restriction in HO Core, the definition is not compositional and replications cannot be nested.

Definition 7.3.2. *Assume a fresh name c. The definition of* input-guarded replication *in HO Core is:*

$$!a(z).\,P \triangleq (a(z).\,c(x).\,x \parallel \overline{c}\langle x\rangle \parallel P) \parallel \overline{c}\langle a(z).\,c(x).\,x \parallel \overline{c}\langle x\rangle \parallel P\rangle$$

where P contains no replications (nested replications are forbidden).

After having been activated by an output message, replication requires an additional τ step to enable the continuation P, i.e., $!a(z).\,P \xrightarrow{a(z)}\xrightarrow{\tau} (!a(z).\,P) \parallel P$.

Counting in HO Core The cornerstone of our encoding is the definition of counters that may be tested for zero. Numbers are represented as nested higher-order processes: the encoding of a number $k+1$ stored in register j, denoted $(\!|\ k+1\ |\!)_j$, is the parallel composition of two processes: $\overline{r_j^{\mathsf{S}}}\langle (\!|\ k\ |\!)_j\rangle$ (the successor of $(\!|\ k\ |\!)_j$) and a flag $\widehat{n_j}$. The encoding of zero comprises such a flag, as well as the message $\overline{r_j^0}$. This way, for instance, $(\!|\ 2\ |\!)_j$ is $\overline{r_j^{\mathsf{S}}}\langle \overline{r_j^{\mathsf{S}}}\langle \overline{r_j^0} \parallel \widehat{n_j}\rangle \parallel \widehat{n_j}\rangle \parallel \widehat{n_j}$.

Registers Registers are counters that may be incremented and decremented. They consist of two parts: their current state and two mutually recursive processes used to generate a new state after an increment or decrement of the register. The state depends on whether the current value of the register is zero or not, but in both cases it consists of a choice between an increment and a decrement. In case of an increment, a message on r_j^{S} is sent containing the current register value, for instance m. This message is then received by the recursive definition of r_j^{S} that creates a new state with value $m+1$, ready for further increment or decrement. In case of a decrement, the behavior depends on the current value, as specified in the reduction relation in Table 7.1. If the current value is zero, then it stays at zero, recreating the state corresponding to zero for further operations using the message on r_j^0, and it spawns a flag $\widehat{z_j}$ indicating that a decrement on a zero-valued register has occurred. If the current value m is strictly greater than zero, then the process $(\!|\ m-1\ |\!)_j$ is spawned. If m was equal to 1, this puts the state of the register to zero (using a message on r_j^0). Otherwise, it keeps the message in a non-zero state, with value $m-1$, using a message on r_j^{S}. In both cases a flag $\widehat{n_j}$ is spawned to indicate that the register was not equal to zero before the decrement. When an increment or decrement has been processed, that is when the new current state has been created, an acknowledgment is sent to proceed with the execution of the next instruction.

Instructions The encoding of instructions goes hand in hand with the encoding of registers. Each instruction $(i : I_i)$ is a replicated process guarded by p_i, which represents the program counter when $p = i$. Once p_i is consumed, the instruction is active and an interaction with a register occurs. In case of an increment instruction, the corresponding choice is sent to the relevant register and, upon reception of the acknowledgment, the next instruction is spawned. In case of a decrement, the corresponding choice is sent to the register, then an acknowledgment is received followed by a choice depending on whether the register was zero, resulting in a jump to the specified instruction, or the spawning of the next instruction otherwise.

The encoding of a configuration of a Minsky machine thus requires a finite number of fresh names (linear on n, the number of instructions).

Definition 7.3.3. *Let N be a Minsky machine with registers $r_0 = m_0$, $r_1 = m_1$ and instructions $(1 : I_1), \ldots, (n : I_n)$. Suppose fresh, pairwise different names r_j^0, r_j^{S}, $p_1, \ldots, p_n$, inc_j, dec_j, ack (for $j \in \{0,1\}$). Given the encodings in Table 7.2, a configuration (i, m_0, m_1) of N is encoded as*

$$\overline{p_i} \parallel [\![r_0 = m_0]\!]_{\mathsf{M}} \parallel [\![r_1 = m_1]\!]_{\mathsf{M}} \parallel \prod_{i=1}^{n} [\![(i : I_i)]\!]_{\mathsf{M}} \,.$$

In HO Core, we write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$, and $P \Uparrow$ if P has an infinite sequence of reductions. Similarly, in Minsky machines $\longrightarrow^*_{\mathrm{M}}$ is the reflexive and transitive closure of $\longrightarrow_{\mathrm{M}}$, and $N \Uparrow_{\mathrm{M}}$ means that N has an infinite sequence of reductions.

Lemma 7.3.4. *Let N be a Minsky machine. We have:*

1. *$N \longrightarrow^*_{\mathrm{M}} N'$ iff $[\![N]\!]_{\mathsf{M}} \longrightarrow^* [\![N']\!]_{\mathsf{M}}$;*
2. *if $[\![N]\!]_{\mathsf{M}} \longrightarrow^* P_1$, $P_1 \longrightarrow P_2$, and $P_1 \longrightarrow P_3$ then $P_2 \equiv P_3$;*
3. *if $[\![N]\!]_{\mathsf{M}} \longrightarrow^* P_1$ then there exists N' such that $P_1 \longrightarrow^* [\![N']\!]_{\mathsf{M}}$ and $N \longrightarrow^*_{\mathrm{M}} N'$;*
4. *$N \Uparrow_{\mathrm{M}}$ iff $[\![N]\!]_{\mathsf{M}} \Uparrow$.*

The results above guarantee that HO Core is Turing complete, and since the encoding preserves termination, it entails the following corollary.

Corollary 7.3.5. *Termination in HO Core is undecidable.*

7.4 Bisimilarity in HO Core

In this section we prove that the main forms of strong bisimilarity for higher-order process calculi coincide in HO Core, and that such a relation is decidable. As a key ingredient for our results, we introduce *open Input/Output (IO) bisimulation*, in which the variable of input prefixes is never instantiated and τ-transitions are not observed. We are not aware of other results on process calculi where processes can perform τ-transitions and yet a bisimulation that does not mention τ-transitions is discriminating enough. (One of the reasons that make this possible is that bisimulation in HO Core is very discriminating.)

We define different kinds of bisimulations by appropriate combinations of the clauses below.

Definition 7.4.1 (HO Core bisimulation clauses, open processes)**.** *A symmetric relation $\mathcal{R}$ on HO Core processes is*

1. *a* τ-bisimulation *if $P \mathcal{R} Q$ and $P \xrightarrow{\tau} P'$ imply that there is Q' such that $Q \xrightarrow{\tau} Q'$ and $P' \mathcal{R} Q'$;*

2. *a* higher-order output bisimulation *if $P \mathcal{R} Q$ and $P \xrightarrow{\overline{a}\langle P''\rangle} P'$ imply that there are Q', Q'' such that $Q \xrightarrow{\overline{a}\langle Q''\rangle} Q'$ with $P' \mathcal{R} Q'$ and $P'' \mathcal{R} Q''$;*

3. *an* output normal bisimulation *if $P \mathcal{R} Q$ and $P \xrightarrow{\overline{a}\langle P''\rangle} P'$ imply that there are Q', Q'' such that $Q \xrightarrow{\overline{a}\langle Q''\rangle} Q'$ with $m.\, P'' \parallel P' \mathcal{R}\, m.\, Q'' \parallel Q'$, where m is fresh;*

4. *an* open bisimulation *if whenever $P \mathcal{R} Q$:*

   - *$P \xrightarrow{a(x)} P'$ implies that there is Q' such that $Q \xrightarrow{a(x)} Q'$ and $P' \mathcal{R} Q'$,*
   - *$P \equiv x \parallel P'$ implies that there is Q' such that $Q \equiv x \parallel Q'$ and $P' \mathcal{R} Q'$.*

Definition 7.4.2 (HO Core bisimulation clauses, closed processes)**.** *A symmetric relation $\mathcal{R}$ on closed HO Core processes is*

1. *an* output context bisimulation *if $P \mathcal{R} Q$ and $P \xrightarrow{\overline{a}\langle P''\rangle} P'$ imply that there are Q', Q'' such that $Q \xrightarrow{\overline{a}\langle Q''\rangle} Q'$ and for all S with $\mathsf{fv}(S) \subseteq \{x\}$, it holds that $S\{^{P''}/x\} \parallel P' \mathcal{R}\, S\{^{Q''}/x\} \parallel Q'$;*

2. *an* input normal bisimulation *if $P \mathcal{R} Q$ and $P \xrightarrow{a(x)} P'$ imply that there is Q' such that $Q \xrightarrow{a(x)} Q'$ and $P'\{\overline{m}/x\} \mathcal{R}\, Q'\{\overline{m}/x\}$, where m is fresh;*

3. closed *if $P \mathcal{R} Q$ and $P \xrightarrow{a(x)} P'$ imply that there is Q' such that $Q \xrightarrow{a(x)} Q'$ and for all closed R, it holds that $P'\{^{R}/x\} \mathcal{R}\, Q'\{^{R}/x\}$.*

A combination of the bisimulation clauses in Definitions 7.4.1 and 7.4.2 is *complete* if it includes exactly one clause for input and output transitions (in contrast, it needs not include a clause for τ-transitions).[1] We will show that all complete combinations

[1] The clauses of Definition 7.4.2 are however tailored to closed processes, therefore combining them with clause 4 in Definition 7.4.1 has little interest.

coincide. We only give a name to those combinations that represent known forms of bisimulation for higher-order processes or that are needed in our proofs. In each case, as usual, a *bisimilarity* is the union of all bisimulations, and is itself a bisimulation (the functions from relations to relations that represent the bisimulation clauses in Definitions 7.4.1 and 7.4.2 are all monotonic).

Definition 7.4.3. Higher-order bisimilarity, *written* $\sim_{\mathtt{HO}}$, *is the largest relation on closed HO Core processes that is a* τ*-bisimulation, a higher-order output bisimulation, and is closed.*

Context bisimilarity, *written* $\sim_{\mathtt{CON}}$, *is the largest relation on closed HO Core processes that is a* τ*-bisimulation, an output context bisimulation, and is closed.*

Normal bisimilarity, *written* $\sim_{\mathtt{NOR}}$, *is the largest relation on closed HO Core processes that is a* τ*-bisimulation, an output normal bisimulation, and an input normal bisimulation.*

IO bisimilarity, *written* $\sim^{\circ}_{\mathtt{IO}}$, *is the largest relation on HO Core processes that is a higher-order output bisimulation and is open.*

Open normal bisimilarity, *written* $\sim^{\circ}_{\mathtt{NOR}}$, *is the largest relation on HO Core processes that is a* τ*-bisimulation, an output normal bisimulation, and is open.*

Environmental bisimilarity (Sangiorgi et al., 2007), a recent proposal of bisimilarity for higher-order calculi, in HO Core roughly corresponds to (and indeed coincides with) the complete combination that is a τ-bisimulation, an output normal bisimulation, and is closed.

Remark 7.4.4. *The input clause of Definition 7.4.2(3) is in the* late *style. It is known (Sangiorgi, 1992) that in calculi of pure higher-order concurrency early and late clauses are equivalent.*

Remark 7.4.5. *In contrast with normal bisimulation (as defined in, e.g., Sangiorgi (1992); Jeffrey and Rathke (2005)), our clause for output normal bisimulation does not use a replication in front of the introduced fresh name. Such a replication would be needed in extensions of the calculus (e.g., with recursion or restriction).*

A bisimilarity on closed processes is extended to open processes as follows.

Definition 7.4.6 (Extension of bisimilarities)**.** *Let* $\mathcal{R}$ *be a bisimilarity on closed HO Core processes. The extension of* $\mathcal{R}$ *to open HO Core processes, denoted* $\mathcal{R}^{\star}$*, is defined by*

$$\mathcal{R}^{\star} = \{(P, Q) : a(x_1).\cdots.a(x_n).\,P\,\mathcal{R}\,a(x_1).\cdots.a(x_n).\,Q\}$$

where $\mathsf{fv}(P) \cup \mathsf{fv}(Q) = \{x_1, \ldots, x_n\}$*, and* a *is fresh in* P, Q*.*

The simplest complete form of bisimilarity is $\sim^{\circ}_{\mathtt{IO}}$. Not only $\sim^{\circ}_{\mathtt{IO}}$ is the less demanding for proofs; it also has a straightforward proof of congruence (see Lanese et al., 2010b). This is significant because congruence is usually a hard problem in bisimilarities for higher-order calculi.

Lemma 7.4.7 (Congruence of $\sim^{\circ}_{\mathtt{IO}}$)**.** *Let* P_1, P_2 *be open HO Core processes.* $P_1 \sim^{\circ}_{\mathtt{IO}} P_2$ *implies:*

1. $a(x).\,P_1 \sim^{\circ}_{\mathtt{IO}} a(x).\,P_2$*;*
2. $P_1 \parallel R \sim^{\circ}_{\mathtt{IO}} P_2 \parallel R$*, for every* R*;*
3. $\overline{a}\langle P_1\rangle \sim^{\circ}_{\mathtt{IO}} \overline{a}\langle P_2\rangle$*.*

Lemma 7.4.8 ($\sim^{\circ}_{\mathtt{IO}}$ is preserved by substitutions)**.** *If* $P \sim^{\circ}_{\mathtt{IO}} Q$ *then for all* x *and* R*, also* $P\{^{R}/x\} \sim^{\circ}_{\mathtt{IO}} Q\{^{R}/x\}$*.*

The most striking property of $\sim^{\circ}_{\mathtt{IO}}$ is its decidability. In contrast with the other bisimilarities, in $\sim^{\circ}_{\mathtt{IO}}$ the size of processes always decreases during the bisimulation game. This is because $\sim^{\circ}_{\mathtt{IO}}$ is an open relation and does not have a clause for τ transitions, hence process copying never occurs.

Lemma 7.4.9. *Relation $\sim^{\circ}_{\mathtt{IO}}$ is decidable.*

Next we show that $\sim^{\circ}_{\mathtt{IO}}$ is also a τ bisimulation. This allows us to prove that $\sim^{\circ}_{\mathtt{IO}}$ coincides with other bisimilarities, starting with $\sim^{\star}_{\mathtt{HO}}$ and to transfer to them its properties, in particular congruence and decidability.

Lemma 7.4.10. *Relation $\sim^{\circ}_{\mathtt{IO}}$ is a τ-bisimulation.*

Proof (Sketch). Suppose $P \sim^{\circ}_{\mathtt{IO}} Q$ and $P \xrightarrow{\tau} P'$. We have to find a matching transition from Q. We can decompose P's transition into an output $P \xrightarrow{\overline{a}\langle R\rangle} P_1$ followed by an input $P_1 \xrightarrow{a(x)} P_2$, with $P' = P_2\{^{R}/x\}$. By definition of $\sim^{\circ}_{\mathtt{IO}}$, Q is capable of matching these transitions, and the final derivative is a process Q_2 with $Q_2 \sim^{\circ}_{\mathtt{IO}} P_2$. Further, as HO Core has no output prefixes (i.e., it is an asynchronous calculus) the two transitions from Q can be combined into a τ-transition, which matches the initial τ-transition from P. We conclude using Lemmas 7.4.7 and 7.4.8. □

Lemma 7.4.11. *$\sim^{\circ}_{\mathtt{IO}}$ and $\sim^{\star}_{\mathtt{HO}}$ coincide.*

We now move to the relationship between $\sim_{\mathtt{HO}}$, $\sim^{\circ}_{\mathtt{NOR}}$, and $\sim_{\mathtt{CON}}$. We begin by establishing a few properties of normal bisimulation, then show a chain of implications between the different bisimulations.

Lemma 7.4.12. *If $m.\,P_1 \parallel P \sim^{\circ}_{\mathtt{NOR}} m.\,Q_1 \parallel Q$, for some fresh m, then we have $P_1 \sim^{\circ}_{\mathtt{NOR}} Q_1$ and $P \sim^{\circ}_{\mathtt{NOR}} Q$.*

Lemma 7.4.13. *$\sim^{\star}_{\mathtt{HO}}$ implies $\sim^{\star}_{\mathtt{CON}}$.*

Lemma 7.4.14. *$\sim_{\mathtt{CON}}$ implies $\sim_{\mathtt{NOR}}$.*

Lemma 7.4.15. *$\sim^{\star}_{\mathtt{NOR}}$ implies $\sim^{\circ}_{\mathtt{NOR}}$.*

Lemma 7.4.16. *$\sim^{\circ}_{\mathtt{NOR}}$ implies $\sim^{\circ}_{\mathtt{IO}}$.*

Lemma 7.4.17. *In HO Core, relations $\sim_{\mathtt{HO}}$, $\sim^{\circ}_{\mathtt{NOR}}$ and $\sim_{\mathtt{CON}}$ coincide on closed processe.*

Proof. This is an immediate consequence of previous results. □

We thus infer that $\sim_{\mathtt{HO}}$ and $\sim_{\mathtt{CON}}$ are congruence relations. Direct proofs of these results proceed by exhibiting an appropriate bisimulation and are usually hard (a sensible aspect being proving congruence for parallel composition). Congruence of higher-order bisimilarity is usually proved by appealing to, and adapting, Howe's method for the λ-calculus (1996). This is the approach followed in, e.g., Baldamus and Frauenstein (1995); Bundgaard et al. (2004); Godskesen and Hildebrandt (2005); Chapter 8 also presents the use of Howe's method for higher-order process calculi with *passivation* constructs.

We then extend the result to all complete combinations of the HO Core bisimulation clauses (Definitions 7.4.1 and 7.4.2).

Theorem 7.4.18. *All complete combinations of the HO Core bisimulation clauses coincide, and are decidable.*

Proof. In Lemma 7.4.17 we have proved that the least demanding combination ($\sim^{\circ}_{\mathtt{IO}}$) coincides with the most demanding ones ($\sim^{\star}_{\mathtt{HO}}$ and $\sim^{\star}_{\mathtt{CON}}$). Decidability then follows from Lemma 7.4.9. □

We find this "collapsing" of bisimilarities in HO Core significant; the only similar result we are aware of is by Cao (2006), who showed that strong context bisimulation and strong normal bisimulation coincide in higher-order π-calculus.

7.5 Barbed Congruence and Asynchronous Equivalences

We now show that the labeled bisimilarities of Section 7.4 coincide with *barbed congruence*, the form of contextual equivalence used in concurrency to justify bisimulation-like relations. Below we use *reduction-closed* barbed congruence (Honda and Yoshida, 1995; Sangiorgi and Walker, 2001), as this makes some technical details simpler; however the results also hold for ordinary barbed congruence as defined in Milner and Sangiorgi (1992). It is worth recalling that the main difference between reduction-closed barbed congruence and the barbed congruence of Milner and Sangiorgi (1992) is quantification over contexts (see (2) in Definition 7.5.1 below). More importantly, we consider the *asynchronous* version of barbed congruence, where barbs are only produced by output messages; we call barbed congruence *synchronous* when inputs contribute too, as in, e.g., Milner and Sangiorgi (1992). We use the asynchronous version for two reasons. First, asynchronous barbed congruence is a weaker relation, which makes the results stronger (they imply the corresponding results for the synchronous relation). Second, asynchronous barbed congruence is more natural in HO Core because it is an asynchronous calculus — it has no output prefix.

Note also that the labeled bisimilarities of Section 7.4 have been defined in the synchronous style. In an *asynchronous labeled bisimilarity* (see, e.g., Amadio et al. (1998)) the input clause is weakened so as to allow, in certain conditions, an input action to be matched also by a τ-action. For instance, input normal bisimulation (Definition 7.4.2(2)) would become:

- if $P \xrightarrow{a(x)} P'$ then, for some fresh name m,
  1. either $Q \xrightarrow{a(x)} Q'$ and $P'\{\overline{m}/x\} \mathcal{R} Q'\{\overline{m}/x\}$;
  2. or $Q \xrightarrow{\tau} Q'$ and $P'\{\overline{m}/x\} \mathcal{R} Q' \parallel \overline{a}\langle\overline{m}\rangle$.

We now define asynchronous barbed congruence. We write $P \downarrow_{\overline{a}}$ (resp. $P \downarrow_a$) if P can perform an output (resp. input) transition at a.

Definition 7.5.1. Asynchronous barbed congruence, $\simeq$, *is the largest symmetric relation on closed processes that is*

1. *a τ-bisimulation (Definition 7.4.1(1));*
2. *context-closed (i.e., $P \simeq Q$ implies $C[P] \simeq C[Q]$, for all closed contexts $C[\cdot]$);*
3. *barb preserving (i.e., if $P \simeq Q$ and $P \downarrow_{\overline{a}}$, then also $Q \downarrow_{\overline{a}}$).*

In synchronous barbed congruence, input barbs $P \downarrow_a$ are also observable.

Lemma 7.5.2. *Asynchronous barbed congruence coincides with normal bisimilarity.*

Proof. The proof basically consists of renaming the messages which may interfere with the observation to messages on fresh names, doing the observation, then renaming back the messages to their original name. Details may be found in Lanese et al. (2010b). □

Remark 7.5.3. *The proof relies on the fact that HO Core has no operators of choice and restriction. In fact, choice would prevent the renaming to be reversible, and restriction*

would prevent the renaming using a context as some names may be hidden. The higher-order aspect of HO Core does not really play a role. The proof could indeed be adapted to CCS-like, or π-calculus-like, languages in which the same operators are missing.

Corollary 7.5.4. *In HO Core asynchronous and synchronous barbed congruence coincide, and they also coincide with all complete combinations of the HO Core bisimulation clauses of Theorem 7.4.18.*

Further, Corollary 7.5.4 can be extended to include the asynchronous versions of the labeled bisimilarities in Section 7.4 (precisely, the *complete asynchronous combinations* of the HO Core bisimulation clauses; that is, complete combinations that make use of an asynchronous input clause as outlined before Definition 7.5.1). This holds because: (i) all proofs of Section 7.4 can be easily adapted to the corresponding asynchronous labeled bisimilarities; (ii) using standard reasoning for barbed congruences, one can show that asynchronous normal bisimilarity coincides with asynchronous barbed congruence; (iii) via Corollary 7.5.4 one can then relate the asynchronous labeled bisimilarities to the synchronous ones.

7.6 Axiomatization and Complexity

We have shown in the previous section that the main forms of bisimilarity for higher-order process calculi coincide in HO Core. We therefore simply call *bisimilarity* such a relation, and write it as $\sim$. Here we present a sound and complete axiomatization of bisimilarity. We do so by adapting to a higher-order setting results by Moller and Milner on unique decomposition of processes (1993), and by Hirschkoff and Pous on an axiomatization for a fragment of (finite) CCS (2008). We then exploit this axiomatization to derive complexity bounds for bisimilarity checking.

Axiomatization

Definition 7.6.1 (Prime decomposition)**.** *A process P is* prime *if $P \not\sim \mathbf{0}$ and $P \sim P_1 \parallel P_2$ imply $P_1 \sim \mathbf{0}$ or $P_2 \sim \mathbf{0}$. When $P \sim \prod_{i=1}^{n} P_i$ where each P_i is prime, we call $\prod_{i=1}^{n} P_i$ a* prime decomposition *of P.*

Proposition 7.6.2 (Cancellation)**.** *For all P, Q, and R, if $P \parallel R \sim Q \parallel R$ then also $P \sim Q$.*

Proposition 7.6.3 (Unique decomposition)**.** *Any process P admits a prime decomposition $\prod_{i=1}^{n} P_i$ which is unique up to bisimilarity and permutation of indices (i.e., given two prime decompositions $\prod_{i=1}^{n} P_i$ and $\prod_{j=1}^{m} Q_j$, then $n = m$ and there is a permutation σ of $\{1, \ldots, n\}$ such that $P_i \sim Q_{\sigma(i)}$ for each $i \in \{1, \ldots, n\}$).*

Both the key law for the axiomatization and the following results are inspired by similar ones by Hirschkoff and Pous (2008) for pure CCS. Using their terminology, we call *distribution law*, briefly (DIS), the axiom schema below (recall that $\prod_1^k Q$ denotes the parallel composition of k copies of Q).

$$a(x).\,(P \parallel \textstyle\prod_1^{k-1} a(x).\,P) \;=\; \prod_1^k a(x).\,P \qquad \text{(DIS)}$$

We then call *extended structural congruence*, written $\equiv_{\mathtt{E}}$, the extension of the structural congruence relation ($\equiv$, Definition 7.2.1) with the axiom schema (DIS). We write $P \rightsquigarrow Q$ when there are processes P' and Q' such that $P \equiv P'$, $Q' \equiv Q$ and Q' is obtained from P' by rewriting a subterm of P' using law (DIS) from left to right. We now state that $\equiv_{\mathtt{E}}$ provides an algebraic characterization of $\sim$ in HO Core.

Definition 7.6.4. *A process P is in* normal form *if it cannot be further simplified in the system $\equiv_{\mathtt{E}}$ by using $\rightsquigarrow$.*

Any process P has a normal form that is unique up to $\equiv$, and which will be denoted by $\mathsf{n}(P)$.

Theorem 7.6.5. *For any processes P and Q, we have $P \sim Q$ iff $\mathsf{n}(P) \equiv \mathsf{n}(Q)$.*

Corollary 7.6.6. *$\equiv_{\mathtt{E}}$ is a sound and complete axiomatization of bisimilarity in HO Core.*

Complexity of bisimilarity checking

To analyze the complexity of deciding whether two processes are bisimilar, one could apply the technique from Dovier et al. (2004), and derive that bisimilarity is decidable in time which is linear in the size of the LTS for $\sim^{\circ}_{\mathtt{IO}}$ (which avoids τ transitions). This LTS is however exponential in the size of the process. A more efficient solution exploits the axiomatization above: one can first normalize processes and then reduce bisimilarity to syntactic equivalence of the obtained normal forms.

For simplicity, we assume a process P is represented as an ordered tree (but we will transform it into a DAG during normalization). In the following, let us denote with $t[m_1, \ldots, m_k]$ the ordered tree with root labeled t and with (ordered) descendants $m_1, \ldots, m_k$. We write $t[]$ for a tree labeled t and without descendants (i.e., a leaf).

Definition 7.6.7 (Tree representation)**.** *Let P be a HO Core process. Its associated ordered tree representation is labeled and defined inductively by*

- $\mathsf{Tree}(\mathbf{0}) = \mathbf{0}[]$
- $\mathsf{Tree}(x) = \mathsf{db}(x)[]$
- $\mathsf{Tree}(\overline{a}\langle Q\rangle) = \bar{a}[\mathsf{Tree}(Q)]$
- $\mathsf{Tree}(a(x).\, Q) = a[\mathsf{Tree}(Q)]$
- $\mathsf{Tree}(\prod_{i=1}^{n} P_i) = \prod_{i=1}^{n}[\mathsf{Tree}(P_1), \ldots, \mathsf{Tree}(P_n)]$

where db *is a function assigning De Bruijn indices (De Bruijn, 1972) to variables. Parallel composition is n-ary, thus we can assume without loss of generality that children of parallel composition nodes are not parallel composition nodes (i.e., we can always flatten them).*

We now describe the normalization steps by characterizing them as reductions as well as pseudocode descriptions. The first step deals with parallel composition nodes: it removes all unnecessary $\mathbf{0}$ nodes, and relabels the nodes when the parallel composition has only one or no descendants.

Normalization step 1. *Let $\rightsquigarrow_{\mathtt{N1}}$ be a transformation rule over trees associated to HO Core processes defined by:*

1. $\prod_{i=1}^{0}[] \rightsquigarrow_{\mathtt{N1}} \mathbf{0}[]$
2. $\prod_{i=1}^{1}[\mathsf{Tree}(P_1)] \rightsquigarrow_{\mathtt{N1}} T$ *if* $\mathsf{Tree}(P_1) \rightsquigarrow_{\mathtt{N1}} T$
3. $\prod_{i=1}^{n}[\mathsf{Tree}(P_1), \ldots, \mathsf{Tree}(P_n)] \rightsquigarrow_{\mathtt{N1}} \prod_{i=1}^{m}[T_{\sigma(1)}, \ldots, T_{\sigma(m)}]$, *if* $\mathsf{Tree}(P_i) \rightsquigarrow_{\mathtt{N1}} T_i$ *for each i, where $m < n$ is the number of trees in $T_1, \ldots, T_n$ that are different from $\mathbf{0}[]$, and where σ is a bijective function from $\{1, \ldots, m\}$ to $\{i \mid i \in \{1, \ldots, n\} \wedge T_i \neq \mathbf{0}[]\}$.*

After this first step, the tree is traversed bottom-up, applying the following two normalization steps.

Normalization step 2. *Let* $\rightsquigarrow_{\mathtt{N2}}$ *be a transformation rule over trees associated to HO Core processes, defined as follows. If the node is a parallel composition, sort all the children lexicographically. If* n *children are equal, leave just one and make* n *references to it.*

The last normalization step applies `DIS` from left to right if possible:

Normalization step 3. *Let* $\rightsquigarrow_{\mathtt{N3}}$ *be a transformation rule over trees associated to HO Core processes, defined by:*

$$a\left[\prod_{i=1}^{k+1}[\mathsf{Tree}(P), \mathsf{Tree}(a(x).\,P), \ldots, \mathsf{Tree}(a(x).\,P)]\right] \rightsquigarrow_{\mathtt{N3}} \prod_{j=1}^{k+1}[\mathsf{Tree}(a(x).\,P), \ldots, \mathsf{Tree}(a(x).\,P)]$$

where $\mathsf{Tree}(a(x).\,P)$ *appears* k *times in the left-hand side, and* $k+1$ *times in the right-hand side.*

The pseudocode for these normalization steps may be found in Lanese et al. (2010b).

We relate now normalization and bisimilarity, and state the complexity of normalization.

Lemma 7.6.8. *Let* P, Q *be processes and* T_P, T_Q *their tree representations normalized according to steps 1, 2 and 3. Then* $P \sim Q$ *iff* $T_P = T_Q$.

Theorem 7.6.9. *Consider two HO Core processes* P *and* Q. $P \sim Q$ *can be decided in time* $O(n^2 \log m)$ *where* $n = \max(\mathbf{size}(P), \mathbf{size}(Q))$ *(i.e., the maximum number of nodes in the tree representations of* P *and* Q*) and* m *is the maximum branching factor in them (i.e., the maximum number of components in a parallel composition).*

Proof. Bisimilarity check proceeds as follows: first normalize the tree representations of the two processes, then check them for syntactic equality.

Normalization step 1 can be performed in time $O(n)$. Normalization step 2 performs a visit of the tree, sorting and removing duplicates from the children of parallel composition nodes. Sorting can be done in $O(n \log m)$ for each parallel composition node. Removing duplicates requires just $O(n)$ time. Normalization step 3 visits the tree too, possibly reconfiguring input nodes. The check for applicability requires one comparison ($O(n)$) and the check that all the other components coincide (simply check that the subtrees have been merged by Normalization step 2: $O(n)$). Applying $\rightsquigarrow_{\mathtt{N3}}$ simply entails collapsing the trees ($O(n)$). Other nodes require no operations.

Thus the normalization for a single node can be done in $O(n \log m)$, and the whole normalization can be done in $O(n^2 \log m)$. □

7.7 Undecidability and Static Restrictions

If the restriction operator is added to HO Core, as in Plain CHOCS or Higher-Order π-calculus, then recursion can be encoded (Thomsen, 1990; Sangiorgi and Walker, 2001) and most of the results in Sections 7.4-7.6 would break. In particular, higher-order and context bisimilarities are different and both undecidable (Sangiorgi, 1992, 1996a).

We discuss here the addition of a limited form of restriction, which we call *static restriction*. These restrictions may not appear inside output messages: in any output

$\overline{a}\langle P\rangle$, P is restriction-free. This limitation is important: it prevents for instance the above-mentioned encoding of recursion from being written. Static restrictions could also be defined as top-level restrictions since, by means of standard structural congruence laws (or similar laws allowing to swap input and restriction), any static restriction can be pulled out at the top-level. Thus the processes would take the form $\boldsymbol{\nu} a_1 \ldots \boldsymbol{\nu} a_n\, P$, where $\boldsymbol{\nu} a_i$ indicates the restriction on the name a_i, and where restriction cannot appear inside P itself. The operational semantics—LTS and bisimilarities—are extended as expected. For instance, one would have bounded outputs as actions, as well as rules

$$\frac{P \xrightarrow{\alpha} P' \qquad z \notin \mathsf{fn}(\alpha)}{\nu z P \xrightarrow{\alpha} \nu z P'}\ \textsc{StRes} \qquad\qquad \frac{P \xrightarrow{\nu \widetilde{v}\, \overline{a}\langle R\rangle} P' \qquad z \in \mathsf{fn}(R) \setminus \widetilde{v}}{\nu z P \xrightarrow{\nu z \widetilde{v}\, \overline{a}\langle R\rangle} P'}\ \textsc{StOpen}$$

defining static restriction and extrusion of restricted names, respectively. Note that there is no need to define how a bounded output interacts with input as every τ transition takes place under the restrictions. Also, structural congruence (Definition 7.2.1) would be extended with the axioms for restriction $\nu z\, \nu w P \equiv \nu w\, \nu z P$ and $\nu z \mathbf{0} \equiv \mathbf{0}$. (In contrast, notice that we do not require the axiom: $\nu z(P \parallel Q) \equiv P \parallel (\nu z Q)$, where z does not occur in P.) We sometimes write $\nu a_1, \ldots, a_n$ to stand for $\nu a_1, \ldots, \nu a_n$.

We show that *four* static restrictions are enough to make undecidable any bisimilarity that has little more than a clause for τ-actions. For this, we reduce the Post correspondence problem (PCP) (Post, 1946) to the bisimilarity of some processes. We call *complete τ-bisimilarity* any complete combination of the HO Core bisimulation clauses (as defined in Section 7.4) that includes the clause for τ actions (Definition 7.4.1(1)); the bisimilarity can even be asynchronous (Section 7.5).

Definition 7.7.1 (PCP)**.** *An* instance *of PCP consists of an alphabet A containing at least two symbols, and a finite list $T_1, \ldots, T_n$ of tiles, where each tile is a pair of words over A. We use $T_i = (u_i, l_i)$ to denote a tile T_i with upper word u_i and lower word l_i. A solution to this instance is a non-empty sequence of indices $i_1, \ldots, i_k$, $1 \leq i_j \leq n$ ($j \in 1 \cdots k$), such that $u_{i_1} \cdots u_{i_k} = l_{i_1} \cdots l_{i_k}$. The decision problem is then to determine whether such a solution exists or not.*

Having (static) restrictions, we can refine the notation for non-nested replications (Definition 7.3.2) and define it in the unguarded case:

$$!P \triangleq \boldsymbol{\nu} c\, (Q_c \parallel \overline{c}\langle Q_c\rangle)$$

where $Q_c = c(x).\,(x \parallel \overline{c}\langle x\rangle \parallel P)$ and P is a HO Core process (i.e., it is restriction-free). It is easy to see that $!P \xrightarrow{\tau} !P \parallel P$.

Now, $!\mathbf{0}$ is a purely divergent process, as it can only make τ-transitions, indefinitely; it is written using only one static restriction. Given an instance of PCP we build a set of processes $P_1, \ldots, P_n$, one for each tile $T_1, \ldots, T_n$, and show that, for each i, P_i is bisimilar to $!\mathbf{0}$ iff the instance of PCP has no solution ending with T_i. Thus PCP is solvable iff there exists j such that P_j is not bisimilar to $!\mathbf{0}$.

The processes $P_1, \ldots, P_n$ execute in two distinct phases: first they build a possible solution of PCP, then they non-deterministically stop building the solution and execute it. If the chosen composition is a solution then a signal on a free channel *success* is sent, thus performing a visible action, which breaks bisimilarity with $!\mathbf{0}$.

The precise encoding of PCP into HO Core is shown in Table 7.3, and described below. We consider an alphabet of two letters, a_1 and a_2. The upper and lower words of a tile are treated as separate *strings*, which are encoded letter by letter. The encoding of a letter is then a process whose continuation encodes the rest of the string, and varies depending on whether the letter occurs in the upper or in the lower word. We use a

LETTERS	$[\![a_1, P]\!]_u = [\![a_2, P]\!]_l = \overline{a}\langle P\rangle$
	$[\![a_2, P]\!]_u = [\![a_1, P]\!]_l = a(x).\,(x \parallel P)$
STRINGS	$[\![a_i \cdot s, P]\!]_w = [\![a_i, [\![s, P]\!]_w]\!]_w$
	$[\![\epsilon, P]\!]_w = P \qquad (\epsilon \text{ is the empty word})$
CREATORS	$C_k = up(x).\,low(y).\,(\overline{up}\langle [\![u_k, x]\!]_u\rangle \parallel \overline{low}\langle [\![l_k, y]\!]_l\rangle)$
STARTERS	$S_k = \overline{up}\langle [\![u_k, \overline{b}]\!]_u\rangle \parallel \overline{low}\langle [\![l_k, b.\,\overline{success}]\!]_l\rangle$
EXECUTOR	$E = up(x).\,low(y).\,(x \parallel y)$
SYSTEM	$P_j = \boldsymbol{\nu} up\, \boldsymbol{\nu} low\, \boldsymbol{\nu} a\, \boldsymbol{\nu} b\, (\, S_j \parallel !\prod_k C_k \parallel E)$

Table 7.3: Encoding of PCP

single channel to encode both letters: for the upper word, a_1 is encoded as $\overline{a}\langle P\rangle$ and a_2 as $a(x).\,(x \parallel P)$, where P is the continuation and x does not occur in P; for the lower word the encodings are switched. In Table 7.3, $[\![a_i, P]\!]_w$ denotes the encoding of the letter a_i with continuation P, with $w = u$ if the encoding is on the upper word, $w = l$ otherwise. Hence, given a string $s = a_i \cdot s'$, its encoding $[\![s, P]\!]_w$ is $[\![a_i, [\![s', P]\!]_w]\!]_w$, i.e., the first letter with the encoding of the rest as continuation. Notice that the encoding of an a_i in the upper word can synchronize only with the encoding of a_i for the lower word.

The whole system P_j is composed by a (replicated) *creator* C_k for each tile T_k, a *starter* S_j that launches the building of a tile composition ending with (u_j, l_j), and an *executor* E. The starter makes the computation begin; creators non-deterministically add their tile to the beginning of the composition. Also non-deterministically, the executor blocks the building of the composition and starts its execution. This proceeds if no difference is found: if both strings end at the same character, then synchronization on channel b can be performed, which in turn, makes action $\overline{success}$ visible. Notice that without synchronizing on b, action $\overline{success}$ could be visible even in the case in which one of the strings is a prefix of the other one.

The encoding of replication requires another restriction, thus P_j has five restrictions. However, names low and a are used in different phases; thus choosing $low = a$ does not create interferences, and four restrictions are enough.

Theorem 7.7.2. *Given an instance of PCP and one of its tiles T_j, there is a solution of the instance of PCP ending with T_j iff P_j is not bisimilar to* $!\mathbf{0}$ *according to any complete τ-bisimilarity.*

Corollary 7.7.3. *Barbed congruence and any complete τ-bisimilarity are undecidable in HO Core with four static restrictions.*

Theorem 7.7.2 actually shows that even *asynchronous barbed bisimilarity* (defined as the largest τ-bisimilarity that is output-barb preserving, and used in the definition of ordinary—as opposed to reduction-closed—barbed congruence) is undecidable. The corollary above then follows from the fact that all the relations there mentioned are at least as demanding as asynchronous barbed bisimilarity.

7.8 Other Extensions

We now examine the impact on decidability of bisimilarity of some extensions of HO Core. We omit the details, including precise statements of the results.

Abstractions An *abstraction* is an expression of the form $(x)P$; it is a parametrized process. An abstraction has a functional type. Applying an abstraction $(x)P$ of type $T \to \diamond$ (where $\diamond$ is the type of all processes) to an argument W of type T yields the

process $P\{^W/x\}$. The argument W can itself be an abstraction; therefore the order of an abstraction, that is, the level of arrow nesting in its type, can be arbitrarily high. The order can also be ω, if there are recursive types. By setting bounds on the order of the types of abstractions, one can define a hierarchy of subcalculi of the Higher-Order π-calculus (Sangiorgi and Walker, 2001); and when this bound is ω, one obtains a calculus capable of representing the π-calculus (for this all operators of the Higher-Order π-calculus are needed, including full restriction).

Allowing the communication of abstractions, as in the Higher-Order π-calculus, one then also needs to add in the grammar for processes an *application* construct of the form $P_1\langle P_2\rangle$, as a destructor for abstractions. Extensions in the LTS would be as follows. Suppose, as in Sangiorgi (1996b), that *beta-conversion* $\succ$ is the least precongruence on HO Core processes generated by the rule

$$(x)P_1\langle P_2\rangle \succ P_1\{^{P_2}/x\}.$$

The LTS could be then extended with a rule

$$\frac{P \succ P_1' \qquad P_1' \xrightarrow{\alpha} Q}{P \xrightarrow{\alpha} Q}\ \textsc{Beta}$$

Notice that with these additions, the characterization of bisimilarity as IO bisimilarity still holds. For a HO Core extended with abstractions and applications, $\sim^{\circ}_{\mathtt{IO}}$ is still a congruence and is preserved by substitutions (by straightforward extensions of the proofs of Lemmas 7.4.7 and 7.4.8). Note that, however, abstraction application may increase the size of processes. If abstractions are of finite type (i.e., their order is smaller than ω) then only a finite number of such applications is possible, and decidability of bisimilarity is preserved. Decidability fails if the order is ω, intuitively because in this case it is possible to simulate the λ-calculus.

Output prefix If we add an output prefix construct $\overline{a}\langle P\rangle.\,Q$ to HO Core, then the proof of the characterization as IO bisimilarity breaks and, with it, the proof of decidability. Decidability proofs can however be adjusted by appealing to results on unique decomposition of processes and axiomatization (along the lines of Section 7.6).

Choice Decidability remains with the addition of a choice operator to HO Core. The proofs require little modifications. The addition of both choice and output prefix is harder. It might be possible to extend the decidability proof for output prefix mentioned above so to accommodate also choice, but the details become much more complex.

Recursion We do not know whether decidability is maintained by the addition of recursion (or similar operators such as replication).

7.9 Concluding Remarks

Process calculi are usually Turing complete and have an undecidable bisimilarity (and barbed congruence). Subcalculi have been studied where bisimilarity becomes decidable but then one loses Turing completeness. Examples are BPA and BPP (see, e.g., (Kučera and Jančar, 2006)) and CCS without restriction and relabeling (Christensen et al., 1994). In this chapter we have identified a Turing complete formalism, HO Core, for which bisimilarity is decidable. We do not know other concurrency formalisms where the same happens. Other peculiarities of HO Core are:

1. it is higher-order, and contextual bisimilarities (barbed congruence) coincide with higher-order bisimilarity (as well as with others, such as context and normal bisimilarities); and

2. it is asynchronous (in that there is no continuation underneath an output), yet asynchronous and synchronous bisimilarities coincide.

We do not know other non-trivial formalisms in which properties (1) or (2) hold (of course (1) makes sense only on higher-order models).

We have also given an axiomatization for bisimilarity. From this we have derived polynomial upper bounds to the decidability of bisimilarity. The axiomatization also intuitively explains why results such as decidability, and the collapse of many forms of bisimilarity, are possible even though HO Core is Turing complete: the bisimilarity relation is very discriminating.

We have used encodings of Minsky machines and of the Post correspondence problem (PCP) for our undecidability results. The encodings are tailored to analyze different problems: undecidability of termination, and undecidability of bisimilarity with static restrictions. The PCP encoding is always divergent, and therefore cannot be used to reason about termination. On the other hand, the encoding of Minsky machines would require at least one restriction for each instruction of the machine, and therefore would have given us a (much) worse result for static restrictions. We find both encodings interesting: they show different ways to exploit higher-order communications for modeling.

We have shown that bisimilarity becomes undecidable with the addition of four static restrictions. We do not know what happens with fewer static restrictions. We also do not know whether the results presented would hold when one abstracts from τ-actions and moves to *weak* equivalences. The problem seems much harder; it reminds us of the situation for BPA and BPP, where strong bisimilarity is decidable but the decidability of weak bisimilarity is a long-standing open problem (Kučera and Jančar, 2006).

Chapter 8

Localities and equivalences

8.1 Introduction

Motivation A natural notion of behavioral equivalence for process calculi is *barbed congruence*. Informally, two processes are barbed-congruent if they behave in the same way (i.e., have the same reductions and the same observables) when placed in similar, but arbitrary, contexts. Due to this quantification on contexts, barbed congruence is unwieldy to use for proofs of equivalence, or to serve as a basis for automated verification tools. One is thus lead to study coinductive characterizations of barbed congruence, typically in the form of bisimilarity relations. For first-order process calculi, such as the π-calculus and its variants, the resulting behavioral theory is well developed, and one can in general readily define bisimilarity relations that characterize barbed congruence.

For higher-order process calculi, the situation is less satisfactory. Simple higher-order calculi, such as HOπ (Sangiorgi, 1992, 1996a), have a well-studied behavioral theory. For HOπ, Sangiorgi has defined *context* and *normal* bisimilarity relations, which both are *sound* with respect to barbed congruence (i.e., are included in barbed congruence) and sometimes *complete* (i.e., they contain barbed congruence), leading to a full characterization. To establish equivalence between processes, context bisimilarity tests all environments which may interact with these processes. For instance, for assessing the equivalence of two processes which consist only of the output of a message on a communication channel a, context bisimilarity needs to consider every interacting system that is capable of doing an input on channel a. Normal bisimilarity improves context bisimilarity by requiring only a single test context. To compare two emitting processes, as above, normal bisimilarity only requires to consider the behavior of the two processes when placed in parallel with a single, finite, particular receiving process. Furthermore, context and normal bisimilarities characterize barbed congruence both in the strong case (where a step from the first process must be simulated by a single step of the second process), and in the weak case (where internal steps are not observable).

Unfortunately, HOπ is not expressive enough to faithfully model concurrent systems with dynamic reconfiguration or strong mobility capabilities. For instance, a running HOπ process cannot be stopped, which prevents the faithful modeling of process failures, of online process replacement, or strong process mobility. It is for this reason that we have seen the emergence of process calculi with (forms of) *process passivation*. Process passivation allows a named process to be stopped and its state captured at any time during its execution. The Kell calculus (Schmitt and Stefani, 2004) and Homer (Bundgaard et al., 2004) are examples of higher-order process calculi with passivation. The behavioral theory of these calculi, or of simpler calculi with localities and migration, is less understood than the one for HOπ, whose proof techniques and relations do not easily carry over, especially in the weak case. We have characterization of weak barbed

Notations:

- X, Y, Z: process variables
- $a, b, \overline{a}, \overline{b}$: names

Syntax:

$$P ::= \mathbf{0} \quad | \quad X \quad | \quad P \mid P \quad | \quad a(X)P \quad | \quad \overline{a}\langle P\rangle P \quad | \quad \nu a.\, P \quad | \quad !P$$

Figure 8.1: Syntax of the Higher-Order π

congruence only in a few cases, such as in Mobile Ambients (Cardelli and Gordon, 1998; Merro and Nardelli, 2005) and its variant NBA (Bugliesi et al., 2005). In the Seal calculus (Vitek and Castagna, 1999; Castagna et al., 2005) and in Homer (Bundgaard et al., 2004; Godskesen and Hildebrandt, 2005), sound weak bisimilarities have been defined in a *delay* style: silent actions are not allowed after an observable one. Because of this limitation, the relations are likely not complete. In the Kell calculus (Schmitt and Stefani, 2004), we propose sound and complete context bisimilarities in the strong case only.

An ideal solution to the characterization problem would be to find a decision algorithm, as described in Chapter 7.[1] The next best solution would be to design a relation akin to normal bisimilarity, which significantly reduces the tests necessary at each step. Finally, a last resort would be to show that contextual bisimilarity is sound and complete in the weak case, for instance by using Howe's method (1996).

In this chapter, we follow the last two steps of this plan. We first define in Section 8.3 a notion of normal bisimulation in an extension of HO Core with localities and passivations, called LHOπP for "Light HOπ with passivation". We next consider adding name restriction (yielding HOπ with passivation, written HOπP), and show that testing large sub-classes of processes is not sufficient, strongly suggesting that defining a form of normal bisimulation for HOπP is impossible. We then show how to apply Howe's method by using a different but equivalent semantics, first by illustrating the approach with HOπ, then applying it to HOπP. Missing proofs and technical details may be found in Lenglet (2010).

8.2 HOπ and HOπP

We first introduce the two main calculi we will be dealing with in this chapter. They may all be seen as extensions of HO Core, adding first restriction (HOπ) then localities (HOπP). We start with the well-known HOπ then illustrate the differences for HOπP.

HOπ, Syntax and Semantics

HOπ (Sangiorgi, 1996a) extends the π-calculus with higher-order communication, which allows process as messages. The syntax of the calculus and some notations can be found in Figure 8.1. As usual, we assume that name restriction $\nu a. \ldots$ binds as far to the right as possible, thus $\nu a.\, P \mid Q$ stands for $\nu a.\, (P \mid Q)$.

In a synchronous higher-order communication $a(X)P \mid \overline{a}\langle Q\rangle R$, the left process $a(X)P$ is waiting for a process (here Q) on name a, and then continues as $\{Q/X\}P$, where $\{Q/X\}P$ is the capture-free substitution of X by Q in P. The right process $\overline{a}\langle Q\rangle R$ sends the process Q on a and then continues as R. In process $a(X)P$, the variable X

[1] Note that this decision algorithm works for the *strong* case only.

is bound. We write $\mathsf{fv}(P)$ the free variables of a process P. If $\mathsf{fv}(P) = \emptyset$, we say that P is *closed*, otherwise it is *open*. In name restriction $\nu a.\, P$, the name a is made local (i.e., bound) to the process P. We write $\mathsf{bn}(P)$ for the bound names of a process P, and $\mathsf{fn}(P)$ for its free names.

Convention on free names and variables We identify processes up to α-conversion of names and variables: process and agents are representative of their α-equivalence class, and are always chosen such that their bound names and variables are distinct from free names and variables. When considering a collection of processes, we assumes that the bound names and bound variables of the processes are chosen to be different from their free names and their free variables. In any discussion or proof, we assume that bound names and bound variables of any process or actions under consideration are chosen to be different from the names and variables occurring free in any other entities under consideration. Note that with this convention, we have $\nu a.\, P \mid Q \equiv P \mid \nu a.\, Q$ without qualification on the free variables of P.

Contextual LTS We now recall the labeled transition system proposed for HOπ by Sangiorgi (1996a). We call it *contextual* since it is used to define *context* bisimilarities. In this LTS, we have three kind of possible evolutions for processes.

- Internal actions labeled by τ, where a process evolves toward a process.
- Message input on a channel a, where a process evolves toward an *abstraction* $(X)Q$. The transition $P \xrightarrow{a} (X)Q$ means that the process P may receive a process R on the name a to continue as $\{R/X\}Q$.
- Message output on a channel a, where a process evolves toward a *concretion* $\nu\widetilde{b}.\, \langle R\rangle Q$. The transition $P \xrightarrow{\overline{a}} \nu\widetilde{b}.\, \langle R\rangle Q$ means that process P may send process R on the name a and continue as Q, and the scope of names $\widetilde{b}$ has to be expanded to encompass the recipient of R. We write $\mathsf{bn}(C) = \widetilde{b}$ the bound names of a concretion, and $o(C)$ the emitted message (here R) of a concretion.

A higher-order communication takes place when a concretion interacts with an abstraction. We define a pseudo-application operator $\bullet$ between an abstraction $F = (X)P$ and a concretion $C = \nu\widetilde{b}.\, \langle R\rangle Q$ by:

$$(X)P \bullet \nu\widetilde{b}.\, \langle R\rangle Q \triangleq \nu\widetilde{b}.\, \{R/X\}P \mid Q$$

We also sometimes use a process application between an abstraction $(X)P$ and a process Q, defined as $(X)P \circ Q \triangleq \{Q/X\}P$.

The LTS rules are given in Figure 8.2, with the exception of the symmetric rules for Par and HO. All the transition rules are straightforward. The transitions are labeled with the names on which the communications may happen, or by τ for an internal evolution. The meta-variable α ranges over all the labels.

Let the set of *agents*, noted A, be the set of processes, abstractions and concretions. A process always evolves towards an agent. Rules Par and Restr require the extension of the parallel composition and restriction operators to all agents, which we define below.

$$\begin{array}{ll}
 & (\nu\widetilde{b}.\, \langle Q\rangle R) \mid P \triangleq \nu\widetilde{b}.\, \langle Q\rangle (R \mid P) \\
(X)Q \mid P \triangleq (X)(Q \mid P) & P \mid (\nu\widetilde{b}.\, \langle Q\rangle R) \triangleq \nu\widetilde{b}.\, \langle Q\rangle (P \mid R) \\
P \mid (X)Q \triangleq (X)(P \mid Q) & \nu a.\, (\nu\widetilde{b}.\, \langle Q\rangle R) \triangleq \nu\widetilde{b}, a.\, \langle Q\rangle R \text{ if } a \in \mathsf{fn}(Q) \\
\nu a.\, (X)Q \triangleq (X)\nu a.\, P & \nu a.\, (\nu\widetilde{b}.\, \langle Q\rangle R) \triangleq \nu\widetilde{b}.\, \langle Q\rangle \nu a.\, R \text{ if } a \notin \mathsf{fn}(Q)
\end{array}$$

$$a(X)P \xrightarrow{a} (X)P \ \textsc{In} \qquad \overline{a}\langle Q\rangle P \xrightarrow{\overline{a}} \langle Q\rangle P \ \textsc{Out} \qquad \frac{P \xrightarrow{\alpha} A}{P \mid Q \xrightarrow{\alpha} A \mid Q} \ \textsc{Par}$$

$$\frac{P \xrightarrow{\alpha} A \qquad \alpha \notin \{a, \overline{a}\}}{\nu a.\, P \xrightarrow{\alpha} \nu a.\, A} \ \textsc{Restr} \qquad \frac{P \xrightarrow{\alpha} A}{!P \xrightarrow{a} A \mid !P} \ \textsc{Replic}$$

$$\frac{P \xrightarrow{a} F \qquad P \xrightarrow{\overline{a}} C}{!P \xrightarrow{\tau} F \bullet C \mid !P} \ \textsc{Replic-HO} \qquad \frac{P \xrightarrow{a} F \qquad Q \xrightarrow{\overline{a}} C}{P \mid Q \xrightarrow{\tau} F \bullet C} \ \text{HO}$$

Figure 8.2: Contextual labeled transition system for HOπ

Contexts A *context* is a process with a hole $\square$. In HOπ, contexts are defined as follows.

$$\mathbb{C} ::= \square \quad | \quad \mathbb{C} \mid P \quad | \quad P \mid \mathbb{C} \quad | \quad a(X)\mathbb{C} \quad | \quad \overline{a}\langle \mathbb{C}\rangle P \quad | \quad \overline{a}\langle P\rangle \mathbb{C} \quad | \quad \nu a.\, \mathbb{C} \quad | \quad !\mathbb{C}$$

Filling the hole of a context $\mathbb{C}$ with a process P yields the process $\mathbb{C}\{P\}$. Unlike substitution, filling a context may result in variable or name capture. A relation that is stable under contexts is called a *congruence*.

Structural congruence Structural congruence $\equiv$ is the smallest congruence verifying the following laws.

$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad P \mid Q \equiv Q \mid P \qquad P \mid \mathbf{0} \equiv P \qquad \nu a.\, \nu b.\, P \equiv \nu b.\, \nu a.\, P$$

$$\nu a.\, \mathbf{0} \equiv \mathbf{0} \qquad \nu a.\, (P \mid Q) \equiv P \mid \nu a.\, Q \qquad !P \equiv P \mid !P$$

HOπ, Equivalences

Barbed congruence A very common notion of observational equivalence is *barbed congruence*, defined by Milner and Sangiorgi (1992). This notion relies on the reduction of the calculus under consideration, and on an *observation predicate*. In the calculi we will study, we are interested in the interaction between a process and its environment, thus the observations are on channels on which communication is possible.

Definition 8.2.1. *The strong observability predicates $P \downarrow_\mu$ is defined as follows*

- *We have $P \downarrow_a$ iff $P \equiv \nu\widetilde{b}.\, a(X)Q \mid R$ with $a \notin \widetilde{b}$.*
- *We have $P \downarrow_{\overline{a}}$ iff $P \equiv \nu\widetilde{b}.\, \overline{a}\langle Q\rangle R \mid S$ with $a \notin \widetilde{b}$.*

Barbed bisimulation is a relation on *closed processes* that relates processes with identical barbs that may keep this property by reduction.

Definition 8.2.2. *A relation $\mathcal{R}$ on closed process is a* strong barbed simulation *iff for all $P \mathcal{R} Q$,*

- *if $P \downarrow_\mu$ then $Q \downarrow_\mu$;*
- *if $P \xrightarrow{\tau} P'$, then there exists Q' such that $Q \xrightarrow{\tau} Q'$ and $P' \mathcal{R} Q'$.*

A relation $\mathcal{R}$ is a strong barbed bisimulation *iff $\mathcal{R}$ and $\mathcal{R}^{-1}$ are both strong barbed simulations. The processes P and Q are* strongly barbed bisimilar *iff there exists a strong barbed bisimulation $\mathcal{R}$ such that $P \mathcal{R} Q$.*

We are ready to define strong barbed congruence.

Definition 8.2.3. *Closed processes P and Q are* strongly barbed congruent*, written $P \sim_b Q$, iff $\mathbb{C}\{P\}$ and $\mathbb{C}\{Q\}$ are strongly barbed bisimilar for every context $\mathbb{C}$.*

Intuitively, strong barbed congruence says that there is no way to distinguish two processes, even by putting them in arbitrary contexts and letting them run for a while.

Up to this point we have worked in the *strong* setting, where each reduction of P is matched by exactly one reduction of Q. In the *weak* case, a reduction of P may be matched by an arbitrary number (possibly zero) of reductions of Q, e.g., to compare a program and an optimized version that needs fewer computation steps. We write $\stackrel{\tau}{\Rightarrow}$ the reflexive and transitive closure of $\stackrel{\tau}{\rightarrow}$.

We define the weak observability predicate by $P \Downarrow_\mu$ iff there exists P' such that $P \stackrel{\tau}{\Rightarrow} P' \downarrow_\mu$. We define barbed simulation as follows.

Definition 8.2.4. *A relation $\mathcal{R}$ on closed process is a* barbed simulation *iff for all $(P, Q) \in \mathcal{R}$:*

- *if $P \downarrow_\mu$ then $Q \Downarrow_\mu$;*
- *if $P \stackrel{\tau}{\rightarrow} P'$, then there exists Q' such that $Q \stackrel{\tau}{\Rightarrow} Q'$ and $(P', Q') \in \mathcal{R}$.*

A relation $\mathcal{R}$ is a barbed bisimulation *iff $\mathcal{R}$ and $\mathcal{R}^{-1}$ are both barbed simulations. The processes P and Q are* barbed bisimilar*, iff there exists a barbed bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$.*

Finally, we define barbed congruence.

Definition 8.2.5. *The closed processes P and Q are* barbed congruent*, written $P \approx_b Q$, iff $\mathbb{C}\{P\}$ and $\mathbb{C}\{Q\}$ are barbed bisimilar for every context $\mathbb{C}$.*

Barbed congruence is a very fine relation, in the sense that it relates only processes which may not be distinguished. Proving that two processes are not barbed congruent amounts to finding a context which distinguishes them. On the other hand, proving that processes actually are barbed congruent is much more difficult, as one as to test every possible context. This is why simpler relations, such as context bisimulations, have been introduced.

Bisimulations for Higher Order Calculi We recall here the definitions and observations of Sangiorgi concerning bisimulations for HOπ (1992; 1996a).

Requiring the contents of the messages to be syntactically identical, as in first order calculi, is clearly too precise, as it would distinguish processes like $\overline{a}\langle\mathbf{0}\rangle$ and $\overline{a}\langle\mathbf{0} \mid \mathbf{0}\rangle$. An approach, developed by Thomsen in the setting of CHOCS and Plain CHOCS (1990; 1993), consists of requiring the emitted messages to be *bisimilar* instead of syntactically equal. The resulting relation is called *higher-order bisimilarity*. In the setting of HOπ, it is defined as follows.

Definition 8.2.6. *A relation $\mathcal{R}$ on closed processes is a higher order simulation iff $P \mathcal{R} Q$ implies:*

- *for all $P \stackrel{\tau}{\rightarrow} P'$, there exists Q' such that $Q \stackrel{\tau}{\rightarrow} Q'$ and $P' \mathcal{R} Q'$;*

- *for all* $P \xrightarrow{a} F$*, there exists* F' *such that* $Q \xrightarrow{a} F'$ *and for all closed processes* R*,* $F \circ R \mathcal{R} F' \circ R$*;*
- *for all* $P \xrightarrow{\overline{a}} \nu\widetilde{b}.\langle R\rangle S$*, there exists* $\nu\widetilde{b'}.\langle R'\rangle S'$ *such that* $Q \xrightarrow{\overline{a}} \nu\widetilde{b'}.\langle R'\rangle S'$*,* $R \mathcal{R} R'$*,* $S \mathcal{R} S'$*.*

A relation $\mathcal{R}$ *is higher order bisimulation iff* $\mathcal{R}$ *and* $\mathcal{R}^{-1}$ *are higher order simulations. Two closed processes* P *and* Q *are higher order bisimilar, noted* $P \dot{\sim} Q$*, iff there exists a higher order bisimulation* $\mathcal{R}$ *such that* $P \mathcal{R} Q$*.*

Unfortunately, this relation is still too fine as it separates the contents of the message from its continuation. Indeed, Sangiorgi has shown that the processes $\overline{a}\langle\mathbf{0}\rangle!b.\,\mathbf{0}$ and $\overline{a}\langle b.\,\mathbf{0}\rangle!b.\,\mathbf{0}$ are barbed congruent, yet they are not higher order bisimilar.

Sangiorgi thus proposes *context* bisimulation as a LTS-based behavioral equivalence. The definition of (early strong) context bisimilarity is as follows.

Definition 8.2.7 (**Early strong context bisimilarity**)**.** *A binary relation* $\mathcal{R}$ *on closed processes is an* early strong context simulation *iff* $P \mathcal{R} Q$ *implies:*

- *for all* $P \xrightarrow{\tau} P'$*, there exists* Q' *such that* $Q \xrightarrow{\tau} Q'$ *and* $P' \mathcal{R} Q'$*;*
- *gor all* $P \xrightarrow{a} F$*, for all closed concretions* C*, there exists* F' *such that* $Q \xrightarrow{a} F'$ *and* $F \bullet C \mathcal{R} F' \bullet C$*;*
- *for all* $P \xrightarrow{\overline{a}} C$*, for all closed abstractions* F*, there exists* C' *such that* $Q \xrightarrow{\overline{a}} C'$ *and* $F \bullet C \mathcal{R} F \bullet C'$*.*

A relation $\mathcal{R}$ *is an* early strong context bisimulation *iff* $\mathcal{R}$ *and* $\mathcal{R}^{-1}$ *are early strong context simulations. Two closed processes* P *and* Q *are* early strong context bisimilar*, noted* $P \sim Q$*, iff there exists an early strong context bisimulation* $\mathcal{R}$ *such that* $P \mathcal{R} Q$*.*

The relation is said to be *early* because the other part of the communication (the concretion in the message input case, the abstraction in the message output case) is given *before* the matching transition. Thus the matching process may change the agent it reduces to depending on this information. The converse approach, where the other part of the communication is given *after* the matching transition, is said to be *late*.

Definition 8.2.8 (**Late strong context bisimilarity**)**.** *A binary relation* $\mathcal{R}$ *on closed processes is a* late strong context simulation *iff* $P \mathcal{R} Q$ *implies*

- *For all* $P \xrightarrow{\tau} P'$*, there exists* Q' *such that* $Q \xrightarrow{\tau} Q'$ *and* $P' \mathcal{R} Q'$*.*
- *For all* $P \xrightarrow{a} F$*, there exists* F' *such that* $Q \xrightarrow{a} F'$ *and for all closed concretions* C*, we have* $F \bullet C \mathcal{R} F' \bullet C$*.*
- *For all* $P \xrightarrow{\overline{a}} C$*, there exists* C' *such that* $Q \xrightarrow{\overline{a}} C'$ *and for all closed abstractions* F*, we have* $F \bullet C \mathcal{R} F \bullet C'$*.*

A relation $\mathcal{R}$ *is a* late strong context bisimulation *iff* $\mathcal{R}$ *and* $\mathcal{R}^{-1}$ *are late strong context simulations. Two closed processes* P *and* Q *are* late strong context bisimilar*, noted* $P \sim_l Q$*, iff there exists a late strong context bisimulation* $\mathcal{R}$ *such that* $P \mathcal{R} Q$*.*

One cas easily show that late bisimilarity is more constraining than early bisimilarity, i.e., $\sim_l \subseteq \sim$. The inverse is generally not true, but for HOπ we actually have $\sim_l = \sim$. Moreover, these relations *characterize* strong barbed congruence: $\sim = \sim_b$.

We now give definitions for the weak case, where internal steps $\xrightarrow{\tau}$ are not observable. For every action γ other than τ, we write $\overset{\gamma}{\Rightarrow}$ for $\overset{\tau}{\Rightarrow}\xrightarrow{\gamma}$ (as higher order steps result

in concretions and abstractions, they may not reduce further; silent steps after this reduction are taken into account in the definition of weak simulation below). We define weak early context bisimilarity as:

Definition 8.2.9 (**Weak early context bisimilarity**). *A binary relation $\mathcal{R}$ on closed processes is an* early weak context simulation *iff $P \mathcal{R} Q$ implies:*

- *for all $P \xrightarrow{\tau} P'$, there exists Q' such that $Q \stackrel{\tau}{\Rightarrow} Q'$ and $P' \mathcal{R} Q'$;*
- *for all $P \xrightarrow{a} F$, for all closed concretions C, there exists F', Q' such that $Q \stackrel{a}{\Rightarrow} F'$, $F' \bullet C \stackrel{\tau}{\Rightarrow} Q'$, and $F \bullet C \mathcal{R} Q'$;*
- *for all $P \xrightarrow{\overline{a}} C$, for all closed abstractions F, there exists C', Q' such that $Q \stackrel{\overline{a}}{\Rightarrow} C'$, $F \bullet C' \stackrel{\tau}{\Rightarrow} Q'$ and $F \bullet C \mathcal{R} Q'$.*

A relation $\mathcal{R}$ is an early weak context bisimulation *iff $\mathcal{R}$ and $\mathcal{R}^{-1}$ are early weak context simulations. Two closed processes P and Q are* early weak context bisimilar*, noted $P \approx Q$, iff there exists an early weak context bisimulation $\mathcal{R}$ such that $P \mathcal{R} Q$.*

The late version, written $\approx_l$, is defined by replacing the two last clauses by:

- for all $P \xrightarrow{a} F$, there exists F' such that $Q \stackrel{a}{\Rightarrow} F'$, and for all closed concretions C, there exists Q' such that $F' \bullet C \stackrel{\tau}{\Rightarrow} Q'$, and $F \bullet C \mathcal{R} Q'$;
- for all $P \xrightarrow{\overline{a}} C$, there exists C' such that $Q \stackrel{\overline{a}}{\Rightarrow} C'$, and for all closed abstractions F, there exists Q' such that, $F \bullet C' \stackrel{\tau}{\Rightarrow} Q'$ and $F \bullet C \mathcal{R} Q'$.

These relations are *correct* in relation to barbed congruence: $\approx \subseteq \approx_b$ and $\approx_l \subseteq \approx_b$. Completeness has been shown in the case of *image finite* processes.

Definition 8.2.10. *A process P is image finite iff:*

- *the set $\{P_i \mid P \stackrel{\tau}{\Rightarrow} P_i\}$ is finite;*
- *for every C and every name a, the set $\{P' \mid P \stackrel{a}{\Rightarrow} F \wedge F \bullet C \stackrel{\tau}{\Rightarrow} P'\}$ is finite;*
- *for every F and every name a, the set $\{P' \mid P \stackrel{\overline{a}}{\Rightarrow} C \wedge F \bullet C \stackrel{\tau}{\Rightarrow} P'\}$ is finite.*

HOπP, Syntax and Semantics

We now describe a simple extension of HOπ with localities and passivation. Our goal is to study the effect of passivation on bisimulations for more complex calculi, such as Homer (Bundgaard et al., 2004) or the Kell Calculus (Schmitt and Stefani, 2004). As these calculi include many other feature, notably ways to control the communication between processes, directly studying them would not only be quite complex, it would not tell us about the direct influence of passivation.

The change to the syntax is quite minor: we simply add localities of the form $a[P]$. As long as passivation does not occur, a locality is a transparent evaluation context: processes inside may reduce and communicate with the outside. At any moment, passivation may occur and the locality becomes a concretion $\langle P \rangle \mathbf{0}$ through a transition on $\overline{a}$. Passivation is a local action τ if and only if there is a process listening on a to receive the locality's contents.

We extend localities to every agent, defining $a[(X)P] \triangleq (X)a[P]$ and $a[\nu\widetilde{b}.\langle Q \rangle R] \triangleq \nu\widetilde{b}.\langle Q \rangle a[R]$ (in this latter case, by our convention we have $a \notin \widetilde{b}$). The labeled transition system of Figure 8.2 is extended with the two following rules.

$$\frac{P \xrightarrow{\alpha} A}{a[P] \xrightarrow{\alpha} a[A]} \ \textsc{Loc} \qquad\qquad a[P] \xrightarrow{\overline{a}} \langle P \rangle \mathbf{0} \ \textsc{Passiv}$$

These definitions imply that a name restriction may cross a locality boundary by scope extrusion when a message is sent whose contents mention the restricted name. However, as we do not extend structural congruence, this is the only case when scope extrusion occurs. We will see in Section 8.4 that this restriction is crucial.

HOπP, Equivalences

Observables in HOπP are names and co-names on which a communication or passivation is enabled, i.e., $P \downarrow_\mu$ iff $P \xrightarrow{\mu}$. The definitions of barbed congruences, $\sim_b$ and $\approx_b$, remain the same as for HOπ (definitions 8.2.3 and 8.2.5).

We cannot directly reuse the context bisimulation from HOπ. Indeed, consider the two following processes.

$$P_1 = \overline{a}\langle \mathbf{0}\rangle !b.\,\mathbf{0} \quad P_2 = \overline{a}\langle b.\,\mathbf{0}\rangle !b.\,\mathbf{0}$$

We have seen that they are barbed congruent (and context bisimilar) in HOπ. However they are note barbed congruent in HOπP. Indeed, the context $\mathbb{C} \triangleq c[\Box] \mid a(X)X \mid c(X)\mathbf{0}$ distinguishes them. We have $\mathbb{C}\{P_1\} \xrightarrow{\tau} c[!b.\,\mathbf{0}] \mid \mathbf{0} \mid c(X)\mathbf{0} \triangleq P_3$ by communication on a. Process $\mathbb{C}\{P_2\}$ can only match this by $\mathbb{C}\{P_2\} \xrightarrow{\tau} c[!b.\,\mathbf{0}] \mid b.\,\mathbf{0} \mid c(X)\mathbf{0} \triangleq P_4$. Triggering passivation on c, we get $P_3 \xrightarrow{\tau} \mathbf{0}$ and $P_4 \xrightarrow{\tau} b.\,\mathbf{0}$. Both processes are not barbed congruent.

In a concretion $\nu\widetilde{a}.\,\langle R\rangle S$, message R may be sent out of a locality c while its continuation S remains in c. If passivation on c is triggered, the continuation S may be discarded (as shown with $\mathbb{C}\{P_1\}$ and $\mathbb{C}\{P_2\}$) or may be put in another context. Passivation may thus split the process R from the continuation S and put them in distinct contexts, which is impossible without passivation. We address this issue following the same approach as in Homer or the Kell Calculus, testing the messages and continuations in distinct *evaluation contexts* $\mathbb{E}$. Evaluation contexts are defined as follows.

$$\mathbb{E} ::= \Box \quad | \quad \nu a.\,\mathbb{E} \quad | \quad \mathbb{E} \mid P \quad | \quad P \mid \mathbb{E} \quad | \quad a[\mathbb{E}]$$

We actually call these contexts, which are use for observation purposes, *bisimulation contexts*. We are now ready to define early strong context bisimilarity for HOπP.

Definition 8.2.11. *A relation $\mathcal{R}$ on closed processes is an early strong context simulation iff $P\ \mathcal{R}\ Q$ implies $\mathsf{fn}(P) = \mathsf{fn}(Q)$ and:*

- *for all $P \xrightarrow{\tau} P'$, there exists Q' such that $Q \xrightarrow{\tau} Q'$ and $P'\ \mathcal{R}\ Q'$;*
- *for all $P \xrightarrow{a} F$, for all C, there exists F' such that $Q \xrightarrow{a} F'$ and $F \bullet C\ \mathcal{R}\ F' \bullet C$;*
- *for all $P \xrightarrow{\overline{a}} C$, for all F, there exists C' telle que $Q \xrightarrow{\overline{a}} C'$ and for all $\mathbb{E}$, we have $F \bullet \mathbb{E}\{C\}\ \mathcal{R}\ F \bullet \mathbb{E}\{C'\}$.*

A relation $\mathcal{R}$ is an early strong context bisimulation iff $\mathcal{R}$ and $\mathcal{R}^{-1}$ are early strong context simulations. Two closed processes P and Q are early strong context bisimilar, noted $P \sim Q$, iff there exists an early strong context bisimulation $\mathcal{R}$ such that $P\ \mathcal{R}\ Q$.

The condition on free names $\mathsf{fn}(P)$ is required because scope extrusion allows to observe the free names of a process.

We now define context bisimulation in the weak case.

Definition 8.2.12. *A relation $\mathcal{R}$ on closed processes is an early weak context simulation iff $P\ \mathcal{R}\ Q$ implies $\mathsf{fn}(P) = \mathsf{fn}(Q)$ and:*

- *for all $P \xrightarrow{\tau} P'$, there exists Q' such that $Q \overset{\tau}{\Rightarrow} Q'$ and $P'\ \mathcal{R}\ Q'$;*

- *for all* $P \xrightarrow{a} F$*, for all* C*, there exist* F', Q' *such that* $Q \overset{a}{\Rightarrow} F'$*,* $F' \bullet C \overset{\tau}{\Rightarrow} Q'$ *and* $F \bullet C \mathcal{R} Q'$*;*
- *for all* $P \xrightarrow{\overline{a}} C$*, for all* F*, there exists* C' *such that* $Q \overset{\overline{a}}{\Rightarrow} C'$ *and for all* $\mathbb{E}$*, there exists* Q' *such that* $F \bullet \mathbb{E}\{C'\} \overset{\tau}{\Rightarrow} Q'$ *and* $F \bullet C \mathcal{R} Q'$*.*

A relation $\mathcal{R}$ *is an early weak context bisimulation iff* $\mathcal{R}$ *and* $\mathcal{R}^{-1}$ *are early weak context simulations. Two closed processes* P *and* Q *are early weak context bisimilar, noted* $P \approx Q$*, iff there exists an early weak context bisimulation* $\mathcal{R}$ *such that* $P \mathcal{R} Q$*.*

We have shown in (Schmitt and Stefani, 2004) that early strong contextual bisimilarity coincide with strong barbed congruence. We describe in 8.7 a proof method to show that early weak context bisimilarity is correct.

Beforehand, we attempt to find a more tractable notion of bisimilarity, based on *normal bisimilarity*. To this end, we start by studying a simpler variant of HOπP where there is no name restriction, then move on to full HOπP.

8.3 Normal bisimulation for LHOπP

In this section, we show that the behavioral theory of a calculus with passivation and without restriction is similar to the one of HOπ: we are able to define simple context and normal bisimilarities which characterize barbed congruence.

Syntax, semantics, and equivalences

The syntax and semantics of the Light Higher-Order π-calculus with Passivation (LHOπP) are the one of HOπP when removing restriction. As before, observables consist of the names and co-names on which communication or passivation is enabled. The definition of barbed congruence is unchanged.

As the definition of context bisimilarity is simpler in this calculus, we re-state it here. It also coincide with strong barbed congruence.

Definition 8.3.1. *A relation* $\mathcal{R}$ *on closed processes is an early strong simulation iff* $P \mathcal{R} Q$ *implies:*

- *for all* $P \xrightarrow{\tau} P'$*, there exists* Q' *such that* $Q \xrightarrow{\tau} Q'$ *and* $P' \mathcal{R} Q'$*;*
- *for all* $P \xrightarrow{a} F$*, for all closed processes* R*, there exists* F' *such that* $Q \xrightarrow{a} F'$ *and* $F \circ R \mathcal{R} F' \circ R$*;*
- *for all* $P \xrightarrow{\overline{a}} \langle R \rangle S$*, there exists* R', S' *such that* $Q \xrightarrow{\overline{a}} \langle R' \rangle S'$*,* $R \mathcal{R} R'$*,* $S \mathcal{R} S'$*.*

A relation $\mathcal{R}$ *is an early strong bisimulation iff* $\mathcal{R}$ *and* $\mathcal{R}^{-1}$ *are early strong simulations. Two closed processes* P *and* Q *are early strong bisimilar, noted* $P \sim Q$*, iff there exists an early strong bisimulation* $\mathcal{R}$ *such that* $P \mathcal{R} Q$*.*

Theorem 8.3.2. $\sim_b = \sim$*.*

We define weak context bisimilarity as usual.

Definition 8.3.3. *A relation* $\mathcal{R}$ *on closed processes is an early weak simulation iff* $P \mathcal{R} Q$ *implies:*

- *for all* $P \xrightarrow{\tau} P'$*, there exists* Q' *such that* $Q \overset{\tau}{\Rightarrow} Q'$ *and* $P' \mathcal{R} Q'$*;*
- *for all* $P \xrightarrow{a} F$*, for all closed processes* R*, there exists* F'*,* Q' *such that* $Q \overset{a}{\Rightarrow} F'$*,* $F' \circ R \overset{\tau}{\Rightarrow} Q'$*, and* $F \circ R \mathcal{R} Q'$*;*

- *for all* $P \xrightarrow{\overline{a}} \langle R \rangle S$*, there exists* R', S'', S' *such that* $Q \overset{\overline{a}}{\Rightarrow} \langle R' \rangle S''$, $S'' \overset{\tau}{\Rightarrow} S'$, $R \mathcal{R} R'$, *and* $S \mathcal{R} S'$.

A relation $\mathcal{R}$ *is an early weak bisimulation iff* $\mathcal{R}$ *and* $\mathcal{R}^{-1}$ *are early weak simulations. Two closed processes* P *and* Q *are early weak bisimilar, noted* $P \approx Q$*, iff there exists an early weak bisimulation* $\mathcal{R}$ *such that* $P \mathcal{R} Q$.

As in the strong case, we have:

Theorem 8.3.4. *Early context bisimilarity is correct.*

The proof of this theorem, which uses Howe's method, can be found in Lenglet (2010). We also have completeness of the early version on image-finite processes (Definition 8.2.10).

Theorem 8.3.5. *Early weak context bisimilarity is complete on image-finite processes.*

Normal bisimulation

In this section, we show that testing one process is enough in the abstraction case and we define a correct and complete bisimulation without any universal quantification, similar to Sangiorgi's normal bisimulation. We write $\overline{m}.\,\mathbf{0}$ for $\overline{m}\langle \mathbf{0} \rangle \mathbf{0}$.

In HOπ, testing abstractions $(X)P$ and $(X)Q$ with a trigger $\overline{m}.\,\mathbf{0}$ (where m does not occur in P, Q) is sufficient to establish context bisimilarity: if $\{\overline{m}.\,\mathbf{0}/X\}P$ and $\{\overline{m}.\,\mathbf{0}/X\}Q$ are context bisimilar, then for all R, $\{R/X\}P$ and $\{R/X\}Q$ are context bisimilar. We first show that this result does not hold in LHOπP. Consider the following processes:

$$P_1 = !a[X] \mid !\overline{a}\langle \mathbf{0} \rangle \mathbf{0} \qquad Q_1 = P_1 \mid X$$

We have $\{\overline{m}.\,\mathbf{0}/X\}P_1 \sim \{\overline{m}.\,\mathbf{0}/X\}Q_1$, but $\{\overline{m}.\,\overline{n}.\,\mathbf{0}/X\}P_1$ and $\{\overline{m}.\,\overline{n}.\,\mathbf{0}/X\}Q_1$ (where m, n do not occur in P, Q) are not bisimilar. Thus we found a process R such that $\{R/X\}P$ and $\{R/X\}Q$ are not strong context bisimilar.

We first give the idea why $P_m = \{\overline{m}.\,\mathbf{0}/X\}P_1$ and $Q_m = \{\overline{m}.\,\mathbf{0}/X\}Q_1 = \overline{m}.\,\mathbf{0} \mid P_m$ are bisimilar. All transitions from P_m are easily matched by Q_m, and reciprocally for Q_m, except for transition $Q_m \xrightarrow{\overline{m}} \equiv P_m$. It can only be matched by $P_m \xrightarrow{\overline{m}} a[\mathbf{0}] \mid P_m = P'_m$. We now prove that P_m and P'_m are bisimilar. We just have to check passivation on a, i.e. transition $P'_m \xrightarrow{\overline{a}} \langle \mathbf{0} \rangle P_m$. It is clearly matched by message sending on a in P_m, i.e. $P_m \xrightarrow{\overline{a}} \langle \mathbf{0} \rangle P_m$. Consequently P_m and Q_m are early strong context bisimilar.

However $P_{m,n} = \{\overline{m}.\,\overline{n}.\,\mathbf{0}/X\}P_1$ and $Q_{m,n} = \{\overline{m}.\,\overline{n}.\,\mathbf{0}/X\}Q_1$ are not bisimilar. We consider the transition $Q_{m,n} \xrightarrow{\overline{m}} \overline{n}.\,\mathbf{0} \mid P_{m,n} = Q'_{m,n}$, which can only be matched by a transition $P_{m,n} \xrightarrow{\overline{m}} a[\overline{n}.\,\mathbf{0}] \mid P_{m,n} = P'_{m,n}$. Passivation of a in $P'_{m,n}$, i.e. transition $P'_{m,n} \xrightarrow{\overline{a}} \langle \overline{n}.\,\mathbf{0} \rangle P_{m,n}$, can only be matched by $Q'_{m,n} \xrightarrow{\overline{a}} \langle \overline{m}.\,\overline{n}.\,\mathbf{0} \rangle Q'_{m,n}$ or $Q'_{m,n} \xrightarrow{\overline{a}} \langle \mathbf{0} \rangle Q'_{m,n}$. Since $\overline{n}.\,\mathbf{0} \nsim \overline{m}.\,\overline{n}.\,\mathbf{0}$ and $\overline{n}.\,\mathbf{0} \nsim \mathbf{0}$, $P'_{m,n}$ and $Q'_{m,n}$ (and consequently $P_{m,n}$ and $Q_{m,n}$) are not bisimilar.

In the previous example, a process $\overline{m}.\,\mathbf{0}$ is not enough to distinguish between process variables inside and outside a locality: a $\xrightarrow{\overline{m}}$ transition from a process $\overline{m}.\,\mathbf{0}$ in a locality can be matched by a $\xrightarrow{\overline{m}}$ transition from a $\overline{m}.\,\mathbf{0}$ outside any locality. This distinction, however, becomes possible with a process $\overline{m}.\,\overline{n}.\,\mathbf{0}$. Suppose we have $\{\overline{m}.\,\overline{n}.\,\mathbf{0}/X\}P$ bisimilar to $\{\overline{m}.\,\overline{n}.\,\mathbf{0}/X\}Q$, with m, n not occurring in P, Q. A $\xrightarrow{\overline{m}}$ transition in P is matched by a $\xrightarrow{\overline{m}}$ transition in Q: the two resulting processes P', Q' may now perform one and

only one $\xrightarrow{\overline{n}}$ transition from a process $\overline{n}.\mathbf{0}$ in an evaluation context. If the process $\overline{n}.\mathbf{0}$ is in a locality a in P', then it can be send in a message on a after a passivation. The process Q' has to match with a message sending on a; since the contents of the messages are pairwise bisimilar, the message from Q' has to contain $\overline{n}.\mathbf{0}$. Consequently the only occurrence of $\overline{n}.\mathbf{0}$ was in an evaluation context in Q' and may be sent in a message on a: it is possible if and only if $\overline{n}.\mathbf{0}$ is in a locality a.

To summarize, when a process $\overline{m}.\overline{n}.\mathbf{0}$ in a locality performs a $\xrightarrow{\overline{m}}$ transition, it has to be matched by a process $\overline{m}.\overline{n}.\mathbf{0}$ in a locality with the same name. More precisely, if a process $\overline{m}.\overline{n}.\mathbf{0}$ in P performs a $\xrightarrow{\overline{m}}$ transition and is matched by a process $\overline{m}.\overline{n}.\mathbf{0}$ in Q, then the locality hierarchies around $\overline{m}.\overline{n}.\mathbf{0}$ in P and Q are the same. This result is a consequence of the following decomposition lemma:

Lemma 8.3.6 (Decomposition)**.** *Let P, Q two open processes such that $\mathsf{fv}(P,Q) \subseteq \{X\}$ and m, n two names which do not occur in P, Q. If $\{\overline{m}.\overline{n}.\mathbf{0}/X\}P \sim \{\overline{m}.\overline{n}.\mathbf{0}/X\}Q$ and $\{\overline{m}.\overline{n}.\mathbf{0}/X\}P \xrightarrow{\overline{m}} \{\overline{n}.\mathbf{0}/X_i\}\{\overline{m}.\overline{n}.\mathbf{0}/X\}P' = P_n$, then there exists Q' such that $\{\overline{m}.\overline{n}.\mathbf{0}/X\}Q \xrightarrow{\overline{m}} \{\overline{n}.\mathbf{0}/X_j\}\{\overline{m}.\overline{n}.\mathbf{0}/X\}Q' = Q_n$ and $P_n \sim Q_n$ (by definition of the bisimulation). Moreover, we are in one of the following cases.*

- *There exist P_1, Q_1 such that $P_n = \overline{n}.\mathbf{0} \mid P_1$, $Q_n = \overline{n}.\mathbf{0} \mid Q_1$ with $P_1 \sim Q_1$.*
- *There exist $a_1, \ldots a_k$, $P_1 \ldots P_{k+1}$, $Q_1 \ldots Q_{k+1}$ such that*

$$P_n = a_1[\ldots a_{k-1}[a_k[\overline{n}.\mathbf{0} \mid P_{k+1}] \mid P_k] \mid P_{k-1} \ldots] \mid P_1$$

  and

$$Q_n = a_1[\ldots a_{k-1}[a_k[\overline{n}.\mathbf{0} \mid Q_{k+1}] \mid Q_k] \mid Q_{k-1} \ldots] \mid Q_1$$

  and for all $1 \leq j \leq k+1$, $P_j \sim Q_j$.

The lemma gives several results on two matching transitions $\{\overline{m}.\overline{n}.\mathbf{0}/X\}P \xrightarrow{\overline{m}} P_n$ and $\{\overline{m}.\overline{n}.\mathbf{0}/X\}Q \xrightarrow{\overline{m}} Q_n$:

- the resulting $\overline{n}.\mathbf{0}$ is only under parallel compositions and localities (and not under replication or choice operators) in P_n, Q_n;
- if $\overline{n}.\mathbf{0}$ is not under a locality in P_n, it is not under a locality in Q_n and the processes in parallel with $\overline{n}.\mathbf{0}$ in P_n, Q_n are bisimilar;
- if $\overline{n}.\mathbf{0}$ is under a locality hierarchy $a_1, \ldots a_k$ in P_n, then it is under the same locality hierarchy in Q_n, and the locality process bodies $P_1, \ldots P_{k+1}$, $Q_1, \ldots Q_{k+1}$ are pairwise bisimilar.

For instance, if we have $\{\overline{n}.\mathbf{0}/X\}P = a[b[\overline{n}.\mathbf{0} \mid P_3] \mid P_2] \mid P_1$, then we have $\{\overline{n}.\mathbf{0}/X\}Q \equiv a[b[\overline{n}.\mathbf{0} \mid Q_3] \mid Q_2] \mid Q_1$ with $P_1 \sim Q_1$, $P_2 \sim Q_2$, $P_3 \sim Q_3$.

From this lemma, we can show that testing abstractions with $\overline{m}.\overline{n}.\mathbf{0}$ is enough, as stated in the following theorem.

Theorem 8.3.7. *Let P, Q two open processes such that $\mathsf{fv}(P,Q) \subseteq \{X\}$ and m, n two names which do not occur in P, Q. If $\{\overline{m}.\overline{n}.\mathbf{0}/X\}P \sim \{\overline{m}.\overline{n}.\mathbf{0}/X\}Q$, then for all closed processes R, we have $\{R/X\}P \sim \{R/X\}Q$*

The theorem allows us to define a bisimulation without any universal quantification, similar to the normal bisimulation of Sangiorgi:

Definition 8.3.8. *A relation $\mathcal{R}$ on closed processes is a normal simulation iff $P \mathcal{R} Q$ implies:*

- *for all $P \xrightarrow{\tau} P'$, there exists Q' such that $Q \xrightarrow{\tau} Q'$ and $P' \mathcal{R} Q'$;*
- *for all $P \xrightarrow{a} F$, there exists F' such that $Q \xrightarrow{a} F'$ and for two names m, n which do not occur in processes P, Q, we have $F \circ \overline{m}.\overline{n}.\mathbf{0} \mathcal{R} F' \circ \overline{m}.\overline{n}.\mathbf{0}$.*
- *for all $P \xrightarrow{\overline{a}} \langle R \rangle S$, there exists R', S' such that $Q \xrightarrow{\overline{a}} \langle R' \rangle S'$, $R \mathcal{R} R'$ and $S \mathcal{R} S'$.*

A relation $\mathcal{R}$ is a normal bisimulation iff $\mathcal{R}$ and $\mathcal{R}^{-1}$ are normal simulations. Two closed processes P and Q are normal bisimilar, noted $P \sim_n Q$, iff there exists a normal bisimulation $\mathcal{R}$ such that $P \mathcal{R} Q$.

As a corollary of Theorem 8.3.7, normal bisimilarity coincides with early bisimilarity.

Corollary 8.3.9. $\sim_n = \sim$

Proof. The inclusion $\sim \subseteq \sim_n$ is easy by definition. The inclusion $\sim_n \subseteq \sim$ is a consequence of Theorem 8.3.7. □

These results may be extended to the weak case:

Theorem 8.3.10. *Let P, Q two open processes such that $\mathsf{fv}(P, Q) \subseteq \{X\}$ and m, n two names which do not occur in P, Q. If $\{\overline{m}.\overline{n}.\mathbf{0}/X\}P \approx \{\overline{m}.\overline{n}.\mathbf{0}/X\}Q$, then for all closed processes R, we have $\{R/X\}P \approx \{R/X\}Q$*

We define weak normal bisimilarity as follows.

Definition 8.3.11. *A relation $\mathcal{R}$ on closed processes is a weak normal simulation iff $P \mathcal{R} Q$ implies:*

- *for all $P \xrightarrow{\tau} P'$, there exists Q' such that $Q \overset{\tau}{\Rightarrow} Q'$ and $P' \mathcal{R} Q'$;*
- *for all $P \xrightarrow{a} F$, there exists F' such that $Q \overset{a}{\Rightarrow} F'$ and for two names m, n which do not occur in processes P, Q, there exists Q' such that $F' \circ \overline{m}.\overline{n}.\mathbf{0} \overset{\tau}{\Rightarrow} Q'$ and $F \circ \overline{m}.\overline{n}.\mathbf{0} \mathcal{R} Q'$.*
- *for all $P \xrightarrow{\overline{a}} \langle R \rangle S$, there exists R', S'', S' such that $Q \overset{\overline{a}}{\Rightarrow} \langle R' \rangle S''$, $S'' \overset{\tau}{\Rightarrow} S'$, $R \mathcal{R} R'$ and $S \mathcal{R} S'$.*

A relation $\mathcal{R}$ is a weak normal bisimulation iff $\mathcal{R}$ and $\mathcal{R}^{-1}$ are weak normal simulations. Two closed processes P and Q are weakly normal bisimilar, noted $P \approx_n Q$, iff there exists a weak normal bisimulation $\mathcal{R}$ such that $P \mathcal{R} Q$.

As in the strong case, we have

Corollary 8.3.12. $\approx_n = \approx$

Hence in a calculus with passivation and without restriction, we can define a suitable bisimulation without any universal quantification. We show in the next section that this does not hold with HOπ, i.e, when adding back restriction.

8.4 Normal bisimulations and HOπP

In this section, we present counter-examples that clearly suggest that defining a notion of normal bisimulation for HOπP is a doomed prospect. More precisely, we prove that testing large sub-classes of processes (*abstraction-free* processes) is not enough to guarantee bisimilarity of abstraction. In Lenglet (2010), another large class of processes, *finite* processes (processes which do not have non-terminating reductions), is also shown not to be enough to test for bisimilarity when the calculus has a choice (+) operator.

Abstraction-free processes

In the following, we omit the trailing zeros to improve readability; m in an agent definition stands for $m.\,\mathbf{0}$. We also write $\nu ab.\,P$ for $\nu a.\,\nu b.\,P$. Let $\mathbf{0}_m \triangleq \nu x.\,x.\,m$. Process $\mathbf{0}_m$ is bisimilar to $\mathbf{0}$ except it has a free name m. We define the following abstractions:

$$\begin{aligned}(X)P &\triangleq (X)\nu nb.\,(b[X \mid \nu m.\,\overline{a}\langle \mathbf{0}_m\rangle(m \mid n \mid \overline{m}.\,\overline{m}.\,p)] \mid \overline{n}.\,b(Y)(Y \mid Y))\\ (X)Q &\triangleq (X)\nu mnb.\,(b[X \mid \overline{a}\langle \mathbf{0}\rangle(m \mid n \mid \overline{m}.\,\overline{m}.\,p)] \mid \overline{n}.\,b(Y)(Y \mid Y))\end{aligned}$$

The two abstractions differ in the process emitted on a and in the position of name restriction on m (inside or outside hidden locality b). An abstraction-free process is a process built with the regular HOπP syntax minus message input $a(X)P$.

Lemma 8.4.1. *Let R be an abstraction-free process. We have $(X)P \circ R \sim (X)Q \circ R$.*

Since R is abstraction-free, it cannot receive the message emitted by P or Q on a; consequently R cannot interact with P or Q. Let $P_{m,R} = \nu nb.\,(b[R \mid m \mid n \mid \overline{m}.\,\overline{m}.\,p] \mid \overline{n}.\,b(Y)(Y \mid Y))$, F be an abstraction, and $\mathbb{E}$ be an evaluation context such that $m \notin \mathsf{fn}(\mathbb{E}, F)$. We now prove that $(X)P \circ R \xrightarrow{\overline{a}} \nu m.\,\langle \mathbf{0}_m\rangle P_{m,R}$ is matched by $(X)Q \circ R \xrightarrow{\overline{a}} \langle \mathbf{0}\rangle \nu m.\,P_{m,R}$, i.e., that we have $\nu m.\,(F \circ \mathbf{0}_m \mid \mathbb{E}\{P_{m,R}\}) \sim F \circ \mathbf{0} \mid \mathbb{E}\{\nu m.\,P_{m,R}\}$. Since $m \notin \mathsf{fn}(\mathbb{E}, F)$, there is no interaction between $F, \mathbb{E}$, and $P_{m,R}$. Moreover, the inert process $\mathbf{0}_m$ does not interfere. Hence the possible transitions from $\nu m.\,(F \circ \mathbf{0}_m \mid \mathbb{E}\{P_{m,R}\})$ are only from $F, \mathbb{E}$, and internal actions in $P_{m,R}$, and are matched by the same transitions in $F \circ \mathbf{0} \mid \mathbb{E}\{\nu m.\,P_{m,R}\}$.

However, abstractions $(X)P$ and $(X)Q$ may have different behaviors with an argument which may receive on a, like $a(Z)q$, where q is a first-order name such that $p \neq q$. By communication on a, we have

$$(X)Q \circ a(Z)q \xrightarrow{\tau} \nu mnb.\,(b[q \mid m \mid n \mid \overline{m}.\,\overline{m}.\,p] \mid \overline{n}.\,b(Y)(Y \mid Y)) \triangleq Q_1$$

Since Q_1 may perform a $\xrightarrow{q}$ transition, $(X)Pa(Z)q$ may only reply by

$$(X)P \circ a(Z)q \xrightarrow{\tau} \nu nb.\,(b[\nu m.\,q \mid m \mid n \mid \overline{m}.\,\overline{m}.\,p] \mid \overline{n}.\,b(Y)(Y \mid Y)) \triangleq P_1$$

Notice that in P_1, the restriction on m remains inside hidden locality b.

After synchronisation on n and passivation on b, we have

$$Q_1(\xrightarrow{\tau})^2 \nu mnb.\,(q \mid q \mid m \mid m \mid \overline{m}.\,\overline{m}.\,p \mid \overline{m}.\,\overline{m}.\,p) \triangleq Q_2$$

(process inside b in Q_1 is duplicated). After two synchronisations on m, we have

$$Q_2(\xrightarrow{\tau})^2 \nu mnb.\,(q \mid q \mid p \mid \overline{m}.\,\overline{m}.\,p) \triangleq Q_3$$

and Q_3 may perform a $\xrightarrow{p}$ transition. These transitions cannot be matched by P_1. Performing the duplication, we have

$$P_1(\xrightarrow{\tau})^2 \nu nb.\,((\nu m.\,q \mid m \mid \overline{m}.\,\overline{m}.\,p) \mid (\nu m.\,q \mid m \mid \overline{m}.\,\overline{m}.\,p)) \triangleq P_2$$

Each copied sub-process $q \mid m \mid \overline{m}.\,\overline{m}.\,p$ of P_2 has its own private copy of m, and we can no longer perform any transition to have the observable p. More generally, the sequence of transitions $Q_1(\xrightarrow{\tau})^4 \xrightarrow{p}$ cannot be matched by P_1, consequently Q_1 and P_1 (and therefore $(X)Q \circ a(Z)q$ and $(X)P \circ a(Z)q$) are not bisimilar.

This counter-example shows that testing abstractions with abstraction-free processes (such as $\overline{m}.\overline{n}.\mathbf{0}$) is not enough to potentially distinguish them. Consequently, we have to test abstractions with processes which performs some kind of message input. Notice that this counter-example relies on scope extrusion; another family of tests that does not rely on scope extrusion but on the number of reductions is presented in Lenglet (2010). We show there that for every number n, there are processes that may not be distinguished after n steps but that may be distinguished after $m > n$ steps.

We thus conjecture that we cannot define a normal bisimilarity in HOπP, i.e., that we cannot define a sound and complete strong bisimilarity with fewer tests than early strong context bisimilarity.

8.5 Contextual Semantics and Howe's Method

A classic and powerful technique to prove the congruence of a relation is Howe's method (1996). Unfortunately it is not adapted to the usual semantics and contextual bisimilarity of HOπ. In this section, we recall the proof scheme of Howe's method, then explain why we cannot use it with early context bisimilarities, which are the usual candidate relations for characterizing barbed congruence in calculi inheriting from the π-calculus.

Howe's Method

Howe's method (Howe, 1996; Baldamus, 1998; Gordon, 1995) is a systematic proof technique to show that a candidate relation $\mathcal{R}$ is a congruence. The method can be divided in three steps.

- Definition of *Howe's closure* $\mathcal{R}^\bullet$ and proofs of its basic properties. Howe's closure $\mathcal{R}^\bullet$ contains $\mathcal{R}$ and is a congruence by construction.
- Proof of a simulation-like property for $\mathcal{R}^\bullet$.
- Conclusive step: proof that $\mathcal{R}$ and $\mathcal{R}^\bullet$ coincide on closed processes. Since $\mathcal{R}^\bullet$ is a congruence, we conclude that $\mathcal{R}$ is a congruence.

The definition of Howe's closure relies on the open extension of $\mathcal{R}$, noted $\mathcal{R}^\circ$: it extends the definition of the relation $\mathcal{R}$ to open processes.

Definition 8.5.1 (**Open extension**). *Let P and Q be two open processes. We have $P\ \mathcal{R}^\circ\ Q$ iff $P\sigma\ \mathcal{R}\ Q\sigma$ for all substitutions σ that close P and Q.*

Howe's closure is inductively defined as the smallest congruence which contains $\mathcal{R}^\circ$ and is closed by right relation composition by $\mathcal{R}^\circ$.

Definition 8.5.2 (**Howe closure**). *Howe's closure $\mathcal{R}^\bullet$ of a relation $\mathcal{R}$ is the smallest relation verifying:*

- $\mathcal{R}^\circ \subseteq \mathcal{R}^\bullet$;
- $\mathcal{R}^\bullet \mathcal{R}^\circ \subseteq \mathcal{R}^\bullet$;
- *for all operators op of the language, if $\widetilde{P}\ \mathcal{R}^\bullet\ \widetilde{Q}$, then $op(\widetilde{P})\ \mathcal{R}^\bullet\ op(\widetilde{Q})$.*

By definition, Howe's closure is a congruence, and the composition with $\mathcal{R}^\circ$ allows some transitivity and gives some additional properties to the relation.

To prove that Howe's closure is a simulation (second step of the method), we need the following property.

Lemma 8.5.3. *Let $\mathcal{R}$ be an equivalence. If $P\ \mathcal{R}^\bullet\ Q$ and $R\ \mathcal{R}^\bullet\ S$, then we have $\{R/X\}P\ \mathcal{R}^\bullet\ \{S/X\}Q$.*

We sketch the proof in order to give an idea on why the transitive item $\mathcal{R}^\bullet\mathcal{R}^\circ\subseteq\mathcal{R}^\bullet$ is needed in Definition 8.5.2. The proof is by induction on the derivation of $P\ \mathcal{R}^\bullet\ Q$. Suppose we have $P\ \mathcal{R}^\circ\ Q$. Since $R\ \mathcal{R}^\bullet\ S$ and $\mathcal{R}^\bullet$ is a congruence, we have $\{R/X\}P\ \mathcal{R}^\bullet\ \{S/X\}P$. Let σ be a substitution that closes P and Q except for X; by open extension definition, we have $\{S/X\}P\sigma\ \mathcal{R}\ \{S/X\}Q\sigma$, i.e., we have $\{S/X\}P\ \mathcal{R}^\circ\ \{S/X\}Q$. Finally we have $\{R/X\}P\ \mathcal{R}^\bullet\mathcal{R}^\circ\ \{S/X\}Q$, hence we have $\{R/X\}P\ \mathcal{R}^\bullet\ \{S/X\}Q$. The other cases are easy using the induction hypothesis.

In our case, we want to prove that a bisimilarity $\mathcal{B}$ is a congruence. By definition, we have $\mathcal{B}^\circ\subseteq\mathcal{B}^\bullet$. To have the reverse inclusion, we prove that $\mathcal{B}^\bullet$ is a bisimulation. However we cannot prove directly that $\mathcal{B}^\bullet$ is symmetric; instead, we prove a simulation property (second step of the method), and we use the following property.

Lemma 8.5.4. *Let $\mathcal{R}$ be an equivalence. Then the reflexive and transitive closure $(\mathcal{R}^\bullet)^*$ of $\mathcal{R}^\bullet$ is symmetric.*

Proof. By proving that $P(\mathcal{R}^\bullet)^{-1}Q$ implies $P(\mathcal{R}^\bullet)^*Q$ for all P, Q. It is done by induction on $P(\mathcal{R}^\bullet)^{-1}Q$. □

Using the simulation result, we can prove that the restriction of $(\mathcal{B}^\bullet)^*$ to closed terms is a bisimulation. Consequently we have $\mathcal{B}\subseteq\mathcal{B}^\bullet\subseteq\ (\mathcal{B}^\bullet)^*\ \subseteq\mathcal{B}$ on closed terms, and we conclude that $\mathcal{B}$ is a congruence.

The main difficulty is to prove the simulation-like property for Howe's closure. In the following subsection, we explain why we cannot use directly Howe's method with early context bisimilarities (Definitions 8.2.7 and 8.2.9).

Communication Problem with Contextual Semantics

We want to prove that $\mathcal{B}^\bullet$ is a simulation. Proving directly that a congruence is a simulation may raise transitivity issues (Schmitt and Stefani, 2004). Howe's method deals with this issue by establishing a stronger simulation result which features some transitivity in its clauses. Given a bisimilarity $\mathcal{B}$ based on a LTS $P \xrightarrow{\lambda} A$, the simulation result follows the pattern below:

Let $P\ \mathcal{B}^\bullet\ Q$. IF $P \xrightarrow{\lambda} A$, then there exists B such that $Q \xrightarrow{\lambda'} B$ and for all $\lambda\ \mathcal{B}^\bullet\ \lambda'$, we have $A\ \mathcal{B}^\bullet\ B$.

This pattern is quite close to a higher-order bisimilarity clause. It supposes that Howe's closure can be extended to labels λ. For instance, suppose we want to apply Howe's method to strong late context bisimilarity $\sim_l$ (definition 8.2.8), which has first been done for Homer in Bundgaard et al. (2004). We extend Howe's closure to abstractions: we have $F \sim_l^\bullet F'$ iff for all C, we have $F \bullet C \sim_l^\bullet F' \bullet C$. We then have:

Lemma 8.5.5. *If $P \sim_l^\bullet Q$, then:*

- *for all $P \xrightarrow{\tau} P'$, there exists Q' such that $Q \xrightarrow{\tau} Q'$ and $P' \sim_l^\bullet Q'$;*
- *for all $P \xrightarrow{a} F$, there exists F' such that $Q \xrightarrow{a} F'$ and $F \sim_l^\bullet F'$;*
- *for all $P \xrightarrow{\overline{a}} C$, there exists C' such that $Q \xrightarrow{\overline{a}} C'$ and for all closed F, F' such that $F \sim_l^\bullet F'$ we have $F \bullet C \sim_l^\bullet F' \bullet C'$.*

Notice that some transitivity is built in the output clause of this simulation-like property: F and C are directly related to F' and C'. Finding a suitable simulation-like property featuring transitivity is more difficult for early context bisimilarity. Sticking to the pattern given earlier, one may think of the following property:

Conjecture 1. *If $P \sim^\bullet Q$, then:*

- *for all $P \xrightarrow{\tau} P'$, there exists Q' such that $Q \xrightarrow{\tau} Q'$ and $P' \sim^\bullet Q'$;*
- *for all $P \xrightarrow{a} F$, for all $C \sim^\bullet C'$, there exists F' such that $Q \xrightarrow{a} F'$ and $F \bullet C \sim^\bullet F' \bullet C'$;*
- *for all $P \xrightarrow{\overline{a}} C$, for all $F \sim^\bullet F'$ there exists C' such that $Q \xrightarrow{\overline{a}} C'$ and $F \bullet C \sim^\bullet F' \bullet C'$.*

These clauses raise several issues. First, we have to find extensions of Howe's closure to abstractions and concretions which fit an early style. Even if we have found such extensions, we have problems to conduct an inductive proof of conjecture 1 with higher-order communication. Suppose we conduct a proof by induction on the derivation of $P \sim^\bullet Q$. Suppose we are in the parallel case, i.e. we have $P = P_1 \mid P_2$ and $Q = Q_1 \mid Q_2$, with $P_1 \sim^\bullet Q_1$ and $P_2 \sim^\bullet Q_2$. Suppose that we have $P \xrightarrow{\tau} P'$, and the transition comes from rule HO: we have $P_1 \xrightarrow{a} F$, $P_2 \xrightarrow{\overline{a}} C$ and $P' = F \bullet C$. We want to find Q' such that $Q \xrightarrow{\tau} Q'$ and $P' \sim^\bullet Q'$. We want to use the same rule HO, hence we have to find F', C' such that $Q \xrightarrow{\tau} F' \bullet C'$. However we cannot use the input clause of the induction hypothesis with P_1, Q_1: to have a F' such that $Q_1 \xrightarrow{a} F'$, we have to find first a concretion C' such that $C \sim^\bullet C'$. We cannot use the output clause with P_2, Q_2 either: to have a C' such that $Q_2 \xrightarrow{\overline{a}} C'$, we have to find first an abstraction F' such that $F \sim^\bullet F'$. We cannot bypass this mutual dependency, therefore the inductive proof of conjecture 1 fails in the higher-order communication case.

Godskesen and Hildebrandt (2005) deal with this issue in Homer by making the concretion clause independent from abstractions. The considered bisimilarity is therefore no longer early, but *input-early*: the input clause is in an early style and the output clause is in a late one. While this approach allows to prove the correction of the bisimilarity, it still does not characterize barbed congruence.

In the following section, we propose a new semantics for HOπ which coincide with the contextual one and which allow the use of Howe's method to prove congruence results with early strong and weak bisimilarities.

8.6 Complementary Semantics for HOπ

We define a LTS $P \xrightarrow{\alpha} P'$ where processes always evolve towards other processes. We have three kinds of transitions: internal actions $P \overset{\tau}{\mapsto} P'$, message input $P \xrightarrow{a,R} P'$, and message output $P \overset{\overline{a},Q}{\longmapsto} P'$. We call this new LTS *complementary* since in the output action, we put the context which complements P in the label λ of the transition (more details below). For higher-order labels $\lambda = a, R$ or $\lambda = \overline{a}, R$ we define $n(\lambda)$ as the name a on which the communication may happen. Rules of the LTS can be found in Figure 8.3, except the symmetric of rules $\textsc{Par}^\pi$ and HO^π. We first detail the form of transitions in the complementary LTS.

Internal action transitions $P \overset{\tau}{\mapsto} P'$ are the same as in the contextual LTS $P \xrightarrow{\tau} P'$. A message input transition $P \xrightarrow{a,R} P'$ means that process P may receive the process R as a message on channel a and become P'. In the contextual style, it means that there exists an abstraction $F = (X)P''$ such that $P \xrightarrow{a} (X)P''$ and $P' = \{R/X\}P''$.

$$a(X)P \xrightarrow{a,R} \{R/X\}P \quad \textsc{In}^\pi \qquad\qquad \frac{Q \xrightarrow{a,R} Q'}{\overline{a}\langle R\rangle S \xmapsto{\overline{a},Q} Q' \mid S} \quad \textsc{Out}^\pi$$

$$\frac{P_1 \overset{\lambda}{\mapsto} P_1'}{P_1 \mid P_2 \overset{\lambda}{\mapsto} P_1' \mid P_2} \quad \textsc{Par}^\pi \qquad \frac{P \overset{\lambda}{\mapsto} P' \qquad a \notin n(\lambda)}{\nu a.\, P \overset{\lambda}{\mapsto} \nu a.\, P'} \quad \textsc{Restr}^\pi \qquad \frac{P \xmapsto{\overline{a},Q} P'}{P \mid Q \overset{\tau}{\mapsto} P'} \quad \text{HO}^\pi$$

Figure 8.3: Complementary LTS for HOπ

Complementary and contextual message input transitions are fundamentally the same, except that the complementary action is written in the early style.

The main difference lies with output action transitions. The transition $P \xmapsto{\overline{a},Q} P'$ means that P may send a message on channel a, Q may receive on a, and the communication on a between P and Q results in P'. Note that it is not the same as writing a contextual transition in an early style $P \xrightarrow{\overline{a},F} F \bullet C$: instead of putting an abstraction F in the label, we put a process Q (without any free process variable). There is a tight correspondence with an output action contextual transition, though: the transition $P \xmapsto{\overline{a},Q} P'$ means that there exists F, C such that $P \xrightarrow{\overline{a}} C$, $Q \xrightarrow{a} F$, and $P' = F \bullet C$.

Rules of the LTS in Figure 8.3 are standard, except rules HO$^\pi$ and $\textsc{Out}^\pi$. In rule HO$^\pi$, the premise $P \xmapsto{\overline{a},Q} P'$ means that P and Q can communicate on a name a and the result is P', i.e., $P \mid Q \xrightarrow{\tau} P'$ (by communication on a). Rule $\textsc{Out}^\pi$ has a premise (unlike its equivalent rule $\textsc{Out}$) since in the conclusion we need the result Q' of the input of process R on channel a by Q.

Remark 8.6.1. *Notice that in a message output $P \xmapsto{\overline{a},Q} P'$, the message itself does not appear in the label or cannot be directly deduced from the transition. It is unusual in higher-order LTS: for instance in the contextual semantics of HOπ, Kell, or Homer, emitted processes appear in concretions. In Mobile Ambients (Merro and Nardelli, 2005), moving ambients also appear in concretions; in the Seal-calculus (Castagna and Nardelli, 2002), moved seals appear in labels (seal freeze P_z or seal chained P^z). Hiding the message in the LTS makes Howe's method easier to apply.*

The correspondence between the complementary LTS and the contextual LTS is exact, and it is established by the following lemma:

Lemma 8.6.2. *Let P be an HOπ process. We have:*

- $P \xrightarrow{\tau} P'$ *iff* $P \overset{\tau}{\mapsto}\equiv P'$;
- *If $P \xrightarrow{a} F$, then for all R we have $P \xrightarrow{a,R} F \circ R$. Conversely, if $P \xrightarrow{a,R} P'$, then there exists F such that $P \xrightarrow{a} F$ and $P' = F \circ R$;*
- *If $P \xrightarrow{\overline{a}} C$, then for all Q such that $Q \xrightarrow{a} F$, we have $P \xmapsto{\overline{a},Q} F \bullet C$. Conversely, if $P \xmapsto{\overline{a},Q} P'$, then there exists F, C such that $P \xrightarrow{\overline{a}} C$, $Q \xrightarrow{a} F$, $P' \equiv F \bullet C$.*

The correspondence between the two LTS is up to structural congruence because of scope extrusion: in Sangiorgi's contextual LTS, scope extrusion is performed iff the name belongs to the free names of the message, while in the complementary LTS, we do not have such a condition. For instance, if we consider $P = \nu b.\, \overline{a}\langle \mathbf{0}\rangle \overline{b}\langle \mathbf{0}\rangle \mathbf{0}$, then we have $P \xrightarrow{\overline{a}} C = \langle \mathbf{0}\rangle \nu b.\, \overline{b}\langle \mathbf{0}\rangle \mathbf{0}$ and for all $F = (X)Q'$, we have $F \bullet C = \{\mathbf{0}/X\}Q' \mid$

$\nu b.\, \bar{b}\langle \mathbf{0}\rangle \mathbf{0} \triangleq P_1$. With the complementary LTS, for all Q such that $Q \xrightarrow{a} F$ we have $P \xmapsto{\overline{a},Q} \nu b.\, (\{\mathbf{0}/X\}Q' \mid \bar{b}\langle \mathbf{0}\rangle \mathbf{0}) \triangleq P_2$. We have $P_1 \neq P_2$ but we have $P_1 \equiv P_2$.

Complementary Bisimilarities

We now define strong complementary bisimilarity and prove its soundness by proving it is a congruence using Howe's method. The result in itself, i.e., the definition of a sound bisimilarity in HOπ, is not new (Sangiorgi, 1992, 1996a). However it allows us to explain why complementary semantics is well suited to apply Howe's method. In the HOπ case, strong complementary bisimilarity is simply the bisimilarity associated to the complementary LTS. Let λ range over complementary LTS labels, i.e. $\lambda = \tau$, $\lambda = a, R$ or $\lambda = \overline{a}, R$, where R is a process.

Definition 8.6.3 (**Strong complementary bisimilarity**). *A binary relation $\mathcal{R}$ on closed processes is a* strong complementary simulation *iff $P \mathcal{R} Q$ implies for all $P \xrightarrow{\lambda} P'$, there exists Q' such that $Q \xrightarrow{\lambda} Q'$ and $P' \mathcal{R} Q'$.*

A relation $\mathcal{R}$ is a strong complementary bisimulation *iff $\mathcal{R}$ and $\mathcal{R}^{-1}$ are strong complementary simulations. Two closed processes P and Q are* strong complementary bisimilar*, noted $P \sim_m Q$, iff there exists a strong complementary bisimulation $\mathcal{R}$ such that $P \mathcal{R} Q$.*

As in context bisimilarity, in the message output case $P \xmapsto{\overline{a},R} P'$, the matching transition $Q \xmapsto{\overline{a},R} Q'$ still depends on a receiving entity (here R). However, instead of considering a context which directly receives the message (an abstraction F), we consider a process R which evolves toward an abstraction. This small nuance allows us to use Howe's method to prove soundness of $\sim_m$. To this end we prove the following simulation-like property for $\sim_m^\bullet$, the Howe closure of $\sim_m$:

Lemma 8.6.4. *Let P, Q be closed processes. If $P \sim_m^\bullet Q$ then:*

- *if $P \overset{\tau}{\mapsto} P'$, then there exists Q' such that $Q \overset{\tau}{\mapsto} Q'$ and $P' \sim_m^\bullet Q'$;*
- *if $P \xrightarrow{a,R} P'$, then for all $R \sim_m^\bullet R'$, there exists Q' such that $Q \xrightarrow{a,R'} Q'$ and $P' \sim_m^\bullet Q'$;*
- *if $P \xmapsto{\overline{a},T} P'$, then for all $T \sim_m^\bullet T'$, there exists Q' such that $Q \xmapsto{\overline{a},T'} Q'$ and $P' \sim_m^\bullet Q'$.*

We do not have the same problem as in Section 8.5 with the higher-order communication case. We recall that in this case, we have $P_1 \mid P_2 \sim_m^\bullet Q_1 \mid Q_2$ with $P_1 \sim_m^\bullet Q_1$, $P_2 \sim_m^\bullet Q_2$ and $P_1 \xmapsto{\overline{a},P_2} P'$. We can apply directly the message output clause of the induction hypothesis: there exists Q' such that $Q_1 \xmapsto{\overline{a},Q_2} Q'$ and $P' \sim_m^\bullet Q'$. We conclude that $Q_1 \mid Q_2 \overset{\tau}{\mapsto} Q'$ (by rule HO$^\pi$) with $P' \sim_m^\bullet Q'$ as wished.

Theorem 8.6.5. *$\sim_m$ is a congruence.*

Proving Lemma 8.6.4 is the only difficult part of the proof of Theorem 8.6.5. The complete proofs can be found in (Lenglet, 2010).

Following the correspondence result between the two LTS (Lemma 8.6.2), we now prove that the bisimilarities have the same discriminating power. The differences in the message output clauses are covered mainly with Lemma 8.6.2. The bisimilarities differ also in how they deal with input actions: complementary bisimilarity tests with a

process while context bisimilarity tests with a concretion. Testing with all concretions includes tests with $\langle P \rangle \mathbf{0}$, which are the same as tests with P (up to $\equiv$). Consequently one inclusion is easy to establish:

Lemma 8.6.6. *We have* $\sim \subseteq \sim_m$.

The proof is done by showing that $\sim$ is a strong complementary bisimilarity (up to $\equiv$). The reverse inclusion requires the congruence result on $\sim_m$ (Theorem 8.6.5).

Lemma 8.6.7. *We have* $\sim_m \subseteq \sim$.

We prove the inclusion by showing that $\sim_m$ is an early strong context bisimulation (up to $\equiv$). In the message input case, we have roughly $\{R/X\}P' \sim_m \{R/X\}Q'$; by congruence it implies that $\nu\widetilde{b}.\,(\{R/X\}P' \mid S) \sim_m \nu\widetilde{b}.\,(\{R/X\}Q' \mid S)$, i.e. $(X)P' \bullet \nu\widetilde{b}.\,\langle R \rangle S \sim_m (X)Q' \bullet \nu\widetilde{b}.\,\langle R \rangle S$. With Theorem 8.6.5, tests with processes are as discriminatory as tests with concretions.

Correspondence also holds in the weak case. In this case, two processes P and Q may evolve independently before interacting with each other. Since a transition $P \overset{\overline{a},Q}{\longmapsto} P'$ includes a communication between P and Q, we have to authorize Q to perform τ-actions before interacting with P in the weak output transition. We define $P \overset{\overline{a},Q}{\Longmapsto} P'$ as $P \overset{\tau}{\Rightarrow} \overset{\overline{a},Q'}{\longmapsto} \overset{\tau}{\Rightarrow} P'$ with $Q \overset{\tau}{\Rightarrow} Q'$. Weak complementary semantics mimics weak context semantics in the following way.

Lemma 8.6.8. *Let P be an HOπ process.*

- *We have $P \overset{\tau}{\Rightarrow} P'$ iff $P \overset{\tau}{\Mapsto} P'$.*
- *Let R be a closed process. If $P \overset{a}{\Rightarrow} F$ and $F \circ R \overset{\tau}{\Rightarrow} P'$ then we have $P \overset{a,R}{\Longrightarrow} F \circ R$. If $P \overset{a,R}{\Longrightarrow} P'$, then there exists F such that $P \overset{a}{\Rightarrow} F$ and $F \circ R \overset{\tau}{\Rightarrow} P'$.*
- *If $P \overset{\overline{a}}{\Rightarrow} C$, then for all Q such that $Q \overset{a}{\Rightarrow} F$ and $F \bullet C \overset{\tau}{\Rightarrow} P'$, we have $P \overset{\overline{a},Q}{\Longmapsto} P'$. If $P \overset{\overline{a},Q}{\Longmapsto} P'$, then there exists F, C such that $P \overset{\overline{a}}{\Rightarrow} C$, $Q \overset{a}{\Rightarrow} F$, and $F \bullet C \overset{\tau}{\Rightarrow} P'$.*

We now give the definition of the weak bisimilarity associated to the complementary LTS.

Definition 8.6.9. *A relation $\mathcal{R}$ on closed processes is a weak complementary simulation iff $P \mathcal{R} Q$ implies that, for all $P \xrightarrow{\lambda} P'$, there exists Q' such that $Q \overset{\lambda}{\Rightarrow} Q'$ and $P' \mathcal{R} Q'$.*

A relation $\mathcal{R}$ is a weak complementary bisimulation iff $\mathcal{R}$ and $\mathcal{R}^{-1}$ are weak complementary simulations. Two closed processes P and Q are weak complementary bisimilar, noted $P \approx_m Q$, iff there exists an weak complementary bisimulation $\mathcal{R}$ such that $P \mathcal{R} Q$.

Using the same proof schemes as in the strong case, we have the following results.

Theorem 8.6.10. *The relation $\approx_m$ is a congruence.*

Lemma 8.6.11. *We have $\approx_m = \approx$.*

We do not detail proofs for weak relations since they are similar to the strong ones. We now turn to HOπP where the congruence result for a weak relation is new.

$$a(X)P \xrightarrow{a,R} \{R/X\}P \quad \textsc{In}_i^p \qquad \frac{P \xrightarrow{a,R} P'}{P \mid Q \xrightarrow{a,R} P' \mid Q} \quad \textsc{Par}_{i\tau}^p \qquad \frac{P \xrightarrow{b,R} P'}{a[P] \xrightarrow{b,R} a[P']} \quad \textsc{Loc}_i^p$$

$$\frac{P \xrightarrow{a,R} P' \qquad b \neq a}{\nu b.\, P \xrightarrow{a,R} \nu b.\, P'} \quad \textsc{Restr}_i^p \qquad \frac{P \overset{\tau}{\mapsto} P'}{P \mid Q \overset{\tau}{\mapsto} P' \mid Q} \quad \textsc{Par}_{i\tau}^p \qquad \frac{P \overset{\tau}{\mapsto} P'}{a[P] \overset{\tau}{\mapsto} a[P']} \quad \textsc{Loc}_{i\tau}^p$$

$$\frac{P \overset{\tau}{\mapsto} P'}{\nu a.\, P \overset{\tau}{\mapsto} \nu a.\, P'} \quad \textsc{Restr}_{i\tau}^p \qquad \frac{P \xrightarrow{\overline{a},Q,\Box}_{\widetilde{b}} P'}{P \mid Q \xrightarrow{\tau} P'} \quad \text{HO}_\tau^p$$

Figure 8.4: Complementary LTS for HOπP: internal and message input actions

8.7 Characterizing Barbed Congruence for HOπP

In this section, we define a complementary semantics for HOπP, and a weak complementary bisimilarity $\approx_m$ which coincides with early weak context bisimilarity $\approx$ (Definition 8.2.12). We then prove that $\approx_m$ is a congruence (and hence sound with respect to weak barbed congruence) using Howe's method. We also prove that $\approx_m$ is complete on image-finite processes, yielding a co-inductive characterization of weak-barbed congruence in a calculus featuring passivation and restriction.

Complementary LTS

As in Section 8.6 we define a complementary LTS which considers processes instead of abstractions in the message output case. However we have two additional issues with HOπP. First, we have to include evaluation contexts $\mathbb{E}$ since they appear in bisimilarity definitions (Definitions 8.2.11 and 8.2.12). Second, handling scope extrusion is more involved than in HOπ, since the scope of restricted names may extend beyond locality boundaries by communication but not by structural congruence. We cannot always extrude names and still have an equivalent semantics (up to $\equiv$) as in HOπ.

Internal action transitions $P \xrightarrow{\tau} P'$ and input action transitions $P \xrightarrow{a,R} P'$ are similar to the corresponding HOπ complementary transitions , except that we have to add rules for localities.. LTS rules dealing with these transitions can be found in Figure 8.4 except the symmetric counterpart of rules $\textsc{Par}_{i\tau}^p$, $\textsc{Par}_{i\tau}^p$, and HO_τ^p. Rule HO_τ^p relies on message output transitions and is explained later.

In HOπP, context bisimilarities test a message output with an abstraction F and a bisimulation context $\mathbb{E}$. As in HOπ, complementary output actions $P \xrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'$ consider a receiving process Q instead of F. We have to add contexts $\mathbb{E}$ in our labels to keep the same discriminating power, and we also use a set of names $\widetilde{b}$ to deal with scope extrusion. Transition $P \xrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'$ means that P is put under context $\mathbb{E}$ and emits a message on a, which is received by Q, i.e. we have $\mathbb{E}\{P\} \mid Q \xrightarrow{\tau} P'$ by communication on a. In the contextual style, it means that there exists F, C such that $P \xrightarrow{\overline{a}} C$, $Q \xrightarrow{a} F$, and $P' = F \bullet \mathbb{E}\{C\}$. Output rules can be found in Figure 8.5, except for the symmetric of rule $\textsc{Par}_o^p$.

Scope extrusion may happen in the process under consideration (e.g. $P = \nu c.\, \overline{a}\langle R\rangle S$ with $c \in \mathsf{fn}(R)$) or because of the bisimulation context $\mathbb{E}$ (e.g. $P = \overline{a}\langle R\rangle S$ and $\mathbb{E} = d[\nu c.\, (\Box \mid \overline{c}\langle \mathbf{0}\rangle \mathbf{0})]$ with $c \in \mathsf{fn}(R)$). We first define auxiliary transitions $P \xrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'$, where we do not allow the latter kind of capture, and we then give rules for general

$$\frac{\mathsf{fn}(R) = \widetilde{b} \qquad Q \xrightarrow{a,R} Q' \qquad \mathsf{bn}(\mathbb{E}) \cap \widetilde{b} = \emptyset}{\overline{a}\langle R\rangle S \xrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} Q' \mid \mathbb{E}\{S\}} \ \textsc{Out}_o^p \qquad \frac{P \xrightarrow{\overline{a},Q,\mathbb{E}\{b[\Box]\}}_{\widetilde{b}} P'}{b[P] \xrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'} \ \textsc{Loc}_o^p$$

$$\frac{\mathsf{fn}(P) = \widetilde{b} \qquad Q \xrightarrow{b,P} Q' \qquad \mathsf{bn}(\mathbb{E}) \cap \widetilde{b} = \emptyset}{b[P] \xrightarrow{\overline{b},Q,\mathbb{E}}_{\widetilde{b}} Q' \mid \mathbb{E}\{\mathbf{0}\}} \ \textsc{Passiv}_o^p \qquad \frac{P_1 \xrightarrow{\overline{a},Q,\mathbb{E}\{\Box \mid P_2\}}_{\widetilde{b}} P'}{P_1 \mid P_2 \xrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'} \ \textsc{Par}_o^p$$

$$\frac{P \xrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P' \qquad c \neq a \qquad c \in \widetilde{b}}{\nu c.\, P \xrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}\setminus c} \nu c.\, P'} \ \textsc{Extr}_o^p$$

$$\frac{P \xrightarrow{\overline{a},Q,\mathbb{E}\{\nu c.\Box\}}_{\widetilde{b}} P' \qquad c \neq a \qquad c \notin \widetilde{b}}{\nu c.\, P \xrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'} \ \textsc{Restr}_o^p \qquad \frac{P \xrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'}{P \xrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'} \ \textsc{CFree}_o^p$$

$$\frac{P \xrightarrow{\overline{a},Q,\mathbb{E}\{\mathbb{F}\}}_{\widetilde{b}} P' \qquad c \in \widetilde{b}}{P \xrightarrow{\overline{a},Q,\mathbb{E}\{\nu c.\mathbb{F}\}}_{\widetilde{b}} \nu c.\, P'} \ \textsc{Capt}_o^p$$

Figure 8.5: Complementary LTS for HOπP: message output actions

output transitions.

Rule $\textsc{Out}_o^p$ deals with message output $\overline{a}\langle R\rangle S \xrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} \mathbb{E}\{S\} \mid Q'$. Premise $Q \xrightarrow{a,R} Q'$ checks that Q may receive R on a, and the resulting process Q' is run in parallel with the continuation S under context $\mathbb{E}$. We check that that $\mathbb{E}$ does not capture free names of R with the side-condition $\mathsf{bn}(\mathbb{E}) \cap \widetilde{b} = \emptyset$. We keep the free names $\widetilde{b}$ of R in the label for scope extrusion.

For instance, let $P = \overline{a}\langle R\rangle S$ and $c \in \mathsf{fn}(R)$. Process $\nu c.\, P$ may emit R on a, but the scope of c has to be expanded to encompass the recipient of R. First premise of rule $\textsc{Extr}_o^p$ checks that P may output a message; here we have $\overline{a}\langle R\rangle S \xrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} \mathbb{E}\{S\} \mid Q'$ with $\widetilde{b} = \mathsf{fn}(R)$. In conclusion, we have $\nu c.\, \overline{a}\langle R\rangle S \xrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}\setminus c} \nu c.\, (\mathbb{E}\{S\} \mid Q')$. Scope of c includes the Q' as wished. We remove c from set $\widetilde{b}$ in the label for observational purposes. The set $\widetilde{b}$ consists of the names which may be extruded. For a concretion $C = \nu\widetilde{a}.\, \langle P_1\rangle P_2$, these names $\widetilde{b_C}$ are the free names of P_1 which are not already bound in $\widetilde{a}$, i.e. $\widetilde{b_C} = \mathsf{fn}(P_1) \setminus \widetilde{a}$.

Suppose now that $P = \overline{a}\langle R\rangle S$ with $c \notin \mathsf{fn}(R)$. Process $\nu c.\, P$ may emit a message, but the scope of c has to encompass the continuation S only: we want to obtain $\nu c.\, P \xrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} \mathbb{E}\{\nu c.\, S\} \mid Q'$. To this end, we consider $P \xrightarrow{\overline{a},Q,\mathbb{E}\{\nu c.\Box\}}_{\widetilde{b}} P'$ as a premise of rule $\textsc{Restr}_o^p$. In process P', the continuation is put under $\mathbb{E}\{\nu c.\, \Box\}$, hence we obtain $\overline{a}\langle R\rangle S \xrightarrow{\overline{a},Q,\mathbb{E}\{\nu c.\Box\}}_{\widetilde{b}} \mathbb{E}\{\nu c.\, S\} \mid Q' = P'$. Process P' is exactly the resulting process we want for $\nu c.\, P$, hence the conclusion of the rule is simply $\nu c.\, P \xrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'$.

Rule for passivation $\textsc{Passiv}_o^p$ is similar to rule $\textsc{Out}_o^p$, while rules $\textsc{Loc}_o^p$, $\textsc{Par}_o^p$ follow the same pattern as rule $\textsc{Restr}_o^p$. Rule $\textsc{CFree}_o^p$ simply means that a transition with a capture-free context is a message output transition. We now explain how te deal with context capture with rule $\textsc{Capt}_o^p$. Suppose $P = \overline{a}\langle R\rangle S$ and $\mathbb{E}' = d[\nu c.\, (\Box \mid \overline{c}\langle \mathbf{0}\rangle \mathbf{0})]$ with $c \in \mathsf{fn}(R)$; we want to obtain $P \xrightarrow{\overline{a},Q,\mathbb{E}'}_{\widetilde{b}} \nu c.\, (d[S \mid \overline{c}\langle \mathbf{0}\rangle \mathbf{0}] \mid Q')$ (with the scope of c

extended out of d). We first consider the transition $P \xrightarrow{\overline{a},Q,\mathbb{E}\{\mathbb{F}\}}_{\widetilde{b}} P'$ without capture on c; in our case we have $P \xrightarrow{\overline{a},Q,d[\Box]}_{\widetilde{b}} d[S \mid \overline{c}\langle\mathbf{0}\rangle\mathbf{0}] \mid Q' = P'$ with $\mathbb{E} = d[\Box]$ and $\mathbb{F} = \Box \mid \overline{c}\langle\mathbf{0}\rangle\mathbf{0}$. Using the rule we have $P \xrightarrow{\overline{a},Q,\mathbb{E}\{\nu c.\mathbb{F}\}}_{\widetilde{b}} \nu c.\, P'$, i.e., $P \xrightarrow{\overline{a},Q,\mathbb{E}'}_{\widetilde{b}} \nu c.\, (d[S \mid \overline{c}\langle\mathbf{0}\rangle\mathbf{0}] \mid Q')$. The scope of c is extended outside $\mathbb{E}$ and includes the recipient of the message as wished.

Premise $P \xrightarrow{\overline{a},Q,\Box}_{\widetilde{b}} P'$ of rule HO^p_τ (Figure 8.4) means that process P sends a message on a to Q without any bisimulation context to surround P, and the result is P'. Consequently we have $P \mid Q \xrightarrow{\tau} P'$ by communication on a, which is precisely the wished conclusion. Names $\widetilde{b}$ are no longer needed for scope extrusion, so we simply forget them.

We now establish the correspondence between the contextual LTS and the complementary LTS.

Lemma 8.7.1. *Let P be an HOπP process. We have:*

- $P \xrightarrow{\tau} P'$ *iff* $P \mapsto^{\tau} P'$;
- *if* $P \xrightarrow{a} F$, *then for all* R *we have* $P \xrightarrow{a,R} F \circ R$; *conversely, if* $P \xrightarrow{a,R} P'$, *then there exists* F *such that* $P \xrightarrow{a} F$ *and* $P' = F \circ R$;
- *if* $P \xrightarrow{\overline{a}} C$, *then for all* Q *such that* $Q \xrightarrow{a} F$ *and for all* $\mathbb{E}$, *we have* $P \xrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} F \bullet \mathbb{E}\{C\}$ *with* $\widetilde{b} = \mathsf{fn}(o(C)) \setminus \mathsf{bn}(C)$; *conversely, if* $P \xrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'$, *then there exists* F, C *such that* $P \xrightarrow{\overline{a}} C$, $Q \xrightarrow{a} F$, $\widetilde{b} = \mathsf{fn}(o(C)) \setminus \mathsf{bn}(C)$, *and* $P' = F \bullet \mathbb{E}\{C\}$.

Complementary Bisimilarities

Strong complementary bisimilarity is defined as follows.

Definition 8.7.2 (**Strong complementary bisimilarity**). *A relation $\mathcal{R}$ on closed processes is a* strong complementary simulation *iff $P \mathcal{R} Q$ implies $\mathsf{fn}(P) = \mathsf{fn}(Q)$ and for all $P \xrightarrow{\lambda} P'$, there exists Q' such that $Q \xrightarrow{\lambda} Q'$ and $P' \mathcal{R} Q'$.*

A relation $\mathcal{R}$ is a strong complementary bisimulation *iff $\mathcal{R}$ and $\mathcal{R}^{-1}$ are strong complementary simulations. Two closed processes P and Q are* strong complementary bisimilar, *noted $P \sim_m Q$, iff there exists a strong complementary bisimulation $\mathcal{R}$ such that $P \mathcal{R} Q$.*

Note that we still have the condition on free names $\mathsf{fn}(P) = \mathsf{fn}(Q)$. We now prove that $\sim_m$ is sound (by proving that it is a congruence) using Howe's method. To this end, we define first $\mathbb{E} \sim_m^\bullet \mathbb{F}$ as the smallest congruence that extends $\sim_m^\bullet$ with $\Box \sim_m^\bullet \Box$.

Definition 8.7.3 (**Howe's closure for evaluation contexts**). *Howe's closure for evaluation contexts is the smallest relation verifying the following laws:*

- $\Box \sim_m^\bullet \Box$;
- *if* $\mathbb{E} \sim_m^\bullet \mathbb{F}$ *and* $P \sim_m^\bullet Q$, *then we have* $\mathbb{E} \mid P \sim_m^\bullet \mathbb{F} \mid Q$;
- *if* $\mathbb{E} \sim_m^\bullet \mathbb{F}$, *then we have* $\nu a.\, \mathbb{E} \sim_m^\bullet \nu a.\, \mathbb{F}$;
- *if* $\mathbb{E} \sim_m^\bullet \mathbb{F}$, *then we have* $a[\mathbb{E}] \sim_m^\bullet a[\mathbb{F}]$.

We prove the following simulation-like property for $\sim_m^\bullet$.

Lemma 8.7.4. *Let P, Q be closed processes. If $P \sim_m^\bullet Q$ then:*

- *if* $P \mapsto^{\tau} P'$, *then there exists* Q' *such that* $Q \mapsto^{\tau} Q'$ *and* $P' \sim_m^\bullet Q'$;

- *if* $P \xrightarrow{a,R} P'$*, then for all* $R \sim_m^\bullet R'$*, there exists* Q' *such that* $Q \xrightarrow{a,R'} Q'$ *and* $P' \sim_m^\bullet Q'$*;*

- *if* $P \xrightarrow{\overline{a},T,\mathbb{E}}_{\widetilde{b}} P'$*, then for all* $T \sim_m^\bullet T'$ *and all* $\mathbb{E} \sim_m^\bullet \mathbb{F}$*, there exists* Q' *such that* $Q \xrightarrow{\overline{a},T',\mathbb{F}}_{\widetilde{b}} Q'$ *and* $P' \sim_m^\bullet Q'$*.*

The proof is by induction on the derivation of $P \sim_m^\bullet Q$. We just detail the communication case: we have $P_1 \mid P_2 \sim_m^\bullet Q_1 \mid Q_2$ with $P_1 \sim_m^\bullet Q_1$, $P_2 \sim_m^\bullet Q_2$ and $P_1 \xrightarrow{\overline{a},P_2,\Box}_{\widetilde{b}} P'$. We can apply directly the message output clause of the induction hypothesis: there exists Q' such that $Q_1 \xrightarrow{\overline{a},Q_2,\Box}_{\widetilde{b}} Q'$ and $P' \sim_m^\bullet Q'$. We conclude that $Q_1 \mid Q_2 \overset{\tau}{\mapsto} Q'$ (by rule HO^p_τ) with $P' \sim_m^\bullet Q'$ as wished.

Theorem 8.7.5. $\sim_m$ *is a congruence.*

We may wonder if strong early context and complementary bisimilarities have the same discriminating power. The output clause of complementary bisimilarity requires that transition $P \xrightarrow{\overline{a},T,\mathbb{E}}_{\widetilde{b}} P'$ has to be matched by a transition $Q \xrightarrow{\overline{a},T,\mathbb{E}}_{\widetilde{b}} Q'$ with the same set of names $\widetilde{b}$ which may be extruded. At first glance, we do not have this requirement for the early strong context bisimilarity. Nevertheless, we prove that both relations coincide.

As in Section 8.6, the first inclusion is easy.

Lemma 8.7.6. *We have* $\sim_m \subseteq \sim$*.*

For the reverse inclusion, we have to prove first the following result on concretion names.

Lemma 8.7.7. *Let* $P \sim Q$*. Let* $P \xrightarrow{\overline{a}} C$*,* F *an abstraction, and* $Q \xrightarrow{\overline{a}} C'$ *such that for all* $\mathbb{E}$*, we have* $F \bullet \mathbb{E}\{C\} \sim F \bullet \mathbb{E}\{C'\}$*. Then we have* $\mathsf{fn}(o(C)) \setminus \mathsf{bn}(C) = \mathsf{fn}(o(C')) \setminus \mathsf{bn}(C')$*.*

We then have the reverse inclusion.

Lemma 8.7.8. *We have* $\sim \subseteq \sim_m$*.*

The proof is done by showing that $\sim$ is a strong complementary bisimilarity.

We can also prove soundness and correspondence between bisimilarities in the weak case. We define $P \overset{\overline{a},Q,\mathbb{E}}{\Longrightarrow}_{\widetilde{b}} P'$ as $P \overset{\tau}{\Rightarrow} \xrightarrow{\overline{a},Q',\mathbb{E}}_{\widetilde{b}} \overset{\tau}{\Rightarrow} P'$ with $Q \overset{\tau}{\Rightarrow} Q'$. Weak complementary bisimilarity is defined as follows.

Definition 8.7.9 (**Weak complementary bisimilarity**)**.** *A relation* $\mathcal{R}$ *on closed processes is a* weak complementary simulation *iff* $P \mathcal{R} Q$ *implies* $\mathsf{fn}(P) = \mathsf{fn}(Q)$ *and for all* $P \xrightarrow{\lambda} P'$*, there exists* Q' *such that* $Q \overset{\lambda}{\Rightarrow} Q'$ *and* $P' \mathcal{R} Q'$*.*

A relation $\mathcal{R}$ *is a* weak complementary bisimulation *iff* $\mathcal{R}$ *and* $\mathcal{R}^{-1}$ *are weak complementary simulations. Two closed processes* P *and* Q *are* weak complementary bisimilar*, noted* $P \approx_m Q$*, iff there exists an weak complementary bisimulation* $\mathcal{R}$ *such that* $P \mathcal{R} Q$*.*

As in the strong case, we have the following results.

Theorem 8.7.10. *The relation* $\approx_m$ *is a congruence.*

Lemma 8.7.11. *We have* $\approx_m = \approx$*.*

Completeness

As for HOπ, we prove that weak complementary bisimilarity $\approx_m$ and weak barbed congruence coincide on *image-finite* processes.

Definition 8.7.12 (Image finite processes). *A closed process P is image finite iff for all label λ, the set $\{P', P \overset{\lambda}{\Rightarrow} P'\}$ is finite.*

Theorem 8.7.13. *Let P, Q be image-finite processes. If $P \approx_b Q$ then $P \approx_m Q$.*

The theorem is proved by contradiction. We define a family of relations $\approx_{m,k}$, with k an integer, which differentiate several levels of bisimulations by stating that processes have to be bisimilar only during the first k steps, and such that $\approx_m = \bigcap_k \approx_{m,k}$.

- We have $P \approx_{m,0} Q$ iff $\mathsf{fn}(P) = \mathsf{fn}(Q)$.
- We have $P \approx_{m,k+1} Q$ iff for $P \xrightarrow{\lambda} P'$, there exists Q' such that $Q \overset{\lambda}{\Rightarrow} Q'$ and $P' \approx_{m,k} Q'$, and conversely for $Q \xrightarrow{\lambda} Q'$.

By induction we prove that if for some k we have $P \not\approx_{m,k} Q$, then there exists a context $\mathbb{C}_k$ such that $\mathbb{C}_k\{P\} \not\approx_b \mathbb{C}_k\{Q\}$. If $P \not\approx_m Q$, then there exists k such that $P \not\approx_{m,k} Q$, hence there exists a context $\mathbb{C}$ such that $\mathbb{C}\{P\} \not\approx_b \mathbb{C}\{Q\}$. Consequently P and Q are not weakly barbed congruent.

8.8 Related Work

Howe's method Howe's method has been originally used to prove congruence in a lazy functional programming language (Howe, 1996). Baldamus and Frauenstein (1995) are the first to adapt the method to process calculi for variants of Plain CHOCS (Thomsen, 1993). They prove congruence of a late delay context bisimilarity in SOCS, a CHOCS-like calculus with static scope, where restricted names follow the emitted processes as in HOπ. They then use it for late and early delay higher-order bisimilarities in SOCD, a calculus with dynamic scoping, where emitted messages may escape the scope of their restricted names.

Bundgaard et al. (2004) adapt Howe's method for their calculus Homer . Homer is a higher-order process calculi featuring hierarchical localities, local names and active process mobility (passivation), for which they prove congruence for a late context strong and delay bisimilarities. Godskesen and Hildebrandt (2005) extend this work for an input-early context delay bisimilarity. Strong input-early bisimilarity is complete with respect to strong barbed congruence, but as stated by the authors themselves, delay input-early bisimilarity is probably not complete with respect to weak barbed congruence because of the delay style.

Behavioral equivalences in higher-order calculi Very few higher-order calculi feature a coinductive characterization of weak barbed congruence. HOπ enjoys a nice behavioral theory: on top of the context bisimilarity (discussed in Section 8.5), Sangiorgi defines *normal* bisimilarity which characterizes weak barbed congruence with fewer tests than context bisimilarity (Section 8.3).

Mobile Ambients (Cardelli and Gordon, 1998) is a calculus with hierarchical localities and subjective linear process mobility. Contextual characterizations of weak barbed congruence have been defined for Mobile Ambients (Merro and Nardelli, 2005) and its variant NBA (Bugliesi et al., 2005). Soundness proofs are done by proving that the smallest congruence which contains weak context bisimilarity is a bisimulation.

Difficulties arise in more expressive process calculi. The Seal calculus (Vitek and Castagna, 1999; Castagna et al., 2005) is a calculus with objective process mobility which allows more flexibility than Mobile Ambients: localities may be stopped, duplicated, and moved up and down in the locality hierarchy. However a process inside a locality cannot be dissociated from its locality boundary. Process mobility requires synchronisation between three processes (a process sending a name a, a receiving process, and a locality named a). Castagna and Nardelli (2002) define a weak delay context bisimilarity called *Hoe bisimilarity* for the Seal calculus, which is similar to normal bisimulation for HOπ in the message sending case, and prove its soundness. The authors point out that Hoe bisimilarity is not complete, not only because of the delay style, but also because of the labels introduced for partial synchronisation which are most likely not observable.

8.9 Conclusion

Chapter 7 seemed to suggest the behavioral theory of minimal process calculi was very simple, or even trivial. Adding passivation does not significantly worsen the situation: we showed in Section 8.3 that a normal bisimilarity can be found. In fact, we even conjecture that strong barbed congruence for LHOπP is decidable. The real culprit is name restriction. By itself, it breaks decidability of strong barbed congruence for HO Core (Section 7.7), and in conjonction with passivation it make the behavioral theory quite complex.

Chapter 9

Conclusion: Toward Certified Analyses

I would be remiss if I did not admit the main topic of this document, static analyses for the manipulation of structured data, came after the fact as I tried to find a way to present the different subjects I had worked on. Nevertheless, it actually is a topic to bind them all: with a couple of exceptions in the domain of language design, it covers the last nine years of my research. Moreover, it also is at the core of the future areas I wish to explore.

More precisely, a first direction I am pursuing is the direct continuation of some of the topics presented here. They can be split in three domains: programming support for component-based distributed applications, understanding of the formal models of bidirectional transformations, and extensions of our XPath satisfiability algorithm.

One goal of the Sardes team continues to be the design of a formal model for component-based distributed programs. At a fundamental level, this work is based on higher-order calculi like the ones presented in Chapter 8. Based on our previous attempts (the M-calculus (Schmitt and Stefani, 2003), the Kell calculus (Schmitt and Stefani, 2004), and Oz/K (Lienhardt et al., 2007)), we have started to understand the main design choices and their interactions. For instance, localities have two purposes: specify the scope of a passivation, but also participate in the encapsulation (the restriction of communication) of processes. From a communication (or service providing) point of view, localities may be shared between several agents. How does this sharing interact with encapsulation and passivation? We have some preliminary answers (Hirschkoff et al., 2005), and are continuing to explore this in an ongoing collaboration with Davide Sangiorgi. Existing analyses like the ones of Chapter 2 also need to be extended to take into account the evolution of the architecture of the system. Finally, we have started to look into adding support for dependable computing constructs, such as transactions, directly in the calculus, in the form of controlled reversible computation.

In the domains of bidirectional languages (Chapters 3 and 4), I have very recently started a collaboration with several persons of the domain to better understand the foundations of lenses. Although this work is very preliminary, it seems that lenses laws naturally spawn from the coalgebra of the costate comonad, or from the algebra of the monad over the slice category. While this may seem very abstract, a better understanding of bidirectional transformations is quite useful, as they are pervasive in the database world.

The third domain which is a direct continuation of previous work is the XPath satisfiability algorithm developed with Pierre Genevès and Nabil Layaïda. We are extending it to deal with counting in XPath expressions, and are applying it to infer the regular type resulting from the application of an expression to an input regular type. An immediate application of this work would be the design of a type system for XQuery that is able to deal precisely with XPath expressions. A second application lies in the modular

verification of hybrid XML documents, where some schemas occur inside other schemas (such as MathML or SVG inside XHTML).

Beyond these short to mid term research projects, my long term goal is to tend toward certified analyses of distributed applications. Such applications are more and more common, typically in the form of small interconnected services that offer features through public APIs. These applications also evolve very rapidly, to fix bugs or offer more services. In this distributed and changing setting, issues like privacy leaks or system unavailability have become commonplace. My goal is thus to apply what I have learned in programming language design and static analyses to build more dependable distributed systems. This long term project also involves a notion of *certified* analysis, where the proof that the analysis and the code implementing it are both shown to be correct using a proof assistant.

While writing computer-certified proofs has long been considered as an exercise involving "academic-size" proofs, recent work has confirmed that theorem provers are now mature enough to deal with large proofs, be it proofs of complex results in mathematics or proofs about programming language semantics (see, e.g., Gonthier and Werner's proof of the 4 color theorem, which required the development of sophisticated prover technology, the certified compiler developed in the ANR project Compcert (Leroy, 2009), or the microkernel seL4 that was certified in Isabelle/HOL (Klein et al., 2009)). The approach I'm aiming for is similar to the extraction of the code of a verifier for proof carrying code from the proof that it is correct (Besson et al., 2009, 2010). Type systems for component-based distributed applications would also benefit from such an approach, for instance to verify the structure of message as in Chapter 2, to make sure a transformation is valid for some data (Chapter 6), or to ensure some protocol is followed.

As a first step toward the certification of some of the proofs of this documents and some future proofs, I have started formalizing the results of Chapter 7 in the Coq proof assistant. The plan is to build upon these reasonably simple proofs to start addressing the more complex questions of passivation and name restriction of Chapter 8, then to fully formalize a calculus for evolutive distributed systems. The day we are given a computer proof that a social network will not divulge our data is still quite far, but I hope this research will bring us closer.

Bibliography

S. Abiteboul, S. Cluet, and T. Milo. Correspondence and translation for heterogeneous data. In *International Conference on Database Theory (ICDT), Delphi, Greece*, 1997. 55

S. Abiteboul, S. Cluet, and T. Milo. A logical view of structure files. *VLDB Journal*, 7 (2):96–114, 1998. 55

S. M. Abramov and R. Glück. The universal resolving algorithm: Inverse computation in a functional language. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837, pages 187–212. Springer-Verlag, 2000. 54

S. M. Abramov and R. Glück. Principles of inverse computation and the universal resolving algorithm. In T. Mogensen, D. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, volume 2566 of *Lecture Notes in Computer Science*, pages 269–295. Springer-Verlag, 2002. 54

L. Afanasiev, P. Blackburn, I. Dimitriou, B. Gaiffe, E. Goris, M. Marx, and M. de Rijke. PDL for ordered trees. *Journal of Applied Non-Classical Logics*, 15(2):115–135, 2005. 115

J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in archjava. In *ECOOP '02 Proceedings of the 16th European Conference on Object-Oriented Programming*. Springer-Verlag, 2002. 28

R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3), July 1997. 27

R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous pi-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998. 130

P. Atzeni and R. Torlone. Management of multiple models in an extensible database design tool. In *Proceedings of EDBT'96, LNCS 1057*, 1996. 56

P. Atzeni and R. Torlone. MDM: a multiple-data model tool for the management of heterogeneous database schemes. In *Proceedings of ACM SIGMOD, Exhibition Section*, pages 528–531, 1997. 56

M. Baldamus. *Semantics and Logic of Higher-Order Processes: Characterizing Late Context Bisimulation*. PhD thesis, Berlin University of Technology, 1998. 152

M. Baldamus and T. Frauenstein. Congruence proofs for weak bisimulation equivalences on higher-order process calculi. Technical report, Berlin University of Technology, 1995. 129, 162

F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, Dec. 1981. 52, 78

P. Barceló and L. Libkin. Temporal logics over unranked trees. In *LICS '05: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, pages 31–40, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2266-1. 115

T. Barros, A. Cansado, E. Madelaine, and M. Rivera. Model-checking distributed components: The vercors platform. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 182, 2007. 7

T. Barsalou, N. Siambela, A. M. Keller, and G. Wiederhold. Updating relational databases through object-based views. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems, Denver, Colorado*, pages 248–257, 1991. 56

M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In *PODS '05: Proceedings of the twenty-fourth ACM Symposium on Principles of Database Systems*, pages 25–36, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-062-0. doi: http://doi.acm.org/10.1145/1065167.1065172. 101, 113

S. Bensalem, M. Bozga, J. Sifakis, and T.-H. Nguyen. Compositional verification for component-based systems and application. In *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 5311 of *Lecture Notes in Computer Science*. Springer, 2008. 7

N. Benton. Embedded interpreters. *Journal of Functional Programming*, 15(4):503–542, 2005. 81

V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Uppsala, Sweden*, pages 51–63, 2003. ISBN 1-58113-756-7. 85, 95, 97

V. Benzaken, G. Castagna, and C. Miachon. A full pattern-based paradigm for XML query processing. In *Practical Aspects of Declarative Languages (PADL), Long Beach, CA*, volume 3350 of *LNCS*, pages 235–252. Springer, Jan. 2005. 95, 97

F. Besson, D. Cachera, T. P. Jensen, and D. Pichardie. Certified static analysis by abstract interpretation. In *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 223–257. Springer-Verlag, Sept. 2009. 166

F. Besson, T. Jensen, D. Pichardie, and T. Turpin. Certified result checking for polyhedral analysis of bytecode programs. In *Proceedings of the 5th International Symposium on Trustworthy Global Computing (TGC 2010)*, Lecture Notes in Computer Science. Springer-Verlag, Feb. 2010. To appear. 166

N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Transactions on Computer Systems (TOCS)*, 16(4), 1998. 7, 8

P. Bidinger, M. Leclercq, V. Quéma, A. Schmitt, and J.-B. Stefani. Dream Types - A Domain Specific Type System for Component-Based Message-Oriented Middleware. In *4th Workshop on Specification and Verification of Component-Based Systems (SAVCBS'05), in association with ESEC/FSE'05*, Lisbon, Portugal, Sept. 2005a. 13

P. Bidinger, A. Schmitt, and J.-B. Stefani. An abstract machine for the Kell calculus. In *7th IFIP International Conference on Formal Methods for Object-Based Distributed Systems (FMOODS)*, volume 3535 of *Lecture Notes in Computer Science*, pages 43–58, Athens, Greece, June 2005b. Best Paper Award. ix, 2

G. Bierman, E. Meijer, and W. Schulte. The essence of data access in C-omega. In *ECOOP*, 2005. 85, 98

S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. Xquery 1.0: An xml query language (second edition), Dec. 2010. `http://www.w3.org/TR/xquery/`. 55, 85, 98

A. Bohannon, J. A. Vaughan, and B. C. Pierce. Relational lenses: A language for updateable views. In *Principles of Database Systems (PODS)*, 2006. Extended version available as University of Pennsylvania technical report MS-CIS-05-27. 30, 55

A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: Resourceful lenses for string data. Technical Report MS-CIS-07-15, Dept. of CIS University of Pennsylvania, Nov. 2007. 67

A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: Resourceful lenses for string data. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 407–419, San Francisco, California, USA, Jan. 2008. ACM. doi: 10.1145/1328897.1328487. ix, 61

C. Brabrand, A. Møller, and M. I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 2007. To appear. Extended abstract in *Database Programming Languages (DBPL)* 2005. 81

V. Braganholo, C. A. Heuser, and C. R. M. Vittori. Updating relational databases through XML views. In *Proc. 3rd Int. Conf. on Information Integration and Web-based Applications and Services (IIWAS)*, 2001. 55

V. Braganholo, S. Davidson, and C. Heuser. On the updatability of XML views over relational databases. In *WebDB 2003*, 2003. 55

E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java. *Software – Practice and Experience*, 36 (11–12), Sept. 2006. 8

R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, 35(8):677–691, 1986. 111

M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication and mobility control in boxed ambients. *Information and Computation*, 202, 2005. 140, 162

M. Bundgaard, J. C. Godskesen, and T. Hildebrandt. Bisimulation congruences for homer — a calculus of higher order mobile embedded resources. Technical Report TR-2004-52, IT University of Copenhagen, 2004. 129, 139, 140, 145, 153, 162

P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *International Conference on Database Theory (ICDT), London, UK*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2001. 66

P. Buneman, S. Khanna, and W.-C. Tan. On propagation of deletions and annotations through views. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems*, pages 150–158, Madison, Wisconsin, USA, 2002. 56

D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Regular xpath: Constraints, query containment and view-based answering for xml documents. In *Proc. of the 2008 Int. Workshop on Logic in Databases (LID 2008)*, 2008. 116

D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. An automata-theoretic approach to regular xpath. In *Proc. of the 12th Int. Symposium on Database Programming Languages (DBPL 2009)*, volume 5708 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2009. 116

Z. Cao. More on bisimulations for higher order pi-calculus. In *Proc. of FoSSaCS'06*, volume 3921 of *Lecture Notes in Computer Science*, pages 63–78. Springer, 2006. 122, 130

L. Cardelli and A. D. Gordon. Mobile ambients. In *FoSSaCS '98*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 1998. 140, 162

L. Cardelli and A. D. Gordon. Types for mobile ambients. In *Proceedings 26th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 1999. 28

C. Carrez, A. Fantechi, and E. Najm. Behaviour contracts for a sound assembly of components. In *Formal Techniques for Networked and Distributed Systems - FORTE 2003*, volume 2767 of *Lecture Notes in Computer Science*, pages 111–126. Springer, 2003. 28

G. Castagna and F. Z. Nardelli. The seal calculus revisited: Contextual equivalence and bisimilarity. In *FSTTCS '02*, volume 2556 of *Lecture Notes in Computer Science*, pages 85–96. Springer, 2002. ISBN 3-540-00225-1. 155, 163

G. Castagna, J. Vitek, and F. Z. Nardelli. The Seal Calculus. *Information and Computation*, 201(1):1–54, 2005. 140, 163

A. S. Christensen, C. Kirkegaard, and A. Møller. A runtime system for XML transformations in Java. In Z. Bellahsène, T. Milo, and e. a. Michael Rys, editors, *Database and XML Technologies: International XML Database Symposium (XSym)*, volume 3186 of *Lecture Notes in Computer Science*, pages 143–157. Springer, Aug. 2004. 98

S. Christensen, Y. Hirshfeld, and F. Moller. Decidable subsets of CCS. *The Computer Journal*, 37(4):233–242, 1994. 136

J. Clark. XSL transformations (XSLT) version 1.0, W3C recommendation, November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116. 98

J. Clark and M. Murata. Relax ng specification. Committee specification, OASIS, Dec. 2001. `http://www.relaxng.org/spec-20011203.html`. 85

E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, volume 131 of *LNCS*, pages 52–71, London, UK, 1981. Springer-Verlag. ISBN 3-540-11212-X. 115

D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Static analysis for path correctness of XML queries. *J. Funct. Program.*, 16(4-5):621–661, 2006. ISSN 0956-7968. 116

S. Conchon and F. Pottier. Join(x): Constraint-based type inference for the join-calculus. In *10th European Symposium on Programming (ESOP)*, volume 2028 of *Lecture Notes in Computer Science*, 2001. 28

S. S. Cosmadakis. Translating updates of relational data base views. Master's thesis, Massachusetts Institute of Technology, 1983. MIT-LCS-TR-284. 56

S. S. Cosmadakis and C. H. Papadimitriou. Updates of relational views. *Journal of the ACM*, 31(4):742–760, 1984. 56

K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 301–312, New York, NY, USA, 1998. ACM. ISBN 1-58113-024-4. doi: http://doi.acm.org/10.1145/289423.289459. 18

Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *VLDB Journal*, 12(1):41–58, 2003. 66

S. Dal Zilio, D. Lugiez, and C. Meyssonnier. A logic you can count on. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 135–146, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-729-X. 101, 116

C. J. Date. *An Introduction to Database Systems (Eighth Edition)*. Addison Wesley, 2003. 56

U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *TODS*, 7(3):381–416, September 1982. 52

N. G. De Bruijn. Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972. 132

E. W. Dijkstra. Program inversion. In F. L. Bauer and M. Broy, editors, *Program Construction, International Summer School, July 26 - August 6, 1978, Marktoberdorf, germany*, volume 69 of *Lecture Notes in Computer Science*. Springer, 1979. ISBN 3-540-09251-X. 54

J. Doner. Tree acceptors and some of their applications. *Journal of Computer and System Sciences*, 4:406–451, 1970. 115

A. Dovier, C. Piazza, and A. Policriti. An efficient algorithm for computing bisimulation equivalence. *Theoretical Computer Science*, 311(1-3):221–256, 2004. 122, 132

J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-03270-8. 111

E. A. Emerson and C. S. Jutla. Tree automata, μ-calculus and determinacy. In *Proceedings of the 32nd annual Symposium on Foundations of Computer Science*, pages 368–377, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press. ISBN 0-8186-2445-0. 115

B. Emir, S. Maneth, and M. Odersky. Scalable programming abstractions for xml services. In *Dependable Systems: Software, Computing, Networks, Research-Results of the DICS Program*, volume 4028 of *Lecture Notes in Computer Science*, pages 103–126, 2006. 98

R. Ennals and D. Gay. Multi-language synchronization. In *European Symposium on Programming (ESOP), Braga, Portugal*, volume 4421 of *Lecture Notes in Computer Science*, pages 475–489. Springer-Verlag, 2007. 81

M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity os. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*. ACM, 2006. 7

D. C. Fallside and P. Walmsley. XML Schema part 0: Primer second edition, W3C recommendation, October 2004. http://www.w3.org/TR/xmlschema-0/. 85, 101

W. Fan, C.-Y. Chan, and M. Garofalakis. Secure XML querying with security views. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 587–598, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-859-8. doi: http://doi.acm.org/10.1145/1007568.1007634. 101

M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *JCSS*, 18(2):194–211, 1979. 115

K. Fisher and R. Gruber. PADS: a domain-specific language for processing ad hoc data. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Chicago, IL*, pages 295–304, 2005. 81

J. N. Foster, B. C. Pierce, and A. Schmitt. *Harmony Programmer's Manual*, 2006. 29

J. N. Foster, M. B. Greenwald, C. Kirkegaard, B. C. Pierce, and A. Schmitt. Exploiting schemas in data synchronization. *Journal of Computer and System Sciences*, 73(4): 669–689, June 2007a. doi: http://dx.doi.org/10.1016/j.jcss.2006.10.024. Extended abstract in *Database Programming Languages (DBPL)* 2005. ix, 2, 32, 52

J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, May 2007b. Preliminary version presented at the *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2004; extended abstract presented at *Principles of Programming Languages (POPL)*, 2005. ix, 5, 31, 36, 37, 46, 48, 56

V. Gapeyev and B. C. Pierce. Paths into patterns. Technical Report MS-CIS-04-25, University of Pennsylvania, Oct. 2004. 94

V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. The Xtatic compiler and runtime system, 2005a. ix

V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. The Xtatic experience. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, Jan. 2005b. University of Pennsylvania Technical Report MS-CIS-04-24, Oct 2004. ix, 61

V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. XML goes native: Run-time representations for Xtatic. In *14th International Conference on Compiler Construction (CC)*, Edinburgh, UK, Apr. 2005c. ix, 61, 87, 91

P. Genevès and N. Layaïda. A system for the static analysis of XPath. *ACM Trans. Inf. Syst.*, 24(4):475–502, 2006. ISSN 1046-8188. doi: http://doi.acm.org/10.1145/1185877.1185882. 115

P. Genevès and N. Layaïda. Deciding XPath containment with MSO. *Data & Knowledge Engineering*, 63(1):108–136, October 2007. 115

P. Genevès and J.-Y. Vion-Dury. Logic-based XPath optimization. In *DocEng '04: Proceedings of the 2004 ACM Symposium on Document Engineering*, pages 211–219, NY, USA, 2004. ACM Press. ISBN 1-58113-938-1. doi: http://doi.acm.org/10.1145/1030397.1030437. 101

P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 342–351, New York, NY, USA, June 2007. ACM Press. ISBN 978-1-59593-633-2. doi: http://doi.acm.org/10.1145/1250734.1250773. ix, 61, 105, 106, 111

J. C. Godskesen and T. Hildebrandt. Extending howe's method to early bisimulations for typed mobile embedded resources with local names. In *FSTTCS '05*, volume 3821 of *Lecture Notes in Computer Science*, pages 140–151. Springer, 2005. 129, 140, 154, 162

A. D. Gordon. Bisimilarity as a theory of functional programming. Mini-course. Notes Series NS-95-3, BRICS, University of Cambridge Computer Laboratory, July 1995. iv+59 pp. 152

G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems (TODS)*, 13(4):486–524, 1988. 53

E. Grädel, W. Thomas, and T. Wilke. *Automata logics, and infinite games: a guide to current research.* Springer-Verlag, New York, NY, USA, 2002. ISBN 3-540-00388-6. 115

M. B. Greenwald, S. Khanna, K. Kunal, B. C. Pierce, and A. Schmitt. Agreeing to agree: Conflict resolution for optimistically replicated data. In *20th International Symposium on Distributed Computing (DISC)*, volume 4167 of *Lecture Notes in Computer Science*, pages 269–283, Stockholm, Sweden, Sept. 2006. doi: 10.1007/11864219_19. ix, 2

M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. Xj: facilitating xml processing in java. In *International World Wide Web Conference*, pages 278–287, 2005. 85, 90, 98

S. J. Hegner. An order-based theory of updates for closed database views. *Annals of Mathematics and Artificial Intelligence*, 40:63–125, 2004. URL `http://www.cs.umu.se/~hegner/Publications/PDF/amai03.pdf`. Summary in *Foundations of Information and Knowledge Systems*, 2002, pp. 230–249. 53

F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):253–289, 1993. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/169701.169692. 28

J. R. Hindley. *Basic simple type theory.* Cambridge University Press, New York, NY, USA, 1997. ISBN 0-521-46518-4. 28

D. Hirschkoff and D. Pous. A distribution law for CCS and a new congruence result for the π-calculus. *Logical Methods in Computer Science*, 4(2), 2008. 122, 131

D. Hirschkoff, T. Hirschowitz, D. Pous, A. Schmitt, and J.-B. Stefani. Component-oriented programming with sharing: Containment is not ownership. In *4th International Conference on Generative Programming and Component Engineering (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 389–404, Tallinn, Estonia, Sept. 2005. ix, 2, 165

M. Hofmann and B. Pierce. Positive subtyping. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 186–197, San Francisco, California, USA, Jan. 1995. Full version in *Information and Computation*, volume 126, number 1, April 1996. Also available as University of Edinburgh technical report ECS-LFCS-94-303, September 1994. 53

K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 151(2):437–486, 1995. 130

K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL*. ACM, 2008. 7

H. Hosoya and M. Murata. Boolean operations and inclusion test for attribute-element constraints. In *Eighth International Conference on Implementation and Application of Automata*, volume 2759 of *Lecture Notes in Computer Science*, pages 201–212. Springer-Verlag, 2003. Preliminary version in PLAN-X 2002. 88, 98

H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language (preliminary report). In D. Suciu and G. Vossen, editors, *International Workshop on the Web and Databases (WebDB)*, May 2000. Reprinted in *The Web and Databases, Selected Papers*, Springer LNCS volume 1997, 2001. 83

H. Hosoya and B. C. Pierce. Regular expression pattern matching. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, 2001. Full version in *Journal of Functional Programming*, 13(6), Nov. 2003, pp. 961–1004. 83, 85, 95, 97

H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003. 83, 97

H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Long Beach, California*, 2005a. 98

H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for xml. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90, jan 2005b. ISSN 0164-0925. 83, 97, 106, 107

D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996. 129, 140, 152, 162

Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bi-directional transformations. In *Partial Evaluation and Program Manipulation (PEPM)*, pages 178–189, 2004. Extended version to appear in *Higher Order and Symbolic Computation*, 2008. 34, 54, 81

G. P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997. 102

K. Inaba and H. Hosoya. MTran, June 2006. http://arbre.is.s.u-tokyo.ac.jp/ kinaba/MTran/. 115

V. Issarny, C. Bidan, and T. Saridakis. Achieving middleware customization in a configuration-based development environment: Experience with the aster prototype. In *Proceedings of the 4th International Conference on Configurable Distributed Systems (CDS)*, 1998. 7

P. Jančar. Undecidability of bisimilarity for Petri nets and some related problems. *Theoretical Computer Science*, 148(2):281–301, 1995. 122

A. Jeffrey and J. Rathke. Contextual equivalence for higher-order pi-calculus revisited. *Logical Methods in Computer Science*, 1(1):1–22, 2005. 122, 128

A. Joolia, T. Batista, G. Coulson, and A. T. A. Gomes. Mapping adl specifications to an efficient and reconfigurable runtime component platform. In *WICSA '05 Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*. IEEE Computer Society, 2005. 7, 28

S. Kawanaka and H. Hosoya. bixid: a bidirectional transformation language for XML. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Portland, Oregon*, pages 201–214, 2006. 81

A. M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *ACM SIGACT–SIGMOD Symposium on Principles of Database Systems, Portland, Oregon*, 1985. 56

M. Kempa and V. Linnemann. On XML objects. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2003. 98

A. J. Kennedy. Functional pearl: Pickler combinators. *Journal of Functional Programming*, 14(6):727–739, 2004. 81

C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of XML transformations in Java. Technical Report RS-03-19, BRICS, May 2003. 85

C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004. 90, 98

N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA 1.4, January 2001. http://www.brics.dk/mona/. 115

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM. 166

E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3), 2000. 7, 8, 27

E. Kohler, R. Morris, and B. Chen. Programming language optimizations for modular router configurations. In *ASPLOS*, 2002. 27

D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27: 333–354, 1983. 111, 115

O. Kupferman and M. Vardi. The weakness of self-complementation. In *Proc. 16th Symp. on Theoretical Aspects of Computer Science*, volume 1563 of *LNCS*, pages 455–466, London, UK, 1999. Springer. 115

A. Kučera and P. Jančar. Equivalence-checking on infinite-state systems: Techniques and results. *TPLP*, 6(3):227–264, 2006. 136, 137

I. Lanese, J. A. Pérez, D. Sangiorgi, and A. Schmitt. On the expressiveness of polyadic and synchronous communication in higher-order process calculi. In S. Abramsky, C. Gavoille, C. Kirchner, F. M. auf der Heide, and P. G. Spirakis, editors, *Proceedings of the 37th International Colloquium on Automata, Languages and Programming (ICALP 2010)*, volume 6199 of *Lecture Notes in Computer Science*, pages 442–453, Bordeaux, France, June 2010a. Springer. ix, 2

I. Lanese, J. A. Pérez, D. Sangiorgi, and A. Schmitt. On the expressiveness and decidability of higher-order process calculi. *Information and Computation*, 2010b. To appear. Extended abstract presented at *Logic in Computer Science (LICS)*, 2008. ix, 119, 123, 128, 130, 133

J. Lechtenbörger. The impact of the constant complement approach towards view updating. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems*, pages 49–55, San Diego, CA, USA, June 9–12 2003. ACM. 53

M. Leclercq, V. Quéma, and J.-B. Stefani. Dream: A component framework for the construction of resource-aware, configurable moms. *IEEE Distributed Systems Online*, 6(9), 2005. 7, 8

M. Leclercq, A. E. Ozcan, V. Quéma, and J.-B. Stefani. Supporting heterogeneous architecture descriptions in an extensible toolset. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 209–219, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi: http://dx.doi.org/10.1109/ICSE.2007.82. 26

S. Lenglet. *Bisimulations dans les calculs avec passivation.* PhD thesis, Université de Grenoble, 2010. 119, 140, 148, 150, 152, 156

S. Lenglet, A. Schmitt, and J.-B. Stefani. Howe's method for calculi with passivation. In M. Bravetti and G. Zavattaro, editors, *Proceedings of the 20th International Conference on Concurrency Theory (CONCUR 2009)*, volume 5710 of *Lecture Notes in Computer Science*, pages 448–462, Bologna, Italy, Sept. 2009a. Springer. ix, 119

S. Lenglet, A. Schmitt, and J.-B. Stefani. Normal bisimulations in process calculi with passivation. In L. de Alfaro, editor, *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS 2009)*, volume 5504 of *Lecture Notes in Computer Science*, pages 257–271, York, United Kingdom, Mar. 2009b. Springer. doi: 10.1007/978-3-642-00596-1_19. ix, 119

X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009. 166

M. Y. Levin and B. C. Pierce. Type-based optimization for regular patterns. In *DBPL '05: Proceedings of the 10th International Symposium on Database Programming Languages*, volume 3774 of *LNCS*, London, UK, August 2005. Springer-Verlag. ISBN 3-540-30951-9. 101

C. Lhoussaine. Type inference for a distributed π-calculus. *Science of Computer Programming*, 50(1-3), Mar. 2004. 28

L. Libkin and C. Sirangelo. Reasoning about xml with temporal logics and automata. In *LPAR '08: Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 97–112, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89438-4. doi: http://dx.doi.org/10.1007/978-3-540-89439-1_7. 116

M. Lienhardt. *Composants et Typage.* PhD thesis, Université Joseph Fourier, LIG, France, 2009. 13, 19, 20, 24, 25, 26

M. Lienhardt, J.-B. Stefani, and A. Schmitt. Oz/k: A kernel language for component-based open programming. In ACM, editor, *6th International Conference on Generative Programming and Component Engineering (GPCE'07)*, pages 43–52, Salzburg, Austria, Oct. 2007. doi: http://dx.doi.org/10.1145/1289971.1289980. ix, 2, 165

M. Lienhardt, A. Schmitt, and J.-B. Stefani. Typing communicating component assemblages. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE'08)*, pages 125–136, Nashville, TN, USA, Oct. 2008. ACM. doi: 10.1145/1449913.1449933. ix, 5

M. Lienhardt, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. Typing component-based communication systems. In *Proceedings of the 11th Formal Methods for Open Object-Based Distributed Systems (FMOODS) & 29th Formal Techniques for Networked and Distributed Systems (FORTE)*, volume 5522 of *Lecture Notes in Computer Science*, pages 167–181, Lisbon, Portugal, June 2009. Springer-Verlag. doi: http://dx.doi.org/10.1007/978-3-642-02138-1_11. ix, 5

X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building reliable, high-performance communication systems from components. In *Proceedings of the seventeenth ACM symposium on Operating systems principles (SOSP)*, 1999. 7

S. Maffeis. Sequence types for the pi-calculus. In *Proceedings of the Third International Workshop on Intersection Types and Related Systems (ITRS 2004)*, volume 136, pages 117–132, Amsterdam, The Netherlands, July 2005. Elsevier Science Publishers B. V. 8, 15, 28

H. Makholm and J. B. Wells. Instant polymorphic type systems for mobile process calculi: Just add reduction rules and close. In *14th European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*. Springer, 2005. 28

M. Marx. Conditional XPath, the first order complete XPath dialect. In *PODS '04: Proceedings of the twenty-third ACM Symposium on Principles of Database Systems*, pages 13–22, New York, NY, USA, 2004a. ACM Press. ISBN 158113858X. doi: http://doi.acm.org/10.1145/1055558.1055562. 115

M. Marx. XPath with conditional axis relations. In *Proceedings of the 9th International Conference on Extending Database Technology*, volume 2992 of *LNCS*, pages 477–494, London, UK, January 2004b. Springer-Verlag. ISBN 3-540-21200-0. 115, 116

Y. Masunaga. A relational database view update translation mechanism. In *VLDB'84*, 1984. 56

S. Matsuoka, S. Takahashi, T. Kamada, and A. Yonezawa. A general framework for bi-directional translation between abstract and pictorial data. *ACM Transactions on Information Systems*, 10(4):408–437, October 1992. 54

J. McCarthy. The inversion of functions defined by turing machines. In C. E. Shannon and J. McCarthy, editors, *Automata Studies, Annals of Mathematical Studies*, pages 177–181. Princeton University Press, 1956. 54

C. M. B. Medeiros and F. W. Tompa. Understanding the implications of view update policies. In *VLDB'85*, 1985. 56

L. Meertens. Designing constraint maintainers for user interaction, 1998. Manuscript. 53, 80

E. Meijer and W. Schulte. Unifying tables, objects and documents. In *Declarative Programming in the Context of OO Languages (DP-COOL)*, Sept. 2003. 85, 98

E. Meijer, W. Schulte, and G. Bierman. Programming with circles, triangles and rectangles. In *XML Conference and Exposition*, Dec. 2003. 85, 98

M. Merro and F. Z. Nardelli. Behavioral theory for mobile ambients. *Journal of the ACM*, 52(6):961–1023, 2005. ISSN 0004-5411. 140, 155, 162

G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/962446.962448. 113, 115

R. Milner and F. Moller. Unique decomposition of processes. *Theoretical Computer Science*, 107(2):357–363, jan 1993. 122, 131

R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proc. 19th ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer Verlag, 1992. 130, 142

M. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967. 121, 124

H. Miranda, A. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *21st International conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, 2001. 7, 8

A. Møller and M. I. Schwartzbach. The design space of type checkers for XML transformation languages. In *International Conference on Database Theory (ICDT)*, volume 3363 of *LNCS*, pages 17–36. Springer-Verlag, Jan. 2005. Invited paper. 98

A. Møller, M. O. Olesen, and M. I. Schwartzbach. Static validation of XSL Transformations. Technical Report RS-05-32, BRICS, 2005. 101

S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bi-directional updating. In *ASIAN Symposium on Programming Languages and Systems (APLAS)*, pages 2–20, Nov. 2004. 54, 81

M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, pages 153–166, 2001. 85

M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4):660–704, 2005. ISSN 1533-5399. doi: http://doi.acm.org/10.1145/1111627.1111631. 106

F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *ICDT '03: Proceedings of the 9th International Conference on Database Theory*, volume 2572 of *LNCS*, pages 315–329, London, UK, 2003. Springer-Verlag. ISBN 3-540-00323-1. 113

J. Niehren and A. Podelski. Feature automata and recognizable sets of feature trees. In *TAPSOFT*, pages 356–375, 1993. 37

A. Ohori and K. Tajima. A polymorphic calculus for views and object sharing. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems, Minneapolis, Minnesota*, 1994. 55

F. J. Oles. Type algebras, functor categories, and block structure. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*. Cambrige University Press, 1985. 53

D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *EDBT '02: Proceedings of the Worshop on XML-Based Data Management*, volume 2490 of *LNCS*, pages 109–127, London, UK, 2002. Springer-Verlag. ISBN 3-540-00130-1. 106

J. Palsberg, M. Wand, and P. O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9:49–67, 1997. ISSN 0934-5043. doi: 10.1007/BF01212524. 26

G. Pan, U. Sattler, and M. Y. Vardi. BDD-based decision procedures for the modal logic K. *Journal of Applied Non-classical Logics*, 16(1-2):169–208, 2006. 108

B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000. 28

B. C. Pierce and J. Vouillon. Unison: A file synchronizer and its specification. Manuscript, 2001. 32

B. C. Pierce, A. Schmitt, and M. B. Greenwald. Bringing harmony to optimism: A synchronization framework for heterogeneous tree-structured data. Technical Report MS-CIS-03-42, University of Pennsylvania, 2003. Superseded by MS-CIS-05-02. 52

E. L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52:264–268, 1946. 20, 122, 134

F. Pottier. A constraint-based presentation and generalization of rows. *LICS*, 0:331, 2003. ISSN 1043-6871. doi: http://doi.ieeecomputersociety.org/10.1109/LICS.2003.1210073. 26

M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959. 81

N. Ramsey. Embedding an interpreted language using higher-order functions and types. In *ACM SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators (IVME), San Diego, CA*, pages 6–14, 2003. 81

A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation (OSDI)*, 2000. 7

D. Rémy. *Type inference for records in natural extension of ML*, pages 67–95. MIT Press, Cambridge, MA, USA, 1994a. ISBN 0-262-07155-X. viii, 8, 12

D. Rémy. *Typing record concatenation for free*, pages 351–372. MIT Press, Cambridge, MA, USA, 1994b. ISBN 0-262-07155-X. viii, 12

E. Roche and Y. Schabes, editors. *Finite-State Language Processing*. MIT Press, 1996. 81

D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis CST–99–93, University of Edinburgh, Dept. of Comp. Sci., 1992. 119, 121, 122, 123, 128, 133, 139, 143, 156

D. Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111(1):120–153, 1994. 122

D. Sangiorgi. Bisimulation for higher-order process calculi. *Information and Computation*, 131(2):141–178, dec 1996a. 123, 133, 139, 140, 141, 143, 156

D. Sangiorgi. π-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science*, 167(2):235–274, 1996b. 123, 136

D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001. 130, 133, 136

D. Sangiorgi, N. Kobayashi, and E. Sumii. Environmental bisimulations for higher-order languages. In *Proc. of LICS'07*, pages 293–302. IEEE Computer Society, 2007. 128

A. Schmitt. Safe Dynamic Binding in the Join Calculus. In R. Baeza-Yates, U. Montanari, and N. Santoro, editors, *Proceedings of IFIP TCS 2002*, volume 96 of *IFIP*, pages 563–575, Montreal, Canada, 2002. Kluwer. [This is the original version that was accepted for publication, before the page cut requested for the final version. This version contains additional examples.]. 15

A. Schmitt and J. Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Global Computing*, volume 3267 of *Lecture Notes in Computer Science*, pages 146–178. Springer Berlin / Heidelberg, Mar. 2004. 139, 140, 145, 147, 153, 165

A. Schmitt and J.-B. Stefani. The M-Calculus: A Higher Order Distributed Process Calculus. In *Proceeding 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003)*, New Orleans, LA, USA, Jan. 2003. 119, 165

P. Schnoebelen. Bisimulation and other undecidable equivalences for lossy channel systems. In *Proc. of TACS'01*, volume 2215 of *Lecture Notes in Computer Science*, pages 385–399. Springer, 2001. 122

M. H. Scholl, C. Laasch, and M. Tresch. Updatable Views in Object-Oriented Databases. In C. Delobel, M. Kifer, and Y. Yasunga, editors, *Proc. 2nd Intl. Conf. on Deductive and Object-Oriented Databases (DOOD)*. Springer, 1991. 55

T. Schwentick. XPath query containment. *SIGMOD Record*, 33(1):101–109, 2004. ISSN 0163-5808. doi: http://doi.acm.org/10.1145/974121.974140. 101, 113

J. Siméon and P. Wadler. The essence of XML. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 1–13, 2003. 87

V. Simonet and F. Pottier. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29 (1), Jan. 2007. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1180475.1180476. 28

C. Szyperski. *Component Software: Beyond Object-Oriented Programming (2nd Edition)*. Addison-Wesley, 2002. 7

N. Tabuchi, E. Sumii, and A. Yonezawa. Regular expression types for strings in a text processing language. In *Workshop on Types in Programming (TIP), Dagstuhl, Germany*, volume 75 of *Electronic Notes in Theoretical Computer Science*, pages 95–113, 2002. 81

Y. Tanabe, K. Takahashi, M. Yamamoto, A. Tozawa, and M. Hagiya. A decision procedure for the alternation-free two-way modal μ-calculus. In *In TABLEAUX 2005*, volume 3702 of *LNCS*, pages 277–291, London, UK, September 2005. Springer-Verlag. ISBN 3-540-28931-3. 115

I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *ACM SIGMOD Symposium on Management of Data (SIGMOD), Santa Barbara, California*, 2001. 55

J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, 1968. 115

B. Thomsen. A calculus of higher order communicating systems. In *POPL '89*, pages 143–154. ACM, 1989. 119, 123

B. Thomsen. *Calculi for Higher Order Communicating Systems*. PhD thesis, Imperial College, 1990. 121, 133, 143

B. Thomsen. Plain chocs: A second generation calculus for higher order processes. *Acta Informatica*, 30(1):1–59, 1993. 119, 123, 143, 162

A. Tozawa. On binary tree logic for XML and its satisfiability test. In *PPL '04: the Sixth JSSST Workshop on Programming and Programming Languages*, Gamagoori, Japan, 2004. Informal Proceedings. 115

R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. *Software – Practice and Experience*, 28(9), July 1998. 7

M. Y. Vardi. Reasoning about the past with two-way automata. In *ICALP '98: Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 628–641, London, UK, 1998. Springer-Verlag. ISBN 3-540-64781-3. 101, 115

J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *ICCL'98: Workshop on Internet Programming Languages*, volume 1686 of *Lecture Notes in Computer Science*, pages 47–77. Springer, 1999. ISBN 3-540-66673-7. 140, 163

P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *ACM Symposium on Principles of Programming Languages (POPL), Munich, Germany*, 1987. 55

S. Weirich. Higher-order intensional type analysis. In *Proceedings of the 11th European Symposium on Programming Languages and Systems (ESOP)*. Springer-Verlag, 2002. 28

XPath 1.0. XML Path Language (XPath) Version 1.0, W3C Recommendation, Nov. 1999. `http://www.w3c.org/TR/xpath`. 92, 105, 106

N. Yoshida and M. Hennessy. Assigning types to processes. *Information and Computation*, 174(2), May 2002. 8, 15, 28

N. Yoshida and V. T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electronic Notes in Theoretical Computer Science*, 171(4), July 2007. 28