A Calculus for Boxing Analysis of Polymorphically Typed Languages

Ph.D. Thesis

Jesper Jørgensen

DIKU, Department of Computer Science
University of Copenhagen

April 1995
(revised May 1996)

ABSTRACT

An important decision when implementing languages with polymorphic types, such as Standard ML or Haskell, is whether to represent data in boxed or unboxed form and when to transform them from one representation to the other. Using a language with explicit representation types and boxing/unboxing operations we axiomatize equationally the set of all explicitly boxed versions, called *completions*, of a given source program. In a two-stage process we give some of the equations a rewriting interpretation that captures eliminating boxing/unboxing operations without relying on a specific implementation or even the semantics of the underlying language. The resulting reduction systems operate on equivalence classes of completions defined by the remaining equations E, which can be understood as moving boxing/unboxing operations along data flow paths in the source program. We call a completion e_{opt} *formally optimal* if every other completion for the same program (and at the same representation type) reduces to e_{opt} under this two-stage reduction.

We show that every source program has formally optimal completions, which are unique modulo E. This is accomplished by first "polarizing" the equations in E and orienting them to obtain two canonical (confluent and strongly normalizing) rewriting systems. The completions produced by algorithms from related work are generally not formally optimal in our sense.

The theory is developed for the polymorphic lambda calculus, and extended to treat primitives and data types, but is also shown adaptable to ML-like languages. An efficient algorithm for finding optimal completions is presented and has been implemented in Standard ML. Our results show that the amount of boxing and unboxing operations can also in practice be substantially reduced in comparison to completions produced by other approaches.

Contents

Preface **viii**
Acknowledgements viii

1 Introduction **1**
1.1 Representation analysis 1
1.2 Coercion calculus 2
1.3 A motivating example 4
1.4 Computing boxing completions 5
1.5 New results 5
1.6 Overview 6
1.6.1 Overview over part I 6
1.6.2 Overview over part II 7
1.6.3 Appendixes 7

I A coercion theory for polymorphic lambda calculus **8**

2 Polymorphic lambda calculus as a framework for boxing analysis **10**
2.1 The polymorphic λ-calculus 10
2.2 Explicitly boxed F_2 11
2.2.1 Representation types 11
2.2.2 Coercions 11
2.2.3 Type inference rules 14
2.3 Coercion and completion congruence 14

3 Coercion calculus **16**
3.1 Equational axiomatization of coercion congruence 16
3.1.1 Coercion equations 16
3.1.2 Coercion reduction 17
3.1.3 Coercion coherence 25
3.1.4 Optimal coercions 26
3.2 Coercion decomposition 27
3.2.1 Prime factorization 27
3.2.2 Positive and negative coercions 29
3.2.3 Polarized factorization 30
3.3 Representation type hierarchy 33
3.4 Summary 36

4 Coherence and reduction of completions **37**

4.1 Equational axiomatization of completion congruence 37

4.1.1 Expression equations 37

4.1.2 Expression coherence 40

4.1.3 Induced coercions and $\beta\eta$-equality 45

4.1.4 Parametricity and Equation 17 46

4.2 Completion reduction 47

4.2.1 $\phi^{\rightarrow}$-reduction modulo $E\psi$ and $\psi^{\rightarrow}$-reduction modulo $E\phi$ 48

4.2.2 $\phi^{\rightarrow}$-reduction modulo E and $\psi^{\rightarrow}$-reduction modulo E 49

4.3 Polarized equational theory 50

4.3.1 Polarized equational theory 50

4.3.2 The polarized versions of rule 17 52

4.3.3 Equivalence of E_p- and E-equivalence 54

4.4 Polarized reduction 56

4.4.1 $E_p^{\leftarrow}\psi^{\rightarrow}$ reduction modulo ϕ 56

4.4.2 $E_p^{\rightarrow}\phi^{\rightarrow}$ reduction modulo ψ 59

4.4.3 $E_p^{\rightarrow}\phi^{\rightarrow}$ reduction 60

4.4.4 $E_p^{\leftarrow}\psi^{\rightarrow}$ reduction 60

4.5 Optimal completions 60

4.5.1 ψ-free and ϕ-free completions 60

4.5.2 Optimal ψ-free and ϕ-free completions 64

4.6 Canonical completions 65

4.6.1 Canonically boxed completion 66

4.6.2 Canonically unboxed completion 68

5 Computing optimal completions **71**

5.1 Rewriting-based algorithm 71

5.1.1 ψ-reduction on ϕ-free completions 72

5.1.2 ϕ-reduction on ψ-free completions 75

5.1.3 An Example 75

5.1.4 Complexity analysis 76

5.2 Graph-based algorithm 77

5.2.1 Principal completions 77

5.2.2 The representation graph 79

5.2.3 Finding optimal completions 82

5.2.4 Finding optimal ψ-free completions 82

5.2.5 Finding optimal ϕ-free completions 86

5.2.6 An example 86

5.2.7 Description of a simple algorithm for finding node assignments 88

5.2.8 Complexity analysis 88

5.3 $\overline{\mathcal{C}}_{p*}$-completions 89

5.3.1 A classification of completions 89

5.3.2 Significance of $\mathcal{C}_{p*}$-completions 90

5.3.3 Existence of $\mathcal{C}_{p*}$-completion 90

5.3.4 The class of $\overline{\mathcal{C}}_{p*}$completions 91

5.3.5 Simple representation graphs 91

5.3.6 Complexity analysis 92

II Boxing analysis for ML-like languages **93**

6 Representation analysis for ML **95**
6.1 ML and explicit boxing . . . 95
6.1.1 Generating coherence conditions for ML . . . 95
6.1.2 Extension of completion reduction to new language features . . . 97
6.2 Recursive and non-recursive definitions . . . 97
6.2.1 Conditionals . . . 97
6.2.2 Let-expressions . . . 98
6.2.3 Fix-expression . . . 98
6.3 Data type constructions . . . 99
6.3.1 Integers . . . 101
6.3.2 Products . . . 101
6.3.3 Sums . . . 101
6.4 Recursive data types . . . 102
6.4.1 Lists . . . 103
6.4.2 Reflexive types . . . 103
6.4.3 $\mathcal{C}_{p*}$-treatment . . . 104
6.5 Imperative language features . . . 104
6.5.1 Polymorphic references . . . 105
6.6 Optimizing the execution efficiency of completions . . . 106
6.6.1 Completion equivalence is not efficiency equivalence . . . 106
6.6.2 Picking an equivalence class representative . . . 106
6.6.3 Moving coercions out of loops . . . 107
6.6.4 Taking control flow information into account . . . 107
6.6.5 Comparison with Peterson's minimum cut based optimization . . . 108
6.6.6 Boxing analysis in the ML-kit compiler . . . 108
6.7 Lazy evaluation and explicit boxing . . . 110
6.8 A larger example . . . 111

7 Implementation of completion inference **118**
7.1 Description of the algorithm . . . 118
7.1.1 The effect of adding recursive primitives . . . 120
7.1.2 Correctness of the algorithm . . . 120
7.1.3 The prototype implementation . . . 122
7.2 Benchmarks . . . 122
7.2.1 $\mathcal{C}_{**}$-completions with polymorphic data structures . . . 123
7.2.2 $\mathcal{C}_{**}$-completions with in-lined primitives . . . 124
7.2.3 $\overline{\mathcal{C}}_{p*}$-completions . . . 124
7.2.4 Comparison with other approaches . . . 125
7.2.5 Conclusion from tests . . . 126

8 Related work **128**
8.1 Representation analysis . . . 128
8.2 Coherence and equivalence . . . 131
8.3 Dynamic typing . . . 131

9 Conclusion and Future Work 132
9.1 Future work . . . 133

A Notation and terminology 135

B Implementation of Boxing Analysis 137
B.1 Signature: `EXPRESSION` . . . 137
B.2 Functor: `Expression` . . . 139
B.3 Signature: `PARSE` . . . 148
B.4 Functor: `Parse` . . . 148
B.5 Signature: `BOXINGANALYSIS` . . . 156
B.6 Functor: `BoxingAnalysis` . . . 157
B.7 Main Program . . . 171

C An Example 175

D Test Programs 179
D.1 Monomorphic insert-sort example . . . 179
D.2 Polymorphic insert-sort example . . . 180
D.3 Flip-list example . . . 181
D.4 Leroy's example . . . 182
D.5 Poulsen's example . . . 182
D.6 Sieve example . . . 183
D.7 Horner example . . . 184
D.8 Mogensen example . . . 186

E Dansk sammenfatning 188
E.1 Boxanalyse . . . 188
E.2 Nye resultater beskrevet i denne afhandling . . . 189

Bibliography 191

Index 194

List of Figures

1 Syntactic categories of F_2 10
2 Typing rules for F_2 10
3 Representation types of explicitly boxed F_2 11
4 Coercion formation rules 12
5 Typing rules for the explicitly boxed F_2 14
6 Equality rules for coercions: Core Equations (C) 16
7 The ϕ and ψ rules for coercions 16
8 Coercion reduction (R) 19
9 Positive and negative coercions 29
10 Summary of coercion calculus 36
11 Equality rules for coercion application 39
12 Equality rules for explicitly boxed expressions (E) 39
13 Polarized equality equations for explicitly boxed expressions (E_p) 51
14 Polarized reduction: $E_p^{\leftarrow}$ 56
15 ϕ-free canonical completion for F_2 66
16 ψ-free canonical completion for F_2 69
17 Principal representation types 78
18 Inference rules for generating principal completions 78
19 Coercion reduction 100
20 Efficient algorithm for finding ψ-free optimal completions 119
21 Efficient algorithm for finding ψ-free optimal completions with primitives 120
22 Performance: benchmarks for $\mathcal{C}_{**}$ with polymorphic data structures 123
23 Performance: benchmarks for $\mathcal{C}_{**}$ with in-lined primitives 124
24 Performance: benchmarks for $\overline{\mathcal{C}}_{p*}$ 124
25 Performance: compared benchmarks 126

Preface

This is a revised version of my Ph.D. thesis. The changes with respect to the original thesis are small correction (language, spelling error, typos, etc.) suggested by the examiners as well as “answers” to questions posed by the examiners at the examination.

The thesis was submitted in fulfillment of the requirements for a Ph.D. degree at DIKU, Department of Computer Science at the University of Copenhagen. It reports work done in the period from June 92 to March 94. My supervisors was Prof. Neil Jones and Lektor Fritz Henglein. The work was funded by a “kandidatstipendium” from the University of Copenhagen.

During my time as a Ph.D. student I have worked in three different areas: efficient program analyses described in [BJ93a, BJ93b], partial evaluation and program transformation described in [GJ94a, GJ94b], and representation analysis in polymorphic functional languages described in this thesis.

The work presented in this report builds on work presented in the paper ”Formally Optimal Boxing”, [HJ94] which is jointly authored with Fritz Henglein on boxing analysis for ML-like languages.

The thesis was successfully defended on the 27 of October 1995. The examiners were: Lektor Fritz Henglein (DIKU), Prof. Chris Hankin (Imperial College, London) and Prof. Simon L. Peyton Jones (University of Glasgow).

Acknowledgements

First of all I would like to thank Fritz who has done most of the actual supervision on this project and with whom I wrote the paper “Formally Optimal Boxing” presented at POPL’94 based on our earlier work on boxing analysis for Core-XML. Fritz is also the one who developed the framework using coercions for dynamic typing on which our work is based.

I would like to thank Neil Jones for his guidance, for creating an excellent research environment in the TOPPS group here at DIKU and for first pointing out the applicability of the framework of dynamic typing to boxing analysis.

A special thanks go to Jakob Rehof with whom I have had numerous discussions on types, typing, coercions calculus, boxing analysis, dynamic typing, etc.

Also a special thanks go to Robert Glück and Robert Harper for reading and commenting a draft of this thesis and to the external examiners Chris Hankin and Simon L. Peyton Jones for their many comments on and suggestions for improvements of the thesis.

Thanks to Eigil Rosager Poulsen from whose master’s thesis many of our test examples are taken.

Thanks to Kristoffer H. Rose for creating XY-pic and helping me with using this very nice package for typesetting of graphs and diagrams in LaTeX (TeX).

I would also like to thank the following people with whom I have had interesting and fruitful discussions on formally optimal boxing or for commenting parts of the draft of my thesis: Amir Ben-Amram, Anders Bondorf, Martin Elsman, Torben Mogensen, Christian Mossin, David Sands, Peter Thiemann and Mads Tofte.

A warm thank to the members of the TOPPS group at DIKU of which I have been a member from my early graduate student days and which always provides an interesting and stimulating environment for doing research in.

Finally, I would like to thank Anne-Marie for her love and support, and our two lovely daughters Mette and Sara for being able to distract me and pull me away from my work when it was taking me nowhere.

Jesper Jørgensen
Leuven, Belgium
May 1996

This thesis is dedicated
to the Earth
and its future
with or without man.

Sometimes I think it would be better off without us.

Chapter 1

Introduction

1.1 Representation analysis

When implementing statically typed high-level programming languages types can be very useful for optimizing the run-time performance of programs. One thing that types can be used for is to determine the representation of data. In Pascal the exact (monomorphic) type of all expressions can be determined at compile-time which results in implementations with very efficient representation of data at run-time. A problem specific to languages with polymorphic types, such as Standard ML or Haskell, is how to represent the actual arguments to polymorphic functions. The polymorphic (generic) parts of arguments to a polymorphic function can be of any type and such a function will usually be called with actual arguments of many different types. It is therefore not always possible to determine the exact type of all expressions at compile-time. There are several possible ways of implementing such polymorphic languages. The predominant one is to use a Lisp-like data representation where data is represented *uniformly*, independent of their actual type, i.e. everything has to fit into one word. There is, however, no need to give up completely, just because one cannot determine how to represent generic parts of arguments to a polymorphic functions; there is still many cases in which one can.

Representation analysis seeks to optimize the run-time representation of elements of data types in high-level programming languages. A representation in which data is represented uniformly will be called *boxed*. A boxed representation of a data structure is usually a pointer to some area in memory where the actual contents of the data structure reside[1]. The point of this representation is that it has the same "size" for all types of data structures. By passing only generic arguments in boxed representation a (truly) polymorphic function can be correctly implemented by a single piece of code since it is guaranteed to never actually inspect the data structure itself. Other operations, however, such as integer addition or a conditional checking the value of a Boolean expression require access to the contents of the data and are penalized by the additional level of indirection introduced by boxing, as they first have to *unbox* the representation; i.e., dereference the pointer, before they can do their actual operation. Furthermore, boxed representations require more space than unboxed representations thus increasing the space demand and garbage collection costs. Parameter passing, on the other hand, is generally more efficient for boxed than unboxed data representations. Thus there are competing

[1]For elements of "small" data types, such as pointer-sized integers, representations may be considered simultaneously boxed and unboxed. In the following we shall think of these as two separate representations with associated trivial boxing and unboxing operations.

demands on the representation of data in a program. A boxed representation can, of course, be transformed to an unboxed representation at run-time, and vice versa. These conversions can, however, contribute substantially to the run-time cost of a program both in terms of time and space.

Boxing analysis is a special representation analysis that seeks to minimize the need for run-time conversions whilst satisfying the representation demands on all data in a program. Boxing analysis can be facilitated by making representation choices and boxing/unboxing operations in a program explicit. This amounts to a translation into a language with explicit boxed and unboxed types and new operations denoting boxing and unboxing operations without, however, changing the "underlying" program. We shall call these explicit boxing and unboxing operations *(representation) coercions*. There are, in principle, many different possible translations for the same program corresponding to different representation choices for the data structures in the program and different needs for representation coercions. We shall refer to any one of these translations as a *completion* of the underlying program. The question then is: which completion should be chosen for a given program?

In a naïve translation every expression is translated to (a computation of) its boxed representation where operations that need to inspect the contents of such a representation use explicit unboxing operations. The rationale for making boxing explicit is that some boxing/unboxing operations can be eliminated in the later transformational stages of such a compiler [PJL91], as for example in the Glasgow Haskell Compiler. Leroy [Ler92] use a similar naïve (canonical) translation that translate every expression to its unboxed representation where polymorphic operations that need parts of their arguments to be boxed use explicit boxing operations to ensure this. Other translations may elide some of these boxing and unboxing operations directly; e.g., the type inference based translations of Poulsen [Pou93].

1.2 Coercion calculus

Beyond offering yet another translation we seek to formulate and answer the more fundamental questions that underlie the very purpose of boxing analysis and, more generally, similar static analyses: Given two completions for the same program, which of them is better? What does it mean for one completion to be "better" than another completion in the first place? Which programs, if any, have "optimal" completions; i.e., completions that are better than any other for the same program? Can such optimal completions be computed, and how? Of course, it doesn't make much sense to compare the quality of completions on the basis of their actual run-time cost on a specific computer assuming a specific language implementation. In any scenario where we take the actual semantics of the language fully into account the answer to the last two questions would be "no" on recursion-theoretic grounds anyway (assuming the language is universal, of course).

If we can pick any one of a collection of completions for a given program it is a fundamental assumption that all completions must be *coherent*; i.e., they have the same observational behavior. Our approach is to assume that we know *nothing else* about the programming language than that any two completions of the same program are coherent. We have chosen the *polymorphic typed lambda calculus* , also called the *second-order lambda calculus* or F_2, as the subject language for our investigation, since many compilers for polymorphic functional languages now use some applied version of this language as intermediate language. For this we show that coherence can be axiomatized by an equational theory; i.e., a theory of equations of the form $t = t'$ where t, t' are completions of the same program (for a given result representation type) or coercions with the same domain and range. This axiomatization includes the two basic equations

$$\texttt{box;unbox} = \iota$$

$$\texttt{unbox;box} = \iota$$

which express that first boxing and then immediately unboxing (or the other way round) a value (boxed value) is observationally indistinguishable from doing nothing at all to the value (; means diagramatical composition and ι is an identity coercion). We interpret these two equations as left-to-right rewriting rules in accordance with our expectation that performing a pair of coercions is operationally more expensive than doing nothing at all. This gives us a rewriting system modulo the remaining equational axioms. These remaining equations intuitively simply "push" coercions back and forth — e.g., from actual argument to formal parameter in a function application — but they do not eliminate them.

The rewriting system gives us a relatively simple — and coarse — notion of quality: if $e' \Rightarrow^* e''$ then e'' is better than e', and if $e' \Rightarrow^* e''$ for *all* completions e' of a given program then e'' is an "optimal" completion (modulo the remaining equational axioms mentioned above). Unfortunately the resulting notion of reduction is not Church-Rosser; i.e., there exist coherent completions that have no common reduct and thus only "locally" optimal completions. We will show that this is due to the fact that a `box;unbox`-redex may only be eliminated at the expense of *introducing* a `unbox;box`-redex, and vice versa.

By giving higher priority to the elimination of `unbox;box`-redexes over `box;unbox` or the other way round, however, we arrive at two *formal optimality* criteria for completions. We show that every program has a formally optimal completion at any given representation type under each of the two choices of priority. This is accomplished by introducing a second equational system E_p equivalent to E and orienting this as left-to-right or right-to-left rewriting rules depending on the *polarity* of the coercions involved. (Any simple-minded orientation of E leads to nonconfluence and nontermination of Knuth-Bendix completion.) The resulting two rewriting systems can be used to compute specific optimal completions.

Formulating boxing analysis in the framework of a formal coercion calculus has the advantage that the results we obtain are extremely general and robust:

1. They apply to any interpretation whatsoever of the underlying programming language; e.g., to a call-by-value, call-by-name, or lazy interpretation of our functional language. In Section 6.7 we will descuss the implications of this for lazy languages.

2. They can be combined with other optimizations unrelated to boxing as the calculus makes few assumptions about the underlying language or its implementation technology.

3. They admit talking about optimality relative to an explicit, formally specified criterion.

4. They leave a great degree of freedom as optimality is accomplished up to a well-defined congruence relation on completions; for example, the notion of optimality is not overcommitted by insisting on syntactic uniqueness.

1.3 A motivating example

Let us first look at a simple informal example to clarify the concept. Consider the polymorphic identity function `id` with type $\forall\alpha.\alpha\to\alpha$ defined by (in ML by a let-binding):

$$\lambda x:\alpha\,.\,x$$

and consider the application `id 5`. In this the argument to `id` must be boxed and if we further assume that the final result of the application must be an unboxed integer, we could insert coercions that ensures that the appropriate changes in representation occur. An *explicitly* boxed expressions (completion) corresponding to the application could then look:

$$(\langle \mathtt{box}_{\mathtt{int}} \to \mathtt{unbox}_{\mathtt{int}}\rangle \mathtt{id})\ \mathtt{5}$$

where an expression of the form $\langle\mathbf{c}\rangle e$ is an application of a coercion $\mathbf{c}$ to an expression e. The coercions $\mathtt{box}_\tau$ and $\mathtt{unbox}_\tau$ respectively boxes and unboxes a value of type τ and the coercion $\mathbf{c}\to\mathbf{d}$ acting on functions changes an (unboxed) function f into a function that first apply $\mathbf{c}$ to its argument then passes the result of this to f and finally applies $\mathbf{d}$ to the output from f. Informally we can write this as $\langle\mathbf{c}\to\mathbf{d}\rangle f \equiv \mathbf{d}\circ f\circ\mathbf{c}$.

The completion above is essentially optimal (we do not consider that `id` could be a no-op), and one can easily write a translation that produces the completion using the definition of `id` with explicit type [Ler92]. Such a translation should for every use (instantiation) of a polymorphic function f insert a coercion $\mathbf{c}$ to make sure that the generic parts of the arguments are properly boxed and that the expression:

$$\langle\mathbf{c}\rangle f$$

has a completely unboxed type (that is, expects unboxed arguments and returns an unboxed result). But if such a translation only considers instantiations of polymorphic functions locally it will not in general produce optimal or very efficient completions. Consider, for example, a slight extension of the expression above `id (id 5)`. Using the simple local completion method we would get:

$$\langle \mathtt{box}_{\mathtt{int}} \to \mathtt{unbox}_{\mathtt{int}}\rangle \mathtt{id}\ (\langle \mathtt{box}_{\mathtt{int}} \to \mathtt{unbox}_{\mathtt{int}}\rangle \mathtt{id}\ \mathtt{5})$$

Writing this out using our informal definition of function coercions we get:

$$\mathtt{unbox}_{\mathtt{int}} \circ \mathtt{id} \circ \mathtt{box}_{\mathtt{int}} \circ \mathtt{unbox}_{\mathtt{int}} \circ \mathtt{id} \circ \mathtt{box}_{\mathtt{int}}(5)$$

from which it is clear that $\mathtt{box}_{\mathtt{int}} \circ \mathtt{unbox}_{\mathtt{int}}$ can be removed altogether. In fact, using the equational theory for completions described in this thesis we may easily show that the completion is equivalent to the optimal completion:

$$\langle \mathtt{unbox}_{\mathtt{int}}\rangle(\mathtt{id}\ (\mathtt{id}\ \langle \mathtt{box}_{\mathtt{int}}\rangle \mathtt{5}))$$

The example is of course quite a trivial one and the same result could be obtained by a simple local (peephole) optimization, but the point is that our method will do similar global optimizations too. Leroy's framework [Ler92] does not achieve these kind of optimizations.

1.4 Computing boxing completions

Although the rewriting-based algorithm gives a way to compute formally optimal completions it is not the most efficient way to do this. We show that an efficient algorithm exist which uses a method similar to that of Poulsen's [Pou93], but which in contrast to Poulsen's algorithm always finds optimal completions. This algorithm makes it easy to see that our work is an improvement Poulsen's. Together with the fact that also Leroy's algorithm [Ler92] does not in general produce optimal completions (see section 1.3) this shows that our framework is an improvement over both of these related works.

1.5 New results

Although polymorphism is at the heart of the problem we attack, it also introduces new challenging problems that needs to be solved. We need to be able to show that all congruent completions are equivalent in the presence of polymorphism. We have solved these problems and believe that this is the most relevant theoretic contribution of our work. Polymorphism is not considered in Henglein's work on dynamic typing [Hen93] and it is possible that the results of our thesis also may prove useful in this and other similar areas, like for instance subtyping.

The boxing algorithm and the quality of its output is apparently the most immediate and practically relevant contribution of our work. It could certainly have been presented, together with the empirical results, independently of the coercion calculus and its formal optimality criteria. But this would have been unsatisfactory in several respects:

With a proliferation of different algorithms for the same problem there is a clear need for a systematic comparison between them. Using exclusively empirical data is unsatisfactory for this purpose as they can only report on system performance where the interaction of boxing with other system properties changes frequently and is difficult to quantify. Our optimality criterion is simple, natural and facilitates a completely formal comparison of boxing completions; furthermore, it makes the basis of comparison explicit and thus, if nothing else, facilitates a substantive criticism of its rationale.

Our boxing algorithm has been developed from a systematic analysis of the coercion calculus and its optimality criterion. Without the general framework it would doubtlessly appear *ad hoc*. It would also have been impossible to say anything about its "robustness" and global properties; for example, the algorithm produces the *same* completion when given either one of the completions of [PJL91, Ler92, Pou93] as its initial input. This follows from the coherence of all completions and the Church-Rosser and strong normalization properties of the rewriting systems.

To summarize, the contributions of this work are:

- a general framework and robust criterion for the quality of boxing completions, which accounts for the costs of boxing/unboxing operations, but abstracts from other language properties and implementation concerns.

- a proof of the existence of *formally optimal (boxing) completions* and their uniqueness modulo an equational theory for moving boxing and unboxing operations along data flow paths. Our notion of formal optimality is independent of any specific properties of the underlying programming language.

- a way to integrate polymorphism and coercion calculus that may turn out to be useful in other contexts.
- a rewriting-based algorithm for computing formally optimal completions, which are uniformly better than those described in [PJL91, Ler92, Pou93] in our (formal) sense.
- an efficient graph based algorithm for computing formally optimal completions, which runs linear in the size of the type checking (type inference for ML) derivation, i.e. add only a constant factor to the this process.
- an experimental implementation of the algorithm and test results for a call-by-value language that support empirically that our completions are also better in practice than those reported in the literature previously.

1.6 Overview

The thesis is divided into two parts. The first part, the chapters 2 to 5, present the theory of the calculus of coercions and completions used in boxing analysis for F_2, and the second part, the chapters 6 and 7, presents boxing analysis for ML like languages and a prototype implementation with benchmarks.

The last two chapters 8 and 9 contains a discussion of related work and a conclusion with suggestions for future work.

1.6.1 Overview over part I

In Chapter 2 we present the basic elements of the calculus of coercions and completions. We present the syntax of coercion and completions and a type system for completions, as well as an inference system (Figure 5) for inferring congruent completions. We also present the important notion of congruence, that formalizes what is meant by two completions being equal, i.e. that two (closed) completion of the same F_2-expression at same representation type are congruent (equal).

The goal of boxing analysis is not just to find any completion, but to find one that is in some sense better that all other. Since coercions are parts of completions, we need to investigate these as well as completions. We need to find out in what sense we may regard one coercion as being better than another, if there exist a best one with regard to some way of measuring quality of coercions, and how to find it (if it exist). So in Chapter 3 we will show how we can define optimality for coercions formally and how we can find optimal coercions. For now, it is enough to think of quality of coercions as "the performance of coercions". We will further define an equality theory on coercions that equates all coercions that formally have the same quality, and show that optimal coercions are unique modulo this equality. Finally, we will show some important properties of coercions that will be needed when we investigate coherence, reduction and optimality of completions in Chapter 4.

In Chapter 4 we will extend our investigation to include completions. That is, we will investigate whether there, based on the way of measuring quality of coercions, exist optimal completions and how to find these (if they exist). We will extend the equality axioms for coercions with equality axioms for expressions to obtain an equality theory on completions. It is obvious that coercion equality must be part of such an equality, since replacing equal coercions in a completion should not result in a completion with a different performance, and such completions should definitely be considered equal by our theory. Intuitively the equality axioms for expressions "move" coercions around in completions without eliminating them. From the equality axioms for expressions we will define several different notions of reduction on completions in a manner similar to what we did for coercions. First we will define $\phi\psi$-reduction on completions as $\phi\psi$ reduction modulo the completion equality. We will show that $\phi\psi$-reduction on completions is not, as one would have hoped, confluent, but we will, however, by giving higher priority to the elimination of `unbox;box`-redexes over `box;unbox` or the other way round, we arrive at two *formal optimality* criteria for completions. We show that there exist reductions systems that may be used in finding such optimal completions and that these systems, fortunately, are canonical. We also show that the two kinds of optimal completions are unique modulo completion equality.

In Chapter 5 we demonstrate how to find optimal completions of the kind described in Chapter 4. We will present two basically different methods to compute optimal completions. The first one is based on reduction using methods developed in Chapters 2, 3 and 4. We will do this by showing how to implement the reductions systems described in Chapter 4. The second method, developed in Chapter 5, is based on transforming the problem of finding optimal completions into what essentially becomes a graph reachability problem. It is on this second method that our prototype implementation describe in Chapter 7 is based.

1.6.2 Overview over part II

The theory as well as the algorithm presented in Part I was for the polymorphic lambda calculus only. The second part of this thesis is concerned with implementation aspect of boxing analysis. So in Chapter 6 we look at how the framework may be extended to include the most common elements of ML-like languages, like polymorphic constants (`map`, `@`, etc.) and languages primitives, like conditionals, let-bindings, recursive functions, etc. We also discuss how to optimize the execution efficiency of completions, that is, finding the best completion is the optimal class.

In chapter Chapter 7 we describe the prototype implementation and presents some benchmarks for this.

1.6.3 Appendixes

Appendix A contains a summary of notation used in this report and we recommend that the reader consults this before reading the thesis.

The other appendices contains the ML code of the implementation (Appendix B), an example of a session with the implementation (Appendix C) and the programs used in the performance test (Appendix D).

Part I

A coercion theory for polymorphic lambda calculus

Chapter 2

Polymorphic lambda calculus as a framework for boxing analysis

The *polymorphic typed lambda calculus*, also called the *second-order lambda calculus* or F_2 was invented independently by Girard and Reynolds as an extension of the usual typed lambda calculus with type abstraction and type application. We use F_2 as an "universal" intermediate languages and will use only little of the calculus of F_2. The idea is that boxing analysis is going to fit into a compiler somewhere after the type inference phase and is going to be performed on an explicitly typed language. This makes F_2 an excellent starting point for investigating boxing analysis for polymorphic languages. In this chapter we present the basic elements of the calculus of coercions and completions. We present the syntax of coercion and completions, as well as an inference system (Figure 5) for inferring congruent completions. We also present the important notion of congruence, that formalizes what is meant by two completions being equal, i.e. that two (closed) completion of the same F_2-expression at same representation type are congruent (equal).

2.1 The polymorphic λ-calculus

The syntactic constructs for *pre-expressions* and types (type schemes) in F_2 are given in Figure 1. We call a pre-expression e *well-formed* (or simply an *F_2-expression*) under *type assignment* Γ, if $\Gamma \vdash e{:}\sigma$ is derivable from the inference rules in Figure 2 for some type σ. It is easy to see that there is at most one typing derivation for e, which also determines σ.

$x \in \mathit{Variable}$; $\alpha \in \mathit{TypeVariable}$

$e ::= x \mid \lambda x{:}\sigma.e \mid e\ e \mid \Lambda\alpha.e \mid e\{\sigma\}$	*Expression*
$\sigma ::= \alpha \mid \sigma \rightarrow \sigma \mid \forall\alpha.\sigma$	*Type*

Figure 1: Syntactic categories of F_2

$$\frac{\Gamma\{x : \sigma_1\}\vdash e:\sigma_2}{\Gamma\vdash\lambda x:\sigma_1 . e:\sigma_1\to\sigma_2} \qquad \frac{\Gamma\vdash e_1:\sigma_1\to\sigma_2 \quad \Gamma\vdash e_2:\sigma_1}{\Gamma\vdash e_1\ e_2:\sigma_2} \qquad \Gamma\{x : \sigma\}\vdash x:\sigma$$

$$\frac{\Gamma\vdash e:\sigma}{\Gamma\vdash\Lambda\alpha . e:\forall\alpha.\sigma}\ \text{if } \alpha \notin FV(\Gamma) \qquad \frac{\Gamma\vdash e:\forall\alpha.\sigma}{\Gamma\vdash e\{\sigma_1\}:\sigma[\sigma_1/\alpha]}$$

Figure 2: Typing rules for F_2

2.2 Explicitly boxed F_2

Explicitly boxed F_2 is a refinement of F_2 in which representation types (boxed/unboxed types) and conversions between these are made explicit. We start by considering the types.

2.2.1 Representation types

Representation types ρ are build from the standard type constructors of F_2, together with one additional type constructor, $[\_]$. Types of the form $[\rho]$ are called *boxed types*; they describe elements that have been boxed. Types with any other top-level type constructor ($\to$ or $\forall.$) are called *unboxed types*; their elements are not boxed. Since doubly boxed representations are useless we prohibit boxed types of the form $[[\rho]]$. Boxed types may otherwise, however, occur inside both boxed and unboxed types. Type variables denoted by metavariables α therefore ranges only over boxed types. Abstract syntax definitions of representation types, boxed and unboxed types are given in Figure 3.

$$\begin{array}{lll} \rho ::= v \mid \pi & & \textit{Representation types} \\ v ::= \rho\to\rho \mid \forall\alpha.\rho & & \textit{Unboxed types} \\ \pi ::= \alpha \mid [v] & & \textit{Boxed types} \end{array}$$

Figure 3: Representation types of explicitly boxed F_2

Definition 2.1 *(Type erasure) The* (underlying) standard type *(or* type erasure*)* $|\rho|$ of representation type ρ is the type arrived at by erasing all occurrences of $[\_]$ in ρ. We say that ρ *represents* $|\rho|$. We say ρ is *valid* for a (closed) F_2-expression e if e has type ρ.

Example 2.1 *(Type erasure)*

The representation types $\forall\alpha.\alpha\to\alpha$, $\forall\alpha.[\alpha\to\alpha]$ and $[\forall\alpha.\alpha\to\alpha]$ all have the same type erasure, namely $\forall\alpha.\alpha\to\alpha$.

2.2.2 Coercions

Representation coercions (or simply *coercions*) are operations that coerce an element from one representation to another. If a coercion **c** coerces elements of type ρ to elements of type ρ' then **c** is said to have *type signature* $\rho \leadsto \rho'$. We will write $\vdash$ **c** : $\rho \leadsto \rho'$ or simply **c**:$\rho \leadsto \rho'$ to state the fact that **c** have *type signature* $\rho \leadsto \rho'$. The primitive coercions are

- $\mathtt{box}_v$: $v \leadsto [v]$, called a *box coercion.*

- $\mathtt{unbox}_v$: $[v] \rightsquigarrow v$, called an *unbox coercion.*

where $\mathtt{box}_v$ coerces an unboxed element of type v to a boxed representation, and $\mathtt{unbox}_v$ takes such a boxed representation and coerces it back to the unboxed representation.

Beyond these primitive coercions we add the identity coercion ι_ρ (at every type ρ), composition of coercions $\mathbf{c}$, $\mathbf{c}'$ written in diagrammatical order $\mathbf{c}\mathtt{;}\mathbf{c}'$ and coercions *induced* by the type constructors. In our case these coercions are:

- $\mathbf{c}\to\mathbf{c}'$, called a *function coercion.*
- $\forall\alpha\,.\,\mathbf{c}$, called an *abstraction coercion.*
- $[\mathbf{c}]$, called a *boxed coercion*[1].

A function coercion $\mathbf{c}\to\mathbf{c}'$ applies to functions f by "wrapping" it with the input coercion $\mathbf{c}$ and the output coercion $\mathbf{c}'$. The result is an unboxed function where $\mathbf{c}$ is applied to the input before it is passed to f and $\mathbf{c}'$ is applied to the result of f before it is returned as the output. An abstraction coercion $\forall\alpha\,.\,\mathbf{c}$ is not as one might expect a polymorphic coercion $\mathbf{c}$ parameterized over the boxed type variable α. Basically, we will require that we may form coercions from any representation type ρ to any other representation type ρ' as long as the two representation types have the same type erasure. Indeed this means that we need coercions from $\forall\alpha.\rho$ to $\forall\alpha.\rho'$. Assume that $\mathbf{c}$ is a coercion from ρ to ρ' then $\forall\alpha\,.\,\mathbf{c}$ is a coercion from $\forall\alpha.\rho$ to $\forall\alpha.\rho'$. A *polymorphic coercion* by analogy with polymorphic function is a coercion parameterized over a type. A boxed coercion $[\mathbf{c}]$ applies $\mathbf{c}$ to the underlying unboxed value of a boxed representation and returns a boxed representation for the result. We will sometimes omit subscripts on coercions when these are not important for the presentation.

The rules for forming coercions are displayed in Figure 4. In the following we will use $\mathbf{c}$, $\mathbf{c}'$, $\mathbf{d}$, etc. to denote arbitrary coercions.

$$\vdash \iota_\rho : \rho \rightsquigarrow \rho \qquad \frac{\vdash \mathbf{c} : {\rho_1}' \rightsquigarrow \rho_1 \quad \vdash \mathbf{c}' : \rho_2 \rightsquigarrow {\rho_2}'}{\vdash \mathbf{c}\to\mathbf{c}' : \rho_1\to\rho_2 \rightsquigarrow {\rho_1}'\to{\rho_2}'} \qquad \frac{\vdash \mathbf{c} : \rho \rightsquigarrow \rho' \quad \vdash \mathbf{c}' : \rho' \rightsquigarrow \rho''}{\vdash \mathbf{c}\mathtt{;}\mathbf{c}' : \rho \rightsquigarrow \rho''}$$

$$\vdash \mathtt{box}_v : v \rightsquigarrow [v] \qquad \vdash \mathtt{unbox}_v : [v] \rightsquigarrow v \qquad \frac{\vdash \mathbf{c} : v \rightsquigarrow v'}{\vdash [\mathbf{c}] : [v] \rightsquigarrow [v']}$$

$$\frac{\vdash \mathbf{c} : \rho \rightsquigarrow \rho'}{\vdash \forall\alpha\,.\,\mathbf{c} : \forall\alpha.\rho \rightsquigarrow \forall\alpha.\rho'}$$

Figure 4: Coercion formation rules

The formation rules for coercions are sufficient to construct coercions that can transform a value from any one of its representations to any other representation:

Proposition 2.2 *Let ρ, ρ' be arbitrary representation types. Then*

$$|\rho| = |\rho'| \Leftrightarrow (\exists\mathbf{c}) \vdash \mathbf{c} : \rho \rightsquigarrow \rho'.$$

[1]We know that this is not a very good name, since it can easily be confused with box coercion $\mathtt{box}_v$, but it was the best we could come up with and it is consistent with calling the type $[v]$ a boxed type

Proof:
"only if": The proof is by structural induction on the common erasure of ρ and ρ'.

Base case: $|\rho| \equiv \alpha$. Then both $\rho \equiv \alpha$ and $\rho' \equiv \alpha$ and we may therefore choose $\mathbf{c}$ to be ι_α.

Inductive case: $|\rho| \equiv \tau_1 \rightarrow \tau_2$. There are four cases that have to be checked according to whether ρ is a boxed type or an unboxed type and ρ' is a boxed type or an unboxed type: (1) Assume that $\rho \equiv \rho_1 \rightarrow \rho_2$ and $\rho' \equiv \rho_1' \rightarrow \rho_2'$. Then by induction there exist coercions $\vdash \mathbf{d}_1 : \rho_1' \rightsquigarrow \rho_1$ and $\vdash \mathbf{d}_2 : \rho_2 \rightsquigarrow \rho_2'$ and we may therefore choose $\mathbf{c}$ to be $\mathbf{d}_1 \rightarrow \mathbf{d}_2$. (2) Assume that $\rho \equiv \rho_1 \rightarrow \rho_2$ and $\rho' \equiv [\rho_1' \rightarrow \rho_2']$. Then by induction there exist coercions $\vdash \mathbf{d}_1 : \rho_1' \rightsquigarrow \rho_1$ and $\vdash \mathbf{d}_2 : \rho_2 \rightsquigarrow \rho_2'$ and we may therefore choose $\mathbf{c}$ to be $\mathbf{d}_1 \rightarrow \mathbf{d}_2 \,\mathtt{;}\, \mathtt{box}_{\rho_1' \rightarrow \rho_2'}$. (3) Assume that $\rho \equiv [\rho_1 \rightarrow \rho_2]$ and $\rho' \equiv \rho_1' \rightarrow \rho_2'$. Then by induction there exist coercions $\vdash \mathbf{d}_1 : \rho_1' \rightsquigarrow \rho_1$ and $\vdash \mathbf{d}_2 : \rho_2 \rightsquigarrow \rho_2'$ and we may therefore choose $\mathbf{c}$ to be $\mathtt{unbox}_{\rho_1 \rightarrow \rho_2} \,\mathtt{;}\, \mathbf{d}_1 \rightarrow \mathbf{d}_2$. (4) Assume that $\rho \equiv [\rho_1 \rightarrow \rho_2]$ and $\rho' \equiv [\rho_1' \rightarrow \rho_2']$. Then by induction there exist coercions $\vdash \mathbf{d}_1 : \rho_1' \rightsquigarrow \rho_1$ and $\vdash \mathbf{d}_2 : \rho_2 \rightsquigarrow \rho_2'$ and we may therefore choose $\mathbf{c}$ to be $\mathtt{[}\mathbf{d}_1 \rightarrow \mathbf{d}_2\mathtt{]}$.

Inductive case: $|\rho| \equiv \forall\alpha.\tau$. Like for function types there are again four cases that has to be checked according to whether ρ is a boxed type or an unboxed type and ρ' is a boxed type or an unboxed type. These are proven proven analogous to the cases for function types.

"if": This is proven by induction over the structure of the coercion $\mathbf{c}$.

Base case: $\mathbf{c} \equiv \iota_\rho$: This will only be the case if the two types ρ and ρ' are the same so the case is trivial.

Base case: $\mathbf{c} \equiv \mathtt{box}_\rho$: This will only be the case if $\rho' \equiv [\rho]$ so obviously $|\rho|{=}|\rho'|$ must hold.

Base case: $\mathbf{c} \equiv \mathtt{unbox}_{\rho'}$: This will only be the case if $\rho \equiv [\rho']$ so obviously $|\rho|{=}|\rho'|$ must hold.

Inductive case: $\mathbf{c} \equiv \mathbf{c}_1 \rightarrow \mathbf{c}_2$: Let $\mathbf{c}_1$ have signature $\rho_1 \rightsquigarrow \rho_1'$ and $\mathbf{c}_2$ have signature $\rho_2 \rightsquigarrow \rho_2'$ then $\mathbf{c}$ has signature $\rho_1' \rightarrow \rho_2 \rightsquigarrow \rho_1 \rightarrow \rho_2'$. By induction $|\rho_1|{=}|\rho_1'|$ and $|\rho_2|{=}|\rho_2'|$ and we have $|\rho| = |\rho_1' \rightarrow \rho_2| = |\rho_1'| \rightarrow |\rho_2| = |\rho_1| \rightarrow |\rho_2'| = |\rho_1 \rightarrow \rho_2'| = |\rho'|$.

Inductive case: $\mathbf{c} \equiv \forall\alpha\,.\,\mathbf{c}_1$: Let $\mathbf{c}_1$ have signature $\rho_1 \rightsquigarrow \rho_1'$ then $\mathbf{c}$ has signature $\forall\alpha.\rho_1 \rightsquigarrow \forall\alpha.\rho_1'$. By induction $|\rho_1|{=}|\rho_1'|$ and we therefore have $|\rho| = |\forall\alpha.\rho_1| = \forall\alpha.|\rho_1| = \forall\alpha.|\rho_1'| = |\forall\alpha.\rho_1'| = |\rho'|$.

Inductive case: $\mathbf{c} \equiv \mathbf{c}_1 \,\mathtt{;}\, \mathbf{c}_2$: Let Let $\mathbf{c}_1$ have signature $\rho \rightsquigarrow \rho''$ and $\mathbf{c}_2$ have signature $\rho'' \rightsquigarrow \rho'$. By induction we have $|\rho| = |\rho''| = |\rho'|$.

Inductive case: $\mathbf{c} \equiv \mathtt{[}\mathbf{c}_1\mathtt{]}$: Let $\mathbf{c}_1$ have signature $\rho_1 \rightsquigarrow \rho_1'$ then $\mathbf{c}$ has signature $[\rho_1] \rightsquigarrow [\rho_1']$. By induction $|\rho_1|{=}|\rho_1'|$ and we therefore have $|\rho| = |[\rho_1]| = |\rho_1| = |\rho_1'| = |[\rho_1']| = |\rho'|$.

■

Indeed Proposition 2.2 holds even without the `[_]`-coercion constructor. We have added `[_]` solely to facilitate coercion factoring and simplification "underneath" boxed representations for $\psi^{\rightarrow}E_p^{\leftarrow}$-reduction modulo ϕ (see Chapter 4), where we may introduce `box;unbox`-redexes in order to eliminate `unbox;box`-redexes.

2.2.3 Type inference rules

The type inference system for explicitly boxed F$_2$ (see Figure 5) is almost identical to the standard type system. There are two noteworthy differences, however.

$$\Gamma\{x : \rho\}\vdash x{:}\rho \qquad \frac{\Gamma\vdash e{:}\rho}{\Gamma\vdash\Lambda\alpha\,.\,e{:}\forall\alpha.\rho}\ \text{if } \alpha \notin FV(\Gamma)$$

$$\frac{\Gamma\vdash e{:}\forall\alpha.\rho}{\Gamma\vdash e\{\pi\}{:}\rho[\pi/\alpha]} \qquad \frac{\Gamma\vdash e_1{:}\rho_1\rightarrow\rho_2 \quad \Gamma\vdash e_2{:}\rho_1}{\Gamma\vdash e_1\ e_2{:}\rho_2}$$

$$\frac{\Gamma\{x : \rho_1\}\vdash e{:}\rho_2}{\Gamma\vdash\lambda x : \rho_1\,.\,e{:}\rho_1\rightarrow\rho_2} \qquad \frac{\Gamma\vdash e{:}\rho \quad \vdash \mathbf{c} : \rho \leadsto \rho'}{\Gamma\vdash\langle\mathbf{c}\rangle e{:}\rho'}$$

Figure 5: Typing rules for the explicitly boxed F_2

1. Quantification in types is only over boxed type variables. The fact that these type variables indeed range over boxed types *only* is captured in the rule for type application: a polymorphically typed expression can only be applied to a boxed type, not an unboxed type.

2. There is an additional rule for applying coercions to expressions: $\langle\mathbf{c}\rangle e$.

Definition 2.3 *(Erasure, completion)*
The *erasure* (or *underlying F_2-expression*) $|e|$ of an explicitly boxed F$_2$-expression e is the (standard) F$_2$-expression arising from e by erasing all occurrences of coercions (including angled brackets) and replacing all representation type occurrences ρ by $|\rho|$. We say e is a *(boxing) completion* of $|e|$ *at type* ρ if e has type ρ.

Example 2.2 *(Expression erasure)*
The erasure of the explicitly boxed F$_2$-expression $\lambda x : [\alpha\rightarrow\alpha]\,.\,\langle\texttt{unbox}_{\alpha\rightarrow\alpha}\rangle x$ is $\lambda x : \alpha\rightarrow\alpha\,.\,x$.

2.3 Coercion and completion congruence

A given F$_2$-expression e can have many completions. Without going into the semantics of F$_2$ we assume that all completions of e at a specific representation type have the same observational (input/output) behavior, but possibly different performance. In fact we shall for now assume *nothing else* about the semantics of explicitly boxed F$_2$, and in fact nothing at all about the semantics of standard F$_2$. To express this formally we define the notion of congruence:

Definition 2.4 *(Coercion and Completion congruence)*
Representation coercions $\mathbf{c}$ and $\mathbf{c}'$ are *congruent*, written $\mathbf{c} \cong \mathbf{c}'$, if they have the same type signature $\rho \leadsto \rho'$. Explicitly boxed expressions e and e' are *congruent*, written $e \cong e'$, if they have the same erasure and representation type (under the same type assignment); that is, they are completions of the same expression at the same representation type under the same type assignment for the free variables.

Intuitively congruent coercions perform the same conversion from one representation to another (determined by their common signature) although two given congruent coercions may look very different. Similarly congruent completions represent different choices of "internal" representation for a given F_2 expression under the same "external" choice of representation (the representation types of the free variables and the expression itself).

Chapter 3

Coercion calculus

In Chapter 2 we gave a system (Figure 5) for inferring congruent completions. The goal of boxing analysis is not just to find one such completion, but to find one that is in some sense better that all other. So we need to investigate in what sense we will regard one completion better than another, if there exist a best one with regard to this way of measuring quality of completions, and how to find it (if it exist). One way that one could consider one completion better than another is with respect to performance of operations related to representation of data. Since coercions are one of the important elements in such considerations we will first investigate coercions by themselves. Since coercions are parts of completions, improving the performance of the coercions in a completion should at least in principle improve the performance of the completion. So in this chapter we will show how we can define optimality for coercions formally and how we can find optimal coercions. For this we develop an equality theory on coercions that equates all coercions that formally have the same performance properties, and use this to define a canonical reduction system on coercions. The normal forms for this system are the optimal coercions. We show these optimal coercions are unique modulo this equality. Finally, we will show some properties about coercions that we will be using when we look at coherence, reduction and optimality of completions in Chapter 4.

3.1 Equational axiomatization of coercion congruence

In this section we shall give an equational axiomatization of coercion congruence. The axioms can be grouped into a set of core equations C and a small group consisting of two axioms that express that boxing composed with unboxing in either order is equal to the identity coercion. We shall show how to construct a reduction relation on coercion from the equational axiomatization and show how this is *canonical* (strongly normalizing and confluent). We will then using the reduction relation prove coherence of coercions. Finally we define what we mean by optimal coercions and show that optimal coercions exist for type signatures.

3.1.1 Coercion equations

Consider the equality axioms in Figures 6 and 7 for coercions. We assume that the coercions on both sides of an equality are well-formed and have the same type signature. We denote the single equality axiom $\texttt{box}\,;\texttt{unbox} = \iota$ by ϕ, and $\texttt{unbox}\,;\texttt{box} = \iota$ by ψ. The equation in Figure 6 will also be called the core coercion equations and will be denoted C.

$$\boxed{\begin{array}{rcll}
(\mathbf{c}\,;\mathbf{c}')\,;\mathbf{c}'' & = & \mathbf{c}\,;(\mathbf{c}'\,;\mathbf{c}'') & (Ass)\\
\mathbf{c}\,;\iota & = & \mathbf{c} & (1)\\
\iota\,;\mathbf{c} & = & \mathbf{c} & (2)\\
(\mathbf{c}_1\!\rightarrow\!\mathbf{c}_2)\,;(\mathbf{c}_1'\!\rightarrow\!\mathbf{c}_2') & = & (\mathbf{c}_1'\,;\mathbf{c}_1)\!\rightarrow\!(\mathbf{c}_2\,;\mathbf{c}_2') & (3)\\
\iota\!\rightarrow\!\iota & = & \iota & (4)\\
\texttt{[}\mathbf{c}_1\texttt{]}\,;\texttt{[}\mathbf{c}_2\texttt{]} & = & \texttt{[}\mathbf{c}_1\,;\mathbf{c}_2\texttt{]} & (5)\\
\texttt{[}\iota\texttt{]} & = & \iota & (6)\\
\forall\alpha\,.\,\iota & = & \iota & (7)\\
(\forall\alpha\,.\,\mathbf{c}_1)\,;(\forall\alpha\,.\,\mathbf{c}_2) & = & \forall\alpha\,.\,(\mathbf{c}_1\,;\mathbf{c}_2) & (8)\\
\texttt{box}_{\upsilon}\,;\texttt{[}\mathbf{c}\texttt{]} & = & \mathbf{c}\,;\texttt{box}_{\upsilon'} & (9)\\
\texttt{[}\mathbf{c}\texttt{]}\,;\texttt{unbox}_{\upsilon} & = & \texttt{unbox}_{\upsilon'}\,;\mathbf{c} & (10)
\end{array}}$$

Figure 6: Equality rules for coercions: Core Equations (C)

$$\boxed{\begin{array}{rcll}
\texttt{box}_{\upsilon}\,;\texttt{unbox}_{\upsilon} & = & \iota & (\phi)\\
\texttt{unbox}_{\upsilon}\,;\texttt{box}_{\upsilon} & = & \iota & (\psi)
\end{array}}$$

Figure 7: The ϕ and ψ rules for coercions

Definition 3.1 *(Coercion equality)*
Let A be a set of coercion equality axioms, we say $\mathbf{c}$ and $\mathbf{c}'$ are *A-equal*, written $A \vdash \mathbf{c} = \mathbf{c}'$, if $\mathbf{c} = \mathbf{c}'$ is derivable from the equality axioms A and the core coercion equations in Figure 6 together with reflexivity, symmetry, transitivity and compatibility of $=$ with arbitrary contexts. If $A \vdash \mathbf{c} = \mathbf{c}'$ does not hold we write $A \vdash \mathbf{c} \neq \mathbf{c}'$.

If A in the definition is empty we say $\mathbf{c}$ and $\mathbf{c}'$ are *equal* and write $\vdash \mathbf{c} = \mathbf{c}'$ or just $\mathbf{c} = \mathbf{c}'$. Note that equality is *not* just syntactic identity; e.g., we have $\vdash \iota_\rho \!\rightarrow\! \iota_\rho = \iota_{\rho\rightarrow\rho}$. We will also speak of A-equality for the equality defined by the set A and talk about A-equivalence classes.

Definition 3.2 *(Proper coercions)* A coercion that is not equal to an identity coercion is called *proper*, that is, a coercion $\mathbf{c}$ is not proper if $\mathbf{c} = \iota$.

Example 3.1 Let us as an example show that $\texttt{[}\mathbf{c}\texttt{]}$ is $\phi\psi$-equal to $\texttt{unbox}\,;\mathbf{c}\,;\texttt{box}$, that is, $\phi\psi \vdash \texttt{[}\mathbf{c}\texttt{]} = \texttt{unbox}\,;(\mathbf{c}\,;\texttt{box})$:

$$\texttt{[}\mathbf{c}\texttt{]} =_2 \iota\,;\texttt{[}\mathbf{c}\texttt{]} =_\psi (\texttt{unbox}\,;\texttt{box})\,;\texttt{[}\mathbf{c}\texttt{]} =_{Ass} \texttt{unbox}\,;(\texttt{box}\,;\texttt{[}\mathbf{c}\texttt{]}) =_9 \texttt{unbox}\,;(\mathbf{c}\,;\texttt{box})$$

so the equality $\phi\psi \vdash \texttt{[}\mathbf{c}\texttt{]} = \texttt{unbox}\,;(\mathbf{c}\,;\texttt{box})$ then follow from transitivity of $\phi\psi$-equality (we have also tacitly used compatibility in the second and the last equality).

3.1.2 Coercion reduction

Our goal is to find completions with a minimum of boxing and unboxing operations and therefore also coercions with a minimum of `box`- and `unbox`-coercions. Let us take a look at the equations for coercions shown in Figure 6 and Figure 7. In Section 3.1.1 we defined an equality theory for coercions based on these equations. What we will show later (coherence of coercions) is that this theory equates all coercions with the same signature. The idea behind coercion reduction is to regard the ϕ and ψ rules as left to right rewrite rules and define reduction as rewriting by these rules modulo the core coercion equations. We want reduction to be canonical (strongly normalizing and confluent) and such that any two congruent coercions reduce to the same normal form. The rewrite system that we obtain this way implements what we will call $\phi\psi$-reduction.

Definition 3.3 *($\phi\psi$-reduction)*
Let now ϕ and ψ stand for the ϕ and ψ rules regarded as left to right rewrite rules. We define $\phi\psi$-reduction to be $\phi\psi$-reduction modulo A-equality[1] where A is empty. Remember that equality on coercions always includes the core equations C from Figure 6.

Clearly, ϕ and ψ eliminate primitive coercions when applied as left-to-right rewriting rules whereas the remaining coercion equations just express differences in the presentation of a coercion. If we accept that boxing and unboxing coercions are more expensive than the identity coercion, then reduction will never make coercions more expensive only less or equally expensive. That is, reduction will in general improve performance of coercions (make them less expensive) and a normal form coercion will be better than all the coercions that reduce to it.

A $\phi\psi$-normal form is an equivalence class and $\phi\psi$-reduction is reduction on A-equivalence classes where A is empty. We will in the following not distinguish between these equivalence classes and their representatives.

Example 3.2 *($\phi\psi$-reduction)*
As an example of $\phi\psi$-reduction we show how we may $\phi\psi$-reduce `unbox;(`**c**`;box)` to `[`**c**`]` where **c** is an arbitrary coercion:

$$\begin{array}{ll}
\texttt{unbox;(}\mathbf{c}\texttt{;box)} & =_9 \\
\texttt{unbox;(box;[}\mathbf{c}\texttt{])} & = Ass \\
\texttt{(unbox;box);[}\mathbf{c}\texttt{]} & \Rightarrow_\psi \\
\iota\texttt{;[}\mathbf{c}\texttt{]} & =_2 \\
\texttt{[}\mathbf{c}\texttt{]} &
\end{array}$$

If we want to find a rewrite system that implements $\phi\psi$-reduction, that is a rewrite system that produces equivalence class representatives of $\phi\psi$-normal form, we can try to find an orientation of the equational system in Figure 6. All the equations of the system have been arranged such that, if we also regard these as left to right rewrite rules then they either simplify the structure of the coercions or move composition inwards such that the ϕ and ψ rules may be applied. Once we have determined an orientation of the equational system, we can attempt to prove strong normalization and confluence. Strong normalization is straightforward for the chosen orientation. We use the fact that the total number of coercions constructors decreases in all rules except rule 3, where the number increases, but in rule 3 the number of function coercion constructors decreases and no other rule increases this number. So reduction must terminate. Once we have strong normalization we can show confluence by showing local (weak)

[1]See Appendix A for a definition of R-reduction modulo E-equality.

confluence (by Newman's Lemma, see [Klo87] for a good presentation of term rewriting). This is done by finding all critical pairs and for these show that they have common reducts. If we do this we find that the term

$$\texttt{unbox};\texttt{box};\,\texttt{[}\mathbf{c}\texttt{]}$$

leads to the critical pair (using rules 9 and ψ)

$$\texttt{unbox};\mathbf{c};\texttt{box}$$
$$\iota;\,\texttt{[}\mathbf{c}\texttt{]}$$

The second reduces to [**c**] by rule 1, but there seems to be no way that one may find a common reduct for the pair if **c** is a proper coercion. We may then try to complete the system by Knuth-Bendix completion, i.e. add one of the two rules:

$$\begin{aligned}\texttt{unbox};\mathbf{c};\texttt{box} &\Rightarrow \texttt{[}\mathbf{c}\texttt{]}\\ \texttt{[}\mathbf{c}\texttt{]} &\Rightarrow \texttt{unbox};\mathbf{c};\texttt{box}\end{aligned}$$

to the system. The first of these will do just fine, since it fits well with our proof of strong normalization. In fact this also holds for the second one, since then number of [_]-constructors decrease in the rule. But as can be seen from Example 3.2 only the first rule implements $\phi\psi$-reduction, so we add this rule to our reduction system. If we check all new critical pairs introduced by adding this new rule we find that they all have common reducts. The resulting rewrite system is shown in Figure 8, and the proofs are given in Lemma 3.9 and Theorem 3.10. The rewrite system contains one equality rule: associativity of coercion composition. This means that coercion reduction is in fact reduction by the named rules of Figure 8 modulo associativity (See Curien and Ghelli [CG90] for orienting and completing associativity).

Definition 3.4 *(Coercion reduction)*
Consider the coercion reduction rules in Figure 8 and call these rules R. We say that a coercion **c** R-reduces to $\mathbf{c}'$ and write write $\mathbf{c} \Rightarrow_R^* \mathbf{c}'$ if **c** reduces to $\mathbf{c}'$ by these rules.

The first result that we will prove about R-reduction is weak (local) confluence (Lemma 3.9). Before we do this we prove a few auxiliary lemmas (Lemmas 3.5, 3.6, 3.7, and 3.8) :

Lemma 3.5 *Given a coercion* **c** *with signature* $\pi \leadsto \upsilon$ *for some types* π *and* υ*then there exists a coercion* **d** *such that* **c***;*box $\Rightarrow^*$ **d** *and* [box*;*c] $\Rightarrow^*$ **d***.*

Proof:
We prove this by induction on the structure of **c**. Since the type signature of **c** is $\pi \leadsto \upsilon$ **c** must have one of the forms: (1) unbox, (2) unbox;$\mathbf{c}'$ or (3) [$\mathbf{c}'$];$\mathbf{c}''$. The lemma is easily verified for the first two cases. For the third case $\mathbf{c}''$ must have the signature $\pi' \leadsto \upsilon'$ for some types π' and υ' and and we have by induction that there exist a coercion $\mathbf{d}'$ such that $\mathbf{c}''$;box $\Rightarrow^*$ $\mathbf{d}'$ and [box;$\mathbf{c}''$] $\Rightarrow^*$ $\mathbf{d}'$. From this we see that either $\mathbf{d}' \equiv \iota$ or $\mathbf{d}' \equiv$ [$\mathbf{d}''$]. In the case of $\mathbf{d}' \equiv \iota$ we have:

c;box $\equiv$[$\mathbf{c}'$];$\mathbf{c}''$;box $\Rightarrow^*$[$\mathbf{c}'$]

and

$$
\begin{array}{rcll}
(\mathbf{c}\,;\mathbf{c}')\,;\mathbf{c}'' & = & \mathbf{c}\,;(\mathbf{c}'\,;\mathbf{c}'') & \\
\mathbf{c}\,;\iota & \Rightarrow & \mathbf{c} & (C1)\\
\iota\,;\mathbf{c} & \Rightarrow & \mathbf{c} & (C2)\\
\iota\rightarrow\iota & \Rightarrow & \iota & (C3)\\
(\mathbf{c}\rightarrow\mathbf{d})\,;(\mathbf{c}'\rightarrow\mathbf{d}') & \Rightarrow & (\mathbf{c}'\,;\mathbf{c})\rightarrow(\mathbf{d}\,;\mathbf{d}') & (C4)\\
\forall\alpha\,.\,\iota & \Rightarrow & \iota & (C5)\\
(\forall\alpha\,.\,\mathbf{c}_1)\,;(\forall\alpha\,.\,\mathbf{c}_2) & \Rightarrow & \forall\alpha\,.\,(\mathbf{c}_1\,;\mathbf{c}_2) & (C6)\\
\texttt{[}\iota\texttt{]} & \Rightarrow & \iota & (C7)\\
\texttt{[}\mathbf{c}\texttt{]}\,;\texttt{[}\mathbf{c}'\texttt{]} & \Rightarrow & \texttt{[}\mathbf{c}\,;\mathbf{c}'\texttt{]} & (C8)\\
\texttt{box}\,;\texttt{[}\mathbf{c}\texttt{]} & \Rightarrow & \mathbf{c}\,;\texttt{box} & (C9)\\
\texttt{[}\mathbf{c}\texttt{]}\,;\texttt{unbox} & \Rightarrow & \texttt{unbox}\,;\mathbf{c} & (C10)\\
\\
\texttt{box}\,;\texttt{unbox} & \Rightarrow & \iota & (\phi^{\rightarrow})\\
\texttt{unbox}\,;\texttt{box} & \Rightarrow & \iota & (\psi^{\rightarrow})\\
\texttt{unbox}\,;\mathbf{c}\,;\texttt{box} & \Rightarrow & \texttt{[}\mathbf{c}\texttt{]} & (\psi'^{\rightarrow})
\end{array}
$$

Figure 8: Coercion reduction (R)

$$\texttt{[box;}\mathbf{c}\texttt{]} \equiv \texttt{[box;[}\mathbf{c}'\texttt{];}\mathbf{c}''\texttt{]} \Rightarrow \texttt{[}\mathbf{c}'\texttt{;box;}\mathbf{c}''\texttt{]} \Rightarrow^* \texttt{[}\mathbf{c}'\texttt{;}\iota\texttt{]} \Rightarrow \texttt{[}\mathbf{c}'\texttt{]}$$

since the only way $\texttt{[box;}\mathbf{c}''\texttt{]} \Rightarrow^* \iota$ is if $\texttt{box;}\mathbf{c}'' \Rightarrow^* \iota$.
In the case of $\mathbf{d}' \equiv \texttt{[}\mathbf{d}''\texttt{]}$ we have:

$$\mathbf{c}\texttt{;box} \equiv \texttt{[}\mathbf{c}'\texttt{];}\mathbf{c}''\texttt{;box} \Rightarrow^* \texttt{[}\mathbf{c}'\texttt{];[}\mathbf{d}''\texttt{]} \Rightarrow \texttt{[}\mathbf{c}'\texttt{;}\mathbf{d}''\texttt{]}$$

and

$$\texttt{[box;}\mathbf{c}\texttt{]} \equiv \texttt{[box;[}\mathbf{c}'\texttt{];}\mathbf{c}''\texttt{]} \Rightarrow \texttt{[}\mathbf{c}'\texttt{;box;}\mathbf{c}''\texttt{]} \Rightarrow^* \texttt{[}\mathbf{c}'\texttt{;}\mathbf{d}''\texttt{]}$$

since the only way $\texttt{[box;}\mathbf{c}''\texttt{]} \Rightarrow^* \texttt{[}\mathbf{d}''\texttt{]}$ is if $\texttt{box;}\mathbf{c}'' \Rightarrow^* \mathbf{d}''$. ■

Lemma 3.6 *Given a coercion* $\mathbf{c}$ *with signature* $v \rightsquigarrow \pi$ *for some types* π *and* v*then there exists a coercion* $\mathbf{d}$ *such that* $\texttt{unbox;}\mathbf{c} \Rightarrow^* \mathbf{d}$ *and* $\texttt{[}\mathbf{c}\texttt{;unbox]} \Rightarrow^* \mathbf{d}$.

Proof:
By induction on the structure of $\mathbf{c}$. Similar to Lemma 3.5. ■

Lemma 3.7 *Given a coercions* $\mathbf{c}$ *and* $\mathbf{c}'$ *then there exists a coercion* $\mathbf{d}$ *such that* $\texttt{[}\mathbf{c}\texttt{;box;}\mathbf{c}'\texttt{]} \Rightarrow^* \mathbf{d}$ *and* $\texttt{[}\mathbf{c}\texttt{];}\mathbf{c}'\texttt{;box} \Rightarrow^* \mathbf{d}$.

Proof:
From Lemma 3.5 we know that there exist a coercion $\mathbf{d}'$ such that $\mathbf{c}'\texttt{;box} \Rightarrow^* \mathbf{d}'$ and $\texttt{[box;}\mathbf{c}'\texttt{]} \Rightarrow^* \mathbf{d}'$. We now have two cases either $\mathbf{d}' \equiv \iota$ or $\mathbf{d}' \equiv \texttt{[}\mathbf{d}''\texttt{]}$ for some $\mathbf{d}''$. In case $\mathbf{d}' \equiv \iota$ we must have $\texttt{box;}\mathbf{c}' \Rightarrow^* \iota$ and therefore:

$\mathtt{[}\mathbf{c};\mathtt{box};\mathbf{c}'\mathtt{]} \Rightarrow^* \mathtt{[}\mathbf{c};\iota\mathtt{]} \Rightarrow \mathtt{[}\mathbf{c}\mathtt{]}$

and

$\mathtt{[}\mathbf{c}\mathtt{]};\mathbf{c}';\mathtt{box} \Rightarrow^* \mathtt{[}\mathbf{c}\mathtt{]};\iota \Rightarrow \mathtt{[}\mathbf{c}\mathtt{]}$

In the case where $\mathbf{d}' \equiv \mathtt{[}\mathbf{d}''\mathtt{]}$ we must have $\mathtt{box};\mathbf{c}' \Rightarrow^* \mathbf{d}''$ and therefore:

$\mathtt{[}\mathbf{c};\mathtt{box};\mathbf{c}'\mathtt{]} \Rightarrow^* \mathtt{[}\mathbf{c};\mathbf{d}''\mathtt{]}$

and

$\mathtt{[}\mathbf{c}\mathtt{]};\mathbf{c}';\mathtt{box} \Rightarrow^* \mathtt{[}\mathbf{c}\mathtt{]};\mathtt{[}\mathbf{d}''\mathtt{]} \Rightarrow \mathtt{[}\mathbf{c};\mathbf{d}''\mathtt{]}$

■

Lemma 3.8 *Given a coercions* $\mathbf{c}$ *and* $\mathbf{c}'$ *then there exists a coercion* $\mathbf{d}$ *such that* $\mathtt{[}\mathbf{c};\mathtt{unbox};\mathbf{c}'\mathtt{]} \Rightarrow^* \mathbf{d}$ *and* $\mathtt{unbox};\mathbf{c};\mathtt{[}\mathbf{c}'\mathtt{]} \Rightarrow^* \mathbf{d}$.

Proof:
Similar to Lemma 3.7. ■

Now that we have Lemmas 3.5, 3.6, 3.7, and 3.8 in place we can prove weak (local) confluence of coercion reduction:

Lemma 3.9 *(Weak (local) confluence of coercion reduction). The coercion reduction system in Figure 8 is weakly confluent.*

Proof:
In general a coercion will have the form: $\mathbf{c}_1;\ldots;\mathbf{c}_n$. We look for overlapping redexes in such terms that may lead to critical pairs. The way the proof is structured is by listing the rules that overlap, the term and show how the critical pair is resolved. We start by finding overlaps between Rule $C1$ and the rest of the rules from $C1$ to ψ', we the look at overlaps between Rule $C2$ and the rules from $C2$ to ψ', and so on up to overlaps between Rule ψ' and itself. Only the rules which may lead to critical pairs are listed.

Rules: $C1$ and ψ'. Term: $\mathtt{unbox};\iota;\mathtt{box}$. Critical pair: $\mathtt{unbox};\mathtt{box}$ and $\mathtt{[}\iota\mathtt{]}$. (1) $\mathtt{unbox};\mathtt{box} \Rightarrow_{\psi} \iota$. (2) $\mathtt{[}\iota\mathtt{]} \Rightarrow_{C7} \iota$.

Rules: $C2$ and ψ'. Term: $\mathtt{unbox};\iota;\mathtt{box}$. Critical pair: $\mathtt{unbox};\mathtt{box}$ and $\mathtt{[}\iota\mathtt{]}$. As above.

Rules: $C3$ and $C4$. Term $\iota\rightarrowtail\iota;\mathbf{c}\rightarrowtail\mathbf{d}$. Critical pair: $\iota;\mathbf{c}\rightarrowtail\mathbf{d}$ and $(\mathbf{c};\iota)\rightarrowtail(\iota;\mathbf{d})$. (1) $\iota;\mathbf{c}\rightarrowtail\mathbf{d} \Rightarrow_{C2} \mathbf{c}\rightarrowtail\mathbf{d}$. (2) $(\mathbf{c};\iota)\rightarrowtail(\iota;\mathbf{d}) \Rightarrow_{C2} (\mathbf{c};\iota)\rightarrowtail\mathbf{d} \Rightarrow_{C1} \mathbf{c}\rightarrowtail\mathbf{d}$.

Rules: $C3$ and $C4$. Term $\mathbf{c}\rightarrowtail\mathbf{d};\iota\rightarrowtail\iota$. Critical pair: $\mathbf{c}\rightarrowtail\mathbf{d};\iota$ and $(\iota;\mathbf{c})\rightarrowtail(\mathbf{d};\iota)$. Proof similar to other case for $C3$ and $C4$.

Rules: $C4$ and $C4$. Term $\mathbf{c}_1\rightarrowtail\mathbf{d}_1;\mathbf{c}_2\rightarrowtail\mathbf{d}_2;\mathbf{c}_3\rightarrowtail\mathbf{d}_3$. Critical pair: $\mathbf{c}_2;\mathbf{c}_1\rightarrowtail\mathbf{d}_1;\mathbf{d}_2;\mathbf{c}_3\rightarrowtail\mathbf{d}_3$ and $\mathbf{c}_1\rightarrowtail\mathbf{d}_1;\mathbf{c}_3;\mathbf{c}_2\rightarrowtail\mathbf{d}_2;\mathbf{d}_3$. (1) $\mathbf{c}_2;\mathbf{c}_1\rightarrowtail\mathbf{d}_1;\mathbf{d}_2;\mathbf{c}_3\rightarrowtail\mathbf{d}_3 \Rightarrow_{C4} \mathbf{c}_3;\mathbf{c}_2;\mathbf{c}_1\rightarrowtail\mathbf{d}_1;\mathbf{d}_2;\mathbf{d}_3$. (2) $\mathbf{c}_1\rightarrowtail\mathbf{d}_1;\mathbf{c}_3;\mathbf{c}_2\rightarrowtail\mathbf{d}_2;\mathbf{d}_3 \Rightarrow_{C4} \mathbf{c}_3;\mathbf{c}_2;\mathbf{c}_1\rightarrowtail\mathbf{d}_1;\mathbf{d}_2;\mathbf{d}_3$.

Rules: $C5$ and $C6$. Term $(\forall\alpha.\mathbf{c});(\forall\alpha.\iota)$. Critical pair: $(\forall\alpha.\mathbf{c});\iota$ and $\forall\alpha.(\mathbf{c};\iota)$. (1) $(\forall\alpha.\mathbf{c});\iota \Rightarrow_{C1} \forall\alpha.\mathbf{c}$. (2) $\forall\alpha.(\mathbf{c};\iota) \Rightarrow_{C1} \forall\alpha.\mathbf{c}$.

Rules: $C5$ and $C6$. Term $(\forall\alpha.\iota);(\forall\alpha.\mathbf{c})$. Critical pair: $\iota;(\forall\alpha.\mathbf{c})$ and $\forall\alpha.(\iota;\mathbf{c})$. Proof similar to other case for $C5$ and $C6$.

Rules: $C6$ and $C6$. Term $(\forall\alpha.\mathbf{c}_1);(\forall\alpha.\mathbf{c}_2);(\forall\alpha.\mathbf{c}_3)$. Critical pair: $(\forall\alpha.\mathbf{c}_1;\mathbf{c}_2);(\forall\alpha.\mathbf{c}_3)$ and $(\forall\alpha.\mathbf{c}_1);(\forall\alpha.\mathbf{c}_2;\mathbf{c}_3)$. Proof similar to the case $C4$ and $C4$.

Rules: $C7$ and $C8$. Term: $[\iota];[\mathbf{c}]$. Critical pair: $\iota;[\mathbf{c}]$ and $[\iota;\mathbf{c}]$. (1) $\iota;[\mathbf{c}] \Rightarrow_{C2} [\mathbf{c}]$. (2) $[\iota;\mathbf{c}] \Rightarrow_{C2} [\mathbf{c}]$.

Rules: $C7$ and $C9$. Term: $\mathtt{box};[\iota]$. Critical pair: $\mathtt{box};\iota$ and $\iota;\mathtt{box}$. (1) $\mathtt{box};\iota \Rightarrow_{C1} \mathtt{box}$. (2) $\iota;\mathtt{box} \Rightarrow_{C2} \mathtt{box}$.

Rules: $C7$ and $C10$. Term: $[\iota];\mathtt{unbox}$. Critical pair: $\iota;\mathtt{unbox}$ and $\mathtt{unbox};\iota$. (1) $\iota;\mathtt{unbox} \Rightarrow_{C2} \mathtt{unbox}$. (2) $\mathtt{unbox};\iota \Rightarrow_{C1} \mathtt{unbox}$.

Rules: $C8$ and $C8$. Term: $[\mathbf{c}_1];[\mathbf{c}_2];[\mathbf{c}_3]$. Critical pair: $[\mathbf{c}_1;\mathbf{c}_2];[\mathbf{c}_3]$ and $[\mathbf{c}_1];[\mathbf{c}_2;\mathbf{c}_3]$. Similar to the proof of case $C4$ and $C4$.

Rules: $C8$ and $C9$. Term: $\mathtt{box};[\mathbf{c}];[\mathbf{d}]$. Critical pair: $\mathtt{box};[\mathbf{c};\mathbf{d}]$ and $\mathbf{c};\mathtt{box};[\mathbf{d}]$. (1) $\mathtt{box};[\mathbf{c};\mathbf{d}] \Rightarrow_{C9} \mathbf{c};\mathbf{d};\mathtt{box}$. (2) $\mathbf{c};\mathtt{box};[\mathbf{d}] \Rightarrow_{C9} \mathbf{c};\mathbf{d};\mathtt{box}$.

Rules: $C8$ and $C10$. Term: $[\mathbf{c}];[\mathbf{d}];\mathtt{unbox}$. Critical pair: $[\mathbf{c};\mathbf{d}];\mathtt{unbox}$ and $[\mathbf{c}];\mathtt{unbox};\mathbf{d}$. (1) $[\mathbf{c};\mathbf{d}];\mathtt{unbox} \Rightarrow_{C10} \mathtt{unbox};\mathbf{c};\mathbf{d}$. (2) $[\mathbf{c}];\mathtt{unbox};\mathbf{d} \Rightarrow_{C10} \mathtt{unbox};\mathbf{c};\mathbf{d}$.

Rules: $C9$ and $C10$. Term: $\mathtt{box};[\mathbf{c}];\mathtt{unbox}$. Critical pair: $\mathbf{c};\mathtt{box};\mathtt{unbox}$ and $\mathtt{box};\mathtt{unbox};\mathbf{c}$. (1) $\mathbf{c};\mathtt{box};\mathtt{unbox} \Rightarrow_{\phi} \mathbf{c};\iota \Rightarrow_{C1} \mathbf{c}$. (2) $\mathtt{box};\mathtt{unbox};\mathbf{c} \Rightarrow_{\phi} \iota;\mathbf{c} \Rightarrow_{C2} \mathbf{c}$.

Rules: $C9$ and ψ. Term: $\mathtt{unbox};\mathtt{box};[\mathbf{c}]$. Critical pair: $\mathtt{unbox};\mathbf{c};\mathtt{box}$ and $\iota;[\mathbf{c}]$. (1) $\mathtt{unbox};\mathbf{c};\mathtt{box} \Rightarrow_{\psi'} [\mathbf{c}]$. (2) $\iota;[\mathbf{c}] \Rightarrow_{C2} [\mathbf{c}]$.

Rules: $C9$ and ψ'. Term: $\mathtt{unbox};\mathbf{c};\mathtt{box};[\mathbf{d}]$. Critical pair: $\mathtt{unbox};\mathbf{c};\mathbf{d};\mathtt{box}$ and $[\mathbf{c}];[\mathbf{d}]$. (1) $\mathtt{unbox};\mathbf{c};\mathbf{d};\mathtt{box} \Rightarrow_{\psi'} [\mathbf{c};\mathbf{d}]$. (2) $[\mathbf{c}];[\mathbf{d}] \Rightarrow_{C8} [\mathbf{c};\mathbf{d}]$.

Rules: $C10$ and ψ. Term: $[\mathbf{c}];\mathtt{unbox};\mathtt{box}$. Critical pair: $\mathtt{unbox};\mathbf{c};\mathtt{box}$ and $[\mathbf{c}];\iota$. (1) $\mathtt{unbox};\mathbf{c};\mathtt{box} \Rightarrow_{\psi'} [\mathbf{c}]$. (2) $[\mathbf{c}];\iota \Rightarrow_{C1} [\mathbf{c}]$.

Rules: $C10$ and ψ'. Term: $[\mathbf{c}];\mathtt{unbox};\mathbf{d};\mathtt{box}$. Critical pair: $\mathtt{unbox};\mathbf{c};\mathbf{d};\mathtt{box}$ and $[\mathbf{c}];[\mathbf{d}]$. (1) $\mathtt{unbox};\mathbf{c};\mathbf{d};\mathtt{box} \Rightarrow_{\psi'} [\mathbf{c};\mathbf{d}]$. (2) $[\mathbf{c}];[\mathbf{d}] \Rightarrow_{C8} [\mathbf{c};\mathbf{d}]$.

Rules: ϕ and ψ. Term: $\mathtt{box};\mathtt{unbox};\mathtt{box}$. Critical pair: $\iota;\mathtt{box}$ and $\mathtt{box};\iota$. (1) $\iota;\mathtt{box} \Rightarrow_{C2} \mathtt{box}$. (2) $\mathtt{box};\iota \Rightarrow_{C1} \mathtt{box}$.

Rules: ϕ and ψ'. Term: $\mathtt{box};\mathtt{unbox};\mathbf{c};\mathtt{box}$. Critical pair: $\iota;\mathbf{c};\mathtt{box}$ and $\mathtt{box};[\mathbf{c}]$. (1) $\iota;\mathbf{c};\mathtt{box} \Rightarrow_{C2} \mathbf{c};\mathtt{box}$. (2) $\mathtt{box};[\mathbf{c}] \Rightarrow_{C9} \mathbf{c};\mathtt{box}$.

Rules: ϕ and ψ'. Term: $\mathtt{unbox};\mathbf{c};\mathtt{box};\mathtt{unbox}$. Critical pair: $\mathtt{unbox};\mathbf{c};\iota$ and $[\mathbf{c}];\mathtt{unbox}$. (1) $\mathtt{unbox};\mathbf{c};\iota \Rightarrow_{C1} \mathtt{unbox};\mathbf{c}$. (2) $[\mathbf{c}];\mathtt{unbox} \Rightarrow_{C10} \mathtt{unbox};\mathbf{c}$.

Rules: ψ and ψ'. Term: $\mathtt{unbox};\mathtt{box};\mathbf{c};\mathtt{box}$. Critical pair: $\iota;\mathbf{c};\mathtt{box}$ and $[\mathtt{box};\mathbf{c}]$. Resolved by Lemma 3.5.

Rules: ψ and ψ'. Term: $\mathtt{unbox};\mathbf{c};\mathtt{unbox};\mathtt{box}$. Critical pair: $\mathtt{unbox};\mathbf{c};\iota$ and $[\mathbf{c};\mathtt{unbox}]$. Resolved by Lemma 3.6.

Rules: ψ' and ψ'. Term: $\mathtt{unbox};\mathbf{c};\mathtt{box};\mathbf{c}';\mathtt{box}$. Critical pair: $[\mathbf{c};\mathtt{box};\mathbf{c}']$ and $[\mathbf{c}];\mathbf{c}';\mathtt{box}$. Resolved by Lemma 3.7.

Rules: ψ' and ψ'. Term: unbox;**c**;unbox;**c**$'$;box. Critical pair: [**c**;unbox;**c**$'$] and unbox;**c**;[**c**$'$]. Resolved by Lemma 3.8.

■

With this result we can prove the following important theorem about coercion reduction:

Theorem 3.10 *The coercion reduction system in Figure 8 has the following properties.*

1. *If* $\mathbf{c} \Rightarrow_R^* \mathbf{c}'$ *then* $\vdash \mathbf{c} \ \Rightarrow_{\phi\psi}^* \mathbf{c}'$.
2. *It is canonical, i.e. strongly normalizing and confluent.*
3. *If* **c** *is a normal form then it has one of the following forms:*
   (a) $\mathbf{c} \equiv \iota$, *or*
   (b) $\mathbf{c} \equiv$ box, *or*
   (c) $\mathbf{c} \equiv$ unbox, *or*
   (d) $\mathbf{c} \equiv \mathbf{c}' \to \mathbf{c}''$, $\mathbf{c} \equiv (\mathbf{c}' \to \mathbf{c}'')$;box, *or* $\mathbf{c} \equiv$ unbox;$(\mathbf{c}' \to \mathbf{c}'')$ *where* $\mathbf{c}'$ *and* $\mathbf{c}''$ *are normal forms of which at least one is proper, or*
   (e) $\mathbf{c} \equiv \forall\alpha.\mathbf{c}'$, $\mathbf{c} \equiv (\forall\alpha.\mathbf{c}')$;box, *or* $\mathbf{c} \equiv$ unbox;$(\forall\alpha.\mathbf{c}')$ *where* $\mathbf{c}'$ *is a normal form and proper, or*
   (f) $\mathbf{c} \equiv$ [$\mathbf{c}'$] *where* $\mathbf{c}'$ *is a normal form and proper.*

Proof:
(1) Since every rule of Figure 8 except $\psi'^{\to}$ corresponds to either a ϕ/ψ-reduction step or an equality from Figure 6 and since we showed in Example 3.2 that unbox;(**c**;box) $\phi\psi$-reduces to [**c**] a reduction sequence $\mathbf{c} \Rightarrow_R^* \mathbf{c}'$ must correspond to a reduction sequence $\vdash \mathbf{c} \ \Rightarrow_{\phi\psi}^* \mathbf{c}'$.

(2) To show that the system is canonical we have to show that it is confluent and strongly normalizing. If a reduction system is strongly normalizing then according to Newman's Lemma it is sufficient to show local confluence of the system. Since this is shown in Lemma 3.9 all we have to show is strong normalization of R: The total number of coercions constructors decrease in all rules except C4. This means that reduction can only fail to terminate if rules C4 can be applied infinitely many times, but this cannot be the case since C4 decreases the number of function coercion constructors and no other rule increases this number. Formally we have that the lexicographical ordering on $\mathcal{Z} \times \mathcal{Z}$ where $\mathcal{Z}$ is the non-negative integers is a well-founded ordering. So the measure (C,I) on coercions, where C is the number of ; in a coercion and I the number of induced coercion constructors, is a well-founded measure on coercions and the strict partial ordering defined by coercion reduction. This ensures that coercions reduction terminates.

(3) It is easy to verify that the coercions listed are normal forms by showing that no reduction rule is applicable to these.

We prove that any normal form **c** must have one of the forms listed. We show this by induction on the structure of the normal form coercion **c**:

Base cases: $\mathbf{c} \equiv \iota$, $\mathbf{c} \equiv$ `box` or $\mathbf{c} \equiv$ `unbox`: Trivial.

Inductive case: $\mathbf{c} \equiv \mathbf{d}\to\mathbf{d}'$: By induction $\mathbf{d}$ and $\mathbf{d}'$ must have one of the forms listed and $\mathbf{c}$ therefore also have one of the forms listed (d first form).

Inductive case: $\mathbf{c} \equiv \forall\alpha.\mathbf{d}$: By induction $\mathbf{d}$ must have one of the forms listed and $\mathbf{c}$ therefore also have one of the forms listed (e first form).

Inductive case: $\mathbf{c} \equiv$ `[`$\mathbf{d}$`]`: By induction $\mathbf{d}$ must have one of the forms listed and $\mathbf{c}$ therefore also have one of the forms listed (f).

Inductive case: $\mathbf{c} \equiv \mathbf{d};\mathbf{d}'$: By induction $\mathbf{d}$ and $\mathbf{d}'$ must have one of the forms listed. The following table covers all the possible combination of $\mathbf{d}$ and $\mathbf{d}'$ and explains for each combination why it is either not possible (nnf = not in normal form, nwf = not well formed according to Figure 4) or what form it corresponds to, e.g d.3 = the third of the forms listed under d in the theorem:

$\mathbf{d}\backslash\mathbf{d}'$	a	b	c	d.1	d.2	d.3	e.1	e.2	e.3	f
a	nnf	nnf	nnf	nnf	nnf	nnf	nnf	nnf	nnf	nnf
b	nnf	nwf	nnf	nwf	nwf	nnf	nwf	nwf	nnf	nnf
c	nnf	nnf	nwf	d.3	nnf	nwf	e.3	nnf	nwf	nwf
d.1	nnf	d.2	nwf	nnf	nnf	nwf	nwf	nwf	nwf	nwf
d.2	nnf	nwf	nnf	nwf	nwf	nnf	nwf	nwf	nwf	nnf
d.3	nnf	nnf	nwf	nnf	nnf	nwf	nwf	nwf	nwf	nwf
e.1	nnf	e.2	nwf	nwf	nwf	nwf	nnf	nnf	nwf	nwf
e.2	nnf	nwf	nnf	nwf	nwf	nwf	nwf	nwf	nnf	nnf
e.3	nnf	nnf	nwf	nwf	nwf	nwf	nnf	nnf	nwf	nwf
f	nnf	nwf	nnf	nwf	nwf	nnf	nwf	nwf	nnf	nnf

■

Theorem 3.10 guarantees that R-normal form coercions exist. Furthermore, from part 3 of the theorem one can see that these normal forms are unique for a given signature. This gives us the following lemma:

Lemma 3.11 *For all ρ, ρ' with $|\rho| = |\rho'|$ there exists a unique R-normal form coercion $\vdash \mathbf{c} : \rho \leadsto \rho'$.*

Proof:

All the forms listed in Theorem 3.10 part 3 are non overlapping and by listing the signatures of all the forms one can see that these are all different from each other. This means that a normal form for a given signature can only match one of the forms and must therefore be unique.

■

Lemma 3.12 *R-normal forms are $\phi\psi$-normal forms.*

Proof:

Given an R-normal form $\mathbf{c}$ and assume that this is not in $\phi\psi$-normal form. Then there exist a coercion $\mathbf{c}'$ such that $\vdash \mathbf{c} \Rightarrow^*_{\phi\psi} \mathbf{c}'$ and $\vdash \mathbf{c} \neq \mathbf{c}'$. But since $\mathbf{c}'$ has the same signature as $\mathbf{c}$ then by Lemma 3.11 we have $\mathbf{c}' \Rightarrow^*_R \mathbf{c}$ and by Theorem 3.10 part 1 that $\vdash \mathbf{c}' \Rightarrow^*_{\phi\psi} \mathbf{c}$ but since $\phi\psi$-reduction cannot increase the number of `box` and `unbox` coercions no ϕ or ψ step can have been involved in $\vdash \mathbf{c} \Rightarrow^*_{\phi\psi} \mathbf{c}'$ only core equation steps. This contradicts the assumption that $\vdash \mathbf{c} \neq \mathbf{c}'$ and $\mathbf{c}$ must therefore be in $\phi\psi$-normal form.

■

Lemma 3.13 *$\phi\psi$-reduction on coercions is canonical; that is, it is strongly normalizing and confluent.*

Proof:
Strong normalization: $\phi\psi$-reduction is reduction on coercions equivalence classes so a reduction step must include one ϕ or ψ reduction step. Since these rules decrease the number of `box`- and `unbox`-coercions reduction must terminate.

Confluence: This follows from the confluence of R-reduction, since if $\mathbf{c}$ have two different reducts then we can always find a common reduct for these by R-reduction and therefore also by $\phi\psi$-reduction (Theorem 3.10 part 1).

■

3.1.3 Coercion coherence

If all congruent coercions are to be semantically equivalent (this is our basic assumption) it is important to know whether our axiomatization of coercions congruence equates exactly the congruent coercions. If this is the case we may replace a coercion by any coercion that we can prove equal to it and be sure that this replacement is semantically safe. Furthermore, if we for a given fixed semantics want to prove that all congruent coercions are equivalent with respect to the semantics, it is enough to prove all the axioms sound with respect to the semantics. The property that all congruent coercions are provably equal is called *coherence* and we give a proof of this:

Theorem 3.14 *(Coherence of coercions) Coercions $\mathbf{c}$ and $\mathbf{c}'$ are congruent if and only if they are $\phi\psi$-equal; i.e., $\mathbf{c} \cong \mathbf{c}'$ iff $\phi\psi \vdash \mathbf{c} = \mathbf{c}'$*

Proof:
"if": This follows trivially from the definition of $\phi\psi \vdash \mathbf{c} = \mathbf{c}'$.

"only if": From Lemma 3.11 it follows that for all ρ, ρ' with $|\rho| = |\rho'|$ there exists a unique R-normal form coercion $\vdash \mathbf{c}_N : \rho \leadsto \rho'$, and that both $\mathbf{c}$ and $\mathbf{c}'$ will R-reduce to this unique coercion. This means that also both $\mathbf{c}$ and $\mathbf{c}'$ will $\phi\psi$-reduce to this unique coercion. Since for any coercion $\mathbf{d}$, if $\vdash \mathbf{d} \Rightarrow^*_{\phi\psi} \mathbf{d}'$ then $\phi\psi \vdash \mathbf{d} = \mathbf{d}'$ it must follow that $\phi\psi \vdash \mathbf{c} = \mathbf{c}_N$ and $\phi\psi \vdash \mathbf{c}' = \mathbf{c}_N$ and therefore by symmetry and transitivity of $=$ we have that $\phi\psi \vdash \mathbf{c} = \mathbf{c}'$.

■

Definition 3.15 *(Inverse coercions)*

Let $\mathbf{c}$ be a coercion then we will call a coercion $\mathbf{d}$ an *inverse* coercion of $\mathbf{c}$ if $\phi\psi \vdash \mathbf{c}\,;\mathbf{d} = \iota$ and $\phi\psi \vdash \mathbf{d}\,;\mathbf{c} = \iota$.

Proposition 3.16 *All coercions have an inverse coercion.*

Proof:

Let $\vdash \mathbf{c} : \rho \leadsto \rho'$ be an arbitrary coercion then by Proposition 2.2 there exist a coercion $\vdash \mathbf{d} : \rho' \leadsto \rho$. Since $\vdash \mathbf{c}\,;\mathbf{d} : \rho \leadsto \rho$ by Theorem 3.14 we must have $\phi\psi \vdash \mathbf{c}\,;\mathbf{d} = \iota$. Similarly we can prove that $\phi\psi \vdash \mathbf{d}\,;\mathbf{c} = \iota$. This shows that $\mathbf{d}$ must be an inverse coercion of $\mathbf{c}$.

■

From the proof of Proposition 3.16 we see that for a given coercion $\vdash \mathbf{c} : \rho \leadsto \rho'$ any coercion $\mathbf{d}$ with signature $\rho' \leadsto \rho$ must be an inverse coercion of $\mathbf{c}$, and that all coercions have infinitely many inverse coercions.

We will in the following use $\mathbf{c}^{-1}$ to denote some arbitrarily chosen inverse coercion of $\mathbf{c}$ if the choice of this is not important.

3.1.4 Optimal coercions

As discussed in Subsection 3.1.2 on coercion reduction if we accept that boxing and unboxing coercions are more expensive than the identity coercion, then $\phi\psi$-reduction will in general improve performance of coercions (make them less expensive) and a normal form coercion will be better than all the coercions that reduce to it. We therefore introduce the following formal definition:

Definition 3.17 *(Formally optimal coercions)*

A coercion $\mathbf{c}$ is *(formally) optimal* if all congruent coercions $\mathbf{c}' \cong \mathbf{c}$ can be reduced to $\mathbf{c}$ by $\phi\psi$-reduction.

From the proof of Lemma 3.11 we see, that given a signature we can easily construct a optimal coercion with that signature. This is a very useful since in a implementation one may which to represent (optimal) coercions simply by their signature. We will see how this is useful in Chapter 5 when we describe how to implement reduction of completion.

Clearly, every coercion equal to an optimal coercion is also optimal. We shall see that, for every type signature $\rho \leadsto \rho'$ with $|\rho| = |\rho'|$, then a (formally) optimal coercion $\mathbf{c}$:$\rho \leadsto \rho'$ exists and are unique modulo core coercion equality, i.e. Figure 6.

Theorem 3.18 *(Existence of formally optimal coercions) Given a type signature $\rho \leadsto \rho'$ with $|\rho| = |\rho'|$, then (formally) optimal coercions exist and are unique modulo core coercion equality.*

Proof:

Existence: By Proposition 2.2 there exist a coercion with signature $\rho \leadsto \rho'$. Let $\mathbf{c}$ be an arbitrary coercion with signature $\rho \leadsto \rho'$ by Lemma 3.11 there exist a unique R-normal form $\mathbf{c}_N$ to which $\mathbf{c}$ R-reduces and by Theorem 3.10 this means that $\mathbf{c}$ also $\phi\psi$-reduces to $\mathbf{c}_N$. By Theorem 3.14 $\mathbf{c}$ and $\mathbf{c}_N$ are congruent and $\mathbf{c}_N$ is therefore a formally optimal coercion with signature $\rho \leadsto \rho'$.

Uniqueness: Let $\mathbf{d}$ be another optimal coercion with signature $\rho \leadsto \rho'$. Since $\mathbf{d}$ is optimal then $\mathbf{c}_N$ $\phi\psi$-reduces to $\mathbf{d}$ and since $\mathbf{c}_N$ is also optimal $\mathbf{d}$ also $\phi\psi$-reduces to $\mathbf{d}$. This means that we have $\mathbf{d} \Rightarrow^*_{\phi\psi} \mathbf{c}_N \Rightarrow^*_{\phi\psi} \mathbf{d}$, but this can only hold if $\vdash \mathbf{d} = \mathbf{c}_N$ because $\phi\psi$-reduction is canonical (Lemma 3.13).

■

3.2 Coercion decomposition

Several results about coercion decomposition will prove useful later. In this section we define some properties of coercions and show some results about coercions with these properties and about coercion decomposition.

3.2.1 Prime factorization

The notion of a *prime* coercion will later prove to be useful. The notion of prime coercions were also used by Rehof in [Reh95]. A prime coercion is informally a coercion that only contains exactly one `box` or one `unbox` coercion. Let us first define what a prime coercion is:

Definition 3.19 *(Prime coercion)*
A coercion $\mathbf{c}$ is called *prime* iff

1. $\mathbf{c}$ is proper
2. if $\vdash \mathbf{c} = \mathbf{d};\mathbf{d}'$ then either
   (a) $\vdash \mathbf{d} = \mathbf{c}$ and $\vdash \mathbf{d}' = \iota$ or
   (b) $\vdash \mathbf{d} = \iota$ and $\vdash \mathbf{d}' = \mathbf{c}$.

It can be verified that the prime coercions can be generated by the following grammar:

$$\varpi \ ::= \ \texttt{box} | \texttt{unbox} | \varpi \rightarrow \iota | \iota \rightarrow \varpi | \forall \alpha . \varpi | \texttt{[}\varpi\texttt{]}$$

We have the following important fact about *prime factoring* of coercions:

Lemma 3.20 *(Prime factoring) Any proper coercion* $\mathbf{c}$ *can be factored into a composition of prime coercions* $\mathbf{c}_1,\ldots,\mathbf{c}_n$*, such that* $\vdash \mathbf{c} = \mathbf{c}_1;\ldots;\mathbf{c}_n$.

Proof:
The proof is by structural induction on the coercion $\mathbf{c}$.

Base cases: $\mathbf{c} \equiv$ `box`: Trivial.

Base cases: $\mathbf{c} \equiv$ `unbox`: Trivial.

Inductive case: $\mathbf{c} \equiv \mathbf{d} \rightarrow \mathbf{d}'$: By induction both $\mathbf{d}$ and $\mathbf{d}'$ can be prime factored if they are proper. Let n be the sum of the number of (prime) coercions that $\mathbf{d}$ and $\mathbf{d}'$ can be prime factored into. We factor $\mathbf{d}$ into $\mathbf{d}_1;\ldots;\mathbf{d}_n$ where the first m of these coercions are the prime factors of $\mathbf{d}$ and the rest of the coercion are identity coercions ι or if $\mathbf{d}$ is not proper then all of the coercions are identity coercions. We similarly factor $\mathbf{d}'$ into $\mathbf{d}_1{}'$;...;$\mathbf{d}_n{}'$ where the last $n - m$ coercions are the prime factors of $\mathbf{d}'$ and the rest of the coercion are identity coercions ι or if $\mathbf{d}'$ is not proper then all of the coercions are identity coercions. Since at least one of the coercions $\mathbf{d}$ and $\mathbf{d}'$ can be prime factored n must be greater than zero. Then we have:

$$\mathbf{c} =$$

$$\mathbf{d} \rightarrow \mathbf{d}' =$$

$$\mathbf{d}_1; ...; \mathbf{d}_n \rightarrow \mathbf{d}_1{}'; ...; \mathbf{d}_n{}' =$$

$$(\mathbf{d}_n \rightarrow \mathbf{d}_1{}'); (\mathbf{d}_1; ...; \mathbf{d}_{n-1} \rightarrow \mathbf{d}_2{}'; ...; \mathbf{d}_n{}') =$$

$$\ldots$$

$$(\mathbf{d}_n \rightarrow \mathbf{d}_1{}'); ...; (\mathbf{d}_1 \rightarrow \mathbf{d}_n{}')$$

Since for all the pairs of coercions $(\mathbf{d}_1,\mathbf{d}_1{}')$,...,$(\mathbf{d}_n,\mathbf{d}_n{}')$ one of the coercions are a prime coercions and the other the identity coercion all the coercions in the last composition must be prime coercions. This shows the case.

Inductive case: $\mathbf{c} \equiv [\mathbf{d}]$: By induction both $\mathbf{d}$ can be factored since $\mathbf{d}$ must be proper if $\mathbf{c}$ is proper. Assume that $\mathbf{d}$ is factored into $\mathbf{d}_1$;...;$\mathbf{d}_n$. We the have:

$$\mathbf{c} =$$

$$[\mathbf{d}] =$$

$$[\mathbf{d}_1; ...; \mathbf{d}_n] =$$

$$[\mathbf{d}_1]; [\mathbf{d}_2; ...; \mathbf{d}_n] =$$

$$\ldots$$

$$[\mathbf{d}_1]; ...; [\mathbf{d}_n] =$$

This proves the case since all the coercions $[\mathbf{d}_1]$,...,$[\mathbf{d}_n]$ must be prime coercions.

Inductive case: $\mathbf{c} \equiv \forall\alpha.\mathbf{d}$: Similar to the case for $\mathbf{c} \equiv [\mathbf{d}]$.

Inductive case: $\mathbf{c} \equiv \mathbf{d};\mathbf{d}'$: If both coercions $\mathbf{d}$ and $\mathbf{d}'$ are proper the prime factoring of $\mathbf{c}$ is simply the composition of the prime factorings of $\mathbf{d}$ and $\mathbf{d}'$. If one of the coercions $\mathbf{d}$ and $\mathbf{d}'$ is not proper then the prime factoring of $\mathbf{c}$ is simply the prime factoring of the other proper coercion.

■

3.2.2 Positive and negative coercions

In this subsection we introduces the notion of *coercion polarity* which will prove important when we in the next chapter (Chapter 4) show how to implement $\phi\psi$-reduction on completions.

Two kinds of coercions called *positive* and *negative coercions* exist and we say that these have positive and negative *polarity.*

Definition 3.21 *(Positive and negative coercions)*

A coercion $\mathbf{c}$ is *positive* if $\mathbf{c}{:}+$ is derivable from the rules in Figure 9 and *negative* if $\mathbf{c}{:}-$ is derivable.

By adding a superscript $+$ or $-$ to a coercion we indicate that a coercion is in fact positive or negative. (If a coercion occurs in, say a reduction rule, these annotations can be regarded as side conditions that have to hold before the rule may be applied. So in the equations of Figure 13 a superscript $+$ (or $-$) on a coercion means that the coercion has to be positive (or negative) for the rule to be applicable.) A coercion may be neither positive nor negative, e.g. $\mathtt{box}_\upsilon\,\mathtt{;}\mathtt{unbox}_\upsilon$. The only coercions that may be both positive and negative are the non proper coercions.

$$
\begin{array}{cccccc}
\iota_\tau : + & \mathtt{box}_\upsilon : + & \dfrac{\mathbf{c}{:}- \quad \mathbf{d}{:}+}{\mathbf{c}\!\to\!\mathbf{d}{:}+} & \dfrac{\mathbf{c}{:}+ \quad \mathbf{d}{:}+}{\mathbf{c}\,\mathtt{;}\,\mathbf{d}{:}+} & \dfrac{\mathbf{c}{:}+}{\mathtt{[}\mathbf{c}\mathtt{]}{:}+} & \dfrac{\mathbf{c}{:}+}{\forall\alpha\,\mathtt{.}\,\mathbf{c}{:}+} \\[2ex]
\iota_\tau : - & \mathtt{unbox}_\upsilon : - & \dfrac{\mathbf{c}{:}+ \quad \mathbf{d}{:}-}{\mathbf{c}\!\to\!\mathbf{d}{:}-} & \dfrac{\mathbf{c}{:}- \quad \mathbf{d}{:}-}{\mathbf{c}\,\mathtt{;}\,\mathbf{d}{:}-} & \dfrac{\mathbf{c}{:}-}{\mathtt{[}\mathbf{c}\mathtt{]}{:}-} & \dfrac{\mathbf{c}{:}-}{\forall\alpha\,\mathtt{.}\,\mathbf{c}{:}-}
\end{array}
$$

Figure 9: Positive and negative coercions

Lemma 3.22 *Coercion reduction preserves polarity; that is, if* $\mathbf{c}$ *is positive or negative and* $\mathbf{c} \Rightarrow_R^* \mathbf{c}'$*, then* $\mathbf{c}'$ *is also positive, respectively negative.*

Proof:

Easily seen by inspecting the rules of R.

■

Lemma 3.23 *Positive and negative coercions are $\phi\psi$-normal forms.*

Proof:

A coercion that is not in $\phi\psi$-normal form must at least contain either a ϕ- or ψ-redex, but such redexes consist of a composition of a positive and a negative coercion and can therefore be neither positive nor negative. Since a positive or a negative coercion cannot contain such non polarized coercions they cannot contain any ϕ- or ψ-redexes and must therefore be in $\phi\psi$-normal form.

■

3.2.3 Polarized factorization

As well as the notion of positive and negative coercions will prove useful when we discuss reduction of completion, so will the notion of polarized factoring and some properties of polarized factoring. There are two kinds of polarized factoring $+/-$- and $-/+$-factoring. Given a coercion $\mathbf{c}$ it will prove important to be able to find an $\phi\psi$-equal coercion that is written as a composition of two coercions, e.g. $\mathbf{c}_1\,;\mathbf{c}_2$, of opposite polarity. If the first coercion is positive and the second negative we say that $\mathbf{c}_1\,;\mathbf{c}_2$ is a $+/-$-factoring of $\mathbf{c}$ and, analogously, if the first coercion is negative and the second positive we say that $\mathbf{c}_1\,;\mathbf{c}_2$ is a $-/+$-factoring of $\mathbf{c}$. In the following we shall define polarized factoring and prove some important facts about it.

Definition 3.24 *(polarized factoring)*
Let $\mathbf{c}$, $\mathbf{c}_1$ and $\mathbf{c}_2$ be coercions, then

1. if $\phi\psi \vdash \mathbf{c} = {\mathbf{c}_1}^+\,;{\mathbf{c}_2}^-$ then we call ${\mathbf{c}_1}^+\,;{\mathbf{c}_2}^-$ a $+/-$-*factoring* of $\mathbf{c}$.
2. if $\phi\psi \vdash \mathbf{c} = {\mathbf{c}_1}^-\,;{\mathbf{c}_2}^+$ then we call ${\mathbf{c}_1}^+\,;{\mathbf{c}_2}^-$ a $-/+$-*factoring* of $\mathbf{c}$.

The following lemma shows that all optimal coercions can be uniquely $+/-$ and $-/+$ factored:

Lemma 3.25 *(Factoring of optimal coercion)*

1. *Every optimal coercion* $\mathbf{c}$ *has a unique* $+/-$*-factoring; that is, there exist unique (modulo core equality)* $\mathbf{d}_1^+$, $\mathbf{d}_2^-$ *such that* $\vdash \mathbf{c} = \mathbf{d}_1^+\,;\mathbf{d}_2^-$.
2. *Every optimal coercion* $\mathbf{c}$ *has a unique* $-/+$*-factoring; that is, there exist unique (modulo core equality)* $\mathbf{d}_1^-$, $\mathbf{d}_2^+$ *such that* $\vdash \mathbf{c} = \mathbf{d}_1^-\,;\mathbf{d}_2^+$.

Proof:
We will prove 1. The proof of 2 is similar. The proof is by induction on the structure of optimal (normal form) coercions (numbers refer to rules of Figure 6):

Base cases: $\mathbf{c} \equiv \iota$, $\mathbf{c} \equiv \texttt{box}$, $\mathbf{c} \equiv \texttt{unbox}$: Trivially shown by use of rules 1 and 2.

Inductive case: $\mathbf{c} \equiv \mathbf{c}' \to \mathbf{c}''$: By induction we have $\vdash \mathbf{c}' = {\mathbf{c}_1}^+\,;{\mathbf{c}_2}^-$ and $\vdash \mathbf{c}'' = {\mathbf{c}_3}^+\,;{\mathbf{c}_4}^-$. From this it follows:

$$\mathbf{c}' \to \mathbf{c}'' = ({\mathbf{c}_1}^+\,;{\mathbf{c}_2}^-) \to ({\mathbf{c}_3}^+\,;{\mathbf{c}_4}^-) =_3 (\mathbf{c}_2 \to \mathbf{c}_3)^+\,;(\mathbf{c}_1 \to \mathbf{c}_4)^-$$

Inductive case: $\mathbf{c} \equiv \mathbf{c}'\,;\texttt{box}$: By induction we have $\vdash \mathbf{c}' = {\mathbf{c}_1}^+\,;{\mathbf{c}_2}^-$. From this it follows:

$$\mathbf{c}'\,;\texttt{box} =_9 \texttt{box}\,;\texttt{[}\mathbf{c}'\texttt{]} = \texttt{box}\,;\texttt{[}{\mathbf{c}_1}^+\,;{\mathbf{c}_2}^-\texttt{]} =_{Ass,7} (\texttt{box}\,;\texttt{[}\mathbf{c}_1\texttt{]})^+\,;\texttt{[}\mathbf{c}_2\texttt{]}^-$$

Inductive case: $\mathbf{c} \equiv \texttt{unbox}\,;\mathbf{c}'$: Similar to the case for $\mathbf{c}'\,;\texttt{box}$.

Inductive case: $\mathbf{c} \equiv \forall\alpha\,.\,\mathbf{c}'$: By induction we have $\vdash \mathbf{c}' = {\mathbf{c}_1}^+\,;{\mathbf{c}_2}^-$. From this it follows:

$$\forall\alpha\,.\,\mathbf{c}' = \forall\alpha\,.\,{\mathbf{c}_1}^+\,;{\mathbf{c}_2}^- =_6 \forall\alpha\,.\,{\mathbf{c}_1}^+\,;\forall\alpha\,.\,{\mathbf{c}_2}^-$$

Inductive case: $\mathbf{c} \equiv \texttt{[}\mathbf{c}'\texttt{]}$: By induction we have $\vdash \mathbf{c}' = \mathbf{c_1}^+ ; \mathbf{c_2}^-$. From this it follows:

$$\texttt{[}\mathbf{c}'\texttt{]} = \texttt{[}\mathbf{c_1}^+ ; \mathbf{c_2}^-\texttt{]} =_7 \texttt{[}\mathbf{c_1}\texttt{]}^+ ; \texttt{[}\mathbf{c_2}\texttt{]}^-$$

In all cases it is easy to verify that the factoring is unique (modulo coercion equality). ■

Notice that polarized factoring is defined with respect to full $\phi\psi$-equality, but that for factoring of optimal coercions this is not needed only pure (core) coercion equality, i.e. without ϕ and ψ.

From the lemma the following corollary immediately follows:

Corollary 3.26 *(Factoring of coercions)*

1. *let* $\mathbf{c}$ *is an arbitrary coercion then there exist coercions* $\mathbf{c_1}$ *and* $\mathbf{c_2}$ *such that* $\mathbf{c_1}$ *is positive and* $\mathbf{c_2}$ *is negative and* $\phi\psi \vdash \mathbf{c} = \mathbf{c_1} ; \mathbf{c_2}$.
2. *let* $\mathbf{c}$ *is an arbitrary coercion then there exist coercions* $\mathbf{c_1}$ *and* $\mathbf{c_2}$ *such that* $\mathbf{c_1}$ *is negative and* $\mathbf{c_2}$ *is positive and* $\phi\psi \vdash \mathbf{c} = \mathbf{c_1} ; \mathbf{c_2}$.

Proof:

Follows from Lemma 3.25 and the fact that a coercions is $\phi\psi$-equal to any of its optimal coercions.

■

An interesting property of polarized factorization is that, if a coercion is a $-/+$- or $+/-$-factoring then we may reduce it to $\phi\psi$-normal form using only one of the rules ϕ resp. ψ:

Lemma 3.27 *Let* $\mathbf{c}_{nf}$ *a* $\phi\psi$*-normal form.*

1. *If* $\phi\psi \vdash \mathbf{c}_{nf} = \mathbf{c}^- ; \mathbf{d}^+$ *for some coercions* $\mathbf{c}$ *and* $\mathbf{d}$ *then* $\vdash \mathbf{c}^- ; \mathbf{d}^+ \Rightarrow_\psi^* \mathbf{c}_{nf}$.
2. *If* $\phi\psi \vdash \mathbf{c}_{nf} = \mathbf{c}^+ ; \mathbf{d}^-$ *for some coercions* $\mathbf{c}$ *and* $\mathbf{d}$ *then* $\vdash \mathbf{c}^+ ; \mathbf{d}^- \Rightarrow_\phi^* \mathbf{c}_{nf}$.

Proof:

We prove (1) here. The proof of (2) is analogous. The proof is by induction on the underlying standard type τ common to all of the representation types in the signatures of all the coercions $\mathbf{c}_{nf}$, $\mathbf{c}$ and $\mathbf{d}$.

Base case: $\tau \equiv \alpha$: In this case both $\mathbf{c}$ and $\mathbf{d}$ is an identity coercion, so the proof is trivial.

Inductive case: $\tau \equiv \tau_1 \to \tau_2$: Since both $\mathbf{c}$ and $\mathbf{d}$ are normal forms (Lemma 3.23) we know from Theorem 3.10, the coercion formation rules (Figure 4) and the definition of positive and negative coercions (Definition 3.21)that either:

$$\vdash \mathbf{c}^- = \mathbf{c_1}^- ; (\mathbf{c_2}^+ \to \mathbf{c_3}^-)$$
$$\vdash \mathbf{d}^+ = (\mathbf{d_1}^- \to \mathbf{d_2}^+) ; \mathbf{d_3}^+$$

where $\mathbf{c_1}$ is either $\texttt{unbox}_\upsilon$ or ι_υ and $\mathbf{d_3}$ is either $\texttt{box}_{\upsilon'}$ or $\iota_{\upsilon'}$ for some representation types υ and υ', or

$$\vdash \mathbf{c}^- = [\mathbf{c_2}^+ \rightarrow \mathbf{c_3}^-]$$

$$\vdash \mathbf{d}^+ = [\mathbf{d_1}^- \rightarrow \mathbf{d_2}^+]$$

We treat the first of these cases first. We have

$$\begin{array}{lll}
\mathbf{c}^- ; \mathbf{d}^+ & = & \\
\mathbf{c_1}^- ; (\mathbf{c_2}^+ \rightarrow \mathbf{c_3}^-) ; (\mathbf{d_1}^- \rightarrow \mathbf{d_2}^+) ; \mathbf{d_3}^+ & =_3 & \\
\mathbf{c_1}^- ; ((\mathbf{d_1}^- ; \mathbf{c_2}^+) \rightarrow (\mathbf{c_3}^- ; \mathbf{d_2}^+)) ; \mathbf{d_3}^+ & \Rightarrow_\psi & (\textit{induction}) \\
\mathbf{c_1}^- ; (\mathbf{c_4} \rightarrow \mathbf{c_5}) ; \mathbf{d_3}^+ & &
\end{array}$$

where $\mathbf{c}_4$ and $\mathbf{c}_5$ are $\phi\psi$-normal forms of $\mathbf{d_1}^- ; \mathbf{c_2}^+$ respectively $\mathbf{c_3}^- ; \mathbf{d_2}^+$. Now we almost finished, since only if $\mathbf{c}_1 \equiv \mathtt{unbox}_\upsilon$ and $\mathbf{d}_3 \equiv \mathtt{box}_{\upsilon'}$ is it not obvious that the derived coercion is equal (equal here means coercion equality) to one of the normal forms listed in Theorem 3.10. If $\mathbf{c}_1 \equiv \mathtt{unbox}_\upsilon$ and $\mathbf{d}_3 \equiv \mathtt{box}_{\upsilon'}$ then we may continue as follows:

$$\begin{array}{ll}
\mathbf{c_1}^- ; (\mathbf{c_4} \rightarrow \mathbf{c_5}) ; \mathbf{d_3}^+ & \equiv \\
\mathtt{unbox}_\upsilon ; (\mathbf{c_4} \rightarrow \mathbf{c_5}) ; \mathtt{box}_{\upsilon'} & =_9 \\
\mathtt{unbox}_\upsilon ; \mathtt{box}_\upsilon ; [\mathbf{c_4} \rightarrow \mathbf{c_5}] & \Rightarrow_\psi \\
\iota_\upsilon ; [\mathbf{c_4} \rightarrow \mathbf{c_5}] & =_2 \\
[\mathbf{c_4} \rightarrow \mathbf{c_5}] &
\end{array}$$

Which proves the first case since $[\mathbf{c_4} \rightarrow \mathbf{c_5}]$ is a $\phi\psi$-normal form. For the second case above we have:

$$\begin{array}{lll}
\mathbf{c}^- ; \mathbf{d}^+ & = & \\
[\mathbf{c_2}^+ \rightarrow \mathbf{c_3}^-] ; [\mathbf{d_1}^- \rightarrow \mathbf{d_2}^+] & =_5 & \\
[(\mathbf{c_2}^+ \rightarrow \mathbf{c_3}^-) ; (\mathbf{d_1}^- \rightarrow \mathbf{d_2}^+)] & =_3 & \\
[(\mathbf{d_1}^- ; \mathbf{c_2}^+) \rightarrow (\mathbf{c_3}^- ; \mathbf{d_2}^+)] & \Rightarrow_\psi & (\textit{induction}) \\
[\mathbf{c_4} \rightarrow \mathbf{c_5}] & &
\end{array}$$

where again $\mathbf{c}_4$ and $\mathbf{c}_5$ are $\phi\psi$-normal forms of $\mathbf{d_1}^- ; \mathbf{c_2}^+$ respectively $\mathbf{c_3}^- ; \mathbf{d_2}^+$. Now it is easy to verify that $[\mathbf{c_4} \rightarrow \mathbf{c_5}]$ is equal to one of the normal forms listed in Theorem 3.10.

Inductive case: $\tau \equiv \forall\alpha.\tau_1$: The proof of this case is completely analogous to the case for functions types.

■

A kind of reverse property of Lemma 3.27 is shown in the following lemma that shows that an arbitrary coercion may be $-/+$-factorized by ϕ-reduction alone and $+/-$-factorized by ψ-reduction alone:

Lemma 3.28 *Let* $\mathbf{c}$ *be an arbitrary coercion, then*

1. *there exist coercions* $\mathbf{c_1}^-$ *and* $\mathbf{c_2}^+$ *such that* $\vdash \mathbf{c} \Rightarrow^*_\phi \mathbf{c_1}^- ; \mathbf{c_2}^+$.

2. *there exist coercions* $\mathbf{c_1}^+$ *and* $\mathbf{c_2}^-$ *such that* $\vdash \mathbf{c} \Rightarrow^*_\psi \mathbf{c_1}^+ ; \mathbf{c_2}^-$.

Proof:

We will prove (1). The proof of (2) is analogous. If $\mathbf{c} \equiv \iota$ then the lemma is trivial. Otherwise, there exist a prime factorization of **c**. Let $\mathbf{d}_1$;...;$\mathbf{d}_n$ be an arbitrary prime factoring of **c**. We prove the lemma by induction on the length of such a prime factorization.

Base case: length = 1: in this case the lemma is also trivial (prime coercions are normal forms).

Inductive case: We have **c**= $\mathbf{d}_1$;...;$\mathbf{d}_n$. Since prime coercions are always either positive or negative there are two cases: $\mathbf{d}_1$ is negative and the lemma follow directly by induction; $\mathbf{d}_1$ is positive, and by induction $\mathbf{d}_2$,...;$\mathbf{d}_n$ may be reduced by ϕ-reduction to ${\mathbf{c}_1}^-;{\mathbf{c}_2}^+$ for some coercions $\mathbf{c}_1$ and $\mathbf{c}_2$. There therefore exist coercions ${\mathbf{d}_2}^-$,${\mathbf{d}_2}^+$ and $\mathbf{c}_{nf}$, where $\mathbf{c}_{nf}$ is in normal form, such that we have:

$$\begin{array}{lll} \mathbf{c} & = & \\ {\mathbf{d}_1}^+;\ldots;\mathbf{d}_n & \Rightarrow_\phi^* & (\mathit{induction}) \\ {\mathbf{d}_1}^+;{\mathbf{c}_1}^-;{\mathbf{c}_2}^+ & =_{\mathit{Lemma}\ 3.27} & \\ \mathbf{c}_{nf};{\mathbf{c}_2}^+ & =_{\mathit{Lemma}\ 3.25} & \\ {\mathbf{d}_2}^-;{\mathbf{d}_2}^+;{\mathbf{c}_2}^+ & & \end{array}$$

which shows that **c** can be $-/+$-factorized by only ϕ-reduction. ■

3.3 Representation type hierarchy

The positive coercions alone define a *subtype hierarchy* on representation types:

Definition 3.29 We define $\rho \leq \rho'$ iff there exists a positive coercion **c** such that $\vdash \mathbf{c} : \rho \leadsto \rho'$.

We may also give an inductively defined ordering $\leq_i$ on representation types:

Definition 3.30 We define $\leq_i$ as the smallest relation on representation types that satisfy the following conditions:

1. $\rho \leq_i \rho$
2. if $\rho_1 \leq_i \rho_2$ and $\rho_2 \leq_i \rho_3$ then $\rho_1 \leq_i \rho_3$
3. $\rho \leq_i [\rho]$
4. if $\rho_1 \leq_i \rho_1'$ and $\rho_2 \leq_i \rho_2'$ then $\rho_1 \to \rho_2 \leq_i \rho_1' \to \rho_2'$
5. if $\rho_1 \leq_i \rho_2$ then $\forall\alpha.\rho_1 \leq_i \forall\alpha.\rho_2$
6. if $\rho_1 \leq_i \rho_2$ then $[\rho_1] \leq_i [\rho_2]$

Notice that the function type constructor is *covariant* in both its arguments. The point that we will prove shortly is that the two orderings are equal. To show this we start by proving the following lemma which summarizes some of the important properties of positive and negative coercions:

Lemma 3.31 *(Positive and negative coercions and $\leq_i$)*

1. *let* $\mathbf{c}$ *be a coercion with signature* $\rho \leadsto \rho'$ *then* $\mathbf{c}{:}{+}$ *implies* $\rho \leq_i \rho'$ *and* $\mathbf{c}{:}{-}$ *implies* $\rho' \leq_i \rho$.
2. *if* $\rho \leq_i \rho'$ *then there exist a positive coercion with signature* $\rho \leadsto \rho'$ *and a negative coercion with signature* $\rho' \leadsto \rho$.
3. *given a positive/negative coercion* $\mathbf{c}$ *then there exist an negative/positive inverse coercion of* $\mathbf{c}$.

Proof:
(1) We will show this by induction on the structure of the coercion $\mathbf{c}$.

Base case: $\mathbf{c} \equiv \iota_\rho$: This case is trivial since we must have $\rho = \rho'$.

Base case: $\mathbf{c} \equiv \texttt{box}_\rho$: In this case $\mathbf{c}$ can only be positive and and we must have $\rho' = [\rho]$. So by rule 3 of Definition 3.30 we have $\rho \leq_i \rho'$.

Base case: $\mathbf{c} \equiv \texttt{unbox}_{\rho'}$: In this case $\mathbf{c}$ can only be negative and and we must have $\rho = [\rho']$. So by rule 3 of Definition 3.30 we have $\rho' \leq_i \rho$.

Inductive case: $\mathbf{c} \equiv \mathbf{d}_1 \to \mathbf{d}_2$: Assume first that $\mathbf{c}$ is positive and that $\rho \equiv \rho_1 \to \rho_2$ and $\rho' \equiv \rho_1' \to \rho_2'$. Then by Definition 3.21 we have that $\mathbf{d}_1$ must be negative and $\mathbf{d}_2$ must be positive and by induction we have $\rho_1 \leq_i \rho_1'$ and $\rho_2 \leq_i \rho_1'$. From this we may use rule 4 of Definition 3.30 to conclude $\rho \leq_i \rho'$. If $\mathbf{c}$ is negative then the proof of the case is analogous.

Inductive case: $\mathbf{c} \equiv [\mathbf{c}']$: Follows by induction and rule 6 of Definition 3.30.

Inductive case: $\mathbf{c} \equiv \forall\alpha.\mathbf{c}'$: Follows by induction and rule 5 of Definition 3.30.

This concludes the proof of (1).

(2) We prove this by induction on the structure of the common erasure of ρ and ρ'. We will denote the positive coercion by $\mathbf{c}_{pos}$ and the negative coercion by $\mathbf{c}_{neg}$.

Base case: $|\rho| \equiv \alpha$: Then $\rho \equiv \rho' \equiv \alpha$ and we choose both $\mathbf{c}_{pos}$ and $\mathbf{c}_{neg}$ to be ι_α.

Inductive case: $|\rho| \equiv \tau_1 \to \tau_2$: There are three case: (a) $\rho \equiv \rho_1 \to \rho_2$ and $\rho' \equiv \rho_1' \to \rho_2'$ for some types ρ_1, ρ_1', etc. and $\rho_1 \leq_i \rho_1'$ and $\rho_2 \leq_i \rho_2'$. By induction there exist a negative coercion $\mathbf{c}_1$ with signature $\rho_1' \leadsto \rho_1$ and a positive coercion $\mathbf{c}_2$ with signature $\rho_2 \leadsto \rho_2'$. From this it follows that we may choose $\mathbf{c}_{pos}$ to be $\mathbf{c}_1 \to \mathbf{c}_2$. Similarly, by induction there exist a positive coercion $\mathbf{d}_1$ with signature $\rho_1 \leadsto \rho_1'$ and a negative coercion $\mathbf{d}_2$ with signature $\rho_2' \leadsto \rho_2$. From this it follows that we may choose $\mathbf{c}_{neg}$ to be $\mathbf{d}_1 \to \mathbf{d}_2$. (b) $\rho \equiv \rho_1 \to \rho_2$ and $\rho' \equiv [\rho_1' \to \rho_2']$ for some types ρ_1, ρ_1', etc. and $\rho_1 \leq_i \rho_1'$ and $\rho_2 \leq_i \rho_2'$. Again by induction there exist a negative coercion $\mathbf{c}_1$ with signature $\rho_1' \leadsto \rho_1$ and a positive coercion $\mathbf{c}_2$ with signature $\rho_2 \leadsto \rho_2'$. We may therefore choose $\mathbf{c}_{pos}$ to be $\texttt{box}_{\rho_1 \to \rho_2}\,;\,[\mathbf{c}_1 \to \mathbf{c}_2]$. Similarly, by induction there exist a positive coercion $\mathbf{d}_1$

with signature $\rho_1 \rightsquigarrow {\rho_1}'$ and a negative coercion $\mathbf{d}_2$ with signature ${\rho_2}' \rightsquigarrow \rho_2$ and we choose $\mathbf{c}_{neg}$ to be $\texttt{[}\mathbf{c}_1 \texttt{->} \mathbf{c}_2\texttt{];unbox}_{\rho_1 \rightarrow \rho_2}$. (c) $\rho \equiv [\rho_1 \rightarrow \rho_2]$ and $\rho' \equiv [{\rho_1}' \rightarrow {\rho_2}']$ for some types ρ_1, ${\rho_1}'$, etc. and $\rho_1 \leq_i {\rho_1}'$ and $\rho_2 \leq_i {\rho_2}'$. By induction there exist a negative coercion $\mathbf{c}_1$ with signature ${\rho_1}' \rightsquigarrow \rho_1$ and a positive coercion $\mathbf{c}_2$ with signature $\rho_2 \rightsquigarrow {\rho_2}'$. So we choose $\mathbf{c}_{pos}$ to be $\texttt{[}\mathbf{c}_1 \texttt{->} \mathbf{c}_2\texttt{]}$. Similarly, by induction there exist a positive coercion $\mathbf{d}_1$ with signature $\rho_1 \rightsquigarrow {\rho_1}'$ and a negative coercion $\mathbf{d}_2$ with signature ${\rho_2}' \rightsquigarrow \rho_2$ and we choose $\mathbf{c}_{neg}$ to be $\texttt{[}\mathbf{c}_1 \texttt{->} \mathbf{c}_2\texttt{]}$.

Inductive case: $|\rho| \equiv \forall\alpha.\tau$: Similar to the previous case.

(3) Let $\mathbf{c}$ a positive coercion with signature $\rho \rightsquigarrow \rho'$. Then by (1) of this lemma $\rho \leq_i \rho'$. By (2) of this lemma there exist a negative coercion with signature $\rho' \rightsquigarrow \rho$ and by using coherence of coercions (Theorem 3.14) it is easy to see that this coercion is an inverse of $\mathbf{c}$.

■

Notice that we could have given an alternative definition of this ordering using the negative coercions: $\rho \leq \rho'$ if there exists a negative coercion $\mathbf{c}$ such that $\vdash \mathbf{c} : \rho' \rightsquigarrow \rho$. This is a consequence of Lemma 3.31 part 3.

Lemma 3.32 *The two orderings $\leq_i$ and $\leq$ define the same partial ordering on representation types.*

Proof:

Follows from Lemma 3.31 parts 1 and 2.

■

Proposition 3.33 *The representation types of any (standard) type τ (i.e., representation types whose type erasure is τ) form a finite lattice under $\leq$.*

Using Definition 3.30 it is easy to give a definition of least upper bound $\sqcup$ and greatest lower bound $\sqcap$ in the representation type lattice:

Definition 3.34 We define the *least upper bound* operation $\sqcup$ in the representation type lattice for a given standard type τ as follows:

$$\begin{array}{lcl}
\alpha \sqcup \alpha & = & \alpha \\
(\rho_1 \rightarrow \rho_2) \sqcup ({\rho_1}' \rightarrow {\rho_2}') & = & (\rho_1 \sqcup {\rho_1}') \rightarrow (\rho_2 \sqcup {\rho_2}') \\
(\forall\alpha.\rho) \sqcup (\forall\alpha.\rho') & = & \forall\alpha.(\rho \sqcup \rho') \\
[v] \sqcup v' & = & [v \sqcup v'] \\
v \sqcup [v'] & = & [v \sqcup v'] \\
[v] \sqcup [v'] & = & [v \sqcup v']
\end{array}$$

and we define the *greatest lower bound* operation $\sqcap$ in the representation type lattice for a given standard type τ as follows:

$$\begin{array}{lcl}
\alpha \sqcap \alpha & = & \alpha \\
(\rho_1 \rightarrow \rho_2) \sqcap (\rho_1' \rightarrow \rho_2') & = & (\rho_1 \sqcap \rho_1') \rightarrow (\rho_2 \sqcap \rho_2') \\
(\forall \alpha.\rho) \sqcap (\forall \alpha.\rho') & = & \forall \alpha.(\rho \sqcap \rho') \\
[v] \sqcap v' & = & v \sqcap v' \\
v \sqcap [v'] & = & v \sqcap v' \\
[v] \sqcap [v'] & = & [v \sqcap v']
\end{array}$$

Since $\sqcup$ and $\sqcap$ are only used on representation types that have the same underlying standard type one can easily prove that these operation are well defined (by induction on the structure of the standard type).

3.4 Summary

The following diagram summarizes some of the important properties of coercion calculus and coercion reduction. The names of coercions in this is not to be considered unique, but like meta variables in a grammar. These will often have conditions on them, like, they have to be positive, etc. A line in the diagram from one coercion to another means that there exists coercions that fulfill the conditions and are equal (modulo the core coercion equations). An arrow decorated with reduction rules from one coercion to another means that there exist coercions such that these fulfill the conditions and the first one reduces to the second one by the reduction rules (modulo the core coercion equations).

$$\begin{tikzcd}
 & \varpi_1; \ldots; \varpi_n \arrow[d, dash] & \\
\mathbf{c}^+ ; \mathbf{c}^- \arrow[dr, "\phi", "*"'] & \mathbf{c} \arrow[l, "\psi", "*"'] \arrow[r, "\phi", "*"'] \arrow[d, "\psi\phi", "R"', "*"] & \mathbf{c}^- ; \mathbf{c}^+ \arrow[dl, "\psi"', "*"] \\
\mathbf{c}^- ; \mathbf{c}^+ \arrow[r, dash] & \mathbf{c}_{nf} \arrow[r, dash] & \mathbf{c}^+ ; \mathbf{c}^-
\end{tikzcd}$$

Figure 10: Summary of coercion calculus

By combining the several lines and arrow one can read of most of the important properties about coercion equality and reduction proven in this chapter. For example, by starting a the node $\mathbf{c}$ going down to $\mathbf{c}_{nf}$ and then either left to $\mathbf{c}^- ; \mathbf{c}^+$ or right to $\mathbf{c}^+ ; \mathbf{c}^-$ one obtains Corollary 3.26.

Chapter 4

Coherence and reduction of completions

In this chapter we will extend what we did for coercions in Chapter 3 to include completions. That is, we will investigate whether, based on the way of measuring quality of coercions that we discussed in Chapter 3, there exist optimal completions and how to find these (if they exist). Since coercions are parts of completions, improving the performance of the coercions in completions will at least not make the performance these worse, but most likely improve it. We will extend the equality axioms for coercions with equality axioms for expressions to obtain an equality theory on completions. It is obvious that coercion equality must be part of such an equality, since replacing equal coercions in a completion should not result in a completion with a different performance, and such completions should definitely be considered equal by our theory. Intuitively the equality axioms "move" coercions around in completions without eliminating these. From the equality axioms for expressions we will define several different forms of reduction on completions in a manner similar to what we did for coercions. First we will define what we wil call $\phi\psi$-reduction on completions as $\phi\psi$ reduction modulo the completion equality. We will then show that $\phi\psi$-reduction on completions is not, as one would have hoped confluent and therefore not canonical. But we show that by giving higher priority to the elimination of `unbox;box`-redexes over `box;unbox` or the other way round, we arrive at two *formal optimality* criteria for completions. We show that there exist reductions systems that may be used to find such optimal completions and that these systems, fortunately, are canonical. We also show that the two kinds of optimal completions are unique modulo completion equality.

4.1 Equational axiomatization of completion congruence

In this section we present an equational axiomatization of completion congruence and show that this is coherent.

4.1.1 Expression equations

We need to extend the equality axioms for coercions with equations for expressions. We should at least require of these new equations that they are strong enough to prove coherence of completion equality. Furthermore, it would seem natural that they should not overlap with coercion equality in the following sense: the equations for expressions should intuitively only move `box`- and `unbox`-coercions around not eliminate or create these. That is, they should at least not contain any "hidden" ϕ or ψ equality steps. What we mean by this is that the equations should be just strong enough to prove coherence, but also so weak that there are no case in which two completions can be proven equal using the extended equality theory (expression equation and coercion equation) in which one of the rules ϕ or ψ have been used, that can also be proven equal not using any of these two rules. This requirement does, however, not seem to be possible to conform to, except on ΛI-terms (see Barendregt [Bar84]), since functions may discard their argument.

Let us now state the new equations and then discuss whether these conform to our requirements. We extend the equality axioms for coercions with the equality axioms for explicitly boxed F$_2$-expressions in Figures 11 and 12. The expressions on both sides of an equality are assumed to be well-formed and to have the same type in a single type environment. In other words, the equations in Figures 11 and 12 should be understood as abbreviations for more complex rules for typed equality. For example, Equation 11 is an abbreviation for:

$$\frac{\Gamma \vdash e{:}\rho \quad \Gamma \vdash \langle \iota_\rho \rangle e{:}\rho}{\Gamma \vdash \langle \iota_\rho \rangle e = e{:}\rho}$$

The special coercions pattern used in Equation 17 of Figure 12 is defined as follows:

Definition 4.1 For every representation type $\rho(\alpha)$, i.e. with free type variable α, we define a coercion pattern $\widehat{\rho}(c,d)$ parameterized by the coercion variables (variables ranging over coercions) c and d in the following way:

$$\begin{array}{ll}
\widehat{\alpha'}(c,d) & = d, \text{ if } \alpha'{=}\alpha \\
 & = \iota_{\alpha'}, \text{ otherwise} \\
\rho_1 \widehat{\rightarrow} \rho_2(c,d) & = \widehat{\rho_1}(d,c) \rightarrow \widehat{\rho_2}(c,d) \\
\widehat{[\rho_1]}(c,d) & = [\widehat{\rho_1}(c,d)] \\
\forall \widehat{\alpha'.}\rho_1(c,d) & = \forall \alpha\,.\,\widehat{\rho_1}(\iota_\alpha,\iota_\alpha) \text{ , if } \alpha'{=}\alpha \\
 & = \forall \alpha'\,.\,\widehat{\rho_1}(c,d), \text{ otherwise}
\end{array}$$

Example 4.1 We will give a few examples of these kind of patterns:

1. let $\rho(\alpha)$ be $\forall \alpha.\alpha \rightarrow \alpha$ then $\widehat{\rho}(c,d)$ will be $\forall \alpha\,.\,\iota_\alpha \rightarrow \iota_\alpha$, i.e. the identity coercion equal to $\iota_{\forall \alpha.\alpha \rightarrow \alpha}$.
2. let $\rho(\alpha)$ be $\alpha \rightarrow (\alpha' \rightarrow \alpha)$ then $\widehat{\rho}(c,d)$ will be $c \rightarrow (\iota_{\alpha'} \rightarrow d)$

We can now define completion equality in an analogous way to coercions equality:

Definition 4.2 *(Completion equality)*
We say e and e' are A-equal, written $A \vdash e = e'$, if $e = e'$ is derivable from the axioms A together with equations in Figures 6 and 11 and rules for reflexivity, symmetry, transitivity and compatibility of $=$.

An important point to remember from this is that any equality defined in this way on completions will include the core coercion equations, but that the equations ϕ and ψ is not part of completion equality just as they were not part of coercions equality. From now on we will refer to the equality axioms of Figure 12 axioms or equations E. So the equality that we are interested in proving coherence for is E-equality.

Let us now see if the axioms of E and 11 conform to our requirements. If we look at Equations 11, 12, 14, 15, and 16 it is clear that these only move `box`- and `unbox`-coercions around. Equation 13 may discard, duplicate or even create new `box`- and `unbox`-coercions. Consider for example a function that never used its argument. We may choose the representation type of the formal parameter of such a function arbitrarily as long as it is a legal representation type of the formal parameters underlying type. Equation 17 is similar to Equation 13 in this respect. The coercion $\mathbf{c}$ in rule 17 can dependent on the type of the expression e and can therefore either be discarded, duplicated or introduced by using the rule. That the equations, when restricted to ΛI-terms, does not contain "hidden" $\phi\psi$-equalities is fairly easy to see for Equations 11 to 16, only for Equation 17 this is not obvious. If we were to use the more liberal equation instead of Equation 17:

$$\langle \mathbf{c} \rangle (e\{\pi\}) = \langle \mathbf{c}' \rangle (e\{\pi'\})$$

we could prove completions equal that we would need $\phi\psi$ equality on coercions to prove if we were only using Equation 17. Below (in Subsection 4.1.4) we will discuss how Equation 17 is related to parametricity and that what Equation 17 describe is essentially a form of "free" theorems, so it is likely that also Equation 17 fulfill the requirement that is does not contain "hidden" $\phi\psi$-equalities.

That the equations also fulfill the first requirement that they are strong enough to prove coherence of completion equality will be shown below in Theorem 4.8.

$$\boxed{\begin{array}{rcll} \langle \iota \rangle e & = & e & (11) \\ \langle \mathbf{c} \rangle \langle \mathbf{d} \rangle e & = & \langle \mathbf{d}\,;\mathbf{c} \rangle e & (12) \end{array}}$$

Figure 11: Equality rules for coercion application

$$\boxed{\begin{array}{rcll} \langle \mathbf{c} \to \mathbf{d} \rangle \lambda x\,.\,e & = & \lambda x\,.\langle \mathbf{d} \rangle (e[\langle \mathbf{c} \rangle x/x]) & (13) \\ (\langle \mathbf{c} \to \mathbf{d} \rangle e)\ e' & = & \langle \mathbf{d} \rangle (e\ (\langle \mathbf{c} \rangle e')) & (14) \\ \Lambda\alpha\,.\langle \mathbf{c} \rangle e & = & \langle \forall\alpha\,.\,\mathbf{c} \rangle \Lambda\alpha\,.\,e & (15) \\ (\langle \forall\alpha\,.\,\mathbf{c} \rangle e)\{\pi\} & = & \langle \mathbf{c}[\pi/\alpha] \rangle (e\{\pi\}) & (16) \\ \langle \widehat{\rho}(\mathbf{c}, \iota_\pi) \rangle (e\{\pi\}) & = & \langle \widehat{\rho}(\iota_{\pi'}, \mathbf{c}) \rangle (e\{\pi'\}) & (17) \\ \multicolumn{4}{l}{\text{Side condition: in equation 17 the type of } e \text{ is } \forall\alpha.\rho} \end{array}}$$

Figure 12: Equality rules for explicitly boxed expressions (E)

Both of the equations 16 and 17 of Figure 12 are concerned with type applications, and one could think that there might be some overlap between these, this is however not the case and they do in fact work quite differently. We will now discuss this difference in detail.

Let us assume that we are given some subexpression $\langle \mathbf{c} \rangle (e\{\pi\})$ in a completion. Assume that the type of e is $\forall\alpha.\rho$. Equation 16 changes the representation type of the operator expression in the type application. On the left hand side of 16 this expressions is $\langle \forall\alpha\,.\,\mathbf{c} \rangle e$ while on the right hand side it is e. This change is independent of the type argument π which is is also unchanged. Equation 17 leaves the representation type of the operator expression in the type application unchanged, but changes the representation type of the type argument instead. This shows that the two equations are independent of each other, that is, work on different parts of the coercions **c**.

Let us give a few examples that illustrates this difference. As an example of the use of rule 15 and 16 consider the following deduction:

$$\begin{gathered}
\langle \mathtt{unbox}_{[[\mathtt{int}]\to[\mathtt{int}]]\to[[\mathtt{int}]\to[\mathtt{int}]]}\rangle((\Lambda\alpha\,.\,\langle\mathtt{box}_{\alpha\to\alpha}\rangle\lambda x\!:\!\alpha\,.\,x)\{[[\mathtt{int}]\to[\mathtt{int}]]\}) \;=_{16}\\
((\langle\forall\alpha\,.\,\mathtt{unbox}_{\alpha\to\alpha}\rangle(\Lambda\alpha\,.\,\langle\mathtt{box}_{\alpha\to\alpha}\rangle\lambda x\!:\!\alpha\,.\,x))\{[[\mathtt{int}]\to[\mathtt{int}]]\} \;=_{15}\\
(\Lambda\alpha\,.\,\langle\mathtt{unbox}_{\alpha\to\alpha}\rangle\langle\mathtt{box}_{\alpha\to\alpha}\rangle\lambda x\!:\!\alpha\,.\,x)\{[[\mathtt{int}]\to[\mathtt{int}]]\} \;=_{12,\psi,11}\\
(\Lambda\alpha\,.\,\lambda x\!:\!\alpha\,.\,x)\{[[\mathtt{int}]\to[\mathtt{int}]]\}
\end{gathered}$$

This shows that rule 16 changes the representation of the polymorphic identity function $\Lambda\alpha\,.\,\lambda x\!:\!\alpha\,.\,x$ by changing the representation type of the operator expression in the type application from $\forall\alpha.[\alpha\to\alpha]$ to $\forall\alpha.\alpha\to\alpha$.

For an example of the use of rule 17 consider the following:

$$\begin{gathered}
\langle\,[\mathtt{unbox}_{\mathtt{int}}\to\mathtt{box}_{\mathtt{int}}]\to[\mathtt{box}_{\mathtt{int}}\to\mathtt{unbox}_{\mathtt{int}}]\rangle((\Lambda\alpha\,.\,\lambda x\!:\!\alpha\,.\,x)\{[[\mathtt{int}]\to[\mathtt{int}]]\}) \;=_{3,12}\\
\langle\iota\to[\mathtt{box}_{\mathtt{int}}\to\mathtt{unbox}_{\mathtt{int}}]\rangle(\langle\,[\mathtt{unbox}_{\mathtt{int}}\to\mathtt{box}_{\mathtt{int}}]\to\iota\rangle((\Lambda\alpha\,.\,\lambda x\!:\!\alpha\,.\,x)\{[[\mathtt{int}]\to[\mathtt{int}]]\})) \;=_{17}\\
\langle\iota\to[\mathtt{box}_{\mathtt{int}}\to\mathtt{unbox}_{\mathtt{int}}]\rangle(\langle\iota\to[\mathtt{unbox}_{\mathtt{int}}\to\mathtt{box}_{\mathtt{int}}]\rangle((\Lambda\alpha\,.\,\lambda x\!:\!\alpha\,.\,x)\{[\mathtt{int}\to\mathtt{int}]\})) \;=_{12,3}\\
\langle\iota\to(\,[\mathtt{unbox}_{\mathtt{int}}\to\mathtt{box}_{\mathtt{int}}]\,;[\mathtt{box}_{\mathtt{int}}\to\mathtt{unbox}_{\mathtt{int}}])\rangle((\Lambda\alpha\,.\,\lambda x\!:\!\alpha\,.\,x)\{[\mathtt{int}\to\mathtt{int}]\}) \;=_{5}\\
\langle\iota\to[(\mathtt{unbox}_{\mathtt{int}}\to\mathtt{box}_{\mathtt{int}})\,;(\mathtt{box}_{\mathtt{int}}\to\mathtt{unbox}_{\mathtt{int}})]\rangle((\Lambda\alpha\,.\,\lambda x\!:\!\alpha\,.\,x)\{[\mathtt{int}\to\mathtt{int}]\}) \;=_{3}\\
\langle\iota\to[(\mathtt{box}_{\mathtt{int}}\,;\mathtt{unbox}_{\mathtt{int}})\to(\mathtt{box}_{\mathtt{int}}\,;\mathtt{unbox}_{\mathtt{int}})]\rangle((\Lambda\alpha\,.\,\lambda x\!:\!\alpha\,.\,x)\{[\mathtt{int}\to\mathtt{int}]\}) \;=_{\phi}\\
\langle\iota\to[\iota\to\iota]\rangle((\Lambda\alpha\,.\,\lambda x\!:\!\alpha\,.\,x)\{[\mathtt{int}\to\mathtt{int}]\}) \;=_{4,6,11}\\
(\Lambda\alpha\,.\,\lambda x\!:\!\alpha\,.\,x)\{[\mathtt{int}\to\mathtt{int}]\}
\end{gathered}$$

which illustrates that rule 17 only changes the representation of the type argument, in this case from $[[\mathtt{int}]\to[\mathtt{int}]]$ to $[\mathtt{int}\to\mathtt{int}]$.

4.1.2 Expression coherence

We assume that all congruent completions are semantically equivalent, and if our axiomatization is coherent then since any two congruent completions of a F_2-expression are observationally indistinguishable then the observational congruence of explicitly boxed F_2 must satisfy $E\phi\psi$-equality, and vice versa. Otherwise one could find two congruent completions with different observable behavior. So we need to show that our axiomatization of completion congruence is coherent. This is what we will do next, but first we need the following definition:

Definition 4.3 *(Head coercion free)*
We call an explicitly boxed F_2-expression *head coercion free* (c.f. [CG90]) if it is not of the form $\langle\mathbf{c}\rangle e$.

and a small lemma that shows that an arbitrary completion is equal to a coercion in which every head coercion free subexpression has exactly one coercion applied to it:

Lemma 4.4 *Let e be an explicitly boxed F_2-expression, then there exists an explicitly boxed F_2-expression e' such that $E\phi\psi \vdash e' = e$ and that all coercions in e' are only applied to head coercion free expressions and every head coercion free subexpression has exactly one coercion applied to it.*

Proof:

Follows from $\langle \mathbf{c} \rangle \langle \mathbf{c}' \rangle e = \langle \mathbf{c}' ; \mathbf{c} \rangle e$, and $e = \langle \iota \rangle e$ (see Figure 11). ■

We also need a lemma about the coercions in equation 17 and for this we need the following terminology:

Definition 4.5 An occurrence o in a type ρ is a subterm in ρ(we assume that we have some way to distinguish different textually equal occurrences) and we write $o \in \rho$ if o is an occurrence in ρ. The n'th occurrence of a type variable α is the n'th textual occurrence of α in ρ when the type is scanned from left to right. Occurrences in a type can be given a polarity. We call o a *positive occurrence* in ρ if:

1. $o = \rho$
2. $\rho = \rho_1 \rightarrow \rho_2$ and o is a positive occurrence in ρ_2 or o is a negative occurrence in ρ_1.
3. $\rho = \forall \alpha . \rho_1$ and o is a positive occurrence in ρ_1.
4. $\rho = [\rho_1]$ and o is a positive occurrence in ρ_1.

and we call o a *negative occurrence* in ρ if:

1. $\rho = \rho_1 \rightarrow \rho_2$ and o is a negative occurrence in ρ_2 or o is a positive occurrence in ρ_1.
2. $\rho = \forall \alpha . \rho_1$ and o is a negative occurrence in ρ_1.
3. $\rho = [\rho_1]$ and o is a negative occurrence in ρ_1.

Definition 4.6 Let α be a type variable and let ρ' and ρ be two (representation) types then we define $\rho[\rho_1/\alpha_-, \rho_2/\alpha_+]$ as the type ρ where all free positive occurrences of α have been replaced by ρ_1 all free negative occurrences of α have been replaced by ρ_2.

We now state and prove the lemma:

Lemma 4.7 *Let* $\mathbf{c}$ *be a coercion with signature* $\pi \rightsquigarrow \pi'$ *and let e be a completion with type* $\forall \alpha . \rho$*, then*

1. $\widehat{\rho}(\iota_\pi, \mathbf{c})$ *has signature* $\rho[\pi/\alpha] \rightsquigarrow \rho[\pi/\alpha_-, \pi'/\alpha_+]$
2. $\widehat{\rho}(\mathbf{c}, \iota_\pi)$ *has signature* $\rho[\pi'/\alpha_-, \pi/\alpha_+] \rightsquigarrow \rho[\pi/\alpha]$

Proof:

The proof is by induction on the structure of ρ.

Base case: $\rho \equiv \alpha$: We look at the two cases: (1) we have $\widehat{\rho}(\iota_\pi,\mathbf{c}) = \mathbf{c}$ which has signature $\pi \rightsquigarrow \pi'$ which is what the lemma states since the occurrence of α is positive. (2) we have $\widehat{\rho}(\mathbf{c},\iota_\pi) = \iota_\pi$ which has signature $\pi \rightsquigarrow \pi$ which is also in accordance with what the lemma states since the occurrence of α is positive.

Base case: $\rho \equiv \alpha'$ ($\alpha' \neq \alpha$): In this case we have $\widehat{\rho}(\iota_\pi,\mathbf{c}) = \iota_{\alpha'}$ and $\widehat{\rho}(\mathbf{c},\iota_\pi) = \iota_{\alpha'}$ which both have signature $\alpha' \rightsquigarrow \alpha'$ which is what the lemma states.

Inductive case: $\rho \equiv \rho_1 \to \rho_2$: We look at part 1 of the lemma, the proof of part 2 is analogous: we have $\widehat{\rho}(\iota_\pi,\mathbf{c}) = \widehat{\rho_1}(\mathbf{c},\iota_\pi) \to \widehat{\rho_2}(\iota_\pi,\mathbf{c})$. By induction we have that $\widehat{\rho_1}(\mathbf{c},\iota_\pi)$, which matches part 2 of the lemma, has signature $\rho_1[\pi'/\alpha_-, \pi/\alpha_+] \rightsquigarrow \rho_1[\pi/\alpha]$ and $\widehat{\rho_2}(\iota_\pi,\mathbf{c})$, which matches part 1 of the lemma, has signature $\rho_2[\pi/\alpha] \rightsquigarrow \rho_2[\pi/\alpha_-, \pi'/\alpha_+]$. This means that $\widehat{\rho}(\iota_\pi,\mathbf{c})$ must have signature:

$$\rho_1[\pi/\alpha] \to \rho_2[\pi/\alpha] \rightsquigarrow \rho_1[\pi'/\alpha_-, \pi/\alpha_+] \to \rho_2[\pi/\alpha_-, \pi'/\alpha_+]$$

But since the positive/negative occurrence of a type variable in ρ_1 is a negative/positive occurrence in $\rho_1 \to \rho_2$ we have by standard properties of substitution:

$$(\rho_1 \to \rho_2)[\pi/\alpha] \rightsquigarrow (\rho_1 \to \rho_2)[\pi/\alpha_-, \pi'/\alpha_+]$$

which is what the lemma states for part 1.

Inductive case: $\rho \equiv [\rho_1]$: We look at part 1 of the lemma, the proof of part 2 is analogous: we have $\widehat{\rho}(\iota_\pi,\mathbf{c}) = [\![\widehat{\rho_1}(\iota_\pi,\mathbf{c})]\!]$. By induction we have that $\widehat{\rho_1}(\iota_\pi,\mathbf{c})$ has signature $\rho_1[\pi/\alpha] \rightsquigarrow \rho_1[\pi/\alpha_-, \pi'/\alpha_+]$ and $\widehat{\rho}(\iota_\pi,\mathbf{c})$ must therefore have signature:

$$[\rho_1[\pi/\alpha]] \rightsquigarrow [\rho_1[\pi/\alpha_-, \pi'/\alpha_+]]$$

using standard properties of substitution is equal to:

$$[\rho_1][\pi/\alpha] \rightsquigarrow [\rho_1][\pi/\alpha_-, \pi'/\alpha_+]$$

which is what the lemma states for part 1.

Inductive case: $\rho \equiv \forall\alpha.\rho_1$: We look at part 1 of the lemma, the proof of part 2 is analogous: we have $\widehat{\rho}(\iota_\pi,\mathbf{c}) = \forall\alpha.\widehat{\rho_1}(\iota_\alpha,\iota_\alpha)$. By induction we have that $\widehat{\rho_1}(\iota_\alpha,\iota_\alpha)$ has signature $\rho_1 \rightsquigarrow \rho_1$ because by the lemma we should substitute α for itself in ρ_1 both for the domain and the range of the signature and $\widehat{\rho}(\iota_\pi,\mathbf{c})$ must therefore have signature:

$$\forall\alpha.\rho_1 \rightsquigarrow \forall\alpha.\rho_1$$

which using standard properties of substitution this is equal to:

$$(\forall\alpha.\rho_1)[\pi/\alpha] \rightsquigarrow (\forall\alpha.\rho_1)[\pi/\alpha_-, \pi'/\alpha_+]$$

which is what the lemma states for part 1.

Inductive case: $\rho \equiv \forall\alpha'.\rho_1$ $(\alpha' \neq \alpha)$: We look at part 1 of the lemma, the proof of part 2 is analogous: we have $\widehat{\rho}(\iota_\pi,\mathbf{c}) = \forall\alpha'.\widehat{\rho_1}(\iota_\pi,\mathbf{c}_1)$. By induction we have that $\widehat{\rho_1}(\iota_\pi,\mathbf{c}_1)$ has signature $\rho_1[\pi/\alpha] \rightsquigarrow \rho_1[\pi/\alpha_-, \pi'/\alpha_+]$ and $\widehat{\rho}(\iota_\pi,\mathbf{c})$ must therefore have signature:

$$\forall\alpha'.(\rho_1[\pi/\alpha]) \rightsquigarrow \forall\alpha'.(\rho_1[\pi/\alpha_-, \pi'/\alpha_+])$$

from which we get:

$$(\forall\alpha'.\rho_1)[\pi/\alpha] \rightsquigarrow (\forall\alpha'.\rho_1)[\pi/\alpha_-, \pi'/\alpha_+]$$

which is what the lemma states for part 1.

■

We will now prove coherence of completions:

Theorem 4.8 *(Coherence of completions) Explicitly boxed expressions e' and e'' are congruent if and only if they are $E\phi\psi$-equal; i.e.,*

$e' \cong e''$ *iff* $E\phi\psi \vdash e' = e''$

Proof:
"if": Assume $E\phi\psi \vdash e' = e''$. By inspection of E we can verify that e' and e'' have the same erasure. Since both e' and e'' are completions at the same type they are congruent, i.e., $e' \cong e''$.

"only if": We will prove this by induction on the structure of the common erasure of e' and e''. From Lemma 4.4 we may without loss of generality assume that in e' and e'' coercions are only applied to head coercion free expressions and that every head coercion free subexpression has exactly one coercion applied to it.

Now assume that we have $\Gamma\vdash e':\rho$ and $\Gamma\vdash e'':\rho$ for some type ρ and that e' and e'' are congruent:

Base case: $|e'| \equiv x$.

Then $e' \equiv \langle\mathbf{c}'\rangle x$ and $e'' \equiv \langle\mathbf{c}''\rangle x$ for some $\mathbf{c}'$ and $\mathbf{c}''$. Let ρ_x be the type of x. Then both $\mathbf{c}'$ and $\mathbf{c}''$ have signature $\rho_x \rightsquigarrow \rho$. So by Theorem 3.14 we must have $\phi\psi \vdash \mathbf{c} = \mathbf{c}'$, from which it follows that $E\phi\psi \vdash \langle\mathbf{c}'\rangle x = \langle\mathbf{c}''\rangle x$.

Inductive case: $|e'| \equiv e_1\{\pi\}$.

Then $e' \equiv \langle\mathbf{c}'\rangle(e_1'\{\pi'\})$ and $e'' \equiv \langle\mathbf{c}''\rangle(e_1''\{\pi''\})$ for some $\mathbf{c}'$, e_1', π', $\mathbf{c}''$, e_1'' and π''. Since e_1' and e_1'' have the same erasure and therefore the same standard type, there must (by Proposition 2.2) exist a coercion $\mathbf{c}$ such that $e_1' \cong \langle\mathbf{c}\rangle e_1''$. Since e_1' and $\langle\mathbf{c}\rangle e_1''$ are congruent we have by induction (since $|e_1'| \equiv |\langle\mathbf{c}\rangle e_1''|$ is a subexpression of $|e'|$) that $E\phi\psi \vdash e_1' = \langle\mathbf{c}\rangle e_1''$. Furthermore, since $\mathbf{c}$ must have signature $\forall\alpha.\rho' \rightsquigarrow \forall\alpha.\rho''$ where $\forall\alpha.\rho'$ is the type of e' and $\forall\alpha.\rho''$ is the type of e_1'' there must exist a coercion $\mathbf{d}$ with signature $\rho' \rightsquigarrow \rho''$ such that $\phi\psi \vdash \mathbf{c} = \forall\alpha.\mathbf{d}$ (by Proposition 2.2 and coercion congruence Theorem 3.14). We therefore have:

$$\begin{array}{ll}
e' & \equiv \\
\langle \mathbf{c}' \rangle (e_1{}'\{\pi'\}) & =_{induction} \\
\langle \mathbf{c}' \rangle ((\langle \mathbf{c} \rangle e_1{}'')\{\pi'\}) & =_{\phi,\psi,C} \\
\langle \mathbf{c}' \rangle ((\langle \forall\alpha\,.\,\mathbf{d} \rangle e_1{}'')\{\pi'\}) & =_{16} \\
\langle \mathbf{d}' \rangle (e_1{}''\{\pi'\}) &
\end{array}$$

where $\mathbf{d}' \equiv \mathbf{d}[\pi'/\alpha]\,;\mathbf{c}'$. So we need to prove that $\langle \mathbf{d}' \rangle (e_1{}''\{\pi'\}) = \langle \mathbf{c}'' \rangle (e_1{}''\{\pi''\})$. Since $e_1{}''\{\pi'\}$ have type $\rho''[\pi'/\alpha]$ there exist by Lemma 4.7, coherence of coercion equality and Proposition 2.2 coercions $\mathbf{d}_1$ and $\mathbf{d}_2$ such that $\phi\psi \vdash \mathbf{d}' = \widehat{\rho''}(\iota,\mathbf{d}_1)\,;\mathbf{d}_2$ where $\mathbf{d}_1$ has signature $\pi' \rightsquigarrow \pi''$. We can therefore continue the above deduction:

$$\begin{array}{ll}
\langle \mathbf{d}' \rangle (e_1{}''\{\pi'\}) & = \\
\langle \widehat{\rho''}(\iota,\mathbf{d}_1)\,;\mathbf{d}_2 \rangle (e_1{}''\{\pi'\}) & =_{12} \\
\langle \mathbf{d}_2 \rangle \langle \widehat{\rho''}(\iota,\mathbf{d}_1) \rangle (e_1{}''\{\pi'\}) & =_{17} \\
\langle \mathbf{d}_2 \rangle \langle \widehat{\rho''}(\mathbf{d}_1,\iota) \rangle (e_1{}''\{\pi''\}) & =_{12} \\
\langle \widehat{\rho''}(\mathbf{d}_1,\iota)\,;\mathbf{d}_2 \rangle (e_1{}''\{\pi''\}) & =_{Theorem\ 3.14} \\
\langle \mathbf{c}'' \rangle (e_1{}''\{\pi''\}) & \equiv \\
e'' &
\end{array}$$

which proves the case.

Inductive case: $|e'| \equiv \Lambda\alpha\,.\,e_1$.

Then $e' \equiv \langle \mathbf{c}' \rangle \Lambda\alpha\,.\,e_1{}'$ and $e'' \equiv \langle \mathbf{c}'' \rangle \Lambda\alpha\,.\,e_1{}''$ for some $\mathbf{c}'$, α, $e_1{}'$, $\mathbf{c}''$ and $e_1{}''$.

It must be the case that $\phi\psi \vdash \mathbf{c}' = (\forall\alpha\,.\,\mathbf{d}')\,;\mathbf{c}$ and $\phi\psi \vdash \mathbf{c}'' = (\forall\alpha\,.\,\mathbf{d}'')\,;\mathbf{c}$ for some coercions $\mathbf{c}$, $\mathbf{d}'$ and $\mathbf{d}''$ where $\mathbf{c}$ is either ι or `box`. This follow from Proposition 2.2 and the formation rules for coercions (Figure 4), $\mathbf{c}$ is `box` if the type of e' (of e'') is boxed and ι otherwise. We now have

$$\begin{array}{ll}
e' & \equiv \\
\langle \mathbf{c}' \rangle \Lambda\alpha\,.\,e_1{}' & =_{\phi,\psi,C} \\
\langle (\forall\alpha\,.\,\mathbf{d}')\,;\mathbf{c} \rangle \Lambda\alpha\,.\,e_1{}' & =_{12} \\
\langle \mathbf{c} \rangle \langle \forall\alpha\,.\,\mathbf{d}' \rangle \Lambda\alpha\,.\,e_1{}' & =_{15} \\
\langle \mathbf{c} \rangle \Lambda\alpha\,.\,\langle \mathbf{d}' \rangle e_1{}' & =_{induction} \\
\langle \mathbf{c} \rangle \Lambda\alpha\,.\,\langle \mathbf{d}'' \rangle e_1{}'' & =_{15} \\
\langle \mathbf{c} \rangle \langle \forall\alpha\,.\,\mathbf{d}'' \rangle \Lambda\alpha\,.\,e_1{}'' & =_{12} \\
\langle (\forall\alpha\,.\,\mathbf{d}'')\,;\mathbf{c} \rangle \Lambda\alpha\,.\,e_1{}'' & =_{\phi,\psi,C} \\
\langle \mathbf{c}'' \rangle \Lambda\alpha\,.\,e_1{}'' & \equiv \\
e'' &
\end{array}$$

Inductive case: $|e'| \equiv \lambda x\!:\!\tau_x\,.\,e_1$.

Then $e' \equiv \langle \mathbf{c}' \rangle \lambda x\!:\!\rho'\,.\,e_1{}'$ and $e'' \equiv \langle \mathbf{c}'' \rangle \lambda x\!:\!\rho''\,.\,e_1{}''$ for some $\mathbf{c}'$, ρ', $e_1{}'$, $\mathbf{c}''$, ρ'' and $e_1{}''$.

Since $\mathbf{c}'$ has type signature $\rho' \rightarrow \rho_1{}' \rightsquigarrow \rho$ and $\mathbf{c}''$ has type signature $\rho'' \rightarrow \rho_1{}'' \rightsquigarrow \rho$ where $\rho_1{}'$ is the type of $e_1{}'$ and $\rho_1{}''$ is the type of $e_1{}''$ there exist coercions $\mathbf{c}_1{}'$, $\mathbf{c}_2{}'$, $\mathbf{c}_1{}''$, $\mathbf{c}_2{}''$ and $\mathbf{c}$ such that $\phi\psi \vdash \mathbf{c}' = \mathbf{c}_1{}' \rightarrow \mathbf{c}_2{}'\,;\mathbf{c}$ and $\phi\psi \vdash \mathbf{c}'' = \mathbf{c}_1{}'' \rightarrow \mathbf{c}_2{}''\,;\mathbf{c}$. In fact, $\mathbf{c}$ is either $\mathtt{box}_\upsilon$ for some unboxed type υ of ρ is a boxed type or ι otherwise. This means that we have

$$\begin{array}{ll}
e' & \equiv \\
\langle \mathbf{c}' \rangle \lambda x : \rho' \,.\, {e_1}' & =_{\phi,\psi,C} \\
\langle {\mathbf{c}_1}' \to {\mathbf{c}_2}' \,;\, \mathbf{c} \rangle \lambda x : \rho' \,.\, {e_1}' & =_{12} \\
\langle \mathbf{c} \rangle \langle {\mathbf{c}_1}' \to {\mathbf{c}_2}' \rangle \lambda x : \rho' \,.\, {e_1}' & =_{13} \\
\langle \mathbf{c} \rangle \lambda x : \rho_d \,.\, \langle {\mathbf{c}_2}' \rangle {e_1}'[\langle {\mathbf{c}_1}' \rangle x / x] & =_{induction} \\
\langle \mathbf{c} \rangle \lambda x : \rho_d \,.\, \langle {\mathbf{c}_2}'' \rangle {e_1}''[\langle {\mathbf{c}_1}'' \rangle x / x] & =_{13} \\
\langle \mathbf{c} \rangle \langle {\mathbf{c}_1}'' \to {\mathbf{c}_2}'' \rangle \lambda x : \rho'' \,.\, {e_1}'' & =_{12} \\
\langle {\mathbf{c}_1}'' \to {\mathbf{c}_2}'' \,;\, \mathbf{c} \rangle \lambda x : \rho'' \,.\, {e_1}'' & =_{\phi,\psi,C} \\
\langle \mathbf{c}'' \rangle \lambda x : \rho'' \,.\, {e_1}'' & \equiv \\
e'' &
\end{array}$$

for some type ρ_d which proves the case.

Inductive case: $|e'| \equiv e_1\ e_2$.

Then $e' \equiv \langle \mathbf{c}' \rangle ({e_1}'\ {e_2}')$ and $e'' \equiv \langle \mathbf{c}'' \rangle ({e_1}''\ {e_2}'')$ for some $\mathbf{c}'$, ${e_1}'$, ${e_2}'$, $\mathbf{c}''$, ${e_1}''$ and ${e_2}''$. We know from Proposition 2.2 that there exist a coercion $\mathbf{c}$ such that $\langle \mathbf{c} \rangle {e_2}'$ and ${e_2}''$ have the same type. Let $\mathbf{c}^{-1}$ be some arbitrarily chosen inverse coercion of $\mathbf{c}$. We then have, under $E\phi\psi$-equality:

$$\begin{array}{ll}
e' & \equiv \\
\langle \mathbf{c}' \rangle ({e_1}'\ {e_2}') & =_{14} \\
(\langle \iota \to \mathbf{c}' \rangle {e_1}')\ {e_2}' & =_{\phi,\psi,C} \\
(\langle (\mathbf{c} \,;\, \mathbf{c}^{-1}) \to \mathbf{c}' \rangle {e_1}')\ {e_2}' & =_{1,3,12} \\
(\langle \mathbf{c} \to \iota \rangle \langle \mathbf{c}^{-1} \to \mathbf{c}' \rangle {e_1}')\ {e_2}' & =_{14} \\
(\langle \mathbf{c}^{-1} \to \mathbf{c}' \rangle {e_1}')\ (\langle \mathbf{c} \rangle {e_2}') & =_{induction} \\
(\langle \mathbf{c}^{-1} \to \mathbf{c}' \rangle {e_1}')\ {e_2}'' & =_{induction} \\
(\langle \iota \to \mathbf{c}'' \rangle {e_1}'')\ {e_2}'' & =_{14} \\
\langle \mathbf{c}'' \rangle ({e_1}''\ {e_2}'') & \equiv \\
e'' &
\end{array}$$

This proves the case and concludes the proof.

■

4.1.3 Induced coercions and $\beta\eta$-equality

It is quite illustrative to consider the connection between induced coercions and $\beta\eta$-equality in F_2 (here we use a little of the calculus of F_2).

Let us first consider the two equations 13 and 14. We could consider either of these as defining induced function coercions. Take for example equation 14:

$$(\langle \mathbf{c} \to \mathbf{d} \rangle e)\ e' = \langle \mathbf{d} \rangle (e\ (\langle \mathbf{c} \rangle e'))$$

If we take this as the defining equation for function coercions, then by using $\beta\eta$-equality we can show that equation 13 must hold:

$$\langle \mathbf{c} \to \mathbf{d} \rangle \lambda x \,.\, e =_{\eta}$$

$$\lambda x \,.\, (\langle \mathbf{c} \to \mathbf{d} \rangle \lambda x \,.\, e)\ x =_{14}$$

$$\lambda x\,.\,\langle\mathbf{d}\rangle((\lambda x\,.e)\ \langle\mathbf{c}\rangle x)$$
$$\lambda x\,.\,\langle\mathbf{d}\rangle(e[\langle\mathbf{c}\rangle x/x])$$

Similarly we could have taken equation 13 as the defining equation for function coercions and then proven that equation 14 holds:

$$(\langle\mathbf{c}\to\mathbf{d}\rangle e)\ e' =_\eta$$
$$(\langle\mathbf{c}\to\mathbf{d}\rangle(\lambda x\,.e\ x))\ e' =_{13}$$
$$(\lambda x\,.\,\langle\mathbf{d}\rangle(e\ \langle\mathbf{c}\rangle x))\ e' =_\beta$$
$$\langle\mathbf{d}\rangle(e\ (\langle\mathbf{c}\rangle e'))$$

η-conversion is of course not safe in general for functional languages because of non-termination, but holds for F_2. Besides that, we are only using η-conversion as a tool to show equality and the resulting equality may of course still hold in languages where η-conversion is unsafe.

We may also show Equations 15 and 16 of Figure 12 equivalent by using η_t, β_t (η_t and β_t-conversion for type abstractions) and α-conversion since:

$$\langle\forall\alpha\,.\,\mathbf{c}\rangle\Lambda\alpha\,.\,e =_\eta$$
$$\Lambda\alpha'\,.\,(\langle\forall\alpha\,.\,\mathbf{c}\rangle\Lambda\alpha\,.e)\{\alpha'\} =_{16}$$
$$\Lambda\alpha'\,.\,\langle\mathbf{c}[\alpha'/\alpha]\rangle((\Lambda\alpha\,.e)\{\alpha'\}) =_{\alpha,\beta_t}$$
$$\Lambda\alpha\,.\,\langle\mathbf{c}\rangle e$$

and

$$(\langle\forall\alpha\,.\,\mathbf{c}\rangle e)\{\pi\} =_\eta$$
$$(\langle\forall\alpha\,.\,\mathbf{c}\rangle(\Lambda\alpha\,.e\{\alpha\}))\{\pi\} =_{15}$$
$$(\Lambda\alpha\,.\,\langle\mathbf{c}\rangle(e\{\alpha\}))\{\pi\} =_{\beta_t}$$
$$\langle\mathbf{c}[\pi/\alpha]\rangle(e\{\pi\})$$

Our theory does not contain $\beta\eta$-equality because we want to use the theory to prove equality between congruent completions and $\beta\eta$-equality does not equate congruent completions, since congruent completions must have the same underlying F_2-expression. But as we showed above the equations in Figure 12 are exactly enough to do this.

4.1.4 Parametricity and Equation 17

Equation 17 of Figure 12 may seem quite strange at first, and it might therefore be enlightening to see its connection to *parametricity* or what Wadler has termed "free theorems" (see Reynolds [Rey83] and Wadler [Wad89]). We will not prove this connection in general, but give an example. Consider the identity function `id` of type $\forall\alpha.\alpha\to\alpha$. If we follow Wadler and derive a "free theorem" we get:

$$c \circ \mathtt{id}_\pi = \mathtt{id}_{\pi'} \circ c$$

where c is a function of type $\pi\to\pi'$. If we regard c as a representation coercion and introduce our notation for coercion application and type application we get

$$\langle \iota \rightarrow c\rangle \texttt{id}\{\pi\} = \langle c \rightarrow \iota\rangle \texttt{id}\{\pi'\}$$

which is identical to the instance of Equation 17 for an expression with the type of the identity function. On the basis of this kind of observation we conjecture that it should be possible to deduce Equation 17 from parametricity.

There is an interesting relation between our Equation 17 and the Axiom C described by Longo, Milsted and Soloviev [LMS93]:

$$(\textbf{Axiom C}) \qquad e\{\tau_1\} = e\{\tau_2\} \;\text{ for } \Gamma\vdash e{:}\forall\alpha.\tau \text{ and } \alpha \notin FV(\tau)$$

If we were to consider coercions as simply F_2-terms then we could change Equation 17 into an equation involving only F_2 syntax. If we defined the pattern $\widehat{\tau}(e_1,e_2)$ on F_2-terms in a similar way to operation on coercions. This will give us the F_2 version of Equation 17:

$$\frac{\Gamma\vdash e{:}\forall\alpha.\tau \quad \Gamma\vdash e_1{:}\tau_1\rightarrow\tau_2}{\widehat{\tau}(e_1,\iota_{\tau_1})(e\{\tau_1\}) = \widehat{\tau}(\iota_{\tau_2},e_1)(e\{\tau_2\})}$$

where ι_τ here is the identity function on type τ. For all types $\tau_1\rightarrow\tau_2$ for which there exist an F_2-expression of that type we have that Axiom C is then a special case of this equation since if $\alpha\notin FV(\tau)$ then $\widehat{\tau}(e_1,\iota_{\tau_1})$ will be $\beta\eta$-equivalent to ι_τ and $\widehat{\tau}(\iota_{\tau_2},e_1)$ will be $\beta\eta$-equivalent to ι_τ and we will get:

$$e\{\tau_1\} = e\{\tau_2\}$$

from the new equation. Longo, Milsted and Soloviev [LMS93] shows that Axiom C is not provable in F_2, but is realized by all models that satisfy Reynolds's parametricity condition. This clearly indicates the relation between our Equation 17 and parametricity. Despite the simple appearance of Axiom C some rather powerful results may be derived from it, but since Axiom C also holds for types $\tau_1\rightarrow\tau_2$ where there exist no F_2-expression of that types it is not clear whether our new equation also can be used to derive similar powerful results. In any case it would be interesting to study the system that one gets from adding the new equation and/or Axiom C to F_2.

4.2 Completion reduction

In Chapter 3 we demonstrated that optimal coercions exist for all signatures. We did this by showing how to implement $\phi\psi$-reduction on coercions by R-reduction. The idea in $\phi\psi$-reduction on coercions is to regard ϕ and ψ as left-to-right rewrite rule modulo coercion equality. Using the basic assumption that boxing and unboxing coercions are more expensive than the identity coercion we argued that reduction will in general improve performance of coercions. It is natural to try an extend $\phi\psi$-reduction to completions and define this in the following way:

Definition 4.9 *($\phi\psi$-reduction)*
Let ϕ and ψ stand for the ϕ and ψ rules regarded as left-to-right rewrite rules. We define $\phi\psi$-reduction on completions to be $\phi\psi$-reduction under E-equality. Remember that E-equality always includes the core equations C from Figure 6.

Defined in this way $\phi\psi$-reduction will at least improve performance of the coercions in completions and we could define what we mean by (formally) optimal completion, in a similar way to optimal coercions, as completions that all other congruent completions $\phi\psi$-reduce to.

Unfortunately, $\phi\psi$-reduction on E-congruence classes is not (equational) Church-Rosser; that is, there are congruent completions that have no common reduct. Consider, for example, the two completions below (in order to give a more readable example we have included two additional type constructors `int` and `bool`, two external primitives `+` and `if` and a few constants `true`, `2` and `5`. The effect of adding such language extensions to F_2 will be discussed in details in Chapter 6. The equational rules needed for the new languages constructs are quite intuitive and will not be explained here. One can give a pure F_2 example, but this will be considerably larger and much more opaque.):

$$\begin{aligned} e_1 \equiv\ & (\lambda id{:}\forall\alpha.\alpha{\to}\alpha\,.\,(\lambda x{:}\texttt{int}.x + \langle\texttt{unbox}\rangle(id\{[\texttt{int}]\}\ \langle\texttt{box}\rangle x)) \\ & \qquad\qquad (\texttt{if true then 2 else } \langle\texttt{unbox}\rangle(id\{[\texttt{int}]\}\ \langle\texttt{box}\rangle \texttt{5}))) \\ & (\Lambda\alpha\,.\,\lambda y{:}\alpha\,.\,y) \end{aligned}$$

$$\begin{aligned} e_2 \equiv\ & (\lambda id{:}\forall\alpha.\alpha{\to}\alpha\,.\,(\lambda x{:}[\texttt{int}].\langle\texttt{unbox}\rangle x + \langle\texttt{unbox}\rangle(id\{[\texttt{int}]\}\ x)) \\ & \qquad\qquad (\texttt{if true then } \langle\texttt{box}\rangle\texttt{2 else } (id\{[\texttt{int}]\}\ \langle\texttt{box}\rangle \texttt{5}))) \\ & (\Lambda\alpha\,.\,\lambda y{:}\alpha\,.\,y) \end{aligned}$$

Neither one of them is reducible to the other by $\phi\psi$-reduction modulo E (they are in fact both in $\phi\psi$-normal form).

4.2.1 $\phi^{\to}$-reduction modulo $E\psi$ and $\psi^{\to}$-reduction modulo $E\phi$

So from the example in the previous subsection it would seem that it is impossible to come up with a notion of optimal completions based on $\phi\psi$-reduction. Using only ϕ or ψ as a reduction rule, reduction modulo E-equality would not help either as the example clearly shows. Examining the example in the previous subsection we see that the main difference between e_1 and e_2 is the representation type of x. In e_1 it is unboxed whereas in e_2 it is boxed. By introducing a `box;unbox`-pair in front of the constant 2 in e_1 we can ψ-reduce e_1 to e_2 (modulo E). Conversely, by introducing an `unbox;box`-pair in front of ($id\{[\texttt{int}]\}\ \langle\texttt{box}\rangle\texttt{5}$) in e_2 we can ϕ-reduce e_2 to e_1 (modulo E). Thus we can trade off a `box;unbox`-redex for an `unbox;box`-redex or visa versa. So only by giving higher priority to the elimination of one kind of redex than to the other we end up with a formal notion of optimality that entails that, for any given representation type, every source F_2-expression has an optimal completion that is unique modulo E-equality. We can do this by using one of the rules ϕ or ψ as a reduction rule and the other as an equality rule. This motivates the following definition:

Definition 4.10 *($\phi^{\to}$-reduction modulo $E\psi$ and $\psi^{\to}$-reduction modulo $E\phi$)* Let $\phi^{\to}$ and $\psi^{\to}$ stand for the ϕ and ψ rules regarded as left-to-right rewrite rules. We define *$\phi^{\to}$-reduction modulo $E\psi$* on completions to be $\phi^{\to}$-reduction under $E\psi$-equality. Similarly, we define *$\psi^{\to}$-reduction modulo $E\phi$* on completions to be $\psi^{\to}$-reduction under $E\phi$-equality.

Let us now give an example of $\psi^{\to}$-reduction modulo $E\phi$:

Example 4.2 *($\psi^{\to}$-reduction modulo $E\phi$)* Let us show how the completion e_1 (from Section 4.2) may be reduced to e_2 (also from Section 4.2) by $\psi^{\to}$-reduction modulo $E\phi$. We will use the following rule for `if`-expressions:

$$\langle\mathbf{c}\rangle\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 = \texttt{if } e_1 \texttt{ then } \langle\mathbf{c}\rangle e_2 \texttt{ else } \langle\mathbf{c}\rangle e_3 \quad (\mathit{if})$$

which will be explained in Chapter 6 Subsection 6.2.1. The reduction sequence is as follows:

$$
\begin{array}{rl}
e_1 \equiv & (\lambda \mathit{id} : \forall\alpha.\alpha\to\alpha\,.\,(\lambda x : \texttt{int}.x \;+\; \langle\texttt{unbox}\rangle(\mathit{id}\{[\texttt{int}]\}\ \langle\texttt{box}\rangle x)) \\
& \qquad\qquad\qquad (\texttt{if true then 2 else } \langle\texttt{unbox}\rangle(\mathit{id}\{[\texttt{int}]\}\ \langle\texttt{box}\rangle\texttt{5}))) \\
& (\Lambda\alpha\,.\,\lambda \mathrm{y}\!: \alpha\,.\,\mathrm{y}) \\
=_{\phi} & (\lambda \mathit{id} : \forall\alpha.\alpha\to\alpha\,.\,(\lambda x : \texttt{int}.x \;+\; \langle\texttt{unbox}\rangle(\mathit{id}\{[\texttt{int}]\}\ \langle\texttt{box}\rangle x)) \\
& \qquad\qquad\qquad (\texttt{if true then } \langle\texttt{box;unbox}\rangle\texttt{2 else } \langle\texttt{unbox}\rangle(\mathit{id}\{[\texttt{int}]\}\ \langle\texttt{box}\rangle\texttt{5}))) \\
& (\Lambda\alpha\,.\,\lambda \mathrm{y}\!: \alpha\,.\,\mathrm{y}) \\
=_{12} & (\lambda \mathit{id} : \forall\alpha.\alpha\to\alpha\,.\,(\lambda x : \texttt{int}.x \;+\; \langle\texttt{unbox}\rangle(\mathit{id}\{[\texttt{int}]\}\ \langle\texttt{box}\rangle x)) \\
& \qquad\qquad\qquad (\texttt{if true then } \langle\texttt{unbox}\rangle\langle\texttt{box}\rangle\texttt{2 else } \langle\texttt{unbox}\rangle(\mathit{id}\{[\texttt{int}]\}\ \langle\texttt{box}\rangle\texttt{5}))) \\
& (\Lambda\alpha\,.\,\lambda \mathrm{y}\!: \alpha\,.\,\mathrm{y}) \\
=_{\mathit{if}} & (\lambda \mathit{id} : \forall\alpha.\alpha\to\alpha\,.\,(\lambda x : \texttt{int}.x \;+\; \langle\texttt{unbox}\rangle(\mathit{id}\{[\texttt{int}]\}\ \langle\texttt{box}\rangle x)) \\
& \qquad\qquad\qquad \langle\texttt{unbox}\rangle(\texttt{if true then } \langle\texttt{box}\rangle\texttt{2 else } (\mathit{id}\{[\texttt{int}]\}\ \langle\texttt{box}\rangle\texttt{5}))) \\
& (\Lambda\alpha\,.\,\lambda \mathrm{y}\!: \alpha\,.\,\mathrm{y}) \\
=_{14} & (\lambda \mathit{id} : \forall\alpha.\alpha\to\alpha\,.\,\langle\texttt{unbox}\to\iota\rangle(\lambda x : \texttt{int}.x \;+\; \langle\texttt{unbox}\rangle(\mathit{id}\{[\texttt{int}]\}\ \langle\texttt{box}\rangle x)) \\
& \qquad\qquad\qquad (\texttt{if true then } \langle\texttt{box}\rangle\texttt{2 else } (\mathit{id}\{[\texttt{int}]\}\ \langle\texttt{box}\rangle\texttt{5}))) \\
& (\Lambda\alpha\,.\,\lambda \mathrm{y}\!: \alpha\,.\,\mathrm{y}) \\
=_{13} & (\lambda \mathit{id} : \forall\alpha.\alpha\to\alpha\,.\,(\lambda x : \texttt{int}.\langle\texttt{unbox}\rangle x \;+\; \langle\texttt{unbox}\rangle(\mathit{id}\{[\texttt{int}]\}\ \langle\texttt{box}\rangle\langle\texttt{unbox}\rangle x)) \\
& \qquad\qquad\qquad (\texttt{if true then } \langle\texttt{box}\rangle\texttt{2 else } (\mathit{id}\{[\texttt{int}]\}\ \langle\texttt{box}\rangle\texttt{5}))) \\
& (\Lambda\alpha\,.\,\lambda \mathrm{y}\!: \alpha\,.\,\mathrm{y}) \\
=_{12} & (\lambda \mathit{id} : \forall\alpha.\alpha\to\alpha\,.\,(\lambda x : \texttt{int}.\langle\texttt{unbox}\rangle x \;+\; \langle\texttt{unbox}\rangle(\mathit{id}\{[\texttt{int}]\}\ \langle\texttt{unbox;box}\rangle x)) \\
& \qquad\qquad\qquad (\texttt{if true then } \langle\texttt{box}\rangle\texttt{2 else } (\mathit{id}\{[\texttt{int}]\}\ \langle\texttt{box}\rangle\texttt{5}))) \\
& (\Lambda\alpha\,.\,\lambda \mathrm{y}\!: \alpha\,.\,\mathrm{y}) \\
\Rightarrow_{\psi} & (\lambda \mathit{id} : \forall\alpha.\alpha\to\alpha\,.\,(\lambda x : \texttt{int}.\langle\texttt{unbox}\rangle x \;+\; \langle\texttt{unbox}\rangle(\mathit{id}\{[\texttt{int}]\}\ x)) \\
& \qquad\qquad\qquad (\texttt{if true then } \langle\texttt{box}\rangle\texttt{2 else } (\mathit{id}\{[\texttt{int}]\}\ \langle\texttt{box}\rangle\texttt{5}))) \\
& (\Lambda\alpha\,.\,\lambda \mathrm{y}\!: \alpha\,.\,\mathrm{y}) \\
\equiv & e_2
\end{array}
$$

The example above shows that by introducing a redex of one kind (say ϕ) we can eliminate a redex of the other kind (ψ). This is an improvement if redexes of the second kind are considered arbitrarily more expensive than redexes of the first kind. But it is not obvious which of the two kinds of redexes should be considered more expensive. Thus we shall pursue two different notions of optimality. In the first we get rid of all ψ-redexes first — even at the cost of introducing additional ϕ-redexes — and then getting rid of all ϕ-redexes without letting ψ-redexes slip back in. In the second we, dually, get rid of all ϕ-redexes first, possibly introducing new ψ-redexes, and then eliminate all ψ-redexes without readmitting ϕ-redexes.

4.2.2 $\phi^{\to}$-reduction modulo E and $\psi^{\to}$-reduction modulo E

To get rid of ϕ-redexes without letting ψ-redexes slip back in we need $\phi^{\to}$-reduction modulo E and analogously to get rid of ψ-redexes without letting ϕ-redexes slip back in we need $\psi^{\to}$-reduction modulo E. So we define this now:

Definition 4.11 *($\phi^{\to}$-reduction modulo E and $\psi^{\to}$-reduction modulo E)*
Let $\phi^{\to}$ and $\psi^{\to}$ stand for the ϕ and ψ rules regarded as left-to-right rewrite rules. We define *$\phi^{\to}$-reduction modulo E* on completions to be $\phi^{\to}$-reduction under E-equality. Similarly, we define *$\psi^{\to}$-reduction modulo E* on completions to be $\psi^{\to}$-reduction under E-equality.

We will introduce the following shorthand notation R/E-reduction or just R/E for R-reduction modulo E where R is a set of rewrite rules and E a set of equations, e.g. we will write $\phi^{\rightarrow}/E\psi$-reduction for $\phi^{\rightarrow}$-reduction modulo $E\psi$.

In Section 4.4 below we will show how to implement $\phi^{\rightarrow}/E\psi$-reduction, $\phi^{\rightarrow}/E$-reduction, $\psi^{\rightarrow}/E\phi$-reduction and $\psi^{\rightarrow}/E$-reduction and then in Section 4.5 we will show how to define the notion of optimal completions using these reduction systems.

4.3 Polarized equational theory

What we have developed so far is an equational theory for completions which enables us to prove all congruent completions for a given F_2-expression equal. What we would like to do is to use this to do $\phi^{\rightarrow}/E\psi$-reduction, $\phi^{\rightarrow}/E$-reduction, $\psi^{\rightarrow}/E\phi$-reduction and $\psi^{\rightarrow}/E$-reduction. But since it is difficult to reason directly about reduction systems on congruence classes defined by an equational theory what we would actually like to do is to characterize E-equality by a canonical term rewriting system that commutes with ψ-reduction and ϕ-reduction such that we can use this to implement $\phi^{\rightarrow}/E\psi$-reduction and $\psi^{\rightarrow}/E\phi$-reduction. Finding a confluent rewriting system for E-equality is not straightforward, however. In particular, the E-equations cannot simply be oriented in one or the other direction, since they will inevitably lead to critical pairs without common reducts. Consider for example the rules of Figure 12 oriented from left to right. In the expression

$$(\langle \mathbf{c} \rightarrow \mathbf{d} \rangle \lambda x \,.\, e)\ e'$$

both rules 13 and 14 are applicable, and Knuth-Bendix completion appears not to terminate. Note that by following one reduction path we might fail to eliminate a box/unbox pair using ϕ or ψ that could be eliminated by following the other path.

In this section we develop a new equational theory that is equivalent to E-equality, but which can easily be oriented in one or the other direction to yield a confluent rewriting system together with ϕ respectively ψ-reduction.

4.3.1 Polarized equational theory

We now define our new axiomatization of completion congruence which takes the polarity of coercions into account. The main idea is to take each rule of Figure 12 and turn it into two rules in which the coercions have been assigned different polarity.

Let us first, as an example, consider rule 13 of Figure 12. This will be turned into the two rules:

$$\langle (\mathbf{c}^{+} \rightarrow \mathbf{d}^{-}) \rangle \lambda x \,.\, e = \lambda x \,.\, \langle \mathbf{d}^{-} \rangle (e[\langle \mathbf{c}^{+} \rangle x / x]) \quad (13^{-})$$

$$\lambda x \,.\, \langle \mathbf{d}^{+} \rangle (e[\langle \mathbf{c}^{-} \rangle x / x]) = \langle (\mathbf{c}^{-} \rightarrow \mathbf{d}^{+}) \rangle \lambda x \,.\, e \quad (13^{+})$$

The two rules are instances of Equation 13 of E, so obviously everything that we can prove with these rules (and the core coercion equations) we can prove with Equation 13 (and the core coercion equations). That the opposite also holds, in the sense that anything that one can prove with rule 13 (and the core coercion equations) can also be proven with the two new rules (and the core coercion equations), will be shown in Theorem 4.13 below.

Why the rules are oriented as they are, has to do with what happens when the rules are regarded as reduction rules. The pragmatic reason is: because it works. If the rules are oriented as they are then we shall see (in Lemma 4.15) that the reduction system becomes confluent. The more intuitive explanation is this: we want positive coercions to move forward along the value flow paths in a program and the negative coercions to move backwards. In this way positive and negative coercions move towards each other and may eventually meet and cancel out (remember that `box`- and unboxc-coercions have diferent polarity).

As an example of how the rules move coercions in this way consider rule 13^- above. In this rule the positive coercion $\mathbf{c}$ moves from a position where it is applied to the argument of the lambda expression to positions where it is applied to all the occurrences of the bound variable x in e, i.e. the coercion "moves" in the same direction as the argument. Similarly coercion $\mathbf{d}$, which is negative, moves from a position where it is applied to the result of the lambda expression to a position where it is applied to the body of the lambda expression. This might not seem as if this corresponds to any real movement of the coercion backwards along the flow path, but it is certainly necessary if the coercion is to move further backwards. This should give some intuition as to why the two rules above are oriented as they are.

$$\begin{array}{rcll}
\langle(\mathbf{c}^+ \to \mathbf{d}^-)\rangle\lambda x\,.\,e &=& \lambda x\,.\langle\mathbf{d}^-\rangle(e[\langle\mathbf{c}^+\rangle x/x]) & (13^-)\\
\lambda x\,.\langle\mathbf{d}^+\rangle(e[\langle\mathbf{c}^-\rangle x/x]) &=& \langle(\mathbf{c}^- \to \mathbf{d}^+)\rangle\lambda x\,.\,e & (13^+)\\[1ex]
\langle\mathbf{d}^-\rangle(e\ (\langle\mathbf{c}^+\rangle e')) &=& (\langle\mathbf{c}^+ \to \mathbf{d}^-\rangle e)\ e' & (14^-)\\
(\langle(\mathbf{c}^- \to \mathbf{d}^+)\rangle e)\ e' &=& \langle\mathbf{d}^+\rangle(e\ (\langle\mathbf{c}^-\rangle e')) & (14^+)\\[1ex]
\langle\forall\alpha\,.\,\mathbf{c}^-\rangle\Lambda\alpha\,.\,e &=& \Lambda\alpha\,.\langle\mathbf{c}^-\rangle e & (15^-)\\
\Lambda\alpha\,.\langle\mathbf{c}^+\rangle e &=& \langle\forall\alpha\,.\,\mathbf{c}^+\rangle\Lambda\alpha\,.\,e & (15^+)\\[1ex]
\langle\mathbf{c}^-[\pi/\alpha]\rangle e\{\pi\} &=& (\langle\forall\alpha\,.\,\mathbf{c}^-\rangle e)\{\pi\} & (16^-)\\
(\langle\forall\alpha\,.\,\mathbf{c}^+\rangle e)\{\pi\} &=& \langle\mathbf{c}^+[\pi/\alpha]\rangle e\{\pi\} & (16^+)\\[1ex]
\langle\widehat{\rho}(\iota,\mathbf{c}^-)\rangle e\{\pi\} &=& \langle\widehat{\rho}(\mathbf{c}^-,\iota)\rangle e\{\pi'\} & (17^-)\\
\langle\widehat{\rho}(\mathbf{c}^+,\iota)\rangle e\{\pi\} &=& \langle\widehat{\rho}(\iota,\mathbf{c}^+)\rangle e\{\pi'\} & (17^+)\\
\multicolumn{4}{l}{\text{Side condition: in rules } 17^- \text{ and } 17^+ \text{ the type of } e \text{ is } \forall\alpha.\rho.}
\end{array}$$

Figure 13: Polarized equality equations for explicitly boxed expressions (E_p)

The equations of the new axiomatization, called E_p, are shown in Figure 13. These equations are those of Figure 12 where all equation have been split into two polarized equations with side conditions on the *polarity* of the coercions occurring in them. We will discuss the last two rules of Figure 13 below (Subsection 4.3.2), where we also explain the notation use in these two rules.

But let us first consider why the new system solves the confluence problem above. Assume that we regard the rules of Figure 13 as left-to-right rewrite rules and we want to rewrite:

$$(\langle\mathbf{c} \to \mathbf{d}\rangle\lambda x\,.\,e)\ e'$$

We cannot always expect to be able to use one of the rules 13^-, 13^+, 14^- or 14^+ immediately. For the rules to be applicable the coercions must have the right polarity. Let us first assume that $\mathbf{c}$ is positive and $\mathbf{d}$ is negative. In that case only rule 13^- is applicable and there is no confluence problem anymore. Then let us assume that the coercions have the reversed polarity, then again only one rule, namely 14^+, is applicable and there is no confluence problem in this case either. If one or more of the coercions are neither positive nor negative or the coercion just does not have the right polarity then none of the rules are applicable directly.

So the confluence problem of the rules in E is not present in E_p. One might think that since the rules are only applicable when the coercions can be assigned polarities then the rules might be too weak, but this is not the case as we will show below. The reason is that any coercion can be split into a composition of two polarized coercions which when combined with the rules of Figure 11 can be used to bring about redexes that fit the rules 13^-, 13^+, 14^- and 14^+. For the example above this means that we may rewrite it to:

$$(\langle\langle \mathbf{c}_1{}^- \to \mathbf{d}_2{}^+\rangle\langle \mathbf{c}_2{}^+ \to \mathbf{d}_1{}^-\rangle\lambda x\,.\,e\rangle)\ e'$$

where $\phi\psi \vdash \mathbf{c} = \mathbf{c}_1{}^-;\mathbf{c}_2{}^+$ and $\phi\psi \vdash \mathbf{d} = \mathbf{d}_1{}^-;\mathbf{d}_2{}^+$. In this case both rule 13^- and 14^+ may be applicable after the splitting, but not involving the same coercions.

4.3.2 The polarized versions of rule 17

In this subsection we will give an explanation of the polarized versions of rule 17 from Figure 12. In ML-typable languages, when we do type inference of an expression e containing a let-bound variable x all we need to know about x is its type. We do not need to know the expression e' to which the variable x is bound, because the type contains all the relevant information. The type we infer for e is exactly the same as we would get had we replaced x by e'.

Similar to this, when rewriting an expression $\langle\mathbf{c}\rangle x\{\pi\}$ we could hope that all the relevant information we need about x is its type and what we want is that the result we get is as we would get had we replaced x with any of its potential actual arguments. Let us look at an example:

$$\langle \mathbf{c}^+ \to \iota\rangle(id\{\pi\})$$

where id is the polymorphic identity function with type $\forall\alpha.\alpha\to\alpha$ (in fact id is the only function of that type). Suppose that we insert the usual definition of id in the expression above:

$$\langle \mathbf{c}^+ \to \iota\rangle((\Lambda\alpha\,.\,\lambda x : \alpha\,.\,x)\{\pi\})$$

If we perform the type application we get:

$$\langle \mathbf{c}^+ \to \iota\rangle\lambda x : \pi\,.\,x$$

which we may rewrite to

$$\langle \iota \to \mathbf{c}^+\rangle\lambda x : \pi'\,.\,x$$

using the rules of Figure 13 except 17^- and 17^+ (in fact only 13^- and 13^+.) From this it would seem natural that we should be able to rewrite the original expression to:

$$\langle \iota \to \mathbf{c}^+\rangle(id\{\pi'\})$$

So the polarized versions of rule 17 have to be able to do this, in the case where we only know the type of an expression, and not the actual expression. That such a rule is also applicable in the case where we know the expression is fine since this is in fact necessary because we will not allow the use of β-conversion for types.

Now we know what the polarized versions of rule 17 have to be able to do, so now we will explain how they do it.

We will call a negative occurrence of a type variable α in a type τ an *input occurrence* and a positive occurrence an *output occurrence*.

To explain the terms input and output occurrence, let us look at an example of a simple polymorphic function (apply):

$$h = \Lambda\alpha\,.\,\Lambda\alpha'\,.\,\lambda f : \alpha \to \alpha'\,.\,\lambda x : \alpha\,.\,f\ x$$

this function has type:

$$\forall\alpha.\forall\alpha'.(\alpha \to \alpha') \to \alpha \to \alpha'$$

Let us forget about the type abstractions and concentrate on the type:

$$(\alpha \to \alpha') \to \alpha \to \alpha'$$

Looking at this type it seems reasonable to call the second occurrence of α an input occurrence, since it corresponds to the type of an input to the function and to call the second occurrence of α' is an output occurrence, since it corresponds to the type of the output from the function. But what about the rest of the occurrences in the type? If we see things from the polymorphic functions point of view then f with type $\alpha\to\alpha'$ is something external which has to be passed a value of type α and which will then return a value of type α'. This means that the input to f is output from the function h and the output from f is input to the function h. If we generalize this then positive occurrences of type variable are output occurrences and negative occurrences of type variable are input occurrences. Intuitively, all a polymorphic function can do with values corresponding to input occurrences is to pass the values to the output (maybe in multiple copies) or throw the values away. Therefore, if all input occurrences corresponding to a given type variable are being coerced with the same coercion then we might as well coerce all the output occurrence corresponding to that type variable with the coercion instead.

This explains the definition of rules 17^- and 17^+:

$$\langle\widehat{\rho}(\iota, \mathbf{c}^-)\rangle e\{\pi\} = \langle\widehat{\rho}(\mathbf{c}^-, \iota)\rangle e\{\pi'\} \quad (17^-)$$

$$\langle\widehat{\rho}(\mathbf{c}^+, \iota)\rangle e\{\pi\} = \langle\widehat{\rho}(\iota, \mathbf{c}^+)\rangle e\{\pi'\} \quad (17^+)$$

since e has type $\forall\alpha.\rho$ and $\widehat{\rho}(c,d)$ is a coercion that has the same structure as the type ρ and where all negative occurrences of α has been replaced by c and all positive occurrences of α by d. This means that the coercion $\widehat{\rho}(\iota,\mathbf{c}^-)$ on the left hand side of in rule 17^- will apply $\mathbf{c}$ to all output occurrences of α while the coercion $\widehat{\rho}(\mathbf{c}^-,\iota)$ on the right hand side of in rule 17^- will apply $\mathbf{c}$ to all input occurrences of α. Therefore, if rule 17^- is regarded as a left-to-right rewrite rule it moves the negative coercion $\mathbf{c}$ from the output to the input of e, i.e backwards and similarly rule 17^+ will moves the positive coercion $\mathbf{c}$ from the input to the output of e, i.e forwards.

4.3.3 Equivalence of E_p- and E-equivalence

Let E_p be the set of equations in Figure 13. We will now show that the polarized equality, i.e. E_p-equality, is equivalent to E-equality, and therefore of course also that $E_p\phi\psi$-equality is equivalent to $E\phi\psi$-equality. To show this we will use the following lemma:

Lemma 4.12 *Let* $\mathbf{c}_1$*,* $\mathbf{c}_2$*,* $\mathbf{d}_1$*, etc. be coercions,* $\forall\alpha.\rho$ *be a representation type and* $\widehat{\rho}(\mathbf{c},\mathbf{d})$ *by the parameterized coercion defined by Definition 4.1. Then the following hold:*

$$\vdash \widehat{\rho}(\mathbf{c}_1\,;\mathbf{c}_2, \mathbf{d}_1\,;\mathbf{d}_2) = \widehat{\rho}(\mathbf{c}_2, \mathbf{d}_1)\,;\widehat{\rho}(\mathbf{c}_1, \mathbf{d}_2)$$

Proof:

The proof is by induction on the structure of ρ.

Base case: $\rho \equiv \alpha$: $\widehat{\rho}(\mathbf{c}_1\,;\mathbf{c}_2, \mathbf{d}_1\,;\mathbf{d}_2) = \mathbf{d}_1\,;\mathbf{d}_2 = \widehat{\rho}(\mathbf{c}_2, \mathbf{d}_1); \widehat{\rho}(\mathbf{c}_1, \mathbf{d}_2)$

Base case: $\rho \equiv \alpha'$ $(\alpha \neq \alpha')$: $\widehat{\rho}(\mathbf{c}_1\,;\mathbf{c}_2, \mathbf{d}_1\,;\mathbf{d}_2) = \iota =_1 \iota; \iota = \widehat{\rho}(\mathbf{c}_2, \mathbf{d}_1); \widehat{\rho}(\mathbf{c}_1, \mathbf{d}_2)$

Inductive case: $\rho \equiv \rho_1 \rightarrow \rho_2$:

$$\begin{array}{ll}
\widehat{\rho}(\mathbf{c}_1\,;\mathbf{c}_2, \mathbf{d}_1\,;\mathbf{d}_2) & = \\
\widehat{\rho_1}(\mathbf{d}_1\,;\mathbf{d}_2, \mathbf{c}_1\,;\mathbf{c}_2) \rightarrow \widehat{\rho_2}(\mathbf{c}_1\,;\mathbf{c}_2, \mathbf{d}_1\,;\mathbf{d}_2) & =_{induction} \\
\widehat{\rho_1}(\mathbf{d}_2, \mathbf{c}_1); \widehat{\rho_1}(\mathbf{d}_1, \mathbf{c}_2) \rightarrow \widehat{\rho_2}(\mathbf{c}_2, \mathbf{d}_1); \widehat{\rho_2}(\mathbf{c}_1, \mathbf{d}_2) & =_3 \\
(\widehat{\rho_1}(\mathbf{d}_1, \mathbf{c}_2) \rightarrow \widehat{\rho_2}(\mathbf{c}_2, \mathbf{d}_1)); (\widehat{\rho_1}(\mathbf{d}_2, \mathbf{c}_1) \rightarrow \widehat{\rho_2}(\mathbf{c}_1, \mathbf{d}_2)) & = \\
\widehat{\rho}(\mathbf{c}_2, \mathbf{d}_1); \widehat{\rho}(\mathbf{c}_1, \mathbf{d}_2) &
\end{array}$$

Inductive case: $\rho \equiv [\rho_1]$:

$$\begin{array}{ll}
\widehat{\rho}(\mathbf{c}_1\,;\mathbf{c}_2, \mathbf{d}_1\,;\mathbf{d}_2) & = \\
[\widehat{\rho_1}(\mathbf{c}_1\,;\mathbf{c}_2, \mathbf{d}_1\,;\mathbf{d}_2)] & =_{induction} \\
[\widehat{\rho_1}(\mathbf{c}_2, \mathbf{d}_1)\,;\widehat{\rho_1}(\mathbf{c}_1, \mathbf{d}_2)] & =_5 \\
[\widehat{\rho_1}(\mathbf{c}_2, \mathbf{d}_1)]\,;[\widehat{\rho_1}(\mathbf{c}_1, \mathbf{d}_2)] & = \\
\widehat{\rho}(\mathbf{c}_2, \mathbf{d}_1); \widehat{\rho}(\mathbf{c}_1, \mathbf{d}_2) &
\end{array}$$

Inductive case: $\rho \equiv \forall\alpha.\rho_1$:

$$\begin{array}{ll}
\widehat{\rho}(\mathbf{c}_1\,;\mathbf{c}_2, \mathbf{d}_1\,;\mathbf{d}_2) & = \\
\forall\alpha'.\widehat{\rho_1}(\iota, \iota) & =_* \\
\iota & =_1 \\
\iota; \iota & =_* \\
\forall\alpha'.\widehat{\rho_1}(\iota, \iota); \forall\alpha'.\widehat{\rho_1}(\iota, \iota) & = \\
\widehat{\rho}(\mathbf{c}_2, \mathbf{d}_1); \widehat{\rho}(\mathbf{c}_1, \mathbf{d}_2) &
\end{array}$$

where the identities marked with a $*$ follows the simple property $\vdash \widehat{\rho}(\iota,\iota) = \iota$ which can easily be checked by induction on the structure of ρ.

Inductive case: $\rho \equiv \forall\alpha'.\rho_1$:

$$\begin{array}{ll}
\widehat{\rho}(\mathbf{c}_1\,;\mathbf{c}_2,\mathbf{d}_1\,;\mathbf{d}_2) & = \\
\forall\alpha'.\widehat{\rho_1}(\mathbf{c}_1\,;\mathbf{c}_2,\mathbf{d}_1\,;\mathbf{d}_2) & =_{induction} \\
\forall\alpha'.\widehat{\rho_1}(\mathbf{c}_2,\mathbf{d}_1)\,;\widehat{\rho_1}(\mathbf{c}_1,\mathbf{d}_2) & =_8 \\
\forall\alpha'.\widehat{\rho_1}(\mathbf{c}_2,\mathbf{d}_1)\,;\forall\alpha'.\widehat{\rho_1}(\mathbf{c}_1,\mathbf{d}_2) & = \\
\widehat{\rho}(\mathbf{c}_2,\mathbf{d}_1);\widehat{\rho}(\mathbf{c}_1,\mathbf{d}_2) &
\end{array}$$

■

We then prove the theorem:

Theorem 4.13 *For all completions e and e':*

$$E \vdash e = e' \quad \textit{iff} \quad E_p \vdash e = e'$$

Proof:
"Only if": This is trivial since all equations of E_p are instances of Equations of E.

"if": All cases except equation 17 are straightforward. The way we prove the theorem for the different equations is basically the same. We take an equation R from E. We then prime factorize all coercions on the left hand side of the equation. This can by Lemma 3.20 be done with only the core coercion equations which is part of E_p. These prime coercions are either positive or negative so we may use the corresponding equations R^+ and R^- to move these in the same direction as the original coercions were moved by the equation R. Finally, we compose the coercion using only the core coercion equations and get the original coercions back. We will show this for equations 14 and 17. First equation 14:

$$\begin{array}{ll}
(\langle \mathbf{c} \rightarrow \mathbf{d} \rangle e)\ e' & =_{\text{prim factoring}} \\
(\langle (\mathbf{c}_1;\ldots;\mathbf{c}_n) \rightarrow (\mathbf{d}_1;\ldots;\mathbf{d}_m) \rangle e)\ e' & =_{1,2,3} \\
(\langle \mathbf{c}_n \rightarrow \iota;\ldots;\mathbf{c}_1 \rightarrow \iota;\ \iota \rightarrow \mathbf{d}_1;\ldots;\iota \rightarrow \mathbf{d}_m \rangle e)\ e' & =_{12,14^-,14^+} \\
\langle \mathbf{d}_1;\ldots;\mathbf{d}_m \rangle (e\ (\langle \mathbf{c}_1;\ldots;\mathbf{c}_n \rangle e')) & =_{\text{prim factoring}} \\
\langle \mathbf{d} \rangle (e\ (\langle \mathbf{c} \rangle e')) &
\end{array}$$

Then we look at Equation 17. This is proven similar to the rest of the equations, except that we use Lemma 4.12 in the proof:

$$\begin{array}{ll}
\langle \widehat{\rho}(\mathbf{c},\iota_\pi) \rangle (e\{\pi\}) & =_{\text{prim factoring}} \\
\langle \widehat{\rho}(\mathbf{c}_1;\ldots;\mathbf{c}_n,\iota_\pi) \rangle (e\{\pi\}) & =_{\text{Lemma 4.12}} \\
\langle \widehat{\rho}(\mathbf{c}_n,\iota_\pi);\ldots;\ \widehat{\rho}(\mathbf{c}_1,\iota_\pi) \rangle (e\{\pi\}) & =_{12,17^-,17^+} \\
\langle \widehat{\rho}(\iota_{\pi'},\mathbf{c}_n);\ldots;\ \widehat{\rho}(\iota_{\pi'},\mathbf{c}_1) \rangle (e\{\pi'\}) & =_{\text{Lemma 4.12}} \\
\langle \widehat{\rho}(\iota_{\pi'},\mathbf{c}_1;\ldots;\mathbf{c}_n) \rangle (e\{\pi'\}) & =_{\text{prim factoring}} \\
\langle \widehat{\rho}(\iota_{\pi'},\mathbf{c}) \rangle (e\{\pi'\}) &
\end{array}$$

■

4.4 Polarized reduction

As explained the purpose of the polarized equation system is that we may use it to obtain two sets of polarized rewrite rules that we will use to implement $\phi/E\psi$- and $\psi/E\phi$-reduction. By regarding the polarized equations as a left-to-right rewrite rules system we obtain a set of rewrite rules which we will call $E_p^{\rightarrow}$ (see Figure 14) which will then be used to implement $E\phi/\psi$-reduction and $E\phi$-reduction.

$$
\begin{array}{rcll}
\lambda x\,.\langle \mathbf{d}^-\rangle(e[\langle \mathbf{c}^+\rangle x/x]) & \Rightarrow & \langle(\mathbf{c}^+ \to \mathbf{d}^-)\rangle \lambda x\,.e & (R13^-)\\
\langle(\mathbf{c}^- \to \mathbf{d}^+)\rangle \lambda x\,.e & \Rightarrow & \lambda x\,.\langle \mathbf{d}^+\rangle(e[\langle \mathbf{c}^-\rangle x/x]) & (R13^+)\\
(\langle \mathbf{c}^+ \to \mathbf{d}^-\rangle e)\ e' & \Rightarrow & \langle \mathbf{d}^-\rangle(e\ (\langle \mathbf{c}^+\rangle e')) & (R14^-)\\
\langle \mathbf{d}^+\rangle(e\ (\langle \mathbf{c}^-\rangle e')) & \Rightarrow & (\langle(\mathbf{c}^- \to \mathbf{d}^+)\rangle e)\ e' & (R14^+)\\
\Lambda\alpha\,.\langle \mathbf{c}^-\rangle e & \Rightarrow & \langle \forall\alpha\,.\mathbf{c}^-\rangle \Lambda\alpha\,.e & (R15^-)\\
\langle \forall\alpha\,.\mathbf{c}^+\rangle \Lambda\alpha\,.e & \Rightarrow & \Lambda\alpha\,.\langle \mathbf{c}^+\rangle e & (R15^+)\\
(\langle \forall\alpha\,.\mathbf{c}^-\rangle e)\{\pi\} & \Rightarrow & \langle \mathbf{c}^-[\pi/\alpha]\rangle e\{\pi\} & (R16^-)\\
\langle \mathbf{c}^+[\pi/\alpha]\rangle e\{\pi\} & \Rightarrow & (\langle \forall\alpha\,.\mathbf{c}^+\rangle e)\{\pi\} & (R16^+)\\
\langle \widehat{\rho}(\mathbf{c}^-, \iota)\rangle e\{\pi'\} & \Rightarrow & \langle \widehat{\rho}(\iota, \mathbf{c}^-)\rangle e\{\pi\} & (R17^-)\\
\langle \widehat{\rho}(\iota, \mathbf{c}^+)\rangle e\{\pi'\} & \Rightarrow & \langle \widehat{\rho}(\mathbf{c}^+, \iota)\rangle e\{\pi\} & (R17^+)
\end{array}
$$

Side conditions: in rules 17^- and 17^+ the type of e is $\forall\alpha.\rho$
in all rules at least one coercion must be proper

Figure 14: Polarized reduction: $E_p^{\leftarrow}$

Similarly, by regarding the polarized equation system as a right-to-left rewrite rules we obtain a set of rewrite rules which we will call $E_p^{\leftarrow}$ which will then be used to implement $E\psi/\phi$-reduction and $E\psi$-reduction.

4.4.1 $E_p^{\leftarrow}\psi^{\rightarrow}$ reduction modulo ϕ

The reduction that implements $\psi/E\phi$-reduction is called $E_p^{\rightarrow}\psi^{\rightarrow}$ reduction modulo ϕ and is defined as follows:

Definition 4.14 *($E_p^{\rightarrow}\psi^{\rightarrow}$ reduction modulo ϕ)*
We define $E_p^{\rightarrow}\psi^{\rightarrow}$ reduction modulo ϕ as rewriting using $E_p^{\rightarrow}$ modulo ϕ on completions and $\phi\psi$-reduction on coercions with the restriction that ϕ-equality is only used in connection with a $E_p^{\rightarrow}$ reduction step. That is, coercion reduction is not modulo ϕ-equality only completion reduction. Remember that ϕ-equality on completions include the core coercions equations Figure 6 and the equations for coercion application Figure 11.

The condition that ψ-equality is only used in connection with a $E_p^{\rightarrow}$ reduction step ensures that trivial non-productive (infinite) reductions like:

$$\langle \texttt{box}\,;\texttt{unbox}\,;\iota\rangle e =_1 \langle \texttt{box}\,;\texttt{unbox}\rangle e =_\phi \langle \texttt{box}\,;\texttt{unbox}\,;\texttt{box}\,;\texttt{unbox}\rangle e \Rightarrow_\psi \langle \texttt{box}\,;\texttt{unbox}\rangle e$$

are excluded.

Lemma 4.15 *(Properties of $\psi^{\to}E_p^{\leftarrow}$-reduction modulo ϕ) $\psi^{\to}E_p^{\leftarrow}$-reduction modulo ϕ is canonical; that is, it is strongly normalizing and confluent.*

Proof:
Strong normalization: Without loss of generality we may assume that every completion has exactly one coercion applied to each subexpression in the underlying F_2-expression (Lemma 4.4).

Let $(\mathbf{c}_1, \ldots, \mathbf{c}_k)$ be the vector of all coercion occurrences in a completion in some particular order such that they are in one-to-one correspondence with the subexpressions of the underlying F_2-expression. Since completion rewriting does not change the underlying F_2-expression, a completion rewriting step corresponds to a rewriting step on this vector.

A $\psi^{\to}$-reduction step operates on a single element of the coercion vector above. By Theorem 3.10 $\phi\psi$-reducing a coercion is strongly normalizing. Thus there can be only finitely many ψ-reduction steps at the beginning of the reduction or after an $E_p^{\leftarrow}$ step is executed.

An $E_p^{\leftarrow}$ step generally operates on several coercions in the coercion vector simultaneously. Consider the type signatures of the coercions in the coercion vector. An $E_p^{\leftarrow}$ step rewrites at least one coercion $\vdash \mathbf{c} : \rho \rightsquigarrow \rho'$ to a new coercion $\mathbf{c}'$ that has domain type or range type properly increased in the subtype hierarchy of Definition 3.29, and an $E_p^{\leftarrow}$ step never rewrites a coercion $\vdash \mathbf{c} : \rho \rightsquigarrow \rho'$ to a new coercion $\mathbf{c}'$ that has domain type or range type properly decreased in the subtype hierarchy of Definition 3.29.

Let us, as an example, consider rule 13^-:

$$\lambda x\,.\langle \mathbf{d}^-\rangle(e[\langle \mathbf{c}^+\rangle x/x]) \Rightarrow \langle(\mathbf{c}^+ \to \mathbf{d}^-)\rangle \lambda x\,.e$$

We assume without loss of generality that x occurs exactly once in e. The vector of coercions must have the form (with signatures of each coercion that we consider added):

$$(\ldots, \iota : \rho_x \to \rho \rightsquigarrow \rho_x \to \rho, \ldots, \mathbf{d}^- : \rho_2 \rightsquigarrow \rho, \ldots, \mathbf{c}^+ : \rho_x \rightsquigarrow \rho_1, \ldots)$$

where ι is the coercion applied to the abstraction. Rewriting using rule 13^- changes this into:

$$(\ldots, (\mathbf{c} \to \mathbf{d})^- : \rho_1 \to \rho_2 \rightsquigarrow \rho_x \to \rho, \ldots, \iota : \rho_2 \rightsquigarrow \rho_2, \ldots, \iota : \rho_1 \rightsquigarrow \rho_1, \ldots)$$

Let us examine the changes in the domain types and range types of the three coercions in the vector that changes. The range type of the second coercion changes from ρ to ρ_2, but this corresponds to the signature of an inverse coercion of $\mathbf{d}$, since $\mathbf{d}$ has negative polarity we may choose this to have positive polarity (Lemma 3.31). We therefore have that $\rho \leq \rho_2$. Similarly, the domain type of the third coercion changes from ρ_x to ρ_1 which corresponds to the signature of the positive coercion $\mathbf{c}$ and so we also have $\rho_x \leq \rho_1$. Finally, it can also be shown that the domain type of the first coercion is increased. All other domain or range types are unchanged, so this means that for rule 13^- at least one coercion in the tuple has its domain type or range type properly increased in the subtype hierarchy (because at least one coercion is proper) and no coercions gets its domain type or range type decreased.

This can be shown to hold for all rules in $E_p^{\leftarrow}$ and since the subtype hierarchy has only finite ascending chains (Proposition 3.33) it follows that $E_p^{\leftarrow}$ steps can only be applied a finite number of times. Thus every $\psi^{\to}E_p^{\leftarrow}$-reduction sequence is finite.

Confluence: Since $\psi^{\to}E_p^{\leftarrow}$-reduction modulo ϕ is strongly normalizing it is, by Newman's Lemma, sufficient to show local confluence: if e has overlapping redexes and reduces by single rewriting steps to e_1 and e_2 then there exists a common reduct e' to which both e_1 and e_2 reduce, possibly in several steps.

Let us consider such triples e, e_1, e_2. By Theorem 3.10 $\psi\phi$-reduction on coercions is confluent. Note that ψ-redexes do not overlap with any $E_p^{\leftarrow}$-redex due to the polarization and orientation of the E_p rules. We only have to worry about overlaps of $E_p^{\leftarrow}$-rules modulo ϕ-equality. There are only two kinds of overlaps:

1. Application of the same rule to the same subexpression, only with different coercions; let us look at one of the rules (the other rules handled analogously), e.g. rule $R13^+$:

   $$\langle(\mathbf{c}_1{}^- \to \mathbf{c}_2{}^+)\,;\mathbf{c}'\rangle\lambda x\,.\,e =_{12} \langle\mathbf{c}'\rangle\langle\mathbf{c}_1{}^- \to \mathbf{c}_2{}^+\rangle\lambda x\,.\,e \Rightarrow_{R13^+} \langle\mathbf{c}'\rangle\lambda x\,.\,\langle\mathbf{c}_2{}^+\rangle(e[\langle\mathbf{c}_1{}^-\rangle x/x])$$

   and

   $$\langle(\mathbf{d}_1{}^- \to \mathbf{d}_2{}^+)\,;\mathbf{d}'\rangle\lambda x\,.\,e =_{12} \langle\mathbf{d}'\rangle\langle\mathbf{d}_1{}^- \to \mathbf{d}_2{}^+\rangle\lambda x\,.\,e \Rightarrow_{R13^+} \mathbf{d}'\,;\lambda x\,.\,\mathbf{d}_2{}^+\,;(e[\langle\mathbf{d}_1{}^-\rangle x/x])$$

   where

   $$\phi \vdash (\mathbf{c}_1{}^- \to \mathbf{c}_2{}^+)\,;\mathbf{c}' = (\mathbf{d}_1{}^- \to \mathbf{d}_2{}^+)\,;\mathbf{d}'$$

   In this case it is sufficient to show that

   $$\phi \vdash \mathbf{c}' \Rightarrow_\psi \mathbf{c}_3{}^+\,;\mathbf{c}_4{}^-$$
   $$\phi \vdash \mathbf{d}' \Rightarrow_\psi \mathbf{d}_3{}^+\,;\mathbf{d}_4{}^-$$

   for some positive $\mathbf{c}_3{}^+$, $\mathbf{d}_3{}^+$ and negative $\mathbf{c}_4{}^-$, $\mathbf{d}_4{}^-$, where $\mathbf{c}_4{}^-$ and $\mathbf{d}_4{}^-$ are equal (by core coercion equality), since then we can apply the same rule again to each of the two different reducts to get a common reduct. By Lemma 3.27 there exist such factorings and since positive and negative coercions are in $\phi\psi$-normal form we have that $\mathbf{c}_4{}^-$ and $\mathbf{d}_4{}^-$ are equal modulo the core coercions equations.

2. Overlaps due to five pairs of adjacent rules in Figure 13: $13^-/13^+$, $14^-/14^+$, $15^-/15^+$, $16^-/16^+$ and $17^-/17^+$. Let us again consider one of these overlaps $13^-/13^+$ (the other cases are handled analogously):

   $$\langle\mathbf{c}_1{}^- \to \mathbf{d}_1{}^+\rangle\lambda x\,.\,\langle\mathbf{d}_2{}^-\rangle(e[\langle\mathbf{c}_2{}^+\rangle x/x])$$

   can be rewritten to

   $$\langle\mathbf{c}_1{}^- \to \mathbf{d}_1{}^+\rangle\langle\mathbf{c}_2{}^+ \to \mathbf{d}_2{}^-\rangle\lambda x\,.\,e$$

   and to

   $$\lambda x\,.\,\langle\mathbf{d}_1{}^+\rangle\langle\mathbf{d}_2{}^-\rangle(e[\langle\mathbf{c}_2{}^+\rangle\langle\mathbf{c}_1{}^-\rangle x/x])$$

For the first reduct we get furthermore

$$\begin{array}{lll}
\langle \mathbf{c}_1{}^- \to \mathbf{d}_1{}^+\rangle\langle \mathbf{c}_2{}^+ \to \mathbf{d}_2{}^-\rangle\lambda x\,.\,e & =_{12} & \\
\langle(\mathbf{c}_2{}^+ \to \mathbf{d}_2{}^-)\,;(\mathbf{c}_1{}^- \to \mathbf{d}_1{}^+)\rangle\lambda x\,.\,e & =_3 & \\
\langle(\mathbf{c}_1{}^-\,;\mathbf{c}_2{}^+) \to (\mathbf{d}_2{}^-\,;\mathbf{d}_1{}^+)\rangle\lambda x\,.\,e & \Rightarrow^*_{\psi} & \text{(Lemma 3.27 and Lemma 3.25)}\\
\langle(\mathbf{c}_3{}^+\,;\mathbf{c}_4{}^-) \to (\mathbf{d}_3{}^+\,;\mathbf{d}_4{}^-)\rangle\lambda x\,.\,e & =_3 & \\
\langle(\mathbf{c}_4{}^- \to \mathbf{d}_3{}^+)\,;(\mathbf{c}_3{}^+ \to \mathbf{d}_4{}^-)\rangle\lambda x\,.\,e & =_{12} & \\
\langle \mathbf{c}_3{}^+ \to \mathbf{d}_4{}^-\rangle\langle \mathbf{c}_4{}^- \to \mathbf{d}_3{}^+\rangle\lambda x\,.\,e & \Rightarrow_{E_p^{\leftarrow}} & \\
\langle \mathbf{c}_3{}^+ \to \mathbf{d}_4{}^-\rangle\lambda x\,.\,\langle \mathbf{d}_3{}^+\rangle(e[\langle \mathbf{c}_4{}^-\rangle x/x]) & &
\end{array}$$

Similarly, we can rewrite the second reduct to the same final completion above.

$$\begin{array}{lll}
\lambda x\,.\,\langle \mathbf{d}_1{}^+\rangle\langle \mathbf{d}_2{}^-\rangle(e[\langle \mathbf{c}_2{}^+\rangle\langle \mathbf{c}_1{}^-\rangle x/x]) & =_{12} & \\
\lambda x\,.\,\langle \mathbf{d}_2{}^-\,;\mathbf{d}_1{}^+\rangle(e[\langle \mathbf{c}_1{}^-\,;\mathbf{c}_2{}^+\rangle x/x]) & \Rightarrow^*_{\psi} & \text{(Lemma 3.27 and Lemma 3.25)}\\
\lambda x\,.\,\langle \mathbf{d}_3{}^+\,;\mathbf{d}_4{}^-\rangle(e[\langle \mathbf{c}_3{}^+\,;\mathbf{c}_4{}^-\rangle x/x]) & =_{12} & \\
\lambda x\,.\,\langle \mathbf{d}_4{}^-\rangle\langle \mathbf{d}_3{}^+\rangle(e[\langle \mathbf{c}_4{}^-\rangle\langle \mathbf{c}_3{}^+\rangle x/x]) & \Rightarrow_{E_p^{\leftarrow}} & \\
\langle \mathbf{c}_3{}^+ \to \mathbf{d}_4{}^-\rangle\lambda x\,.\,\langle \mathbf{d}_3{}^+\rangle(e[\langle \mathbf{c}_4{}^-\rangle x/x]) & &
\end{array}$$

This completes the proof. ■

4.4.2 $E_p^{\rightarrow}\phi^{\rightarrow}$ reduction modulo ψ

The reduction that implements $\phi/E\psi$-reduction is called $E_p^{\rightarrow}\phi^{\rightarrow}$ reduction modulo ψ and is defined as follows:

Definition 4.16 *($E_p^{\rightarrow}\phi^{\rightarrow}$ reduction modulo ψ)*
We define $E_p^{\rightarrow}\phi^{\rightarrow}$ reduction modulo ψ as rewriting using $E_p^{\rightarrow}$ modulo ψ on completions and $\phi\psi$-reduction on coercions with the restriction that ψ-equality is only used in connection with a $E_p^{\rightarrow}$ reduction step. That is, coercion reduction is not modulo ψ-equality only completion reduction. Remember that ψ-equality on completions include the core coercions equations Figure 6 and the equations for coercion application Figure 11.

The condition that ψ-equality is only used in connection with a $E_p^{\rightarrow}$ reduction step is introduce here for the same reason as for $E_p^{\leftarrow}\psi^{\rightarrow}$ reduction modulo ϕ.

Lemma 4.17 *(Properties of $\phi^{\rightarrow}E_p^{\rightarrow}$-reduction modulo ψ) $\phi^{\rightarrow}E_p^{\rightarrow}$-reduction modulo ψ is canonical; that is, it is strongly normalizing and confluent.*

Proof:
Analogous to Lemma 4.15.

■

4.4.3 $E_p^{\rightarrow}\phi^{\rightarrow}$ reduction

The reduction that implements ϕ/E-reduction is called $E_p^{\rightarrow}\phi^{\rightarrow}$ reduction and is defined as follows:

Definition 4.18 *($E_p^{\rightarrow}\phi^{\rightarrow}$ reduction)*
We define $E_p^{\rightarrow}\phi^{\rightarrow}$ reduction as rewriting using $E_p^{\rightarrow}$ on completions and $\phi\psi$-reduction on coercions.

4.4.4 $E_p^{\leftarrow}\psi^{\rightarrow}$ reduction

The reduction that implements ψ/E-reduction is called $E_p^{\leftarrow}\psi^{\rightarrow}$ reduction and is defined as follows:

Definition 4.19 *($E_p^{\leftarrow}\psi^{\rightarrow}$ reduction)*
We define $E_p^{\leftarrow}\psi^{\rightarrow}$ reduction as rewriting using $E_p^{\leftarrow}$ on completions and $\phi\psi$-reduction on coercions.

4.5 Optimal completions

Recall our discussion at the end of Subsection 4.2.1 on how to find optimal completions. We conjectured that by using $\phi/E\psi$-reduction we could get rid of all ϕ-redexes first — even at the cost of introducing additional ψ-redexes — and then next by ψ/E-reduction get rid of all ψ-redexes without letting ϕ-redexes slip back in. Dually, that by using $\psi/E\phi$-reduction we could get rid of all ψ-redexes first — even at the cost of introducing additional ϕ-redexes — and then next by ϕ/E-reduction get rid of all ϕ-redexes without letting ψ-redexes slip back in. In this way we would obtain two kind of $\phi\psi$-normal forms. In this section we will show that this scheme for finding $\phi\psi$-normal forms is indeed feasible and we will define two kinds of (formally) optimal completions as being these two kind of $\phi\psi$-normal forms.

4.5.1 ψ-free and ϕ-free completions

The class of normal forms under $\phi^{\rightarrow}$-reduction modulo $E\psi$ and $\psi^{\rightarrow}$-reduction modulo $E\phi$ are called ϕ-free and ψ-free completions and are define as follows:

Definition 4.20 *(ψ-free completions, ϕ-free completions)*

1. We say a completion e is *ψ-free* if every congruent completion $e' \cong e$ ψ-reduces to e under $E\phi$-equality; i.e, $E\phi \vdash e' \Rightarrow_{\psi}^{*} e$.

2. We say a completion e is *ϕ-free* if every congruent completion $e' \cong e$ ϕ-reduces to e under $E\psi$-equality; i.e., $E\psi \vdash e' \Rightarrow_{\phi}^{*} e$.

Because of the strong global requirement that *all* congruent completions must be ψ-reducible modulo $E\phi$ to e for e to be called ψ-free it is not even clear that ψ-free completions (or ϕ-free completions) exist. This can be shown, however, and this is what we will do next.

To prove that ψ-free completions exists we need a few lemmas. First a lemma that shows commutativity of $\psi^{\rightarrow}$ and $E_p^{\leftarrow}/\phi$:

Lemma 4.21 *$\psi^{\rightarrow}$ and $E_p^{\leftarrow}/\phi$ commutes, that is the following diagram commutes:*

$$\begin{tikzcd}
e \arrow[r, "\psi^{\rightarrow}"] \arrow[d, "E_p^{\leftarrow}/\phi"] & e' \arrow[d, dotted, "E_p^{\leftarrow}/\phi", "*"'] \\
e_1 \arrow[r, dotted, "\psi^{\rightarrow}", "*"'] & e_1{}'
\end{tikzcd}$$

Proof:

In the proof we check that $\psi^{\rightarrow}$ commutes with each of the rules of $E_p^{\leftarrow}$ and with $\phi^{\rightarrow}$ (introducing a ϕ redex by ϕ-equality does not affect a $\psi^{\rightarrow}$ redex, so $\psi^{\rightarrow}$ commutes with ϕ-equality if it commutes with $\phi^{\rightarrow}$). For each of the rules of $E_p^{\leftarrow}$ it is easy to see that $\psi^{\rightarrow}$ redexes do not overlap with any $E_p^{\leftarrow}$ due to the polarization of the $E_p^{\leftarrow}$ rules. This means that an $E_p^{\leftarrow}$ reduction step cannot remove or move a $\psi^{\rightarrow}$ redex. We therefore have that $\psi^{\rightarrow}$ commutes with each of the rules of $E_p^{\leftarrow}$. That $\psi^{\rightarrow}$ commutes with $\phi^{\rightarrow}$ is easy to show. If the redexes are non overlapping it is trivial, and if the redexes overlap we have two cases:

1.

$$\texttt{unbox}; \texttt{box}; \texttt{unbox} \Rightarrow_{\phi} \texttt{unbox}; \iota =_1 \texttt{unbox}$$

$$\texttt{unbox}; \texttt{box}; \texttt{unbox} \Rightarrow_{\psi} \iota; \texttt{unbox} =_2 \texttt{unbox}$$

2.

$$\texttt{box}; \texttt{unbox}; \texttt{box} \Rightarrow_{\phi} \iota; \texttt{box} =_2 \texttt{box}$$

$$\texttt{box}; \texttt{unbox}; \texttt{box} \Rightarrow_{\psi} \texttt{box}; \iota =_1 \texttt{box}$$

which shows that $\psi^{\rightarrow}$ commutes with $\phi^{\rightarrow}$ and concludes the lemma.

■

We next prove the lemma that helps us prove the uniqueness of ψ-free normal forms:

Lemma 4.22 *Let e and e' be two (congruent) coercions such that e $\psi/E\phi$-reduces to e'. Let $\overline{e}$ and $\overline{e'}$ be $E_p^{\leftarrow}/\phi$-normal forms of e and e', then $\overline{e}$ $E_p^{\leftarrow}\psi^{\rightarrow}/\phi$-reduces to $\overline{e'}$. The lemma can be visualized by the following diagram:*

$$\begin{tikzcd}
e \arrow[r, "\psi/E\phi", "*"'] \arrow[d, "E_p^{\leftarrow}/\phi", "*"'] & e' \arrow[d, "E_p^{\leftarrow}/\phi", "*"'] \\
\overline{e} \arrow[r, dotted, "E_p^{\leftarrow}\psi^{\rightarrow}/\phi", "*"'] & \overline{e'}
\end{tikzcd}$$

Proof:

Since E-equality and E_p-equality are equivalent (Theorem 4.13) we use E_p-equality instead of E-equality in the proof. We prove the lemma by induction on the length of the $\psi/E_p\phi$-reduction of e to e'. Let in the this proof $\overline{e_1}$ be the $E_p^{\leftarrow}/\phi$-normal form of e_1 and $\equiv$ be syntactical identity.

Let the first reduction step be a $\psi^{\rightarrow}$-reduction step and let e_1 be the reduct of e by the reduction step. Then by the commutativity of $\psi^{\rightarrow}$-reduction (Lemma 4.21) and $E_p^{\leftarrow}/\phi$-reduction and by induction there must exists a coercion $e_1{}'$ such that the following diagram commutes:

$$\begin{tikzcd}
e \arrow[rr, "\psi^{\rightarrow}"] \arrow[d, "E_p^{\leftarrow}/\phi"', "*"] & & e_1 \arrow[r, "\psi/E_p\phi", "*"'] \arrow[d, dotted, "E_p^{\leftarrow}/\phi", "*"'] \arrow[dl, dotted, "E_p^{\leftarrow}/\phi", "*"'] & e' \arrow[d, "E_p^{\leftarrow}/\phi", "*"'] \\
\overline{e} \arrow[r, dotted, "\psi^{\rightarrow}", "*"'] & e_1{}' \arrow[r, dotted, "E_p^{\leftarrow}/\phi", "*"'] & \overline{e_1} \arrow[r, dotted, "E_p^{\leftarrow}\psi^{\rightarrow}/\phi", "*"'] & \overline{e'}
\end{tikzcd}$$

from which the result of the lemma follows for this case.

Let the first step be a E_p-equality step, then since E_p-equality is obviously the equivalence relation generated by $E_p^{\leftarrow}$, we may without loss of generality assume that the this is a $E_p^{\leftarrow}$ step or $E_p^{\rightarrow}$ step. We look at the case where it is a $E_p^{\leftarrow}$ step first. Let e_1 be the reduct of e by the reduction step. By induction and since $E_p^{\leftarrow}/\phi$ is confluent there exist a coercion $e_1{}'$ such that the following diagram commutes:

$$\begin{tikzcd}
e \arrow[r, "E_p^{\leftarrow}"] \arrow[d, "E_p^{\leftarrow}/\phi"', "*"] & e_1 \arrow[r, "\psi/E_p\phi", "*"'] \arrow[d, dotted, "E_p^{\leftarrow}/\phi", "*"'] & e' \arrow[d, "E_p^{\leftarrow}/\phi", "*"'] \\
\overline{e} \arrow[r, dotted, dash, "\equiv"] & \overline{e_1} \arrow[r, dotted, "E_p^{\leftarrow}\psi^{\rightarrow}/\phi", "*"'] & \overline{e'}
\end{tikzcd}$$

from which the result of the lemma follows for this case.

Next we look at the case where it is a $E_p^{\rightarrow}$ step. Let e_1 be the reduct of e by the reduction step then e_1 must $E_p^{\leftarrow}/\phi$ reduce to $\overline{e}$ so by induction the following diagram commutes:

$$\begin{tikzcd}
e \arrow[d, "E_p^{\leftarrow}/\phi"', "*"] & e_1 \arrow[l, "E_p^{\leftarrow}"'] \arrow[r, "\psi/E_p\phi", "*"'] \arrow[d, dotted, "E_p^{\leftarrow}/\phi", "*"'] & e' \arrow[d, "E_p^{\leftarrow}/\phi", "*"'] \\
\overline{e} \arrow[r, dotted, dash, "\equiv"] & \overline{e_1} \arrow[r, dotted, "E_p^{\leftarrow}\psi^{\rightarrow}/\phi", "*"'] & \overline{e'}
\end{tikzcd}$$

from which the result of the lemma follows for this case.

Finally, we look at the case where the first step is a ϕ-equality step. Let e_1 be the reduct of e by the reduction step then e_1 must $E_p^{\leftarrow}/\phi$ reduce to $\overline{e}$ so by induction the following diagram commutes:

$$\begin{tikzcd}
e \arrow[r, dash, "\phi"] \arrow[d, "E_p^{\leftarrow}/\phi", "*"'] & e_1 \arrow[r, "\psi/E_p\phi", "*"'] \arrow[d, dotted, "E_p^{\leftarrow}/\phi", "*"'] & e' \arrow[d, "E_p^{\leftarrow}/\phi", "*"'] \\
\overline{e} \arrow[r, dotted, dash, "\equiv"] & \overline{e_1} \arrow[r, dotted, "E_p^{\leftarrow}\psi^{\rightarrow}/\phi", "*"'] & \overline{e'}
\end{tikzcd}$$

from which the result of the lemma follows for this case.

■

We can now prove that ψ- and ϕ-free completions exist and that these are unique modulo $E_p\phi$-equality:

Theorem 4.23 *(Existence and uniqueness of ψ-, resp. ϕ-free completions) Let e be a (closed) F_2-expression and let ρ be a valid representation type for e. Then*

1. *e has a ψ-free completion e' at ρ and e' is uniquely determined up to $E\phi$-equality.*
2. *e has a ϕ-free completion e'' at ρ and e'' is uniquely determined up to $E\psi$-equality.*

Proof:

We only give a proof of 1. The proof of 2 is analogous. (It requires lemmas analogous to Lemma 4.15 and Lemma 4.22.)

Consider all the (congruent) completions of e at ρ. By Theorems 4.8 and 4.13 we know that they are all $E_p\phi\psi$-equal. If we consider two arbitrary completions e_1 and e_n at ρ then we know that $e_1 = e_2 \ldots = e_{n-1} = e_n$ for some completion $e_2, \ldots, e_{n-1}$ where $=$ means equal by one rule from $E_p\phi\psi$. This means that for each $1 \leq$ k $<$ n either $E\phi \vdash e_k \Rightarrow_\psi e_{k+1}$ or $E\phi \vdash e_{k+1} \Rightarrow_\psi e_k$. From this and from the fact that $\psi^{\rightarrow}E_p^{\leftarrow}$-reduction modulo ϕ is confluent (Lemma 4.15) it follows that they must all have a common reduct. This proves that any two (congruent) completions of e at ρ have a common reduct, so from the fact that $\psi^{\rightarrow}E_p^{\leftarrow}$-reduction modulo ϕ is canonical (Lemma 4.15 again) all completions of e at ρ must have a common normal form e_{nf} which must then be a ψ-free completion of e.

Let e_c be another ψ-free completion of e. Let $\overline{e_c}$ and $\overline{e_{nf}}$ be $E_p^{\leftarrow}/\phi$-normal forms of e_c and e_{nf}. Since e_c is ψ-free we have that e_{nf} $\psi/E\phi$-reduces to e_c and therefore by Lemma 4.22 that $\overline{e_c}$ $E_p^{\leftarrow}\psi^{\rightarrow}/\phi$-reduces to $\overline{e_{nf}}$. Similarly, since e_{nf} is ψ-free we have that e_c $\psi/E\phi$-reduces to e_{nf} and therefore by Lemma 4.22 that $\overline{e_{nf}}$ $E_p^{\leftarrow}\psi/\phi$-reduces to $\overline{e_c}$. Since $E_p^{\leftarrow}\psi^{\rightarrow}/\phi$-reduction is strongly normalizing we must have $E \vdash \overline{e_{nf}} = \overline{e_c}$ and therefore $E\phi \vdash e_{nf} = e_c$.

■

Intuitively, a ψ-free completion "prefers" to keep data in a boxed representation and unboxes a representation only when it is sure the unboxed value is required by some operation and not required boxed anymore. This way passing arguments to polymorphic functions and returning their results can be expected to be efficient whereas operations requiring unboxed data such as integer operations may be inefficient due to the cost of unboxing arguments and boxing the results.

Dual to this, a ϕ-free completion prefers to keep data in unboxed representation; it boxes a value only when it is sure to be required due to a call to a polymorphic function. Thus primitive operations will generally be executed fast as no coercions need to be performed for neither the arguments nor the result, but calls to polymorphic functions may be expensive due to the need for boxing (parts of) the arguments and unboxing (parts of) the result.

Since the degree of polymorphism in a program tends to be greatest when higher-order functions are used a ψ-free completion will generally be better for higher-order programs, especially if there is little "ground type" processing such as arithmetic operations. On the other hand, ϕ-free completions will generally do best where there is little polymorphism and/or lots of operations on ground types.

4.5.2 Optimal ψ-free and ϕ-free completions

The two constructions above for ψ-free and ϕ-free completion are canonical since they use a universal standard representation (maximally boxed or minimally boxed) for all data independent of their context. They are not "optimal" since they typically contain many ϕ-, respectively ψ-redexes modulo E_p-equality. We shall now set out to construct *optimal* ψ-free and ϕ-free completions, which have no remaining (ψ *or*ϕ) redexes — and are thus $\psi\phi$-normal forms modulo E.

Definition 4.24 *(Optimal ψ-free/ϕ-free completions)*

1. A completion e is a *(formally) optimal ψ-free completion* if e is ψ-free and every congruent ψ-free completion e' ϕ-reduces to e modulo E; i.e., $E \vdash e' \Rightarrow^*_\phi e$.
2. A completion e is a *(formally) optimal ϕ-free completion* if e is ϕ-free and every congruent ϕ-free completion e' ψ-reduces to e modulo E; i.e., $E \vdash e' \Rightarrow^*_\psi e$.

We have shown that ψ-free completions are $E_p\phi$-equal, and ϕ-free completions are $E_p\psi$-equal. As a result we obtain our main theorem:

Theorem 4.25 *(Existence of ψ-free and ϕ-free optimal completions) Let e be a (closed) F_2-expression and let ρ be a valid representation type for e. Then e has both an optimal ψ-free completion and an optimal ϕ-free completion at ρ. Furthermore, both are unique modulo E.*

Proof:

We prove this for ϕ-free completions. The proof for ψ-free completions is analogous.

For the proof we employ again a rewriting system. This time we use $E_p^{\leftarrow}\psi^{\rightarrow}$-reduction. This rewriting system operates on equivalence classes defined by the coercion equations in Figure 6 and the application equations in Figure 11. We shall however only consider $E_p^{\leftarrow}\psi^{\rightarrow}$-reduction on ϕ-free completions. Since all ϕ-free completions are $E_p\psi$-equivalent we have that $E_p^{\leftarrow}\psi^{\rightarrow}$-reduction on ϕ-free completions is closed. To see this, assume that e is a ϕ-free completion and e $E_p^{\leftarrow}\psi^{\rightarrow}$-reduces to e'. Then since e is a ϕ-free all congruent completions $E_p^{\leftarrow}\psi^{\rightarrow}$ reduces modulo ϕ to e, which means that they also $E_p^{\leftarrow}\psi^{\rightarrow}$ reduces modulo ϕ to e', and therefore that e' is a ϕ-free completion.

We need to prove that $E_p^{\leftarrow}\psi^{\rightarrow}$-reduction ϕ-free completions is canonical. That $E_p^{\leftarrow}\psi^{\rightarrow}$-reduction is strongly normalizing follows from the fact that $E_p^{\leftarrow}\psi^{\rightarrow}/\phi$-reduction is strongly normalizing (Lemma 4.15). So all we need to show is local confluence of $E_p^{\leftarrow}\psi^{\rightarrow}$-reduction on ϕ-free completions. If we examine the proof of local confluence of $E_p^{\leftarrow}\psi^{\rightarrow}/\phi$-reduction (Lemma 4.15) then the crucial step was to show that if

$$\phi \vdash \mathbf{c} = \mathbf{d}^+ \,; \mathbf{c}'$$

then we could show that there existed coercions $\mathbf{c}_3{}^+$ and $\mathbf{c}_4{}^-$ such that

$$\phi \vdash \mathbf{c}' \Rightarrow_\psi \mathbf{c}_3{}^+ \,; \mathbf{c}_4{}^-$$

and that this factoring was unique (modulo the core coercion equations). What we need to show here is, that if

$$\vdash \mathbf{c} = \mathbf{d}^+ \,; \mathbf{c}'$$

then there exist coercions $\mathbf{c}_3{}^+$ and $\mathbf{c}_4{}^-$ such that

$$\vdash \mathbf{c}' \Rightarrow_\psi \mathbf{c}_3{}^+ \,; \mathbf{c}_4{}^-$$

But since we are only looking at ϕ-free completions we know that if $\mathbf{c}_{nf}$ is a normal form of $\mathbf{c}$ then

$$\vdash \mathbf{c}' \Rightarrow_\psi \mathbf{c}_{nf}$$

and since normal form coercions can be uniquely $+/-$-factorized (Lemma 3.25) then we can take $\mathbf{c}_3{}^+ \,; \mathbf{c}_4{}^-$ to be this unique $+/-$-factoring. This shows local confluence of $E_p^{\leftarrow}\psi^{\rightarrow}$-reduction on ϕ-free completions and therefore also that $E_p^{\leftarrow}\psi^{\rightarrow}$-reduction on ϕ-free completions is canonical. So by the same kind of argumentations as we used in the proof of Theorem 4.23 we may show that optimal ψ-free completions exist.

To prove uniqueness of optimal ψ-free completions modulo E-equality we need to prove a result similar to Lemma 4.22. That is, we should prove that the following diagram commutes:

$$\begin{tikzcd}
e \arrow[r, "\psi/E", "*"'] \arrow[d, "E_p^{\leftarrow}"', "*"] & e' \arrow[d, "E_p^{\leftarrow}", "*"'] \\
\overline{e} \arrow[r, dotted, "E_p^{\leftarrow}\psi^{\rightarrow}", "*"'] & \overline{e'}
\end{tikzcd}$$

where e and e' are ψ-free completions and $\overline{e}$ and $\overline{e'}$ are $E_p^{\leftarrow}$-normal forms of e respectively e'. A proof of this can easily be constructed by inspection of Lemma 4.22). We can then prove uniqueness of optimal ϕ-free completions analogously the way we proved uniqueness of ϕ-free completions in Theorem 4.23.

■

4.6 Canonical completions

The rules for $\psi^{\rightarrow}E_p^{\leftarrow}$-reduction modulo ϕ suggest an explicit construction of ψ-free completions: take an arbitrary completion and execute the ψ-reduction system until a normal form is reached. Analogously for ϕ-free completions.

An simpler method consists of devising syntax-directed translations that produce a ψ-free or ϕ-free completion directly. The canonical construction of a ψ-free completion consists of keeping all data in their "maximally" boxed representation (i.e., representing a standard type by the maximal type in the representation type hierarchy) and boxing a unboxed value as soon as it is produced by some operation and unboxing it just before it is consumed by some operation.

The canonical construction of a ϕ-free completion consists of keeping all data in their "minimally" boxed representation where an unboxed value is only boxed just before it is passed to a polymorphic function and unboxed immediately after it is returned from a polymorphic function. This is actually the construction described by Leroy [Ler92].

In the following two sections we will present these two kinds of canonical completions: ϕ-free and ψ-free canonical completions .

4.6.1 Canonically boxed completion

The first kind of canonical completions are ϕ-free canonical completions. We want to define a translation from F_2-expressions into ϕ-free canonical completion. This amounts to translating expressions of the form $e\{\sigma\}$ into $\langle \mathbf{c} \rangle (e'\{[\rho]\})$ where $\mathbf{c}$ is the canonical coercion that makes sure that the representation type of the translated expression is completely without boxed type constructors $[\_]$ and ρ is identical to σ. We have, however, to be careful when performing this step since we do not allow boxing of a boxed value. If σ is a type variable α then the type $[\alpha]$ would be an illegal type. What we really have to do is only to box ρ if this is not already a boxed type or equivalently σ is not a type variable.

The ϕ-free canonical completion correspond almost to the same completions that Leroy obtains from his translation [Ler92]. The only difference is that Leroy does allow boxing of boxed values. So his translation will always do as the translation described above.

We write $\overline{\sigma}$ for the embedding of a standard type σ into its syntactically equivalent representation type. We extend this type embedding in the natural way to a homomorphism on type assignments such that: $x{:}\overline{\sigma} \in \overline{\Gamma}$ iff $x{:}\sigma \in \Gamma$.

Furthermore we define an operation $\lceil\_\rceil$ that only inserts a boxed type constructor around a representation type when this is not a boxed type variable:

Definition 4.26 Given a representation type ρ we define $\lceil\rho\rceil$ to be the smallest boxed type larger than ρ. That is, if ρ is a boxed type, we have $\lceil\rho\rceil \equiv \rho$ and if ρ is an unboxed type, we have $\lceil\rho\rceil \equiv [\rho]$

We can now give a formal definition of ϕ-free canonical completions:

Definition 4.27 *(ϕ-free canonical completion).*

We say that an explicitly boxed F_2-expression e_c is a *ϕ-free canonical completion* of a F_2-expression e if $\Gamma \vdash e{:}\sigma \Rightarrow e_c$ is derivable from the inference system of Figure 15 for some type environment Γ and some type σ.

There are two things that we should show about this definition: first, that ϕ-free canonical completions are really completions and second, that they are indeed ϕ-free. We will prove this in the following two lemmas.

Lemma 4.28 *ϕ-free canonical completions are well-typed w.r.t. the inference system in Figure 5, i.e. if $\Gamma \vdash e{:}\sigma \Rightarrow e_c$ then $\overline{\Gamma} \vdash e_c{:}\overline{\sigma}$.*

$$\dfrac{\Gamma\{x : \sigma_1\}\vdash e{:}\sigma_2 \Rightarrow e'}{\Gamma\vdash \lambda x : \sigma_1 \,.\, e{:}\sigma_1 \nrightarrow \sigma_2 \Rightarrow \lambda x : \overline{\sigma_1} \,.\, e'}$$

$$\dfrac{\Gamma\vdash e_1{:}\sigma_1 \nrightarrow \sigma_2 \Rightarrow {e_1}' \quad \Gamma\vdash e_2{:}\sigma_1 \Rightarrow {e_2}'}{\Gamma\vdash e_1\ e_2{:}\sigma_2 \Rightarrow {e_1}'\ {e_2}'}$$

$$\dfrac{\Gamma\vdash e{:}\sigma \Rightarrow e'}{\Gamma\vdash \Lambda\alpha \,.\, e{:}\forall\alpha.\sigma \Rightarrow \Lambda\overline{\alpha} \,.\, e'}\ \alpha \notin FV(\Gamma)$$

$$\dfrac{\Gamma\vdash e{:}\forall\alpha.\sigma \Rightarrow e' \quad \vdash_c \mathbf{c}{:}\overline{\sigma}[\lceil\overline{\sigma_1}\rceil/\overline{\alpha}] \rightsquigarrow \overline{\sigma}[\overline{\sigma_1}/\overline{\alpha}]}{\Gamma\vdash e\{\sigma_1\}{:}\sigma[\sigma_1/\alpha] \Rightarrow \langle\mathbf{c}\rangle e'\{\lceil\overline{\sigma_1}\rceil\}}$$

$$\Gamma\{x : \sigma\}\vdash x{:}\sigma \Rightarrow x$$

Figure 15: ϕ-free canonical completion for F_2

Proof:

The proof is by induction on the structure of the F_2-expression e or equivalent on the structure of the derivation of $\Gamma\vdash e{:}\sigma$.

Base case: x. From $\Gamma\{x : \sigma\}\vdash x{:}\sigma \Rightarrow x$ it is obvious that $\overline{\Gamma}\{x : \overline{\sigma}\}\vdash x{:}\overline{\sigma}$ holds.

Inductive case: $\lambda x : \sigma_x . e_1$. By induction we have from $\Gamma\{x : \sigma_x\}\vdash e_1{:}\sigma_1 \Rightarrow {e_1}'$ that $\overline{\Gamma}\{x : \overline{\sigma_x}\}\vdash e_1{:}\overline{\sigma_1}$ which means that we have:

$$\dfrac{\overline{\Gamma}\{x : \overline{\sigma_x}\}\vdash e_1{:}\overline{\sigma_1}}{\overline{\Gamma}\vdash \lambda x : \overline{\sigma_x} \,.\, e_1{:}\overline{\sigma_1} \nrightarrow \overline{\sigma}}$$

which proves the case, since $\overline{\sigma_1 \rightarrow \sigma} = \overline{\sigma_1} \nrightarrow \overline{\sigma}$.

Inductive case: $e_1\ e_2$. By induction we have from $\Gamma\vdash e_1{:}\sigma_2 \nrightarrow \sigma \Rightarrow {e_1}'$ and $\Gamma\vdash e_2{:}\sigma_2 \Rightarrow {e_2}'$ that $\overline{\Gamma}\vdash {e_1}'{:}\overline{\sigma_2 \rightarrow \sigma}$ and $\overline{\Gamma}\vdash {e_2}'{:}\overline{\sigma_2}$. This means that we have:

$$\dfrac{\overline{\Gamma}\vdash {e_1}'{:}\overline{\sigma_2} \nrightarrow \overline{\sigma} \quad \overline{\Gamma}\vdash {e_2}'{:}\overline{\sigma_2}}{\overline{\Gamma}\vdash {e_1}'\ {e_2}'{:}\overline{\sigma}}$$

since $\overline{\sigma_2 \rightarrow \sigma} = \overline{\sigma_2} \nrightarrow \overline{\sigma}$.

Inductive case: $\Lambda\alpha . e_1$. By induction we know from $\Gamma\vdash e_1{:}\sigma_1 \Rightarrow {e_1}'$ that $\overline{\Gamma}\vdash {e_1}'{:}\overline{\sigma_1}$ and from the fact that e is well-typed we know that $\alpha \notin FV(\overline{\Gamma})$. This means that we have:

$$\dfrac{\overline{\Gamma}\vdash {e_1}'{:}\overline{\sigma_1}}{\overline{\Gamma}\vdash \Lambda\alpha \,.\, e_1{:}\forall\alpha.\overline{\sigma_1}}\ \alpha \notin FV(\overline{\Gamma})$$

which proves the case, since $\forall\alpha.\overline{\sigma_1} = \overline{\forall\alpha.\sigma_1}$.

Inductive case: $e_1\{\sigma_2\}$. By induction we know from $\Gamma\vdash e_1{:}\forall\alpha.\sigma_1 \Rightarrow {e_1}'$ that $\overline{\Gamma}\vdash {e_1}'{:}\overline{\forall\alpha.\sigma_2}$ from which we have:

$$\dfrac{\dfrac{\overline{\Gamma}\vdash {e_1}':\forall\alpha.\overline{\sigma_1}}{\overline{\Gamma}\vdash {e_1}'\{\lceil\overline{\sigma_2}\rceil\}:\overline{\sigma_1}[\lceil\overline{\sigma_2}\rceil/\alpha]} \quad \vdash \mathbf{c}:\overline{\sigma_1}[\lceil\overline{\sigma_2}\rceil/\alpha] \leadsto \overline{\sigma_1}[\overline{\sigma_2}/\alpha]}{\overline{\Gamma}\vdash\langle\mathbf{c}\rangle {e_1}'\{\lceil\overline{\sigma_2}\rceil\}:\overline{\sigma_1}[\overline{\sigma_2}/\alpha]}$$

which proves the case, since $\overline{\sigma_1}[\overline{\sigma_2}/\alpha] = \overline{\sigma_1[\sigma_2/\alpha]}$ and $\forall\alpha.\overline{\sigma_1} = \overline{\forall\alpha.\sigma_1}$.

■

Notice that the representation type ρ of a ϕ-free canonical completion is always completely unboxed, i.e. $\rho \equiv \overline{|\rho|}$.

Lemma 4.29 *ϕ-free canonical completions are ϕ-free.*

Proof:

Since the representation type of all sub-expressions of ϕ-free canonical completions are as unboxed as they can possibly be, ϕ-free canonical completions must be normal forms w.r.t. $\phi^{\rightarrow}E_p^{\leftarrow}$-reduction modulo ψ, since $\phi^{\rightarrow}E_p^{\leftarrow}$-reduction modulo ψ always decreases at least one representation type of one sub-expression. So ϕ-free canonical completions must be ϕ-free.

■

4.6.2 Canonically unboxed completion

The second kind of canonical completions are ψ-free canonical completions. Again we want to define a translation from F_2-expressions into ψ-free canonical completion. To describe this translation we need the following definition:

Definition 4.30 *(Maximally boxed representation type)*

Given a type σ we define $\overline{\overline{\sigma}}$ to be the largest representation type representing σ and which we extend naturally to a homomorphism on type assignment: $x : \overline{\overline{\sigma}} \in \overline{\overline{\Gamma}}$ iff $x : \sigma \in \Gamma$.

In ψ-free canonical completion values should kept as boxed as possible. This can be ensured by boxing values as soon as they are created and not unboxing values before they are consumed by operations that need unboxed values. In F_2 the values are functions (if we consider values created by type abstractions as functions too). These are produce by abstractions and type abstractions, which means that the translation should insert an application of a box coercion on all abstractions and type abstractions. The only places where values are consumed are at applications and type applications so the translation should insert an application of an unbox coercion on the operator expression in all these.

Let us consider what happens with ordinary abstraction and application. Since ordinary function values are created by abstractions, we should translate $\lambda x\!:\!\sigma_1\,.\,e$ into $\langle \mathtt{box}_{\overline{\overline{\sigma_1}}\twoheadrightarrow\overline{\overline{\sigma_2}}}\rangle \lambda x\!:\!\overline{\overline{\sigma_1}}\,.\,e'$ where $\sigma_1\twoheadrightarrow\sigma_2$ is the type of $\lambda x\!:\!\sigma_1\,.\,e$ and e' is the translation of e.

Similarly the translation should also insert an application of an unbox coercion at the operator expression in all application, that is, translate $e_1\ e_2$ into $(\langle \mathtt{unbox}_{\overline{\overline{\sigma_1}}\twoheadrightarrow\overline{\overline{\sigma_2}}}\rangle {e_1}')\ {e_2}'$ where $\sigma_1\twoheadrightarrow\sigma_2$ is the type of e_1 and ${e_i}'$ is the translation of e_i.

Type abstractions and type application is treated similarly as can be seen from Figure 16. We can now give a formal definition of ψ-free canonical completions:

Definition 4.31 *(ψ-free canonical completion).*

We say that an explicitly boxed F_2-expression e_c is a *ψ-free canonical completion* of a F_2-expression e if $\Gamma\vdash e{:}\sigma\Rightarrow e_c$ is derivable from the inference system of Figure 16 for some type environment Γ and some type σ.

$$\frac{\Gamma\{x : \sigma_1\}\vdash e{:}\sigma_2\Rightarrow e'}{\Gamma\vdash\lambda x:\sigma_1\,.\,e{:}\sigma_1\twoheadrightarrow\sigma_2\Rightarrow\langle\mathtt{box}_{\overline{\overline{\sigma_1}}\twoheadrightarrow\overline{\overline{\sigma_2}}}\rangle\lambda x:\overline{\overline{\sigma_1}}\,.\,e'}$$

$$\frac{\Gamma\vdash e_1{:}\sigma_1\twoheadrightarrow\sigma_2\Rightarrow e_1{}' \quad \Gamma\vdash e_2{:}\sigma_1\Rightarrow e_2{}'}{\Gamma\vdash e_1\ e_2{:}\sigma_2\Rightarrow(\langle\mathtt{unbox}_{\overline{\overline{\sigma_1}}\twoheadrightarrow\overline{\overline{\sigma_2}}}\rangle e_1{}')\ e_2{}'}$$

$$\frac{\Gamma\vdash e{:}\sigma\Rightarrow e'}{\Gamma\vdash\Lambda\alpha\,.\,e{:}\forall\alpha.\sigma\Rightarrow\langle\mathtt{box}_{\forall\alpha.\overline{\overline{\sigma}}}\rangle\Lambda\alpha\,.\,e'}\ \text{if } \alpha\ \notin\ FV(\Gamma)$$

$$\frac{\Gamma\vdash e{:}\forall\alpha.\sigma\Rightarrow e'}{\Gamma\vdash e\{\sigma_1\}{:}\sigma[\sigma_1/\alpha]\Rightarrow(\langle\mathtt{unbox}_{\forall\alpha.\overline{\overline{\sigma}}}\rangle e')\{\overline{\overline{\sigma_1}}\}}$$

$$\Gamma\{x : \sigma\}\vdash x{:}\sigma\Rightarrow x$$

Figure 16: ψ-free canonical completion for F_2

Lemma 4.32 *ψ-free canonical completions are well-typed w.r.t. the inference system in Figure 5, i.e. if $\Gamma\vdash e{:}\sigma$ and $\Gamma\vdash e{:}\sigma\Rightarrow e'$ then $\overline{\overline{\Gamma}}\vdash e{:}\overline{\overline{\sigma}}$.*

Proof:

The proof is by induction on the structure of the F_2-expression e or equivalent on the structure of the derivation of $\Gamma\vdash e{:}\sigma$.

Base case: x. From $\Gamma\{x : \sigma\}\vdash x{:}\sigma\Rightarrow x$ it is obvious that $\overline{\overline{\Gamma}}\{x : \overline{\overline{\sigma}}\}\vdash x{:}\overline{\overline{\sigma}}$ holds.

Inductive case: $\lambda x:\sigma_x\,.\,e_1$. By induction we know from $\Gamma\{x : \sigma\}\vdash e_1{:}\sigma_1\Rightarrow e_1{}'$ that $\overline{\overline{\Gamma}}\{x : \overline{\overline{\sigma_x}}\}\vdash e_1{}'{:}\overline{\overline{\sigma_1}}$ which gives us:

$$\frac{\dfrac{\overline{\overline{\Gamma}}\{x : \overline{\overline{\sigma_x}}\}\vdash e_1{}'{:}\overline{\overline{\sigma_1}}}{\overline{\overline{\Gamma}}\vdash\lambda x:\overline{\overline{\sigma_x}}\,.\,e_1{}'{:}\overline{\overline{\sigma_x}}\twoheadrightarrow\overline{\overline{\sigma_1}}} \qquad \vdash\mathtt{box}_{\overline{\overline{\sigma_x}}\twoheadrightarrow\overline{\overline{\sigma_1}}}:\overline{\overline{\sigma_x}}\twoheadrightarrow\overline{\overline{\sigma_1}}\leadsto[\overline{\overline{\sigma_x}}\twoheadrightarrow\overline{\overline{\sigma_1}}]}{\overline{\overline{\Gamma}}\vdash\langle\mathtt{box}_{\overline{\overline{\sigma_x}}\twoheadrightarrow\overline{\overline{\sigma_1}}}\rangle\lambda x:\overline{\overline{\sigma_x}}\,.\,e_1{}'{:}[\overline{\overline{\sigma_x}}\twoheadrightarrow\overline{\overline{\sigma_1}}]}$$

which prove the case since $[\overline{\overline{\sigma_x}}\twoheadrightarrow\overline{\overline{\sigma_1}}]=\overline{\overline{\sigma_x\rightarrow\sigma_1}}$.

Inductive case: $e_1\ e_2$. By induction we have from $\Gamma\vdash e_1{:}\sigma_2\twoheadrightarrow\sigma\Rightarrow e_1{}'$ and $\Gamma\vdash e_2{:}\sigma_2\Rightarrow e_2{}'$ that $\overline{\overline{\Gamma}}\vdash e_1{}'{:}\overline{\overline{\sigma_2\rightarrow\sigma}}$ and $\overline{\overline{\Gamma}}\vdash e_2{}'{:}\overline{\overline{\sigma_2}}$ which means we have:

$$\frac{\dfrac{\overline{\overline{\Gamma}}\vdash e_1{}'{:}[\overline{\overline{\sigma_2}}\twoheadrightarrow\overline{\overline{\sigma}}] \quad \vdash\mathtt{unbox}_{\overline{\overline{\sigma_2}}\twoheadrightarrow\overline{\overline{\sigma}}}:[\overline{\overline{\sigma_2}}\twoheadrightarrow\overline{\overline{\sigma}}]\leadsto\overline{\overline{\sigma_2}}\twoheadrightarrow\overline{\overline{\sigma}}}{\overline{\overline{\Gamma}}\vdash\langle\mathtt{unbox}_{\overline{\overline{\sigma_2}}\twoheadrightarrow\overline{\overline{\sigma}}}\rangle e_1{}'{:}\overline{\overline{\sigma_2}}\twoheadrightarrow\overline{\overline{\sigma}}} \qquad \overline{\overline{\Gamma}}\vdash e_2{}'{:}\overline{\overline{\sigma_2}}}{\overline{\overline{\Gamma}}\vdash(\langle\mathtt{unbox}_{\overline{\overline{\sigma_2}}\twoheadrightarrow\overline{\overline{\sigma}}}\rangle e_1{}')\ e_2{}'{:}\overline{\overline{\sigma}}}$$

which proves the case, since $\overline{\overline{\sigma_2 \rightarrow \sigma}} = [\overline{\overline{\sigma_2}} \nrightarrow \overline{\overline{\sigma}}]$.

Inductive case: $\Lambda\alpha\,.\,e_1$. By induction we have from $\Gamma \vdash e_1 : \sigma_1 \Rightarrow e_1{}'$ that $\overline{\overline{\Gamma}} \vdash e_1{}' : \overline{\overline{\sigma_1}}$ which means we have:

$$\dfrac{\dfrac{\overline{\overline{\Gamma}} \vdash e_1{}' : \overline{\overline{\sigma_1}}}{\overline{\overline{\Gamma}} \vdash \Lambda\alpha\,.\,e_1{}' : \forall\alpha.\overline{\overline{\sigma_1}}} \qquad \vdash \mathtt{box}_{\forall\alpha.\overline{\overline{\sigma_1}}} : \forall\alpha.\overline{\overline{\sigma_1}} \leadsto [\forall\alpha.\overline{\overline{\sigma_1}}]}{\overline{\overline{\Gamma}} \vdash \langle \mathtt{box}_{\forall\alpha.\overline{\overline{\sigma_1}}} \rangle \Lambda\alpha\,.\,e_1{}' : [\forall\alpha.\overline{\overline{\sigma_1}}]}$$

which proves the case, since $[\forall\alpha.\overline{\overline{\sigma_1}}] = \overline{\overline{\forall\alpha.\sigma_1}}$ and $\alpha \notin FV(\Gamma) \Rightarrow \alpha \notin FV(\overline{\overline{\Gamma}})$.

Inductive case: $e_1\{\sigma_2\}$. By induction we have from $\Gamma \vdash e_1 : \forall\alpha.\sigma_1 \Rightarrow e_1{}'$ that $\overline{\overline{\Gamma}} \vdash e_1{}' : \overline{\overline{\forall\alpha.\sigma_1}}$ which means we have:

$$\dfrac{\dfrac{\overline{\overline{\Gamma}} \vdash e_1{}' : [\forall\alpha.\overline{\overline{\sigma_1}}] \qquad \vdash \mathtt{unbox}_{\forall\alpha.\overline{\overline{\sigma_1}}} : [\forall\alpha.\overline{\overline{\sigma_1}}] \leadsto \forall\alpha.\overline{\overline{\sigma_1}}}{\overline{\overline{\Gamma}} \vdash (\langle \mathtt{unbox}_{\forall\alpha.\overline{\overline{\sigma_1}}} \rangle e_1{}') : \forall\alpha.\overline{\overline{\sigma_1}}}}{\overline{\overline{\Gamma}} \vdash \langle \mathtt{unbox}_{\forall\alpha.\overline{\overline{\sigma_1}}} \rangle e_1{}' \{\overline{\overline{\sigma_2}}\} : \overline{\overline{\sigma_1}}[\overline{\overline{\sigma_2}}/\alpha]}$$

which proves the case, since $\overline{\overline{\forall\alpha.\sigma_1}} = [\forall\alpha.\overline{\overline{\sigma_1}}]$ and $\overline{\overline{\sigma_1}}[\overline{\overline{\sigma_2}}/\alpha] = \overline{\overline{\sigma_1[\sigma_2/\alpha]}}$.

■

Lemma 4.33 *ψ-free canonical completions are ψ-free.*

Proof:

Since the representation type of all sub-expressions of ψ-free canonical completions are as boxed as they can possibly be, ψ-free canonical completions must be normal forms w.r.t. $\psi^{\rightarrow} E_p^{\leftarrow}$-reduction modulo ϕ, since $\psi^{\rightarrow} E_p^{\leftarrow}$-reduction modulo ϕ always increases at least one representation type of one sub-expression. So ψ-free canonical completions must be ψ-free.

■

Chapter 5

Computing optimal completions

We have now established that there exist optimal completions in the form of two E-equivalence classes: ϕ-free optimal completions and ψ-free optimal completions. We used a reduction system to prove these things, but this was still on a fairly theoretical level. In this chapter we will show how to compute optimal completions. We will present two basically different methods to compute optimal completions. The first one is based on reduction using methods developed in Chapters 2, 3 and 4. We will describe how to implement $\psi^{\rightarrow}E_p^{\leftarrow}$-reduction on ϕ-free completions to find ϕ-free optimal completions and $\phi^{\rightarrow}E_p^{\rightarrow}$-reduction on ψ-free completions to find ψ-free optimal completions. The second method is developed in this chapter and is based on transforming the problem of finding optimal completions into what essentially becomes a graph reachability problem. It is on this second method that our prototype implementation describe in Chapter 7 is based.

5.1 Rewriting-based algorithm

The goal of reduction is to find optimal ψ-free or ϕ-free completions. We can always find a congruent ψ-free or ϕ-free completion for a given completion, since we can just start from the underlying F_2-expression and construct the canonically unboxed (ψ-free) or canonically boxed (ϕ-free) completion. These canonical completions are not necessarily the same completions that we get by $E_p^{\leftarrow}\psi^{\rightarrow}$-reduction modulo ϕ and $E_p^{\rightarrow}\phi^{\rightarrow}$-reduction modulo ψ on some other congruent completion. In fact, this will very seldom be the case, since the canonical completions somehow represents the worst cases (most boxed or most unboxed). But since we have to find some completion from a given F_2-expression to start with we might as well choose a canonical one. That a canonical completion is just as good a starting point as any other completion can be seen by Theorem 4.25: reduction will eventually lead to the same optimal ψ-free (or ϕ-free) completions as the ones that we would have obtained by starting from an arbitrary completion using first $E_p^{\leftarrow}\psi^{\rightarrow}/\phi$-reduction followed by $E_p^{\rightarrow}\phi^{\rightarrow}$-reduction (or first $E_p^{\rightarrow}\phi^{\rightarrow}/\psi$-reduction followed by $E_p^{\leftarrow}\psi^{\rightarrow}$-reduction). This means that $E_p^{\leftarrow}\psi^{\rightarrow}$-reduction modulo ϕ and $E_p^{\rightarrow}\phi^{\rightarrow}$-reduction modulo ψ is not something that we need to think about how to do in an actual implementation.

This means that we only need to work out how to implement $E_p^{\leftarrow}\psi^{\rightarrow}$-reduction and $E_p^{\rightarrow}\phi^{\rightarrow}$-reduction.

5.1.1 ψ-reduction on ϕ-free completions

Let us consider $E_p^{\leftarrow}\psi^{\rightarrow}$-reduction first. The first observation that we may make is that since we start reduction with ϕ-free completions and $E_p^{\leftarrow}\psi^{\rightarrow}$-reduction never introduces any ϕ-redexes (see the proof of Theorem 4.25), we only need to work out how to do $E_p^{\leftarrow}\psi^{\rightarrow}$-reduction on ϕ-free completions. We may therefore use full $\psi\phi$-reduction (Figure 8) for coercion. This is a very useful property that enable us to use type signatures as representations of coercions, in which case coercion reduction is no longer something we need to do, since we can always regain the optimal coercions by construction the canonical coercions (see the comment after Lemma 3.11).

We will here outline how $E_p^{\leftarrow}\psi^{\rightarrow}$-reduction on ϕ-free completions could be implemented:

Input: a ϕ-free completion e.

1. reduce all coercions in e to $\phi\psi$-normal form using R-reduction. This is legal since no coercion can contain any ϕ-redex because e is ϕ-free. Actually, since we use signatures instead of coercions this operation consists of replacing all coercions by their signatures, but it is conceptually nice to think of it as doing reduction and as if we are still working with coercions terms. If we start from the canonical ϕ-free completion this step is not necessary.

2. normalize e such that every head coercion free subexpression in e has exactly one coercion applied to it. We use equations 11 and 12 for this.

3. $+/-$-factorize all coercions in e and split all coercion applications into two coercion applications using Equation 12. This means that every head coercion free subexpression e_1 now has exactly two coercion applied to is, as follows:

   $$\langle \mathbf{c}^-\rangle\langle \mathbf{d}^+\rangle e_1$$

   We call completions of this form $+/-$-*normalized.*

4. now look for a $E_p^{\leftarrow}$-redex in e and perform a $E_p^{\leftarrow}$-reduction step on e if such a redex is found. Stop if no redex is found.

5. after the $E_p^{\leftarrow}$-reduction step the completion e may (in the worst case) contain subexpressions of the form

   $$\langle \mathbf{c}_1{}^+\rangle\langle \mathbf{c}_2{}^-\rangle\langle \mathbf{d}_1{}^+\rangle\langle \mathbf{d}_2{}^-\rangle e_1$$

   by Equation 12 these are rewritten to

   $$\langle \mathbf{d}_2{}^-\,;\mathbf{d}_1{}^+\,;\mathbf{c}_2{}^-\,;\mathbf{c}_1{}^+\rangle e_1$$

   and $+/-$-factorized using ψ-reduction (Lemma 3.28).

6. goto step 4.

$E_p^{\leftarrow}\psi^{\rightarrow}$-reduction as described here is of course not very efficient. In an efficient implementation one would not factor all coercions but only the proper ones and then when looking for $E_p^{\leftarrow}$-redexes take into account that one can always add identity coercions when needed.

We will in the following look at how to implement the different elements in this process. Especially, since there are some complications with the rules 13^-, 17^- and 17^+, we will consider how to implement these rules. But first we will consider how to implement $+/-$-factorization.

Implementing $+/-$-factorization

When coercions are represented by their type signature $+/-$-factorization becomes fairly easy. Assume that $\mathbf{c}$ is a coercion that we want to $+/-$-factorize into $\mathbf{c}_1{}^+;\mathbf{c}_2{}^-$ where both $\mathbf{c}_1$ and $\mathbf{c}_2$ are optimal (canonical) coercions and such that $\mathbf{c} \Rightarrow_\psi \mathbf{c}_1{}^+;\mathbf{c}_2{}^-$. Since also $\mathbf{c}_1$ and $\mathbf{c}_2$ are going to be represented by their signature all we have to do is to find the representation type corresponding to the common range type of $\mathbf{c}_1$ and domain have type of $\mathbf{c}_2$. Let us call this type ρ. Since $\mathbf{c}_1$ has to be positive, ρ has to be greater than or equal to the domain type ρ_1 of $\mathbf{c}_1$ and similarly, since $\mathbf{c}_2$ has to be negative, ρ has to be greater than or equal to the range type ρ_2 of $\mathbf{c}_2$. We cannot make ρ too large, because we need $\mathbf{c} \Rightarrow_\psi \mathbf{c}_1{}^+;\mathbf{c}_2{}^-$ to hold. If we were to introduce a boxed subtype in ρ where the corresponding subtype in neither ρ_1 nor ρ_2 is boxed then we need ϕ-equality to prove $\mathbf{c}_1{}^+;\mathbf{c}_2{}^-$ equivalent to $\mathbf{c}$, since ϕ-equality is the only way that a new "intermediate" and "more boxed" type can be introduced, i.e. in $\mathtt{box}_\upsilon;\mathtt{unbox}_\upsilon = \iota_\upsilon$ the range type $[\upsilon]$ of $\mathtt{box}_\upsilon$ is more boxed than common domain and range type of ι_υ. This means that we have to make ρ the smallest type greater than or equal to ρ_1 and ρ_2, that is:

$$\rho = \rho_1 \sqcup \rho_2$$

Implementing rule 13^-

Consider rule 13^-:

$$\lambda x\,.\langle \mathbf{d}^-\rangle(e[\langle \mathbf{c}^+\rangle x/x]) \Rightarrow \langle(\mathbf{c}^+ \rightarrow \mathbf{d}^-)\rangle\lambda x\,.e$$

Assume that the potential redex looks something like:

$$\lambda x\,.\langle \mathbf{d}\rangle(\ldots\langle \mathbf{c}_i{}^+\rangle x\ldots)$$

with n occurrences of x in the body of the abstraction with n positive coercions $\mathbf{c}_1$ ... $\mathbf{c}_n$ one at each occurrence. It might be that these are not all the same and that we therefore cannot apply the rule. However, if we can find a proper coercion $\mathbf{c}$ such that for each of the coercions $\mathbf{c}_i$ there exists a positive coercion $\mathbf{c}_i{}'$ such that:

$$\vdash \mathbf{c}_i = \mathbf{c}\,;\mathbf{c}_i{}'$$

then we may split each of the coercion applications at the occurrences of x into

$$\langle \mathbf{c}_i{}'^+\rangle\langle \mathbf{c}^+\rangle x$$

for some coercions $\mathbf{c}_i{}'$ and then apply rule 13^-. The largest possible coercion $\mathbf{c}$ that fulfills this criteria can be found in the following way. Let the range type of $\mathbf{c}$ be ρ. If the range type of the coercion $\mathbf{c}_i$ are ρ_i then we have $\rho_x \leq \rho \leq \rho_i$ for all i. This means that we may choose $\mathbf{c}$ to be the canonical coercion with the signature:

$$\rho_x \rightsquigarrow \rho_1 \sqcap \ldots \sqcap \rho_n$$

where ρ_x is the type of x and $\sqcap$ is the greatest lower bound in the representation type lattice (see Proposition 3.33).

Another thing that might seem like a problem with rule 13^- is that if x does not occur free in the body of e then it seems like one can keep pulling coercions out from the abstraction. But this is not the case since for every coercion we pull out, the type of the formal parameter will decrease, and this cannot go on forever.

Implementing rule 17^- and rule 17^+

The way we handle rule 17^- and 17^+ is similar to rule 13^-. Let us look at one of them, 17^+ (the other is handled similarly):

$$\langle \widehat{\rho}(\iota, \mathbf{c}^+) \rangle e\{\pi\} \Rightarrow \langle \widehat{\rho}(\mathbf{c}^+, \iota) \rangle e\{\pi'\}$$

In this rule the type of e is $\forall \alpha . \rho$ for some type variable α. If we look at a potential redex for this rule:

$$\langle \mathbf{d}^+ \rangle e\{\pi\}$$

then it is not likely that $\mathbf{d}$ matches $\widehat{\rho}(\iota,\mathbf{c}^+)$ since this demands a lot of the structure of $\mathbf{d}$. So again we must factor out a proper part of $\mathbf{d}$ that has the right form, that is, it matches $\widehat{\rho}(\iota,\mathbf{c}^+)$ for some proper coercion $\mathbf{c}$. Assume that this is possible then we will get a reduction of the form:

$$\begin{gathered}\langle \mathbf{d}^+ \rangle e\{\pi\} = \\ \langle \mathbf{d}'^+ \rangle \langle \widehat{\rho}(\iota, \mathbf{c}^+) \rangle e\{\pi\} \Rightarrow \\ \langle \mathbf{d}'^+ \rangle \langle \widehat{\rho}(\mathbf{c}^+, \iota) \rangle e\{\pi'\} = \\ \langle \mathbf{d}''^+ \rangle e\{\pi'\}\end{gathered}$$

In the splitting of $\mathbf{d}$ we are only allowed to use the pure coercion equality (without ψ and ϕ), that is:

$$\vdash \mathbf{d} = \widehat{\rho}(\iota, \mathbf{c}^+) ; \mathbf{d}'$$

this also means that, since $\mathbf{d}$ is positive then so must $\mathbf{d}'$ be. The coercion $\mathbf{c}$ in this equation will have signature $\pi \rightsquigarrow \pi'$ which means that the coercion $\widehat{\rho}(\iota,\mathbf{c}^+)$ must have signature $\rho[\alpha/\pi] \rightsquigarrow \rho[\alpha^-/\pi][\alpha^+/\pi']$ where α^-/α^+ are all the negative/positive occurrences of α in ρ. So the question of finding the new coercion $\mathbf{d}''$ can be rephrased to the question of finding the largest boxed type π' such that $\mathbf{d}'$ is still positive. If the type of the whole expression is ρ', then the condition that π' must fulfill can be written:

$$\rho[\alpha^-/\pi][\alpha^+/\pi'] \leq \rho'$$

This means that we may go over the two types structurally and collect all constraint on π'. This will give us a set of constraints of the form:

$$\pi' \leq \rho_i$$

We may then find the maximal solution to the set of constraints by setting:

$$\pi' \;= \rho_1 \sqcap \ldots \sqcap \rho_n$$

This works because ρ regarded as a function of α is monotone with respect to our ordering on representation types. We then perform the reduction if π' is different from π.

Let us finally give an example to illustrate the use of rule 17^{-}. Let *id* be the identity function and consider the following expression:

$$\langle [\mathtt{box}_{\mathtt{int}} \rightarrow \iota_{\mathtt{int}}] \rightarrow [\mathtt{unbox}_{\mathtt{int}} \rightarrow \mathtt{box}_{\mathtt{int}}] \rangle id\{[\mathtt{int} \rightarrow \mathtt{int}]\}$$

The type of the expression is $[[\mathtt{int}]\!\mapsto\!\mathtt{int}]\!\mapsto\![[\mathtt{int}]\!\mapsto\![\mathtt{int}]]$. This gives only one constraint on π':

$$\pi' \leq [[\mathtt{int}] \rightarrow [\mathtt{int}]]$$

which means that we may set π' to $[[\mathtt{int}]\!\mapsto\![\mathtt{int}]]$. Since one may use signature to represent coercions then the reduction step is now very easy, because $\mathbf{d}''$ is now simply the canonical coercion from $[[\mathtt{int}]\!\mapsto\![\mathtt{int}]]\!\mapsto\![[\mathtt{int}]\!\mapsto\![\mathtt{int}]]$ to $[[\mathtt{int}]\!\mapsto\!\mathtt{int}]\!\mapsto\![[\mathtt{int}]\!\mapsto\![\mathtt{int}]]$, that is, the coercion $[\iota_{[\mathtt{int}]}\!\rightarrow\!\mathtt{box}_{\mathtt{int}}]\!\rightarrow\!\iota_{[[\mathtt{int}]\mapsto[\mathtt{int}]]}$.

5.1.2 ϕ-reduction on ψ-free completions

$\phi^{\rightarrow}E_p^{\rightarrow}$-reduction is implemented completely analogue to $E_p^{\leftarrow}\psi^{\rightarrow}$-reduction we just use $\phi^{\rightarrow}$ instead of $\psi^{\rightarrow}$,$E_p^{\rightarrow}$ instead of $E_p^{\leftarrow}$, $-/+$-factorization instead of $+/-$-factorization, $\sqcup$ instead of $\sqcap$ and $\sqcap$ instead of $\sqcup$.

5.1.3 An Example

Let us look at a small example of $E_p^{\leftarrow}\psi^{\rightarrow}$-reduction. The example is the following F_2-expression:

$$\mathtt{id\{int\}\ (id\{int\}\ 5)}$$

where `id` is the polymorphic identity function $\Lambda\alpha.\lambda x\!:\!\alpha.x$. The ϕ-free canonical completions of this is:

$$(\langle\mathtt{box} \rightarrow \mathtt{unbox}\rangle\mathtt{id\{int\}})\ ((\langle\mathtt{box} \rightarrow \mathtt{unbox}\rangle\mathtt{id\{int\}})\ \mathtt{5})$$

We will now describe what happens when we apply reduction to this. There is no need to reduce the coercions to normal form (Step 1), since all coercions are in normal form, so this leaves the completions unchanged. If we perform step 2 and 3 we get the $-/+$-normal form:

$$\langle\iota\rangle\langle\iota\rangle((\langle\mathtt{box} \rightarrow \mathtt{unbox}\rangle\langle\iota\rangle\mathtt{id\{int\}})\ \langle\iota\rangle\langle\iota\rangle((\langle\mathtt{box} \rightarrow \mathtt{unbox}\rangle\langle\iota\rangle\mathtt{id\{int\}})\ \langle\iota\rangle\langle\iota\rangle\mathtt{5}))$$

If we look for a $E_p^{\leftarrow}$-redexes in this term we find two $R14^{-}$-redexes. If we performs the first one of these we get:

$$\langle\iota\rangle\langle\iota\rangle\langle\mathtt{unbox}\rangle((\langle\iota\rangle\mathtt{id\{int\}})\ \langle\mathtt{box}\rangle\langle\iota\rangle\langle\iota\rangle((\langle\mathtt{box} \rightarrow \mathtt{unbox}\rangle\langle\iota\rangle\mathtt{id\{int\}})\ \langle\iota\rangle\langle\iota\rangle\mathtt{5}))$$

If we $-/+$-normalize this completion (step 5) we get:

$$\langle\texttt{unbox}\rangle\langle\iota\rangle(((\langle\iota\rangle\langle\iota\rangle\texttt{id}\{\texttt{int}\})\ \langle\iota\rangle\langle\texttt{box}\rangle(((\langle\texttt{box}\to\texttt{unbox}\rangle\langle\iota\rangle\texttt{id}\{\texttt{int}\})\ \langle\iota\rangle\langle\iota\rangle\texttt{5}))$$

We the go to step 4 and find one more $R14^{-}$-redex. After the reduction step we get:

$$\langle\texttt{unbox}\rangle\langle\iota\rangle(((\langle\iota\rangle\langle\iota\rangle\texttt{id}\{\texttt{int}\})\ \langle\iota\rangle\langle\texttt{box}\rangle\langle\texttt{unbox}\rangle(((\langle\texttt{box}\to\texttt{unbox}\rangle\langle\iota\rangle\texttt{id}\{\texttt{int}\})\ \langle\texttt{box}\rangle\langle\iota\rangle\langle\iota\rangle\texttt{5}))$$

After yet another $-/+$-normalization we get:

$$\langle\texttt{unbox}\rangle\langle\iota\rangle(((\langle\iota\rangle\langle\iota\rangle\texttt{id}\{\texttt{int}\})\ \langle\iota\rangle\langle\iota\rangle(((\langle\iota\rangle\texttt{id}\{\texttt{int}\})\ \langle\iota\rangle\langle\texttt{box}\rangle\texttt{5}))$$

which contains no $E_p^{\leftarrow}$-redexes. The normal form completion is the found by "cleaning up" this completion using rule 11 and 12 (in this case 11 is enough):

$$\langle\texttt{unbox}\rangle((\texttt{id}\{\texttt{int}\})\ ((\texttt{id}\{\texttt{int}\})\ \langle\texttt{box}\rangle\texttt{5}))$$

5.1.4 Complexity analysis

Let us reason about the complexity of a possible implementation of reduction of completion based on the description above. In the part of the proof of Lemma 4.15, where we show strong normalization of $\psi^{\to}E_p^{\leftarrow}$-reduction modulo ϕ, we showed that an $E_p^{\leftarrow}$ step rewrites at least one coercion $\vdash \mathbf{c} : \rho \rightsquigarrow \rho'$ to a new coercion $\mathbf{c}'$ that has domain type or range type properly increased in the subtype hierarchy of Definition 3.29. The same argument holds for $\psi^{\to}E_p^{\leftarrow}$-reduction (not modulo ϕ) as well as $\phi^{\to}E_p^{\to}$-reduction (not modulo ψ), which is actually the kind of reduction that an implementation should perform. Now this is the same as saying, that for an $E_p^{\leftarrow}$ step the representation type of at least one subexpression of the term that is rewritten, is properly increased. If the program size is n measured in terms of the number of subexpressions in the program (not including the size of type annotations) and the maximum size of any type of a subexpression in the program is m, then the maximal number of $E_p^{\leftarrow}$ step that can be performed is $\Theta(nm)$. Before each step one has to find the next redex. We expect that it may be possible to do this in constant time given a work list of pending redexes. Furthermore, one might also in the worst case have to perform $\Theta(k)$ $\sqcup/\sqcap$-operations (to do $+/-$ or $-/+$ factorization, finding the coercion to "pull out" of an abstraction, etc.) before doing the next $E_p^{\leftarrow}$ step, where k is the maximum number of non-defining occurrences of each variable in programs, e.g. free variables x in bodies of abstractions $\lambda x.e$. Each of these operations may take time $\Theta(m)$ in the worst case. This gives a another factor $\Theta(km)$ which should be multiplied with the number of $E_p^{\leftarrow}$ steps giving the total worst case time complexity $\Theta(nkm^2)$. The factor k may in the worst case be proportional to n so one may also express the worst case time complexity less informatively by $\Theta(n^2m^2)$.

If we are performing reductions on programs that correspond to ML-typable programs then the size n used above is also the size of the corresponding ML-programs. It is a well-known fact that ML-typing can produce types that are at least exponential in the size of the programs that are being typed. This means, that m may be exponential in n and that time complexity of $E_p^{\leftarrow}$-reduction modulo ϕ for ML-programs may be exponential in the size of the ML-programs. On the other hand, ML-programmers know that this worst case behaviour hardly ever occurs in real programs, only in contrived examples. In general m will be quite small and so will the number k of $\sqcup$-operations which have to be performed for each $E_p^{\leftarrow}$ step also be. So one should expect that the actual time complexity would be more like $\Theta(nm^2)$ or $\Theta(n)$, if we regard m as a small constant factor independent of the size of programs (that types in large programs are not more complex than types in small programs).

We may also consider the space complexity of a possible implementation of completion reduction. Since coercions are represented by signatures (pairs of representation types), the size of a the terms that we do reduction on may be of the order $\Theta(nm)$. So the worst case space complexity for the reduction approach must be $\Theta(nm)$, but is in the average case more likely to be $\Theta(n)$.

5.2 Graph-based algorithm

In this section we will present a different algorithm for finding optimal completions. The algorithm is not based on a rewrite system as the ones described in Section 5.1, but on finding an assignment of so called box values to a set of box variables extracted from a F_2-expression.

The algorithm can be summarized as follows. First a *principal completion*[1] is generated from the F_2-expression. This is a special kind of completion parameterized over variables called *box variables.* In a principal completion every head coercion free expression has a coercion applied to them and coercions are represented by their type signatures. The first type in these signatures corresponds to the type of the head coercion free expression itself, while the second type corresponds to the type of the expression with the coercion. In all types in the principal completions all type constructors are annotated with a unique box variable b or a *box value* (U or B). When box values are substituted for the box variable in a principal completion a new expression is obtained, which is equivalent to a given completion of the original F_2-expression. If a type constructor is annotated with U/B then the constructed type is unboxed/boxed respectively. Given an F_2-expression e and a completion $\overline{e}$ (with only canonical coercions) of e at some given type, it is always possible to find a substitution ς such that we may obtain $\overline{e}$ by applying ς to the box variables in the principal completion of e.

From the principal completion of an F_2-expression e we generate a graph, called a *representation* graph, where the box variables correspond to the nodes and an edge in the graph corresponds to a change in representation.

Representation graphs are used to find an assignment of box values to the set of box variables, by minimizing the number of edges that correspond to an actual change in representation. These assignments are substitutions which when applied to the principal completion of e result in an expression equivalent to an optimal completion of e. We will show how to find assignments of box values that correspond to both optimal ψ-free completions and optimal ϕ-free completions.

5.2.1 Principal completions

Before we describe how to obtain principal completions we need to explain the notation used in principal completions. The syntax of representation types and coercions in principal completions is slightly different from representation types and coercions in ordinary completions. *Principal representation types* ϱ are ordinary F_2-types with a *box annotation* on all type constructors. Box annotations are either a *box value* $\beta \in \{U, B\} = BoxVal$ or a *box variable* b. Box variables range over box values. The syntax of types in principal completions is shown in Figure 17.

[1]This name is borrowed from Poulsen [Pou93] although our principal completions are somewhat different than those of Poulsen; they serve the same purpose.

We say that a principal representation type is *ground* if it contains no box variables. All representation types can be represented by ground principal representation types. Boxed representation types $[v]$ are represented by a type having a B annotated on the topmost type constructor of the representation of v, and coercions are represented by their type signatures. For example, if ρ_i is represented by the type ϱ_i, then the type $[\rho_1 \to \rho_2]$ is represented by $\varrho_1 \to^B \varrho_2$, and the coercion $\texttt{box}_{\texttt{int}}$ is represented by $\texttt{int}^U \leadsto \texttt{int}^B$.

Type erasure for principal representation types is defined similarly to type erasure for ordinary representation types (we use the same notation).

$\alpha \in \mathit{TypeVariable}$; $\beta \in \mathrm{BoxVal}$; $b \in \mathit{BoxVariable}$	
$\varrho ::= \alpha \mid \varrho \to^{\mu} \varrho \mid \forall^{\mu}\alpha.\varrho$	*Principal representation types*
$\beta ::= U \mid B$	*Box value*
$\mu ::= b \mid \beta$	*Box annotation*

Figure 17: Principal representation types

Expressions in principal completions have the same syntax as explicitly boxed F$_2$-expressions except that the types are now principal representation types and that coercion application is written $<\varrho \leadsto \varrho'> e$. If both types are ground, then $<\varrho \leadsto \varrho'> e$ is the application of the optimal coercion **c** with signature $\rho \leadsto \rho'$ to e, where ρ and ρ' are the types represented by ϱ and ϱ'. In Chapter 3 Theorem 3.18 we showed that for a given signature optimal coercions are unique (modulo core coercion equality) and therefore type signatures can be used to represent coercions, since we have a way of finding optimal coercions from these. This is what we will do here. Of course $<\varrho \leadsto \varrho'> e$ will in general be parameterized over box variables, but when we substitute primitive box annotations for these we get an application of a given optimal coercion. By substituting U's and B's for all box variables in a principal completion one obtains a representation of a completion for the original F$_2$-expression. The point is, that this will not give us the completion directly, but if we replace all types ϱ by the type they represent and replace type signatures by their corresponding canonical coercions, then we obtain a real F$_2$-completion.

$$\tilde{\Gamma}\{x : \varrho_x\} \vdash <\varrho_x \leadsto \varrho'> x : \varrho' \qquad \frac{\tilde{\Gamma}\{x : \varrho_x\} \vdash \tilde{e} : \varrho}{\tilde{\Gamma} \vdash <\varrho_x \to^U \varrho \leadsto \varrho'> \lambda x : \varrho_x . \tilde{e} : \varrho'}$$

$$\frac{\tilde{\Gamma} \vdash \tilde{e}_1 : \varrho_2 \to^U \varrho \qquad \tilde{\Gamma} \vdash \tilde{e}_2 : \varrho_2}{\tilde{\Gamma} \vdash <\varrho \leadsto \varrho'> (\tilde{e}_1\ \tilde{e}_2) : \varrho'} \qquad \frac{\tilde{\Gamma} \vdash \tilde{e} : \varrho}{\tilde{\Gamma} \vdash <\forall^U \alpha.\varrho \leadsto \varrho'> \Lambda\alpha . \tilde{e} : \varrho'}$$

$$\frac{\tilde{\Gamma} \vdash \tilde{e} : \forall^U \alpha.\varrho}{\tilde{\Gamma} \vdash <\varrho[\varrho_1^B/\alpha] \leadsto \varrho'> e\{\varrho_1^B\} : \varrho'}$$

Side condition: all distinct types in a derivation have distinct box variables.

Figure 18: Inference rules for generating principal completions

Figure 18 shows an inductive system for generating principal completions. In this figure expressions $\tilde{e}$, etc. are principal completions. A principal representation type ϱ with a superscript U/B, e.g. ${\varrho_1}^B$, is a principal representation type where the box annotation on the topmost type constructor is U/B respectively. If the principal representation type is a type variable α then the is no topmost type constructor and ${\alpha_1}^B \equiv \alpha$.

Definition 5.1 *(Principal Completions)*
Let e be a F_2-expression with typing judgement $\Gamma \vdash e{:}\sigma$ then $\tilde{e}$ is a *principal completion* of e if $|\tilde{e}| \equiv e$ and if:

$$\tilde{\Gamma} \vdash \tilde{e}{:}\varrho$$

is derivable from the rules of Figure 18, where $\tilde{\Gamma}$ is a type assignment obtained from Γ by replacing every type assumption $x{:}\sigma_x$ in Γ by $x{:}\varrho_x$ where ϱ_x is a copy of σ_x where all type constructors have been annotated with box values. Similarly, ϱ is a copy of σ where all type constructor have been annotated with box values.

A few comments on this definition are in order. Principal completions, as we have defined them here, are defined with respect to fixed assumptions on the representation type of the completion and its free variables. We could have used boxed variables instead of box values in $\tilde{\Gamma}$ and ϱ to get an alternative definition. This is probably what one would want to do if one were interested in using boxing analysis in separate compilation, but we have chosen to use the slightly simpler approach, since we are not concerned with separate compilation in this thesis.

For a given F_2-expression e we will from now on write $\tilde{e}$ for one of its principal completions (if the choice of type assignment and representation type is not important). If ς is a box value assignment for which the set of box variables in $\tilde{e}$ is a subset of $\mathbf{dom}(\varsigma)$ we write $\varsigma(\tilde{e})$ for the corresponding completion obtained by applying ς to all box variables in $\tilde{e}$, replacing the principal representation types by ordinary representation types and inserting canonical coercions for the type signatures in $\tilde{e}$. That $\varsigma(\tilde{e})$ is a completion for any ς is straightforward to verify, but can all completions be obtained by applying some substitution ς to the principal completion? The answer is: yes, but only if we limit what we mean by "all completions". In the completion obtained by substitution all coercions will be optimal coercions. So we cannot obtain completions with non optimal coercions in them, but since we are really not interested in these completions this is not a problem and we might as well limit the completions that we want to consider to the ones with only optimal coercions. Any of these completions is completely determined by the types of all its subexpressions and in the derivation of a principal completion all types of subexpressions have fresh box variables on all type constructors (the first type in all type signatures). Therefore, we may choose the representation type of each subexpression completely independently of the representation type of all other subexpressions for a given expression. We therefore have the following proposition:

Proposition 5.2 *Let e be a F_2-expression with principal completion $\tilde{e}$ and let $\overline{e}$ be a completion of e in which all coercions are canonical. Then there exists a substitution ς such that $\varsigma(\tilde{e}) \equiv \overline{e}$. Conversely, for every substitution ς such that ς is defined for all box variables in $\tilde{e}$ then $\varsigma(\tilde{e})$ is a completion of e.*

5.2.2 The representation graph

The purpose of the algorithm is to find *optimal completions*, which minimize the changes in representation that can occur at run-time. If we were to state this in relation to principal completions, it means that in all subexpressions of the principal completion of the form $<\varrho \rightsquigarrow$

$\varrho' > \tilde{e}$ the two types ϱ and ϱ' should be as close to each other as possible. This means that in all signatures $\varrho \leadsto \varrho'$ in the principal completion we should find all pairs of box variables occurring at the same position in the two types (on the same type constructor) and collect these into a set of ordered pairs $\mathcal{S}$. We should then find a substitution for the box variables such that the number of pairs that are mapped to different box values is minimal.

We will use a directed representation graph $\mathcal{G}$ instead of the set $\mathcal{S}$, where there is a node in $\mathcal{G}$ for every box variable in $\mathcal{S}$ and there is an edge in $\mathcal{G}$ between two nodes corresponding to box variables b and b', if the pair (b,b') is in the set $\mathcal{S}$. A node $\mathcal{G}$ is a pair of a box variable and a value in $BoxVal_\perp$ and there is at most one node in $\mathcal{G}$ for each box variable. This means that the set of node $\mathcal{N}$ in the graph corresponds to a partial function from box variables to box values. For each pair (μ,b) in $\mathcal{S}$, where μ is a box value, we generate a new fresh node n in $\mathcal{G}$, add an edge from n to the node corresponding to b, set the value of n to μ, and collect all these new nodes into a set $\mathcal{A}$, called the *sources* of the graph. Similarly, for each pair (b,μ) in $\mathcal{S}$, where μ is a box value, we generate a new fresh node n in $\mathcal{G}$, add an edge from the node corresponding to b to n to $\mathcal{G}$, set the value of n to μ, and collect all these new nodes into a set $\mathcal{Z}$, called the *sinks* of the graph. We will say that ς is a *box value assignment* to a representation graph, if ς assigns a box value to every node in the graph and it extends $\mathcal{N}$ regarded as a function, that is, if for every node $(b,\beta) \in \mathcal{N}$ we have $\varsigma(b) \geq \beta$. By setting the value (the second component) of all the remaining nodes in a graph with undefined values to box values we can use graphs to represent box value assignments to the set of box variables in the corresponding principal completion. In the following we will not distinguish between nodes and their corresponding box variables when it is clear from the context what we are referring to.

Definition 5.3 *(Representation graph, sources and sinks)*

Let $\tilde{e}$ be a principal completion w.r.t. a type assignment $\tilde{\Gamma}$ and a representation type ϱ. The *representation graph* $\mathcal{G}$ corresponding to $\tilde{e}$ is a four tuple $(\mathcal{N},\mathcal{E},\mathcal{A},\mathcal{Z})$ where

1. $\mathcal{N}$ is the set of nodes of the graph, where a node is a pair of a box variable and a value in $BoxVal_\perp$ and there is only one node in $\mathcal{N}$ for each box variable. We will use the notation $b_\perp$ to denote $(b,\perp)$.

2. $\mathcal{E}$ is the set of (directed) edges of the graph, represented by pairs of nodes.

3. $\mathcal{A} \subseteq \mathcal{N}$ is a set of nodes called the *sources* of the graph.

4. $\mathcal{Z} \subseteq \mathcal{N}$ is a set of nodes called the *sinks* of the graph.

and[2]

$$(\mathcal{N},\mathcal{E},\mathcal{A},\mathcal{Z}) = \bigcup_{<\rho \leadsto \rho' > \tilde{e}' \in \Sigma(\tilde{e})} c(\rho,\rho')$$

where

[2]Here the operator $\cup$ is extended in the natural way to work on tuples of sets, e.g. $(S_1,S_2) \cup (S_3,S_4) = (S_1 \cup S_3, S_2 \cup S_4)$, etc.

$$
\begin{array}{lll}
c(\rho_1 \rightarrow^b \rho_2, {\rho_1}' \rightarrow^{b'} {\rho_2}') & = & (\{b_\perp, {b_\perp}'\}, \{(b_\perp, {b_\perp}')\}, \emptyset, \emptyset) \cup c({\rho_1}', \rho_1) \cup c(\rho_2, {\rho_2}') \\
c(\rho_1 \rightarrow^b \rho_2, {\rho_1}' \rightarrow^{\beta} {\rho_2}') & = & (\{b_\perp, n\}, \{(b_\perp, n)\}, \emptyset, \{n\}) \cup c({\rho_1}', \rho_1) \cup c(\rho_2, {\rho_2}') \\
 & & \text{where } n \text{ is a fresh node with value } \beta \\
c(\rho_1 \rightarrow^{\beta} \rho_2, {\rho_1}' \rightarrow^{b} {\rho_2}') & = & (\{b_\perp, n\}, \{(n, b_\perp)\}, \{n\}, \emptyset) \cup c({\rho_1}', \rho_1) \cup c(\rho_2, {\rho_2}') \\
 & & \text{where } n \text{ is a fresh node with value } \beta \\
c(\forall^b \alpha.\rho, \forall^{b'} \alpha.\rho') & = & (\{b_\perp, {b_\perp}'\}, \{(b_\perp, {b_\perp}')\}, \emptyset, \emptyset) \cup c(\rho, \rho') \\
c(\forall^b \alpha.\rho, \forall^{\beta} \alpha.\rho') & = & (\{b_\perp, n\}, \{(b_\perp, n)\}, \emptyset, \{n\}) \cup c(\rho, \rho') \\
 & & \text{where } n \text{ is a fresh node with value } \beta \\
c(\forall^{\beta} \alpha.\rho, \forall^{b} \alpha.\rho') & = & (\{b_\perp, n\}, \{(n, b_\perp)\}, \{n\}, \emptyset) \cup c(\rho, \rho') \\
 & & \text{where } n \text{ is a fresh node with value } \beta \\
c(\alpha, \alpha) & = & (\emptyset, \emptyset, \emptyset, \emptyset)
\end{array}
$$

Besides source and sink nodes "internally" in the principal completion the sets $\mathcal{A}$ and $\mathcal{Z}$ also contain source and sink nodes corresponding to the input and output of the principal completion. When we talk about a given principal completion it is always assumed that this is with respect to a given type assignment $\tilde{\Gamma}$ and a given type $\underset{\sim}{\rho}$. We may then regard all the types in $\tilde{\Gamma}$ as input types and $\underset{\sim}{\rho}$ as the output type of the principal completion and all the nodes corresponding to the box variables of these types (input and output types) are then source or sink nodes. One of these input/output nodes is a source node if there is an outgoing edge from the node and a sink node if there is an incoming edge to the node in the representation graph. In an actual implementation we have to assign values to the input/output nodes by annotating $\underset{\sim}{\rho}$ and the types in $\tilde{\Gamma}$ with box values. Actually, there is no limitation on how to do this, but in our implementation we have chosen to assume that the "world" is unboxed (e.g. to output values one need these to be unboxed). That is, all values that are part of the input or the output of a F_2-expression must be unboxed. This assumption is fair as long as we are only considering representations in single entities like a single F_2-expression. If we were to extend our framework to work under some form of separate compilation we should probably use another assumption and modify the framework slightly.

In the following we will often show pictures of representation graphs or parts of these. The following lemma will help motivate the way we present graphs graphically:

Lemma 5.4 *Given a box value assignment ς to a principal completion $\tilde{e}$ then to every edge (b_1,b_2) in the corresponding representation graph where $\varsigma(b_1) \neq \varsigma(b_2)$, there corresponds exactly one proper primitive coercion* $\mathbf{c}$ *in $\varsigma(\tilde{e})$. If*

- *$\varsigma(b_1)=U$ and $\varsigma(b_2)=B$ then* $\mathbf{c} \equiv$ `box`
- *$\varsigma(b_1)=B$ and $\varsigma(b_2)=U$ then* $\mathbf{c} \equiv$ `unbox`

Proof:

We will prove the lemma when $\varsigma(b_1)=U$ and $\varsigma(b_2)=B$. The proof for $\varsigma(b_1)=B$ and $\varsigma(b_2)=U$ is similar. Assume that $(b_1,b_2) \in c(\rho,\rho').2$ where $<\rho \rightsquigarrow \rho'> \tilde{e}'$ is a subexpression of $\tilde{e}$. We will prove the statement by induction on the structure of the common erasure of ρ and ρ'.

Case: $|\rho| = \alpha$. Cannot be the case.

Case: $|\rho| = \tau \to \tau'$. Let $\rho \equiv \rho_1 \to^{b} \rho_2$ and $\rho' \equiv {\rho_1}' \to^{b'} {\rho_2}'$, there are now three different possibilities, either (1) $b = b_1$ and $b' = b_2$ in which case it is easy to see that the canonical coercion $\mathbf{c}_{\rho,\rho'}{:}\rho \rightsquigarrow \rho'$ (Theorem 3.10) must have the form $\mathbf{c}$;`box` so the statement holds for this case, (2 and 3) $(b_1,b_2) \in \mathrm{c}({\rho_1}',\rho_1).2$ or $(b_1,b_2) \in \mathrm{c}(\rho_2,{\rho_2}').2$ in which cases the statement follow by induction and from the fact that the canonical coercion $\mathbf{c}_{\rho,\rho'}{:}\rho \rightsquigarrow \rho'$ is equal to either $\mathbf{c}_{{\rho_1}',\rho_1} \to \mathbf{c}_{\rho_2,{\rho_2}'}$;`box` or $\mathbf{c}_{{\rho_1}',\rho_1} \to \mathbf{c}_{\rho_2,{\rho_2}'}$, where $\mathbf{c}_{{\rho_1}',\rho_1}$ is the canonical coercion with signature ${\rho_1}' \rightsquigarrow \rho_1$ and $\mathbf{c}_{\rho_2,{\rho_2}'}$ the canonical coercion with signature $\rho_2 \rightsquigarrow {\rho_2}'$.

Case: $|\rho| = \forall\alpha.\tau$. Proven similar to the case for function types.

■

We may therefore show nodes and edges of a representation graph graphically as follow:

$$\begin{tikzcd} b_1 : \beta \arrow[r, "\mathrm{c}"] & b_2 : \beta' \end{tikzcd}$$

where β and β' are box values of the two nodes respectively and $\mathbf{c}$ is the corresponding coercion as explained by Lemma 5.4. If the coercion is an identity coercion we will leave it out. An example could be:

$$\begin{tikzcd} b_1 : U \arrow[r, "\texttt{box}"] & b_2 : B \end{tikzcd}$$

where we annotate the nodes b_1 and b_2 with their box values in the assignment and we annotate the edges with the corresponding primitive coercion.

5.2.3 Finding optimal completions

There are of course many ways that one might attack the problem of finding the best box value assignment to the set of box variables generated from a principal completion just as there are many forms of assignments that may rightly claim to be optimal, since as described in Chapter 4 the resulting optimal completion that we get from the reduction systems is a representative for an equivalence class of $\phi\psi$-normal forms. So any algorithm that gives as result another representative for the same equivalence class as that given by one of the reduction systems, is in principal just as good (but not in practice).

Poulsen [Pou93] calls the edges in his graph "constraints" and he uses the following criteria for finding the best assignment: among the substitutions that solve the most constraints by equality choose the one that assigns U to most of the box variables in the constraint set.

We shall in this section describe two ways that will lead to assignments corresponding to optimal ψ- respectively ϕ-free completions. The two ways are in a sense dual in that the one is obtainable from the other by interchanging U and B, and we shall only describe the one in detail.

5.2.4 Finding optimal ψ-free completions

Before presenting the algorithm in Subsection 5.2.7 we will give a specification of its input and output, and prove that the output will in fact be an optimal ψ-free completion. The specification is:

1. Let the input to the algorithm be the representation graph $\mathcal{G}$ generated from the principal completion $\tilde{e}$.

2. The output from the algorithm is $\varsigma(\tilde{e})$ where $\varsigma(b) = B$, if a node b lies on the path between a source with box value B and a sink with box value B, otherwise $\varsigma(b) = U$.

We will first show that the output of the algorithm is a $\psi\phi$-normal form.

Given a principal completion $\tilde{e}$ and a box value assignment ς we refer to $\varsigma(\tilde{e})$ as the corresponding completion. Given a completion e and a principal completion $\tilde{e}$ we refer to the assignment ς, where $e{=}\varsigma(\tilde{e})$, as the corresponding assignment. We say that we have changed an assignment ς in a *single-step* to ς', if the value of ς and ς' only differ in one box variable and we say that a single-step change to an assignment is *legal* if the two corresponding completions are E-equivalent.

Lemma 5.5 *Let e be an F_2-expression with principal completion $\tilde{e}$, ς' an assignment of box values to the box variables in $\tilde{e}$ and e' the corresponding completion. Any use of an E-equality step $e'{=}_E e''$ in a derivation can be split up into a sequence of E-equivalences*

$$e' = e_1 = ... = e_n = e''$$

such that corresponding sequence of assignments

$$\varsigma', \varsigma_1, ..., \varsigma_n, \varsigma''$$

consists of single-step changes.

Proof:

Consider a coercion $\mathbf{c}$ in a completion with only optimal coercions. By Lemma 3.20 we know that we can factorize $\mathbf{c}$ into prime coercions: $\vdash \mathbf{c} = \mathbf{c}_1 ; \mathbf{c}_2 ; ... ; \mathbf{c}_n$.

If we look at a given E-equality step $e'{=}_R e''$, we may split all the coercions in e' that are moved by R into compositions of prime coercions. Assume that there are n of these. We may then use Equation 11 of Figure 11 to make sure that applications of these coercions contains only prime coercions. We can then by use of Equation 10 of Figure 11 and n usages of R in which only one prime coercion is moved show that e' is E-equal to e''.
Let us, as an example, look at Equation 13 of Figure 12:

$$(\langle \mathbf{c} \to \mathbf{d} \rangle e)\ e' = \langle \mathbf{d} \rangle (e\ (\langle \mathbf{c} \rangle e'))$$

we may show that:

$$\begin{gathered}
(\langle \mathbf{c} \to \mathbf{d} \rangle e)\ e' = \\
(\langle \iota \to \mathbf{d}_l \rangle ... \langle \iota \to \mathbf{d}_1 \rangle \langle \mathbf{c}_1 \to \iota \rangle ... \langle \mathbf{c}_k \to \iota \rangle e)\ e' = \\
\langle \mathbf{d}_l \rangle ((\langle \iota \to \mathbf{d}_{l-1} \rangle ... \langle \iota \to \mathbf{d}_1 \rangle \langle \mathbf{c}_1 \to \iota \rangle ... \langle \mathbf{c}_k \to \iota \rangle e)\ e') = \\
\langle \mathbf{d}_l \rangle ... \langle \mathbf{d}_1 \rangle ((\langle \mathbf{c}_1 \to \iota \rangle ... \langle \mathbf{c}_k \to \iota \rangle e)\ e') = \\
\langle \mathbf{d} \rangle ((\langle \mathbf{c}_n \to \iota \rangle ... \langle \mathbf{c}_1 \to \iota \rangle e)\ e') = \\
\langle \mathbf{d} \rangle ((\langle \mathbf{c}_2 \to \iota \rangle ... \langle \mathbf{c}_k \to \iota \rangle e)\ (\langle \mathbf{c}_1 \rangle e')) = \\
\langle \mathbf{d} \rangle (e\ (\langle \mathbf{c}_k \rangle ... \langle \mathbf{c}_1 \rangle e')) = \\
\langle \mathbf{d} \rangle (e\ (\langle \mathbf{c} \rangle e'))
\end{gathered}$$

Moving a coercion with only one proper primitive coercion corresponds to changing the values of just one box variable in the corresponding box value assignment. ■

Theorem 5.6 *The completion found by the algorithm is a $\psi\phi$-normal form.*

Proof:
If the output from the algorithm is not in normal form then the corresponding completion must be E-equivalent to a completion which contains a ψ or ϕ-redex. This means that we should be able to change the corresponding assignment in single-step changes (only changing the assignment in one variable at a time) to an assignment corresponding to that completion. Actually, no such assignment exist, since we use canonical completions, but instead there must exist an assignment where in the graph there are nodes and edges like:

$$\begin{tikzcd} b_1:B \arrow[r, "\texttt{unbox}"] & b_2:U \arrow[r, "\texttt{box}"] & b_3:B \end{tikzcd}$$

or

$$\begin{tikzcd} b_1:U \arrow[r, "\texttt{box}"] & b_2:B \arrow[r, "\texttt{unbox}"] & b_3:U \end{tikzcd}$$

where changing the box value of b_2 will correspond to performing one (or several) ψ/ϕ-reduction step(s). What we have to show is that there is no way that we can change the output from the algorithm in legal single-step changes to an assignment that contains one of the two configurations shown in the diagrams above. If this is the case then there is no completion with a ψ or a ϕ-redex that is E-equivalent to the completion corresponding to the assignment.

Consider a path in the graph from a source b_1 to a sink b_n both with box value B. We will, without loss of generality, assume that the subgraph that we are considering has the form:

$$\begin{tikzcd} b:U \arrow[dr, "\texttt{box}"] & & & & b':U \\ b_1:B \arrow[r] & b_2:B \arrow[r] & b_3:B \arrow[r, dotted, no head] & b_{n-1}:B \arrow[ur, "\texttt{unbox}"] \arrow[r] & b_n:B \end{tikzcd}$$

There may be more edges than we have shown and the nodes b and b' are not necessarily sources or sinks. We will first look at the part of the graph that lies on the path between b_1 and b_n. The only situations where we may have an edge with different box values at its ends are the two shown. First we will look only at the part of the graph shown in the picture. It can be seen that is not possible the change the box values of any of the nodes b_2 to b_n by a legal single-step change. Imagine that we change the box values of b_2 to U. We would then get the graph:

$$\begin{tikzcd} b:U \arrow[dr] & & & & b':U \\ b_1:B \arrow[r, "\texttt{unbox}"] & b_2:U \arrow[r, "\texttt{box}"] & b_3:B \arrow[r, dotted, no head] & b_{n-1}:B \arrow[ur, "\texttt{unbox}"] \arrow[r] & b_n:B \end{tikzcd}$$

This will introduce a new coercion `unbox` somewhere in the corresponding completion, but there is no E-rule that can introduce a primitive coercion not already present in a completion (E-rules only move coercions around).

We will then look at the subgraph not including the path. Let us look at the node b. All the nodes in the part of the graph that come before b must have box value U. We could try to "push" the `box`-coercion backwards by setting the value of b to B. This might be possible as long as there are no outgoing edges to a node with value U. Assume that this is the case for b. Then this would look like:

$$\begin{tikzcd} {} \arrow[r] & b:U \arrow[rr] \arrow[dr, "\mathtt{box}"] & & b'':U \\ & {} \arrow[r] & b_2:B \arrow[r] & {} \end{tikzcd}$$

If we set the value of b to B we will introduce an `unbox`-coercion on the edge from b to b'' and as we argued above this cannot correspond to a legal single-step change. If the values of b'' were B or there were no outgoing edge from b except to b_2 then there is no problem in setting the values of b to B because no ψ/ϕ-redex is introduced, and if we try to "push" the `box`-coercion further backwards from b we may use the argument as we have just used to inductively show that we cannot introduce any ψ/ϕ-redex by legal single step changes. A similar argument can be used to show that this is also the case for the node b', and this means the original graph must correspond to a $\psi\phi$-normal form. ■

Notice that in the proof we only use the fact that some possible changes to the assignment cannot correspond to E-equalities between completions, but not that the possible changes do correspond to a E-equalities between completions. This seems to be much harder to show, but fortunately this is not needed.

We will give an informal proof of the following:

Theorem 5.7 *(Correctness of the algorithm) The completion found by the algorithm is an optimal ψ-free completion.*

Proof:

We will show that the completion $\overline{e}$ corresponding to the assignment produced by the algorithm is a normal form under ψ-reduction modulo ϕ. Since the completion is also a $\phi\psi$-normal form, it must be an optimal ψ-free completion. Assume that we introduced a `box;unbox` pair somewhere in the completion $\overline{e}$. In the graph this would correspond to changing a part of the graph of the form:

$$\begin{tikzcd} b_1:U \arrow[r] & b_2:U \arrow[r] & b_3:U \end{tikzcd}$$

into

$$\begin{tikzcd} b_1:U \arrow[r, "\mathtt{box}"] & b_2:B \arrow[r, "\mathtt{unbox}"] & b_3:U \end{tikzcd}$$

since one can only use ϕ-equality at points where the type is unboxed. One can then try to move the box coercion backwards in the graph and the unbox coercion forward in the graph (using E-equality) in the hope that they both get eliminated by ψ-reduction steps. Moving a coercion might duplicate it or even eliminate it. We will assume for now that the coercions are not eliminated by pure E-equality and treat this special case later. There cannot be both boxed sources before b_2 in the graph and boxed sinks after b_2 in the graph, because then b_2 would have box value B. Therefore, at least all of the graph after b_2 or before b_2 must be unboxed. This means that in moving the coercion we can only hope to eliminate one of them. Assume, without loss of generality, that we succeed in eliminating the `box` coercion. This will happen if it meets an `unbox` coercion (or several, if it is duplicated), the original `unbox` coercion cannot be eliminated and we may move it back the same way that we moved it forward and then further back the same way as we moved the `box` coercion to the point where the `box` coercion was eliminated by a `unbox` coercion. This shows that we cannot by using ϕ-equality obtain a new completion that is not E equivalent to the original completion. Similarly, we may show that using ϕ-equality several times before moving the coercions will achieve nothing either, since if a coercion, say `box`, created by one use of ϕ-equality gets eliminated by an `unbox` coercion from another coercion pair `box;unbox` created by another use of ϕ-equality, it is, in a sense, just replaced by the `box` coercion from this pair. Finally, if the coercions can be eliminated by pure E-equality nothing is gained either, since if we can eliminate a coercion by E-equality we can also introduce it again by E-equality. This shows that the completion produce by the algorithm is a normal form under ψ-reduction modulo ϕ and therefore also ψ-free. ∎

5.2.5 Finding optimal ϕ-free completions

The specification of the algorithm that finds optimal ϕ-free completions is completely dual to the one for the algorithm that finds optimal ψ-free completions. All one does is to replace B by U, U by B in the specification and one gets the specification for the algorithm that finds ϕ-free optimal completions.

Similarly we have the theorem:

Theorem 5.8 *(Correctness of the algorithm) The completion found by the algorithm is an optimal ψ-free completions.*

5.2.6 An example

We have chosen a small example to show a principal completion, its corresponding representation graph and how the different assignments can be found. The example is the following F$_2$-expression:

```
id{int} (id{int} 5)
```

where `id` is the identity function which we assume is defined as a primitive. The principal completion is then:

$$\begin{aligned}
&\langle \mathtt{int}^{b_8} \leadsto \mathtt{int}^{b_{18}}\rangle(((\langle \mathtt{int}^{B(b_4)} \rightarrow^{b_3} \mathtt{int}^{B(b_5)} \leadsto \mathtt{int}^{b_7} \rightarrow^{U(b_6)} \mathtt{int}^{b_8}\rangle \\
&\qquad (((\langle \forall^{b_0}\alpha.\alpha \rightarrow^{b_1} \alpha \leadsto \forall^{U(b_2)}\alpha.\alpha \rightarrow^{b_3} \alpha\rangle \mathtt{id})\{\mathtt{int}^B\})) \\
&\qquad ((\langle \mathtt{int}^{b_{15}} \leadsto \mathtt{int}^{b_7}\rangle(((\langle \mathtt{int}^{B(b_{11})} \rightarrow^{b_1 0} \mathtt{int}^{B(b_{12})} \leadsto \mathtt{int}^{b_{14}} \rightarrow^{U(b_{13})} \mathtt{int}^{b_{15}}\rangle \\
&\qquad\qquad (((\langle \forall^{b_0}\alpha.\alpha \rightarrow^{b_1} \alpha \leadsto \forall^{U(b_9)}\alpha.\alpha \rightarrow^{b_1 0} \alpha\rangle \mathtt{id})\{\mathtt{int}^B\})) \\
&\qquad\qquad \langle \mathtt{int}^{U(b_{16})} \leadsto \mathtt{int}^{b_{14}}\rangle 5))
\end{aligned}$$

where $\tilde{\Gamma}$ is $\{\mathtt{id}{:}\forall^{U(b_1)}\alpha.\alpha \rightarrow^{U(b_0)} \alpha\}$ and the type of the completion $\tilde{\tau}$ is $\mathtt{int}^{U(b_{18})}$. The box variables in parenthesis after box values represents the nodes generated for the corresponding sources and sinks. This gives the following representation graph:

$$\begin{tikzcd}
b_{16}:U \arrow[r] & b_{14} \arrow[r] & b_{11}:B & & \\
b_{12}:B \arrow[r] & b_{15} \arrow[r] & b_{17} \arrow[r] & b_7 \arrow[r] & b_4:B \\
b_5:B \arrow[r] & b_8 \arrow[r] & b_{18}:U & & \\
b_1:U \arrow[r] \arrow[dr] & b_1 0 \arrow[r] & b_{13}:U & b_0:U \arrow[r] \arrow[dr] & b_9:U \\
 & b_3 \arrow[r] & b_6:U & & b_2:U
\end{tikzcd}$$

Any assingment must have:

$$\varsigma(b_0) = \varsigma(b_1) = \varsigma(b_2) = \varsigma(b_6) = \varsigma(b_9) = \varsigma(b_{13}) = \varsigma(b_{16}) = \varsigma(b_{18}) = U$$

$$\varsigma(b_4) = \varsigma(b_5) = \varsigma(b_{11}) = \varsigma(b_{12}) = B$$

Any optimal assignment ς to this set must have $\varsigma(b_3){=}U$ and $\varsigma(b_1 0){=}U$ as well as $\varsigma(b_7){=}B$, $\varsigma(b_{15}){=}B$ and $\varsigma(b_{17}){=}B$. This leaves us with four possible assignments according to the four possible assignments of box values to b_8 and b_{14}. If we solve the set by the algorithm for finding optimal ψ-free completions we get:

$$\varsigma(b_3) = U,\ \varsigma(b_1 0) = U,\ \varsigma(b_7) = B,\ \varsigma(b_{15}) = B,\ \varsigma(b_{17}) = B,\ \varsigma(b_8) = U,\ \varsigma(b_{14}) = U$$

which corresponds to the completion:

$$(\langle \iota_{\texttt{int}} \to \texttt{unbox}_{\texttt{int}} \rangle(\texttt{id}\{\texttt{int}\}))\ ((\langle \texttt{box}_{\texttt{int}} \to \iota_{\texttt{int}} \rangle(\texttt{id}\{\texttt{int}\}))\ \texttt{5})$$

If we solve the set by the algorithm for finding optimal ϕ-free completions we get:

$$\varsigma(b_3) = U,\ \varsigma(b_1 0) = U,\ \varsigma(b_7) = B,\ \varsigma(b_{15}) = B,\ \varsigma(b_{17}) = B,\ \varsigma(b_8) = B,\ \varsigma(b_{14}) = B$$

which corresponds to the completion:

$$\langle \texttt{unbox}_{\texttt{int}} \rangle(\texttt{id}\{\texttt{int}\}\ (\texttt{id}\{\texttt{int}\}\ \langle \texttt{box}_{\texttt{int}} \rangle \texttt{5}))$$

The example does unfortunately not show the more intriguing problems involved in finding an assignment, but we hope that it at least give some insight into the mechanisms involved.

5.2.7 Description of a simple algorithm for finding node assignments

We will describe an algorithm for finding assignments corresponding to optimal ψ-free completions. The job of the algorithm is to find those node that lie on a path between a source node and a sink node with box value B. This can be done in two phases. In the first phase one starts at all source nodes with box value B and visit all nodes that can be reach from these source nodes following all edges in the forward direction giving these nodes a mark indicating that they lie on a path coming from a source node with box value B. In the second phase one one starts at all sink nodes with box value B and visit all nodes that can be reached from these sink nodes following all edges in the backward direction giving these nodes a mark indicating that they lie on a path leading to a sink node with box value B. After this a node is assigned box value B if it has been marked with both kinds of marks and U otherwise.

An algorithm for finding assignments corresponding to optimal ϕ-free completions works analogously to the one for optimal ψ-free completions. One simply replace B with U and replace U with B in the above description and then the algorithm finds optimal ϕ-free completions. Finding the final assignment can be done during the second phase of the algorithm.

Marking the graph can be done by a simple reachability algorithm. Since the graph may contain cycles one needs some form of visited mark that indicates that a node has already been visited to make the algorithm terminate, but since the algorithm already marks all the node it visit with a mark, this can be used.

5.2.8 Complexity analysis

The time complexity of the algorithm depends on two things: the time complexity of the graph generation and the time complexity of the reachability algorithm. The time complexity of the graph generation is linear in the size of the graph. We will therefore first work out the size of the graph.

Size of the representation graph

We will work out the size of the representation graph in terms of the size n of the program being analyzed and the maximum size m of the type of any subexpression in the program. The number of nodes in the representation graph must be of the order $\Theta(nm)$ since we generate a node for each type constructor in the type of each subexpression in the program. In the worst case the number of edges in a graph can be quadratic in the number of nodes in a graph, but this will not be the case here. The number of outgoing edges from a node is of the order $\Theta(k)$, where k is the number described above in Subsection 5.1.4. So the number of edges in the representation graph can in the worst case be of the order $\Theta(nkm)$.

Complexity of reachability algorithm

The algorithm presented is linear in the size of the graph (in fact the number of edges in the graph). The generation of the graph is linear in the size of the generated graph. The two phases of the algorithm have the same complexity. In each phase a given node in the graph will be visited at most as many times as there are incoming edges to the node. The time complexity of the algorithm is therefore $\Theta(n_{edge})$ where n_{edge} is the number of edges in the graph, since all other operations performed by the algorithm can be performed in constant time.

Complexity

Since the number of edges in the representation graph in the worst case can be of the order $\Theta(nkm)$ the worse case time complexity of the algorithm is $\Theta(nkm)$.

If we again (like in Subsection 5.1.4) consider what happens when we run the algorithm on programs that corresponds to ML-typable programs, then the number of edges in the graph will in general be much smaller. The maximal out- or in-degree of nodes in the graph will depend on the maximal number of non-defining occurrences of a given variable in the program and one will therefore expect the number of edges in the graph to be of the order $\Theta(nm)$. So one should expect that the actual time complexity would be more like $\Theta(nm)$ or $\Theta(n)$, if we regard m as a small constant factor independent of the size of programs.

The space complexity of the graph based algorithm is given by the size of

the graph, so it must be $\Theta(nkm)$ in the worst case. But the average case space complexity is more likely to be $\Theta(nm)$ or $\Theta(n)$ based on the same assumptions as were used for the time complexity.

5.3 $\overline{\mathcal{C}}_{p*}$-completions

In this section we will discuss a subclass of completions called $\overline{\mathcal{C}}_{p*}$-completions. These completions have some very nice properties and it turns out that by a slight modification we can make our algorithm from Section 5.2.7 produce $\overline{\mathcal{C}}_{p*}$-completions. It is therefore interesting to know something more concrete about these completions, like for instance, does there always exist such completions, does there exist optimal $\overline{\mathcal{C}}_{p*}$-completions, etc.

5.3.1 A classification of completions

Henglein [Hen92] has given a classification of completions in dynamic typing. He divides completions into four classes according to where coercions may be placed within the completions and the form of these coercions. This classification can also be use to classify our completions. The four classes are:

- $\mathcal{C}_{pf}$, the class of completions with only primitive (p) coercions placed at fixed (f) positions in the completions, namely only at constructor (e.g. abstractions) and destructor points (e.g. applications).
- $\mathcal{C}_{p*}$, the class of completions with only primitive coercions, but where these can be placed anywhere $(*)$ in the completions.
- $\mathcal{C}_{*f}$, the class of completions with arbitrary $(*)$ coercions, but where these are placed at fixed positions in the completions (same places as in $\mathcal{C}_{pf}$).
- $\mathcal{C}_{**}$, the class of completions with no constraint on either the coercions or where these can be placed.

We will call a completion in class $\mathcal{C}$ where $\mathcal{C}$ is one of the classes above a $\mathcal{C}$-completion. The optimal completions considered so far is in general $\mathcal{C}_{**}$-completions, but the subclass $\mathcal{C}_{p*}$ of $\mathcal{C}_{**}$ is also interesting as we will explain now.

5.3.2 Significance of $\mathcal{C}_{p*}$-completions

We have not yet discussed how coercions can be implemented , but in general the primitive coercions `box` and `unbox` are the only coercions that can be implemented directly. The coercion `box` is implemented as a primitive operation that converts values from the unboxed representation to the boxed representation, and the coercion `unbox` is implemented as primitive operation that converts values from the boxed representation to the unboxed representation. The identity coercion does not give rise to any operation in the implementation. Induced coercions, however, introduce an extra overhead in the implementation in terms of new abstractions and applications (*stub code* in Leroy's terminology). For example, to get rid of function coercions one would use the following rule:

$$\langle \mathbf{c} \to \mathbf{d} \rangle e \quad = \quad \texttt{let } f : {\rho'}_x \to \rho \texttt{ = } e \texttt{ in } \lambda x : \rho_x . \langle \mathbf{d} \rangle (f\ (\langle \mathbf{c} \rangle x))$$

which introduces a new abstraction and a new application. Similarly, to get rid of $\forall \alpha . \mathbf{c}$ one would use the rule:

$$\langle \forall \alpha . \mathbf{c} \rangle e \quad = \quad \texttt{let } f : \forall \alpha . \rho \texttt{ = } e \texttt{ in } \Lambda \alpha . \langle \mathbf{c} \rangle (f\{\alpha\})$$

which also introduces new code. The last kind of induced coercion $[\mathbf{c}]$ is eliminated using the equality $\texttt{unbox};\mathbf{c};\texttt{box} = [\mathbf{c}]$ and rule 12 from Figure 11. So when one implements $\mathcal{C}_{**}$-completions extra (stub) code will usually have to be inserted. This is not the case with $\mathcal{C}_{p*}$-completions. Furthermore, $\mathcal{C}_{p*}$-completions will contain no coercions on recursive data structures, like $\mathbf{map}(\texttt{unbox})$, which are also undesirable as we will discuss later in Subsection 6.4.

5.3.3 Existence of $\mathcal{C}_{p*}$-completion

It is not obvious that there always exists a $\mathcal{C}_{p*}$-completion for a given F_2-expression. In fact, if we insist on a fixed representation type of a completion there exist F_2-expressions that have no completion in $\mathcal{C}_{p*}$. Take for instance the identity function, this has no $\mathcal{C}_{p*}$-completion at type $\texttt{int} \to \texttt{int}$. But we know from Section 4.6 that canonical ψ-free completions only have primitive coercions, so these must be in $\mathcal{C}_{p*}$. This means that if we insist on having fixed assumptions on the representation type of a completion of a given closed F_2-expression e, then all we can be sure of is that there exist a completion of the form $\langle \mathbf{c} \rangle \overline{e}$ where $\overline{e}$ is a $\mathcal{C}_{p*}$-completion of e. We therefore have the following:

Proposition 5.9 *(Existence of $\mathcal{C}_{p*}$-completion)*

1. *Any F_2-expression has a completion in $\mathcal{C}_{p*}$.*

2. *If e is a closed F_2-expression and ρ any valid representation type for e, then there exist a completion $\overline{e} \in \mathcal{C}_{p*}$ and a coercion $\mathbf{c}$ such that $\langle \mathbf{c} \rangle \overline{e}$ is a completion of e at ρ.*

It is even less obvious whether the classes of optimal completions will always overlap $\mathcal{C}_{p*}$. The following example shows that there exists ϕ-free optimal completions that are not E-equivalent with any term in $\mathcal{C}_{p*}$ (we will use integers and pairs in the example to make it simpler; one can easily come up with a similar pure F_2 example, but it would be somewhat larger):

$(\lambda f : \forall\alpha.(\alpha\rightarrow\alpha)\rightarrow\alpha\rightarrow\alpha\,.\,\lambda g : \texttt{int}\rightarrow\texttt{int}\,.\,(f\{\texttt{int}\}\ g\ 5\,\texttt{,}\,g\ 5))$
$\quad(\Lambda\alpha\,.\,\lambda h : \alpha\rightarrow\alpha\,.\,\lambda x : \alpha\,.\,h\ x)$
$\quad(\lambda x : \texttt{int}\,.\,x)$

The ϕ-free optimal completion of this term at representation type `int*int` is:

$(\lambda f : \forall\alpha.(\alpha\rightarrow\alpha)\rightarrow\alpha\rightarrow\alpha\,.$
$\quad\lambda g : \texttt{int}\rightarrow\texttt{int}\,.\,((\langle\texttt{unbox}\rangle(f\{[\texttt{int}]\}\ (\langle\texttt{unbox}\rightarrow\texttt{box}\rangle g)\ (\langle\texttt{box}\rangle 5))\,\texttt{,}\,g\ 5)$
$\quad(\Lambda\alpha\,.\,\lambda h : \alpha\rightarrow\alpha\,.\,\lambda x : \alpha\,.\,h\ x)$
$\quad(\lambda x : \texttt{int}\,.\,x)$

There is no way that we may get rid of the induced coercion $\texttt{unbox}\rightarrow\texttt{box}$ in this completion by using E-equality alone. It is, however, very often the case that the optimal classes of completions contains elements from $\mathcal{C}_{p*}$. Even if this is not the case one would expect that the best completion in $\mathcal{C}_{p*}$ often is just as good or better than the optimal completion (any in the optimal class), because the completions in $\mathcal{C}_{p*}$ has no extra stub code inserted. Only experiments with a real implementation may show this.

5.3.4 The class of $\overline{\mathcal{C}}_{p*}$completions

Since we cannot be sure that there always exist $\mathcal{C}_{p*}$-completions at a given type we will study the following class of completions instead:

$$\overline{\mathcal{C}}_{p*} = \{\langle\mathbf{c}\rangle e \mid e \ \in \mathcal{C}_{p*} \wedge e \text{ closed} \wedge \mathbf{c} \text{ optimal}\}$$

If an expression e is not a closed expression one may close it by adding a lambda-abstraction to e for all free variables in e. One may then apply our approach to this closed expression instead. Afterwards one may then have to move some induced coercions into e (using rule 13 from E) before removing the lambda-abstractions again. $\overline{\mathcal{C}}_{p*}$-completions may only contain stub code related to boxing and unboxing external objects. This could be boxing and unboxing of input and output values or of input and output to primitives (introduced as free variables). We could, therefore, say that $\overline{\mathcal{C}}_{p*}$-completions have no *internal* stub code.

5.3.5 Simple representation graphs

An interesting fact is, that we by a slight modification of our algorithm, can make it produce optimal $\overline{\mathcal{C}}_{p*}$-completions. As explained in Section 5.2.2, if we are given a box value assignment ς to the box variables in a given principal completion $\tilde{e}$, then to every edge in the corresponding representation graph there corresponds a primitive coercion in the $\varsigma(\tilde{e})$. It is not hard to determine which edges in a representation graph will correspond to coercions which will be part of an induced coercion in $\varsigma(\tilde{e})$ for any ς(e.g. $\mathbf{c}$ is not directly applied to some sub-expression $\tilde{e}'$ in $\tilde{e}$. If one removes all such edges from the representation graph and for each edge merge the two node into one new node by setting the outgoing edges from the new node to be the union of the outgoing edges from the two original node (except possibly the edge that we removed) and the incoming edges to the new node to be the union of the incoming edges to the two original node (again, except possibly the edge that we removed), then the remaining edges of the graph will correspond to only applications of primitive coercion. If we use the algorithm that finds optimal ψ-free completions on the reduced graph we will call the completions that we find optimal ψ-free $\overline{\mathcal{C}}_{p*}$-completions and if we use the algorithm that finds optimal ϕ-free completions on the reduced graph we will call the completions that we find optimal ϕ-free $\overline{\mathcal{C}}_{p*}$-completions.

5.3.6 Complexity analysis

Now the graph is much smaller, since it will only contain one node for each program point (subexpression) of the original F_2-expression. This means that the reachability algorithm will run in time $\Theta(n)$, where n is the size of the program as defined in Subsection 5.1.4. It is, however, not clear whether it is possible to construct the graph in a time proportional to its size, since the number of edges that have to be contracted may be as large as for $\mathcal{C}_{**}$-completions. So the complexity of the algorithm for $\overline{\mathcal{C}}_{p*}$-completions may be as bad as for $\mathcal{C}_{**}$-completions, but we conjecture that it should be possible to do better for $\mathcal{C}_{p*}$-completions.

Part II

Boxing analysis for ML-like languages

Chapter 6

Representation analysis for ML

Although F_2 is a elegant theoretical polymorphic language it is not specially well suited as an actual programming language, because it has some less attractive properties [Wel94]: both *type inference* (given an ordinary lambda expression e, is there an F_2-expression e' typable in F_2 with $|e'|=e$), and *type-checking* (given an ordinary lambda expression e, a type assignment Γ and a type τ, is there a typing in F_2 ending in $\Gamma \vdash e' : \tau$ where $|e'|=e$) is undecidable. If we are given a F_2-expression and remove all type related syntax (type abstraction, type application and type annotations) we will get an ordinary lambda expression. There exist no algorithm that can in general reconstruct a principal F_2-expression that we may instantiate to the original F_2-expression. This is, however, the case for a subset of F_2 for which the underlying expressions (where all type related syntax is erased) corresponds to pure core part of ML with no primitive types. In [HJ94] we presented a boxing analysis for a small ML like language, that was not based on boxing analysis for F_2, but was developed directly for that language.

Today many implementations of SML use an intermediate language very close to F_2 extended with certain language primitives such as recursion, case-expression, exceptions, etc. So it is our belief that integrating our boxing analysis, as presented in the thesis, into such systems should be possible. In this chapter we will look at how adding new languages features to F_2 affects our framework, especially, coherence and reduction. The features that we will look at are the most common elements of ML-like languages.

6.1 ML and explicit boxing

Boxing analysis of ML should be performed sometime after type inference in a compiler. The input language of the boxing analysis should be an explicitly typed form of ML that is constructed by the type inference module of the compiler and the core of this language should be F_2. The essential problem that has to be solved is how adding primitives like `+`, `sqrt`, `if`, `fix`, etc. to F_2 affects our framework. That is, how does this affect coherence, reduction, etc.

6.1.1 Generating coherence conditions for ML

Extending our work to handle language primitives and polymorphic constants is straightforward and elegant. We will show how one can derive very natural equations for language primitives and polymorphic constants directly from equation 17 of Figure 12. Assume first that p is a polymorphic constant of type σ, then the typing rule for p is:

$$\Gamma \vdash \mathrm{p}{:}\sigma$$

If we add assumptions for all the polymorphic constants of a language to the initial type assignment in typing derivations, then there is no difference in the way that polymorphic constants and free variables are typed and our proof of coherence still holds. If one assumes that polymorphic constants are always fully instantiated, like in ML, then all we need is Equation 17 for polymorphic constants. That is, polymorphic constants (`::`,`@`,`map` etc.), and therefore also monomorphic constants (`+`,`-`,`*`,`sqrt` etc., since these are just special cases of polymorphic constants), can be treated by Equation 17 of Figure 12.

If P is a language primitive (like `if` or `fix`) instead of a polymorphic constant, we could try also to use Equation 17 directly, but this would be too conservative. While we may assume that polymorphic constants are essentially free variables with a given polymorphic type σ, which we may type with the rule above, then language primitives P usually have their own typing rule:

$$\frac{\Gamma \vdash e_1{:}\sigma_1 \quad \ldots \quad \Gamma \vdash e_n{:}\sigma_n}{\Gamma \vdash \mathrm{P}(e_1,\ldots,e_n){:}\sigma_2}$$

Then for language primitives, like conditionals, we would be free to choose any representation type for the arguments of P, even for what corresponds to the generic part of the arguments (those parts that would be boxed had P been a polymorphic constant). So if we use equation 17 directly on P, regarding P as a polymorphic function instantiated to some type(s) and applied to n arguments, then we would not always be able to prove coherence. Assume for the sake of argumentation that we can give P a type $\forall\alpha.(\tau_1 * \ldots * \tau_n) \to \tau$ for some types τ, τ_1, up to τ_n. Then if we regard P as a free variable with that type and if P always occurs instantiated and fully applied in programs, we may introduce the following equivalence:

$$(P\{\tau'\})(e_1, \ldots, e_n) = P(e_1, \ldots, e_n)$$

Then when we have a typing of

$$\Gamma \vdash \mathrm{P}(e_1,\ldots,e_n){:}\tau[\tau'/\alpha]$$

with the rule above, then we also have an equivalent typing of

$$\Gamma \vdash (\mathrm{P}\{\tau'\})\ (e_1,\ldots,e_n){:}\tau[\tau'/\alpha]$$

without the specific typing rule for P. Since we want to allow τ' to have any representation, also unboxed, we may use equation 17 without the restriction of τ' to boxed types. In this way we obtain an equation for P:

$$\langle \widehat{\rho}(\mathbf{c}, \iota_{\rho_1}) \rangle (P\{\rho_1\}) = \langle \widehat{\rho}(\iota_{\rho_1{}'}, \mathbf{c}) \rangle (P\{\rho_1{}'\})$$

which is equivalent to

$$\langle (\mathbf{c}_1, \ldots, \mathbf{c}_n) \to \mathbf{c} \rangle (P\{\rho_1\}) = \langle (\mathbf{c}_1', \ldots, \mathbf{c}_n') \to \mathbf{c}' \rangle (P\{\rho_1{}'\})$$

for some coercions $\mathbf{c}_1$,...,$\mathbf{c}_n$, $\mathbf{c}$, $\mathbf{c}_1'$,...,$\mathbf{c}_n'$ and $\mathbf{c}'$. Form which we may deduce an equation for P of the form:

$$\langle \mathbf{c} \rangle P(\langle \mathbf{c}_1 \rangle e_1, \ldots, \langle \mathbf{c}_n \rangle e_n) = \langle \mathbf{c}' \rangle P(\langle \mathbf{c}_1' \rangle e_1, \ldots, \langle \mathbf{c}_n' \rangle e_n)$$

There is nothing in the proof of coherence (Theorem 4.8) that depends on the argument type ρ in a type application $e\{\rho\}$ having to be boxed, this is only a requirement from the typing rules, that is, on completions. This means that the proof of coherence of completions will also work in the presence of language primitives.

In this way one can develop specific equations for language primitives like conditional, fix-point operator, pairing, and primitive operations and be sure that coherence is preserved in the presence of these. We will give some examples below in Subsection 6.2.

6.1.2 Extension of completion reduction to new language features

Extending completion reduction to new language features is now almost trivial, since we know how to obtain the polarized versions of Equation 17. We simply repeat what we did for Equation 17 for the two polarized equations 17^- and 17^+. We will also give some examples of this below.

6.2 Recursive and non-recursive definitions

Let us now give an example of the method outlined above for generating coherence conditions and reduction rules (polarized equations) for language primitives.

6.2.1 Conditionals

The first example is the conditional. The type of the conditional `if _ then _ else _` is $\forall\alpha.(\texttt{bool} * \alpha * \alpha) \to \alpha$. We assume for the example the existence of a base type `bool` and a tuple type. Treating the conditional the way describe above yields

$$\langle \iota_{bool*\rho'*\rho'} \to \mathbf{c}\rangle(\texttt{if \_ then \_ else \_})\{\rho'\} = \langle(\iota_{bool}, \mathbf{c}, \mathbf{c}) \to \iota_\rho\rangle(\texttt{if \_ then \_ else \_})\{\rho\}$$

for any coercion $\mathbf{c} : \rho' \to \rho$. Applying both sides of the equation to argument (e_1, e_2, e_3) we obtain the natural equation

$$\langle\mathbf{c}\rangle(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) = \texttt{if } e_1 \texttt{ then } \langle\mathbf{c}\rangle e_2 \texttt{ else } \langle\mathbf{c}\rangle e_3$$

Doing the same for equations 17^- and 17^+ we obtain the natural polarized equations for conditionals:

$$\langle\mathbf{c}^-\rangle\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 = \texttt{if } e_1 \texttt{ then } \langle\mathbf{c}^-\rangle e_2 \texttt{ else } \langle\mathbf{c}^-\rangle e_3$$

$$\texttt{if } e_1 \texttt{ then } \langle\mathbf{c}^+\rangle e_2 \texttt{ else } \langle\mathbf{c}^+\rangle e_3 = \langle\mathbf{c}^+\rangle\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$$

Since these equations follow from Equation 17 and the other equations of E then all the results that we proved for completions reduction also hold for a language extended with conditions and the equations above.

Note that case-expressions can be handled by using the following equivalence:

$$\texttt{case } x \texttt{ of } \{C(\overline{x}) \texttt{ => } e_C\}_C$$

$$\equiv$$

$$\texttt{if } C_1?(x) \texttt{ then } C_1(\overline{x}) \texttt{ => } e_C[C_1\texttt{-}i(x)/x_i] \texttt{ else if } C_2?(x) \texttt{ then} \ldots$$

Where $C?$ is a primitive testing whether the argument is a value the kine constructed by C, e.g. `nil?` test whether its argument is an empty list and C`-`i a primitive selecting the ith argument of a value $C(\overline{v})$, e.g. `cons-1` is the same as `hd`[1]. Since we already know how to handle such primitives it is easy to see that case-expressions can be handle directly within our framework using the above equivalence.

6.2.2 Let-expressions

Let-expressions can be handled by a simple equivalence

$$\texttt{let } x:\tau \texttt{ = } e_1 \texttt{ in } e_2 \equiv (\lambda x:\tau\,.\,e_2)e_1$$

By using this equivalence and the equations of E one can derive the following equation for let-expressions:

$$\langle \mathbf{d}\rangle\texttt{let } x \texttt{ = } \langle \mathbf{c}\rangle e \texttt{ in } e' = \texttt{let } x \texttt{ = } e \texttt{ in } \langle \mathbf{d}\rangle(e'[\langle \mathbf{c}\rangle x/x^i])$$

A different and more complicated equation was used by Henglein and the author in [HJ94] since we did not have abstraction coercion, i.e. $\forall\alpha\,.\,\mathbf{c}$, in our framework.

6.2.3 Fix-expression

We derive an Equation for (`fix` x e) in the same way that we derive a rule for conditionals. The type of `fix` regarded as a primitive is $\forall\alpha.(\alpha\!\to\!\alpha)\!\to\!\alpha$ so using rule 17 we get:

$$\langle(\mathbf{c}\to\iota)\to\mathbf{c}\rangle\texttt{fix} \;=\; \langle(\iota\to\mathbf{c})\to\iota\rangle\texttt{fix}$$

if we apply `fix` to some abstraction $\lambda x\,.\,e$ we get:

$$\langle \mathbf{c}\rangle(\texttt{fix } \langle \mathbf{c}\to\iota\rangle\lambda x\,.\,e) = \texttt{fix } \langle\iota\to\mathbf{c}\rangle\lambda x\,.\,e$$

which gives us the equation:

$$\langle \mathbf{c}\rangle(\texttt{fix } x \; e[\langle \mathbf{c}\rangle x/x]) = (\texttt{fix } x \; \langle \mathbf{c}\rangle e)$$

for the other syntax of `fix`. Similarly, we may derive the following polarized equations for `fix`:

$$\begin{aligned}\langle \mathbf{c}^-\rangle(\texttt{fix } x \; e[\langle \mathbf{c}^-\rangle x/x]) &= (\texttt{fix } x \; \langle \mathbf{c}^-\rangle e)\\ (\texttt{fix } x \; \langle \mathbf{c}^+\rangle e) &= \langle \mathbf{c}^+\rangle(\texttt{fix } x \; e[\langle \mathbf{c}^+\rangle x/x])\end{aligned}$$

[1] In general we will not use these forms for common data types like list, etc. but rather the usual names, like `hd`, `tl`, etc.

6.3 Data type constructions

Adding new type constructors, like `int`, `bool`, `*`, etc. to a language will in general imply adding new coercion constructors in order to ensure coherence of coercions. The way we add new constructors is through *datatype declarations*. A datatype declaration has the form:

$$\texttt{datatype}\ \alpha_1 ... \alpha_n\ T = C_1 \tau_1 \mid ... \mid \tau_m$$

where T is the type constructor defined and C_i are the value constructors of the type (used to construct value of the type). The types define in this way may even be reflexive, i.e. the type constructor may have contravariant arguments (see Subsection 6.4.2 below). If we add a new type constructor T, such that $T(\tau_1,...,\tau_n)$ is a type constructed by this, then we will have to add a new coercion constructor $\mathbf{T}$, such that $\mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n)$ is a coercion constructed by this. If we do this we have to add a new formation rule for the new coercion constructor:

$$\frac{\vdash \mathbf{c}_i : \rho_i \rightsquigarrow {\rho_i}'}{\vdash \mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n) : T(\rho_1,...,\rho_n) \rightsquigarrow T({\rho_1}',...,{\rho_n}')}$$

The equations that we have to add to the equations for coercions in Figure 6 are the following (functorial) equations:

$$\mathbf{T}(\iota, ..., \iota) = \iota$$

$$\mathbf{T}(\mathbf{c}_1, ..., \mathbf{c}_n) ; \mathbf{T}(\mathbf{d}_1, ..., \mathbf{d}_n) = \mathbf{T}(\mathbf{c}_1 ; \mathbf{d}_1, ..., \mathbf{c}_n ; \mathbf{d}_n)$$

In terms of category theory type constructors can be seen as functors, and the equations above are simply the two conditions that must hold for a functor. Put in a different way the coercion constructor $\mathbf{T}$ is just the *polytypic function*[Jeu95] usually called `map`. That is, the analogue function of the function `map` from the datatype list on any datatype.

We need to check that it is sensible to add new data type constructors in this way. That is, we have to check that the results about coercions from Chapter 2 and Chapter 3 still holds after adding the new coercion constructor. The means checking Proposition 2.2:

Proposition 6.1 *(Proposition 2.2 for a language extended with data types) Let ρ, ρ' be arbitrary representation types. Then*

$$|\rho| = |\rho'| \Leftrightarrow (\exists \mathbf{c}) \vdash \mathbf{c} : \rho \rightsquigarrow \rho'.$$

Proof:
"only if": The proof is by structural induction on the common erasure of ρ and ρ'. All other cases are as in Proposition 2.2:

Inductive case: $|\rho| \equiv T(\tau_1,...,\tau_n)$. Like for function types there are again four cases that have to be checked according to whether ρ is a boxed type or an unboxed type and ρ' is a boxed type or an unboxed type. These are proven proven similarly, to the case of function types.

"if": This is proven by induction over the structure of the coercion $\mathbf{c}$. All other cases are as in Proposition 2.2:

Inductive case: $\mathbf{c} \equiv \mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n)$: Let $\mathbf{c}_i$ have signature $\rho_i \rightsquigarrow \rho_i'$ then $\mathbf{c}$ has signature $T(\rho_1,...,\rho_n) \rightsquigarrow T(\rho_1',...,\rho_n')$. By induction $|\rho_i|=|\rho_i'|$ and we therefore have $|\rho| = |\mathrm{T}(\rho_1, ..., \rho_n)| = T(|\rho_1|,...,|\rho_n|) = T(|\rho_1'|,...,|\rho_n'|) = |\mathrm{T}(\rho_1', ..., \rho_n')| = |\rho'|$.

■

To be able to perform coercion reduction in the presence of new induced coercion constructors one simply adds the two equation for induced coercion constructors as left to right rewrite rules (see Figure 19) to R (see Figure 8).

$$\boxed{\begin{array}{rcll} \mathbf{T}(\iota, ..., \iota) & \Rightarrow & \iota & (C11) \\ \mathbf{T}(\mathbf{c}_1, ..., \mathbf{c}_n)\,;\mathbf{T}(\mathbf{d}_1, ..., \mathbf{d}_n) & \Rightarrow & \mathbf{T}(\mathbf{c}_1\,;\mathbf{d}_1, ..., \mathbf{c}_n\,;\mathbf{d}_n) & (C12) \end{array}}$$

Figure 19: Coercion reduction

We should then check Lemma 3.9 (weak confluence) and Theorem 3.10. First the new version of Lemma 3.9:

Lemma 6.2 *(Weak confluence of coercion reduction). The coercion reduction system in Figure 8 extended with the rules in Figure 19 is weakly confluent.*

Proof:

There can be no critical pairs involving the two new rules and any of the old rules since none of the old rules contain the new coercion constructor $\mathbf{T}$. So any critical pairs have to involve the two new rules (and associativity).

Rules: $C11$ and $C12$. Term $\mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n);\mathbf{T}(\iota,...,\iota)$. Critical pair: $\mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n);\iota$ and $\mathbf{T}(\mathbf{c}_1;\iota,...,\mathbf{c}_n;\iota)$. (1) $\mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n);\iota \Rightarrow_{C1} \mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n)$. (2) $\mathbf{T}(\mathbf{c}_1;\iota,...,\mathbf{c}_n;\iota) \Rightarrow_{C1} \mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n;\iota) \Rightarrow_{C1} ... \Rightarrow_{C1} \mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n)$.

Rules: $C11$ and $C12$. Term $\mathbf{T}(\iota,...,\iota);\mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n)$. Critical pair: $\iota;\mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n)$ and $\mathbf{T}(\iota;\mathbf{c}_1,...,\iota;\mathbf{c}_n)$. (1) $\iota;\mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n) \Rightarrow_{C2} \mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n)$. (2) $\mathbf{T}(\iota;\mathbf{c}_1,...,\iota;\mathbf{c}_n) \Rightarrow_{C2} \mathbf{T}(\mathbf{c}_1,...,\iota;\mathbf{c}_n) \Rightarrow_{C2} ... \Rightarrow_{C2} \mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n)$.

Rules: $C12$ and $C12$. Term $\mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n);\mathbf{T}(\mathbf{c}_1',...,\mathbf{c}_n');\mathbf{T}(\mathbf{c}_1'',...,\mathbf{c}_n'')$. Critical pair: $\mathbf{T}(\mathbf{c}_1;\mathbf{c}_1',...,\mathbf{c}_n;\mathbf{c}_n');\mathbf{T}(\mathbf{c}_1'',...,\mathbf{c}_n'')$ and $\mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n);\mathbf{T}(\mathbf{c}_1';\mathbf{c}_1'',...,\mathbf{c}_n';\mathbf{c}_n'')$. Proof similar to the case $C4$ and $C4$ in Lemma 3.9.

■

Before we can prove coherence of coercions with data types we need to prove Theorem 3.10 for the combined coercion reduction system R' (Figure 8 and Figure 19). Parts 1 and 2 of Theorem 3.10 are unchanged except that we replace R by R'. In Part 3 of Theorem 3.10 we just need to add one more alternative to the characterization of normal forms:

- $\mathbf{c} \equiv \mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n)$, $\mathbf{c} \equiv \mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n)\,;$`box`, or $\mathbf{c} \equiv$ `unbox`$\,;\mathbf{T}(\mathbf{c}_1,...,\mathbf{c}_n)$ where $\mathbf{c}_1,...,\mathbf{c}_n$ are normal forms of which at least one is proper.

We will now give some examples of the effect of adding new type constructors.

6.3.1 Integers

For base types like `int` the two equation for induced coercions degenerates into two equations like:

$$\begin{aligned}\mathbf{int} &= \iota_{\texttt{int}}\\ \mathbf{int} &= \mathbf{int}\,;\mathbf{int}\end{aligned}$$

where the second one obviously follows from the first one. So `int` is an example of a type that we do not need a new coercion constructor for. Similar equations holds for other nullary type constructors, e.g. `real`, `string` and `unit`. The equations for expressions are trivial instantiations of Equation 11, e.g.:

$$\langle\iota_{\texttt{int}}\rangle 1 = 1$$

6.3.2 Products

For composite types like pairs we get the slightly more interesting but quite intuitive equations:

$$\begin{aligned}(\iota_{\rho},\ \iota_{\rho'}) &= \iota_{\rho*\rho'}\\ (\mathbf{c}_1\,;\mathbf{d}_1,\ \mathbf{c}_2\,;\mathbf{d}_2) &= (\mathbf{c}_1,\ \mathbf{c}_2)\,;(\mathbf{d}_1,\ \mathbf{d}_2)\end{aligned}$$

The equations for expression will be:

$$\begin{aligned}\langle\mathbf{c}\rangle(\texttt{fst}\ e) &= \texttt{fst}(\langle(\mathbf{c},\ \iota)\rangle e)\\ \langle\mathbf{c}\rangle(\texttt{snd}\ e) &= \texttt{snd}(\langle(\iota,\ \mathbf{c})\rangle e)\\ \langle(\mathbf{c},\ \mathbf{d})\rangle(e_1\texttt{,}e_2) &= (\langle\mathbf{c}\rangle e_1\texttt{,}\langle\mathbf{d}\rangle e_2)\end{aligned}$$

6.3.3 Sums

Base types like integers, reals, etc. can essentially also be considered as sumtypes. If we assume that the boolean type `bool` is the sumtype defined as:

$$\texttt{bool} = \mathbf{true} + \mathbf{false}$$

That is, the boolean type can be considered a sumtype with the two nullary type constructors `true` and `false`. Then `bool`can be treated the same way as integers. So we get the equations:

$$\begin{aligned}\mathbf{bool} &= \iota_{\texttt{bool}}\\ \mathbf{bool} &= \mathbf{bool}\,;\mathbf{bool}\end{aligned}$$

which is completely analogue to the equations for integers. But not all sum types are as simple as the booleans, so let us look at a datatype well known to many ML programmers:

$$\texttt{datatype}\ \alpha\ \texttt{option} = \texttt{Some of}\ \alpha \mid \texttt{None}$$

According to the general treatment of datatypes there should be a coercions constructor **option** for this type. The equations that holds for this should be:

$$\begin{aligned}\mathbf{option}(\iota_\rho) &= \iota_\rho\ \texttt{option}\\ \mathbf{option}(\mathbf{c});\mathbf{option}(\mathbf{d}) &= \mathbf{option}(\mathbf{c};\mathbf{d})\end{aligned}$$

It could be illustrative to look at how **option** should be implemented:

```
option(c) = λv.case v of
                 Some x => Some(c(x))
               | None => None
```

So for sum types the case-expressions naturally pops up in the implementation of the coercions constructor. This is not strange, because this is what one needs to implement "map" on sum types. This may be generalized to any non recursive sum type.

The equations for expressions are straightforward:

$$\begin{aligned}\langle\mathbf{option}(\mathbf{c})\rangle\texttt{None} &= \texttt{None}\\ \langle\mathbf{option}(\mathbf{c})\rangle(\texttt{Some}\ e) &= \texttt{Some}\ (\langle\mathbf{c}\rangle e)\end{aligned}$$

6.4 Recursive data types

Leroy [Ler90] pointed out that recursive data types may introduce an efficiency problem. If we introduce lists then the corresponding coercion constructor to the list type constructor is `map`. If we allow elements of lists to be unboxed, which is completely legal in our approach, then we might end up having coercions of the form `map(box)` in our completions. Such coercions do not seem very practical, since copying a list is not a very efficient process.

There are two reasons why Leroy wanted to exclude recursive coercion constructors from his framework. One is that it could be incorrect when recursive data structures can be physically updated (mutable data structures) and the other is that Leroy can only place coercion at instantiation points which make it impossible for him to avoid such inefficient coercions. For these reasons Leroy requires that generic components of recursive data structures be systematically boxed (wrapped in Leroy's terminology). In our approach there is no need to be so restrictive, since a language like ML has no mutable data structures. Furthermore, in our framework we are not restricted in where we may place coercions, so we may, as a last resort, place these where the recursive data structures are created and consumed to avoid recursive coercions.

However, if we choose to require that generic components of recursive data structures be systematically boxed, then there is a very simple method to do this in our approach: we just make the primitive operations working on recursive data structures polymorphic constants. In the case of list this could amount to the following assignments of types:

$$\begin{aligned}&\texttt{::} &&: \forall\alpha.\alpha * \alpha\ \texttt{list} \rightarrow \alpha\ \texttt{list}\\ &\texttt{nil} &&: \forall\alpha.\alpha\ \texttt{list}\\ &\texttt{hd} &&: \forall\alpha.\alpha\ \texttt{list} \rightarrow \alpha\\ &\texttt{tl} &&: \forall\alpha.\alpha\ \texttt{list} \rightarrow \alpha\ \texttt{list}\\ &\texttt{null} &&: \forall\alpha.\alpha\ \texttt{list} \rightarrow \texttt{bool}\end{aligned}$$

There is however one slight problem associated with this method. Since we assume that the world is unboxed, any type of a completion should be completely unboxed and this should hold for completions with a recursive data type as well. This means that in order to produce well typed completions in such cases we have to insert a coercion that unboxes the components of the recursive data type. This coercion could be placed as the outermost coercion in the completion and one may then regard the work performed by this coercion as the work that some implicit print routine needs to perform in order to present the output to the outside world, e.g. the user. A similar problem occurs when programs can take recursive data structure as input.

There are obviously cases where recursive data types with unboxed components, e.g. lists of unboxed integers, would be more efficient, e.g. in monomorphic programs, and below (in Subsection 6.4.3) we will explain how to avoid expensive induced coercions altogether.

6.4.1 Lists

As an example of a recursive data type let us consider lists. If we add a new type constructor **list** for lists we also have to add a new constructor on coercions. This constructor is for list essentially `map`, since if e evaluates to a list then $\langle\mathbf{list}(\mathbf{c})\rangle e$ will correspond to applying the coercion $\mathbf{c}$ to each of the elements of the list that e evaluates to. We have to extend Figure 6 with some new equations for this new coercion constructor as explained in Section 6.3. For list these equations are

$$\begin{aligned}\mathbf{list}(\iota_\rho) &= \iota_{\rho\ list}\\ (\mathbf{list}(\mathbf{c}))\,;(\mathbf{list}(\mathbf{d})) &= \mathbf{list}(\mathbf{c}\,;\mathbf{d})\end{aligned}$$

These equations should be well-known for the ordinary `map` function on lists and the equations for expressions involving the primitives above are:

$$\begin{aligned}\langle\mathbf{c}\rangle(\texttt{hd}\ e) &= \texttt{hd}(\langle\mathbf{list}(\mathbf{c})\rangle e)\\ \langle\mathbf{list}(\mathbf{c})\rangle(\texttt{tl}\ e) &= \texttt{hd}(\langle\mathbf{list}(\mathbf{c})\rangle e)\\ \langle\mathbf{list}(\mathbf{c})\rangle\,\texttt{[]} &= \texttt{[]}\\ \langle\mathbf{list}(\mathbf{c})\rangle(e_1\texttt{::}e_2) &= \langle\mathbf{c}\rangle e_1\texttt{::}\langle\mathbf{list}(\mathbf{c})\rangle e_2\\ \texttt{null}(\langle\mathbf{list}(\mathbf{c})\rangle e) &= \texttt{null}\ e\end{aligned}$$

6.4.2 Reflexive types

We will now show that all of our theory for datatypes also holds for reflexive types. Consider the datatype defined as follows:

$$\texttt{datatype}\ \alpha_1\ \alpha_2\ \texttt{fun}\ =\ \texttt{Func of}\ \alpha_1\ \texttt{->}\ \alpha_2$$

This type constructor is contravariant in its first argument (α_1). If we write down the formation rule for the corresponding coercion constructor it is:

$$\frac{\vdash \mathbf{c}_1:\rho_1 \leadsto \rho_1{}' \quad \vdash \mathbf{c}_2:\rho_2 \leadsto \rho_2{}'}{\vdash \mathbf{fun}(\mathbf{c}_1,\mathbf{c}_2):\texttt{fun}(\rho_1,\rho_2) \leadsto \texttt{fun}(\rho_1{}',\rho_2{}')}$$

ones first thought is that this is wrong because it should be similar the formation rule for function coercions. But in effect we have not defined **fun** and we can define this a follows:

$$\langle\mathbf{fun}(\mathbf{c},\mathbf{d})\rangle(\texttt{Func}\ e) = \texttt{Func}(\langle\mathbf{c}^{-1}\!\to\!\mathbf{d}\rangle e)$$

which is essentially the equation that we will deduce for the constructor `Func` using the method describe in Section 6.1. There is of course one small problem with this definition and that is that the inverse of a coercion is not unique, but first of all we may just choose this to be any arbitrarily chosen inverse, and second as we will explain in the next subsection, we actually do not need induced coercion like **fun**(**c**,**d**) and there is therefore no need to worry about how to implement these.

Let us round of with a slightly more involved example. Consider the datatype declaration:

$$\texttt{datatype}\ \alpha\ \texttt{dyn}\ =\ \texttt{Atom of}\ \alpha \mid \texttt{Func of}\ \alpha\ \texttt{dyn -> }\alpha\ \texttt{dyn}$$

This gives us the coercion equations:

$$\begin{aligned}\mathbf{dyn}(\iota_\rho) &= \iota_\rho\ \texttt{dyn}\\ \mathbf{dyn}(\mathbf{c};\mathbf{d}) &= \mathbf{dyn}(\mathbf{c});\mathbf{dyn}(\mathbf{d})\end{aligned}$$

and the equation that we get for `Atom` and `func` are:

$$\begin{aligned}\langle\mathbf{dyn}(\mathbf{c})\rangle(\texttt{Atom}\ e) &= \texttt{Atom}\ (\langle\mathbf{c}\rangle e)\\ \langle\mathbf{dyn}(\mathbf{c})\rangle(\texttt{Func}\ e) &= \texttt{Func}\ (\langle\mathbf{dyn}(\mathbf{c})^{-1}\to\mathbf{dyn}(\mathbf{c})\rangle e)\end{aligned}$$

This means, for instance, that we can prove the following:

$$\begin{aligned}\langle\mathbf{dyn}(\mathbf{c})\rangle(\texttt{Func}\ (\lambda x.x)) &= \texttt{Func}\ (\langle\mathbf{dyn}(\mathbf{c})^{-1}\to\mathbf{dyn}(\mathbf{c})\rangle(\lambda x.x))\\ &= \texttt{Func}\ (\lambda x.\langle\mathbf{dyn}(\mathbf{c})\rangle\langle\mathbf{dyn}(\mathbf{c})^{-1}\rangle x)\\ &= \texttt{Func}\ (\lambda x.\langle\mathbf{dyn}(\mathbf{c})^{-1};\mathbf{dyn}(\mathbf{c})\rangle x)\\ &= \texttt{Func}\ (\lambda x.\langle\iota\rangle x)\\ &= \texttt{Func}\ (\lambda x.x)\end{aligned}$$

6.4.3 $\mathcal{C}_{p*}$-treatment

In general induced coercions may be very expensive, like we explained for list above, so it would seem that there is a problem with these. But there is a very simple way to get rid of this problem. $\overline{\mathcal{C}}_{p*}$-completions cannot contain any coercion on recursive data types, since these would then not be primitive. An algorithm that finds $\overline{\mathcal{C}}_{p*}$-completions would therefore ensure that completions have no expensive recursive coercions in them. The only induced coercions that would remain in a $\overline{\mathcal{C}}_{p*}$-completion are then the ones related to input and output, and as explained above in Subsection 6.4 these are unavoidable, and in fact make explicit the operations that an actual implementation should perform on input and output.

6.5 Imperative language features

At first, one would think that introducing imperative features in a language, should break coherence, since Equation 17 follows from parametricity and imperative features should restrict parametricity. But since all we require is that all congruent completions are semantically equivalent, then we could still use the general framework in the presence of imperative features, this would just demand more of the concrete semantics that one intends to use. Since the primitive coercions are not imperative, $\overline{\mathcal{C}}_{p*}$-completions will not contain any imperative coercions, and we can then use any semantics that only assigns meaning to $\overline{\mathcal{C}}_{p*}$-completions, but is equivalent to a full semantics (for $\mathcal{C}_{**}$-completions) on $\overline{\mathcal{C}}_{p*}$-completions. In this way the semantics on

$\overline{\mathcal{C}}_{p*}$-completions may be much simpler than the full semantics, since it does not have to assign meanings to induced coercion, and especially, to imperative coercions. We will give one example of how to handle an imperative feature in ML, polymorphic references. Other examples could be exceptions.

6.5.1 Polymorphic references

In Standard ML reference types are written τ `ref` and the primitive function `ref`, with type $\forall$`'_`α`.'_`$\alpha\to$`'_`α `ref` constructs elements of this type. Furthermore, ML contains the assignment operator `:=`, that has the type $\forall$`'`α`.'`α `ref*'`$\alpha\to$`unit` and the de-refence function `!`, that has type $\forall$`'`α`.'`α `ref`$\to$`'`α. If we treat reference types and the associated primitives in a similar manner as discussed in Section 6.2 we should add a new coercion constructor `ref(c)` corresponding to the reference type constructor. We would the have rules for composition of reference coercions and the identity coercions on references similar to the rules for list types. Furthermore, we would have to add the following rules which may be deduced the same way as, say, the rules for if-expressions and fix-expressions:

$$\texttt{ref}\ (\langle \mathbf{c}\rangle e) = \langle \texttt{ref(c)}\rangle(\texttt{ref}\ e)$$

$$\langle \texttt{ref(c)}\rangle e\texttt{:=}\langle \mathbf{c}\rangle e' = e\texttt{:=}e'$$

$$\texttt{!}\langle \texttt{ref(c)}\rangle e = \langle \mathbf{c}\rangle(\texttt{!}e)$$

These rules work fine as long as we do not consider the semantics of coercions. The problem occurs when one starts to consider what meaning to assign to a reference coercion `ref(c)`. The natural way is to think of `ref(c)` as a coercion that operates on a reference by changing the representation of the object that the reference points to. This is, however, not a good interpretation of the action of `ref(c)`, since the presence of references breaks referential transparency. Consider the following completion:

$$\texttt{let}\ x\ \texttt{=}\ \texttt{ref}\ \langle\texttt{box}\rangle 3\ \texttt{in}\ (x'\texttt{:=!}x\texttt{;!}x)$$

This can we rewritten, by using the rules above and the rules of E, to:

$$\texttt{let}\ x\ \texttt{=}\ \texttt{ref 3 in}\ (x'\texttt{:=!}(\langle\texttt{ref(box)}\rangle x)\texttt{;!}(\langle\texttt{ref(box)}\rangle x))$$

Here in the second application of `ref(box)` to x the value coerced is already represented boxed, since the first application of `ref(box)` to x has changed the value of x from an unboxed value to a boxed value. A better interpretation, due to Henglein, of `ref(c)` is that $\langle$`ref(c)`$\rangle e$ evaluates to the pair of the coercion **c** and the reference computed by evaluating e. In such an interpretation also the action of `!` should be interpreted in a slightly non-standard way. That is, if e evaluates to $(\mathbf{c}, v)$ then `!`e evaluates to $[\![\mathbf{c}]\!](v)$, where $[\![\mathbf{c}]\!]$is the interpretation of **c**. This shows that it is possible to give `ref(c)` an interpretation, although this might not be an efficient one. If one restricts completions to be $\overline{\mathcal{C}}_{p*}$-completions then these will not contain any reference coercions, since $\overline{\mathcal{C}}_{p*}$-completions contains only primitive coercions, then a referenced values cannot “change” representation. If we restrict a semantics with the interpretation of `ref(c)` as discussed above to only $\overline{\mathcal{C}}_{p*}$-completions then we may disregard the fact that references are interpreted as a pair of a coercions and a standard interpretation of a reference, since the coercion will always be the identity coercion, and just represents references.

6.6 Optimizing the execution efficiency of completions

Until now all we have assumed about execution efficiency of completions is that for coercions the composition of an `unbox`- and a `box`-coercion in either direction is less efficient than a identity coercion ι. This does not take into account what happens in a realistic implementation of a polymorphic functional language. In this section we will discuss some of these aspects of boxing analysis.

The position of coercions in a completion may effect the execution efficiency of completions in many ways. A placement of a coercion in a loop may cause the coercion to be executed billions of times more than had it been placed outside the loop, so where coercions are placed in a completion may have a direct effect on execution efficiency. But also other factors than how often a coercions is performed matters for execution efficiency. If too many unboxed values are live at the same time, then this may effect register allocation in a negative way and may therefore degrade performance of completions. Also other factors have influence on choosing the right completion in a concrete implementation and in the remaining part of this section we will discusses some of these.

6.6.1 Completion equivalence is not efficiency equivalence

In a class of optimal completions the execution efficiency of the completions may differ considerably in an actual implementation. Consider for example, Equation 13 of Figure 12, this may, if used from left to right, duplicate coercions when x occurs free more than once in e, or it may throw coercions away if x does not occur free inside e. If a coercion is duplicated it may degrade execution efficiency, if the coercion ends up being performed more than once. As our argumentation shows completion equivalence and efficiency equivalence is not the same thing. But since it is, on recursion-theoretic grounds, impossible in general to find optimal completions with respect to execution efficiency, our theory does at least tell us that there are two classes of formally optimal completions and that the only way that we may change execution efficiency in these is by moving coercions around. This reassures us that, if we for other efficiency reasons, choose to move coercions, we cannot introduce new coercions (if we are careful with the use of Equations 13 and 17). So the important consideration in optimizing the execution efficiency of completions is to find out where to place the remaining coercions, or picking the best equivalence class representative.

6.6.2 Picking an equivalence class representative

Choosing the best completion with respect to execution efficiency is not a trivial job. First of all this depends on what one wants to optimize with respect to and on the kind of execution model one has in mind. But in any case boxing analysis makes sure that all directly unnecessary representation shifts are removed. It could then be up to a subsequent phase (on the basis of E-equivalence) to find the best completion by using other criteria.

On the one hand, one might expect that boxing as late as possible will decrease the pressure on memory, since boxed values take up more space. On the other hand, this might put more pressure on the use of registers, since unboxed values may take up more registers, e.g. an unboxed pair of integers may use two registers, while a boxed pair only take up one.

By the same reasoning unboxing as early as possible should decrease the pressure on memory and increase the pressure on registers, while unboxing as late as possible should have the opposite effect.

Accessing an unboxed value may be considerably faster that accessing a boxed value (specially in todays computers where accessing a register is much faster, to say the least, than accessing a value in memory). This should indicate that if one wants a completion that is best with respect to execution efficiency, then one should choose a completion that keeps values unboxed for as long as possible, as long as this does not affect register allocation too much.

This discussion should indicate that it is not a trivial job to pick the best equivalence class representative, and that this may depend on many factors. Below, in Subsection 6.6.6 we will discuss more about doing this in a concrete compiler. In the next three subsections we will discuss some more ideas on how to optimize execution efficiency.

6.6.3 Moving coercions out of loops

In an implementation of our analysis for a realistic language one should take the actual semantics of the considered language into account. For example, our analysis does not distinguish between E-equivalent completions so any of these are equally good choices for an optimal completion. This can in some situations lead to a very poor choice of representation. Let us for the rest of the discussion assume that we are considering a language with call-by-value semantics. Consider the function `fromto` used in the `sieve` program shown in Appendix D. Here written in SML syntax:

```
fun fromto f t=
    if f > t then
       []
    else
       f :: (fromto (f + 1) t)
```

This function takes two integers `f` and `t` as argument and returns the list of integers from `f` and up to `t`. If we call this function in the following way `fromto 1 (id 1000)` then we may either insert an unbox coercion unboxing the result of `(id 1000)` immediately or we may place the unbox coercion inside `fromto` where we test whether `f` is greater than `t` which controls termination of the function. In the first case the unbox coercion is performed only once, while in the second case it is performed 1000 times. The two corresponding completions are E-equivalent, but it makes quite a difference which one we choose.

6.6.4 Taking control flow information into account

The graph used by the graph based algorithm described in Section 5.2 is essentially a value-flow graph and does not take control flow information into account. Consider the following completion:

$$\langle \mathbf{c} \to \mathbf{d} \rangle \lambda x\,.\,\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$$

by equation 13, this is equivalent to

$$\lambda x\,.\,\langle \mathbf{d} \rangle (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)[\mathbf{c}/x]$$

If x occurs exactly once in both e_2 and e_3 but not in e_1 then "moving" **c** in to the applied occurrences of x by Equation 13, cannot make the execution efficiency of the completion worse (only better, since there may exist execution paths in e_2 and/or e_3 that do not contain **c**), since the new occurrences of **c** will lie on different execution paths. To recognize this one needs control flow information, so the representation graph as it is, is not adequate for this kind of optimization. One way to add this kind of information could be to annotate nodes as being *conjunctive* (respectively *disjunctive*), where for a conjunctive node the successor nodes may lie on the same execution path for some execution, and for a disjunctive node this cannot be the case. The annotation of nodes should then be based on control flow information of the corresponding completion (or underlying expression).

In some cases one could imagine that common subexpression elimination could eliminate the problem.

6.6.5 Comparison with Peterson's minimum cut based optimization

Another way to take the actual semantics into account is to use a method similar to [Pet89].

One could devise algorithms for computing optimal boxing completions, which also take account of control dependencies and carefully place coercions at points where they get executed with minimum run-time frequency. We expect that an analyses such as Peterson's could be used for this purpose. Our representation graph is essentially the same as Peterson's cost graph if we assign cost factors to all edges. These factors could be computed as the cost of the coercion corresponding to the edges (`box`, `unbox` or ι) times some appropriate execution frequency of the edges. The total cost is then the sum of the costs of all the edges. This can be found by cutting the graph into two parts: one part where the nodes have box value B and one where the nodes have box value U. All the edges connecting nodes in the same part of the graph will have cost 0 (if we assume that the cost of ι is 0) and only the cost of the edges that connects node from the two parts will contribute to the total cost of the representation. Finding the best cut can be accomplished by a well known graph algorithm, *mincut*.

A scheme for finding the execution frequency of edges may take many different things into account, such as strictness or neededness of arguments or occurrence counting similar to that used by the partial evaluator Similix [Bon90].

6.6.6 Boxing analysis in the ML-kit compiler

The ML-kit compiler[BRTT93] is a prototype SML compiler under construction at DIKU based on Talpin and Tofte's [TT94, TT93] region inference. Region inference provides the basis for a stack-allocation like implementation of typed call-by-value functional languages. The stack-like behaviour is expressed by the fact that all the values in a program, including function closures, are put into regions at runtime, that are then allocated and deallocated in a stack-like manner. Region inference infers the scope of regions and for all value constructing expressions, like `5`, `(2,3)`, $\lambda x.e'$, etc., what region to put the value in. After region inference programs are annotated by inserting two special target language constructs. The first one of these is the *letregion-expression*

$$\texttt{letregion}\ \varrho\ e\ \texttt{end}$$

which marks the scope of a region (ϱ) and the second one is the at-expression:

$$e\ \texttt{at}\ \varrho$$

which indicates into which region (ϱ) the values computed by e is going to be put at runtime. The idea is then that boxed values go into regions while unboxed values does not, i.e. may go into registers or on the stack.

An implementation of boxing analysis is currently being inserted into the ML-kit compiler. In the ML-kit compiler we have in the present version decided that the boxing analysis is to be placed after the region inference phase. One reason for this was that it did not seem that there was anything to be gained by putting before the region inference phase, and since the boxing analysis is being added to a version of the compiler that was all ready working, it would seem a waist of effort to have to modify the region inference phase essentially for notthing. Putting the boxing analysis after the region inference phase does, however, mean that one has to be careful, since adding `box`-coercions or moving `box`-coercions around in a program may change where at-expressions are placed, since `box`-coercions are value creating. Even worse, moving coercions around may cause the information inferred by the region inference to be incorrect. The region inference infers what is called *effects*. For an expression e, this corresponds intuitively to the effect on regions when the expression is executed, that is, whether a value has been *put* into the region or the region has been accessed to *get* a value from it. So moving coercions around may affect this information.

To ensure that no such problem occurs we have chosen to compute a special kind of completions. These completions are $\overline{\mathcal{C}}_{p*}$-completions where `box`-coercions can only be placed at fixed positions in a program, namely at value creating expressions (sources), but `unbox`-coercions can be placed anywhere. The implementation of course infers the best positions for the `unbox`-coercions, by placing these as early as possible and if a value never is needed boxed, then no `box`-coercion is inserted on the place where it is constructed. These kind of completions are of course not in general optimal (since this is already the case for $\overline{\mathcal{C}}_{p*}$-completions), but the conflict between boxing operations and region inference is gone. If boxing analysis finds out that no `box`-coercion needs to be inserted at some value creating point in a program, then this corresponds to not having an at-expression at that point. But since it is safe to remove regions that are never used (by an at-expression), removing an at-expression is also a safe operation.

We expect to get fairly good results with the outlined method, since it has the following nice properties:

1. it does not box more than the present implementation, since `box`-coercions are only inserted where values were already being created boxed anyway. This will mean that performance of programs are not being made worse by boxing analysis.

2. for completely monomorphic (parts of) programs no `box`-coercions and `unbox`-coercions are inserted.

3. for monomorphic parts of programs where there is no use of boxed values later in the lifetime of a value this value is unboxed for the rest of its life.

There is a very simple algorithm for finding the completions used by the ML-kit compiler; one simply by default sets all box values in the representation graph to U and then do a backwards reachability analysis from all boxed sinks, marking the reached node with box value B.

The addition of boxing analysis has made it necessary to change parts of the back-end of the compiler and this work is not completed yet, and we do therefore not have any result showing the effect of boxing analysis in the ML-kit compiler. When such results are available we hope to report on the successful effect of boxing analysis in a region based compiler.

6.7 Lazy evaluation and explicit boxing

At first sight it seem that explicit boxing combined with lazy evaluation would not lead to any optimization since in principal all values would be closures and therefore always represented uniformly. This does not change the fact that our framework is independent of the implementation of the underlying programming language, only that the benefit may be bigger in some languages than in others.

However, if combined with a strictness analysis boxing analysis is able to describe exactly the same optimizations as described by Peyton Jones and Launchbury in [PJL91], without changing the underlying structure of programs. But our framework also goes beyond this, because boxing analysis can do further optimization in the presence of polymorphism. Let us look at an example of this. In the framework of Jones and Launchbury type constructor, e.g. `Int`, are used as constructors of boxed values, that is as box coercions, e.g. `Int` is their equivalent of our coercion $\mathtt{box}_{\mathtt{int}}$. Similarly their equivalent of our unbox coercions are case expression of the form:

```
case e of Int t# -> t#
```

which for this case is equivalent to our coercion $\mathtt{unbox}_{\mathtt{int}}$. Consider the following simple program:

```
f x y z = fst(id(x+y,True))*z
```

Now if the strictness analysis has told us that `f` is strict in all its arguments, the definition of `f` above is in Jones and Launchbury's framework equivalent to:

```
f x y z = case x of
            Int x# -> case y of
                        Int y# -> case z of
                                    Int z# -> f# x# y# z#
f# x# y# z# = let x = x#
                  y = y#
                  z = z#
              in
                  fst(id(x+y,True))*z
```

where the annotation `#` on variables serve the purpose of making the presentation clearer by indicating that these will be bound to unboxed values, but are otherwise ignored. The function `f#` can by their transformations (essentially deforestation of the constructors and destructors of boxed values) be translated into:

```
f# x# y# z# = case id (Pair (case (#x +# y#) of #t -> Int #t,True)) of
                Pair p# -> case #p of
                             (f,_) -> case f of
                                        Int t# -> case (t# *# z#) of
                                                    r# -> Int r#
```

This is not optimal, since the result of the addition is boxed and then unboxed again (`f`) before multiplied with `z#`. With our framework would be able to obtain the following result:

```
f# x# y# z# = case id (Pair (case (#x +# y#) of #t -> #t,True)) of
                 Pair p# -> case #p of
                              (t#,_) -> case (t# *# z#) of
                                          r# -> Int r#
```

which is optimal, since this is equivalent to:

```
f# x# y# z# =⟨box⟩fst(⟨unbox⟩(id ⟨box⟩(#x +# y#,True)))*# z#
```

which is the result that our analysis will be able to produce.

Another utilisation of boxing analysis for lazy languages would be to be able to have both boxed and unboxed closures and thereby save space in an implementation. We will not pursue this idea further here, but leave this to further research.

6.8 A larger example

The examples used so far has deliberately been keep minimal in order to make the presentation as clear as possible. In this section we will present an example that combines and illustrates many of the ideas presented in this chapter. We will use a slightly different syntax, namely the one use in our implementation. That is, we will use `Fn a =>` e for type abstraction and `fn a =>` e for ordinary abstraction. To further simplify the presentation we will leave out types from coercions. The program that we will use contains a function `assoc` that will look up an element in an association list using a predicate `p` to test for success. The entire program looks as follows:

```
case ((Fn a =>
        Fn b =>
         fix (fn assoc:a->b list->(a->b->Bool)->b option =>
               fn x:a =>
                fn l:b list =>
                 fn p:a->b->Bool =>
                  if null l then
                     None
                  else if p x (hd l) then
                          Some (hd l)
                       else
                          assoc x (tl l) p))
       {int} {int*int}
       5 [(5,42)] (fn x:int => fn y:int*int => x = fst y))
of
   None => 0
   Some p => snd p
```

What the output of this program is is left to the reader to figure out. Now a possible completion for this program at type `int` would be:

```
case <unbox>
       ((Fn a =>
          Fn b =>
           fix (fn assoc:a->b list->(a->b->Bool)->[b option] =>
                  fn x:a =>
                   fn l:b list =>
                    fn p:a->b->Bool =>
                     if null l then
                        <box>None
                     else if p x (hd l) then
                             <box>(Some (hd l))
                          else
                             assoc x (tl l) p))
         {[int]} {[[int]*[int]]}
         (<box>5) [<box>(<box>5,<box>42)]
         (fn x:[int]=>fn y:[[int]*[int]]=><unbox>x=<unbox>fst(<unbox>y)))
  of
    None => 0
    Some p => <unbox>snd(<unbox>p)
```

We will demonstrate how we can prove this equal to the optimal completion:

```
  case (Fn a => Fn b =>
         fn assoc:a->b list->(a->b->Bool)->b option =>
          fn x:a =>
           fn l:b list =>
            fn p:a->b->Bool =>
             if null l then
                None
             else if p x (hd l) then
                     Some (hd l)
                  else
                     assoc x (tl l) p)
        {[int]} {[int*int]}
        (<box>5) [<box>(5,42)]
        (fn x:[int]=>fn y:[int*int]=><unbox>x=fst(<unbox>y))
  of
    None => 0
    Some p => snd(<unbox>p)
```

This completions does not box 5 and 42 and unbox the final result as the original completion does. Furthermore, it does not box the "return" values (`None` and `Some (hd l)`) in the body of `Assoc` and unbox these again in the case-expression which the original completion does.

We will use the following abbreviations for parts of the program to make the presentation easier:

```
Assoc = Fn a => Fn b => fix F

F = fn assoc:a->b list->(a->b->Bool)->[b option] =>
     fn x:a =>
      fn l:b list =>
       fn p:a->b->Bool =>
        If

If = if null l then
        <box>None
     else if p x (hd l) then
             <box>(Some (hd l))
          else
             assoc x (tl l) p
```

The program can then be written:

```
case <unbox>(Assoc {[int]} {[[int]*[int]]}
             (<box>5) [<box>(<box>5,<box>42)]
             (fn x:[int]=>fn y:[[int]*[int]]=><unbox>x=<unbox>fst(<unbox>y)))
of
  None => 0
  Some p => <unbox>snd(<unbox>p)
```

We concentrate on this part first. Now from equation 17 we have the following equality:

```
(Assoc {[int]}) {[[int]*[int]]}} =
  <i->list([(unbox,unbox)])->(i->[(box,box)]->i)->[option([(box,box)])]>
    ((Assoc {[int]}) {[int*int]}})
```

Which used on our program after a few applications of Equation 14 and 11 gives:

```
case <unbox><[option([(box,box)])]>
       ((Assoc {[int]} {[int*int]})
        (<box>5) (<list([(unbox,unbox)])>[<box>(<box>5,<box>42)])
        (<i->[(box,box)]->i>(fn x:[int]=>
                              fn y:[[int]*[int]]=>
                               <unbox>x=<unbox>fst(<unbox>y))))
of
  None => 0
  Some p => <unbox>snd(<unbox>p)
```

We consider the parts of this one by one starting with:

`<list([(unbox,unbox)])>[<box>(<box>5,<box>42)]`	$=_{\text{Def.}}$
`<list([(unbox,unbox)])>(<box>(<box>5,<box>42)::[])`	$=_{::}$
`<[(unbox,unbox)]><box>(<box>5,<box>42)::<list([(unbox,unbox)])>[]`	$=_{12}$
`<box;[(unbox,unbox)]>(<box>5,<box>42)::<list([(unbox,unbox)])>[]`	$=_{9}$
`<(unbox,unbox);box>(<box>5,<box>42)::<list([(unbox,unbox)])>[]`	$=_{12}$
`<box><(unbox,unbox)>(<box>5,<box>42)::<list([(unbox,unbox)])>[]`	$=_{\text{rule for pairs}}$
`<box>(<unbox><box>5,<unbox><box>42)::<list([(unbox,unbox)])>[]`	$=_{12,\phi}$
`<box>(5,42)::<list([(unbox,unbox)])>[]`	$=_{[]}$
`<box>(5,42)::[]`	$=_{\text{Def.}}$
`[<box>(5,42)]`	

and then:

```
<i->[(box,box)]->i>
 (fn x:[int]=>fn y:[[int]*[int]]=><unbox>x=<unbox>fst(<unbox>y))         $=_{13}$
fn x:[int]=>
 <[(box,box)]->i>(fn y:[[int]*[int]]=><unbox>x=<unbox>fst(<unbox>y))     $=_{13}$
fn x:[int]=>fn y:[int*int]=><unbox>x=<unbox>fst(<unbox><[(box,box)]>y)  $=_{12}$
fn x:[int]=>fn y:[int*int]=><unbox>x=<unbox>fst(<[(box,box)];unbox>y)   $=_{10}$
fn x:[int]=>fn y:[int*int]=><unbox>x=<unbox>fst(<unbox;(box,box)>y)     $=_{12}$
fn x:[int]=>fn y:[int*int]=><unbox>x=<unbox>fst(<(box,box)><unbox>y)    $=_{\texttt{fst}}$
fn x:[int]=>fn y:[int*int]=><unbox>x=<unbox><box>fst(<unbox>y)          $=_{12,\phi}$
fn x:[int]=>fn y:[int*int]=><unbox>x=fst(<unbox>y)
```

If we apply these results in the program we get:

```
case <unbox><[option([(box,box)])]>(Assoc {[int]} {[(int,int)]}
                                    (<box>5) [<box>(5,42)]
                                    (fn x:[int]=>
                                      fn y:[int*int]=><unbox>x=fst(<unbox>y)))
of
  None => 0
  Some p => <unbox>snd(<unbox>p)
```

This can by using Equation 10 and 12 further be proven equal to:

```
case <option([(box,box)])>
       <unbox>(Assoc {[int]} {[(int,int)]}
               (<box>5) [<box>(5,42)]
               (fn x:[int]=>fn y:[int*int]=><unbox>x=fst(<unbox>y)))
of
  None => 0
  Some p => <unbox>snd(<unbox>p)
```

Using a rule for case-expressions one can show this equal to:

```
case <unbox>(Assoc {[int]} {[(int,int)]}
             (<box>5) [<box>(5,42)]
             (fn x:[int]=>fn y:[int]=><unbox>x=fst(<unbox>y)))
of
  None => 0
  Some p => <unbox>snd(<unbox><[(box,box)]>p)
```

We then consider the second branch of the case-expression:

```
<unbox>snd(<unbox><[(box,box)]>p)   $=_{12}$
<unbox>snd(<[(box,box)];unbox>p)    $=_{10}$
<unbox>snd(<unbox;(box,box)>p)      $=_{12}$
<unbox>snd(<(box,box)><unbox>p)     $=_{\texttt{snd}}$
<unbox><box>snd(<unbox>p)           $=_{12,\phi}$
snd(<unbox>p)
```

and apply this in the program:

```
case <unbox>(Assoc {[int]} {[(int,int)]}
             (<box>5) [<box>(5,42)]
             (fn x:[int]=>fn y:[int*int]=><unbox>x=fst(<unbox>y)))
of
  None => 0
  Some p => snd(<unbox>p)
```

We will now concentrate on the "test"-expression of the case-expressions and by using Equation 11 and 14 we prove that this is equal to:

```
(<i->i->i->unbox>(Assoc {[int]} {[(int,int)]}))
 (<box>5) [<box>(5,42)]
 (fn x:[int]=>fn y:[int*int]=><unbox>x=fst(<unbox>y))
```

We then handle the type applications as follows:

```
<i->i->i->unbox>(Assoc [int] [(int,int)])                         $=_{16}$
(<∀b.i->i->i->unbox>(Assoc [int])) [(int,int)]                    $=_{16}$
(<∀a.∀b.i->i->i->unbox>Assoc) [int] [(int,int)]                   $=_{\text{Def.}}$
<∀a.∀b.i->i->i->unbox>(Fn a => Fn b => fix F) [int] [(int,int)]   $=_{15}$
(Fn a => <∀b.i->i->i->unbox>(Fn b => fix F)) [int] [(int,int)]    $=_{15}$
(Fn a => Fn b => <i->i->i->unbox>(fix F)) [int] [(int,int)]
```

Then using the rule for `fix` we have:

```
<i->i->i->unbox>(fix F)                                          $=_{\psi,11,12,3,4}$
<i->i->i->unbox>(fix <(i->i->i->unbox)->i><(i->i->i->box)->i>F)  $=_{\texttt{fix}}$
fix <i->(i->i->i->unbox)><(i->i->i->box)->i>F                    $=_{12,3}$
fix <(i->i->i->box)->(i->i->i->unbox)>F
```

Inserting the definition of `F` we get:

```
<(i->i->i->box)->(i->i->i->unbox)>
  (fn assoc:a->b list->(a->b->Bool)->[b option] =>
    fn x:a =>
     fn l:b list =>
      fn p:a->b->Bool => If)
```

which we by using Equation 13 several times can change into:

```
  (fn assoc:a->b list->(a->b->Bool)->b option =>
    <i->i->i->unbox>
      fn x:a =>
       fn l:b list =>
        fn p:a->b->Bool =>
         if null l then
            <box>None
         else if p x (hd l) then
                 <box>(Some (hd l))
              else
                 (<i->i->i->box>assoc) x (tl l) p)
```

The if-expression:

```
if null l then
   <box>None
else if p x (hd l) then
        <box>(Some (hd l))
     else
        <box>(assoc x (tl l) p))
```

is by the equations for conditionals equal to:

```
<box>(if null l then
         None
      else if p x (hd l) then
              Some (hd l)
           else
              assoc x (tl l) p)
```

Call the new if-expression `If'` we can by several applications of Equation 13 show that:

```
fn assoc:a->b list->(a->b->Bool)->b option =>
 fn x:a =>
  <i->i->unbox>
    fn l:b list =>
     fn p:a->b->Bool =>
      <box>If'
```

is equal to

```
fn assoc:a->b list->(a->b->Bool)->b option =>
 fn x:a =>
  <i->i->unbox>
    fn l:b list =>
     fn p:a->b->Bool =>
     <unbox><box>If'
```

and by use of Equation 12 and ϕ we further that it is equal to:

```
fn assoc:a->b list->(a->b->Bool)->b option =>
 fn x:a =>
  <i->i->unbox>
    fn l:b list =>
     fn p:a->b->Bool =>
      If'
```

So we finally obtain the optimal completion:

```
case (Fn a => Fn b =>
       fn assoc:a->b list->(a->b->Bool)->b option =>
        fn x:a =>
         fn l:b list =>
          fn p:a->b->Bool =>
           if null l then
              None
           else if p x (hd l) then
                   Some (hd l)
                else
                   assoc x (tl l) p)
      {[int]} {[int*int]}
      (<box>5) [<box>(5,42)]
      (fn x:[int]=>fn y:[int*int]=><unbox>x=fst(<unbox>y))
of
  None => 0
  Some p => snd(<unbox>p)
```

Chapter 7

Implementation of completion inference

In this chapter we describe the algorithm used in the prototype implementation. The two phase reachability algorithm explained in Subsection 5.2.7 is implemented as a depth first search algorithm. In this the graph is traversed depth first by a recursive function and a node is updated (assigned boxed values) when the function has returned from all the successors of the node. After the description of the algorithm (Section 7.1) we presents some benchmarks of the prototype implementation (Section 7.2).

7.1 Description of the algorithm

In this section we present the algorithm implemented by the prototype implementation. The algorithm implements the two reachability phases describe in Section 5.2.7 as a depth first search algorithm.

In the implementation of the algorithm a node (box variable) is represented by *node structure*. Each node structure contains a *value* field containing the box value of the node, a *successor* field containing a list of nodes representing the outgoing edges from the node, and a *visited* field containing a boolean. So we use the graph to represent the box value assignment. To start with, all nodes get assigned an initial box value. This value depends on what kind of optimal completions we want to find. If we are generating ψ-free optimal completions the initial value will be U and if we are generating ϕ-free optimal completions it will be B. The visited field indicates whether a node has not been examined (visited) before and is initially set to false for all nodes except source and sink nodes.

Figure 20 shows an efficient algorithm for finding ψ-free optimal completions that works for pure F_2. We will in the next subsection discuss what changes are needed to the algorithm when we add primitives to F_2. For nodes the three fields that contains the current box value, the list of outgoing edges and the visited tag are called `v`, `succ` and `seenB4`. The function `lub` is the least upper bound operation on the 2-point domain $U \sqsubseteq B$. The function `generate` generates the representation graph and a principal completion from the input F_2-expression in the following way:

- it generates a principal completion in which the box variables are represented by node structures. This is done by slightly modifying a type checking algorithm for F_2 such that it implements the inference rules of Figure 18.

- instead of generating the pair (b,b'), i.e. an edge in the representation graph, it adds b' to b.`succ`.
- for all source and sink nodes it sets the `v` field to the corresponding value of the node and collects the source nodes with value B in a list A_B.
- it sets the `v` field to U for all other nodes.
- it sets the `seenB4` field to false except for all sources and sinks for which it is set to true.

```
generate;
worklist := $A_B$;
while worklist <> [] do
  remove b from worklist;
  for b' in b.succ do findB b';
end while

function findB b =
  if not(b.seenB4) and b.succ <> [] then
     b.seenB4:=True;
     b.v := foldr lub U (map findB b.succ)
  end if;
  return b.v
end function
```

Figure 20: Efficient algorithm for finding ψ-free optimal completions

When the algorithm terminates it is easy to generate the optimal completion from the principal completion. It is also fairly easy to see that the efficient algorithm implements the algorithm described in Section 5.2.3, so we will not show this here, but following the presentation below of the complete algorithm (including primitives) we will give an informal proof of this for the complete algorithm.

In the special cases where a node, that is not a source or a sink node, does not have any source and/or any sink nodes the algorithm will set the value of the node to the default value. This is just fine since in the two cases data is either never created or never used, which means that the choice of representation is irrelevant. However, there are cases where a different choice than the default value might be better. Suppose we are finding ψ-free optimal completions for the expression:

$$(\lambda x : \texttt{int}.\texttt{1})\ (id\{\texttt{int}\}\ \texttt{5})$$

we will then find the optimal completion:

$$(\lambda x : \texttt{int}.\texttt{1})\ (\langle \texttt{box}_{\texttt{int}} \rightarrow \texttt{unbox}_{\texttt{int}} \rangle id\{\texttt{int}\}\ \texttt{5})$$

but if we are allowed to set the representation type of x to [`int`] the we would get the following more efficient completion:

$$(\lambda x:\texttt{[int]}.\texttt{1})\ ((\texttt{box}_{\texttt{int}}\rightarrow\iota)id\{\texttt{int}\}\ \texttt{5})$$

That we only get the former completion is in complete agreement with our definition of formally optimal completions, since the two completions are E-equivalent and are therefore both in the optimal equivalent class. If we want the algorithm to gives us the more efficient completion, we have to take further properties into account, like the actual semantics of the language.

7.1.1 The effect of adding recursive primitives

The algorithm described above works for pure F_2 and also works if we extend F_2 with some primitives (non-recursive). The problem is that the algorithm assumes that when a node in the graph has been visited then its box value field has also been updated. This works fine when there are no cycles in the graph and this will be the case as long as we stick to pure F_2.

If we add recursive primitives like `fix`, `map`, etc. then this is no longer the case and the algorithm has to be modified. The simplest way to do this is to use a union/find based method. We add an equivalence class pointer to each node such that we have to follow this pointer (using the usual path compression used in union/find) in order to find the box value of a node. In this way certain nodes will be equivalence class representatives and hold the box values of their class. We will change the function `findB` such that it no longer returns a box value, but an equivalence class pointer. In a similar way we will change `lub` such that it works on equivalence class pointers, and return the one of the input pointer which points to a node with a box value which is equal to the least upper bound of the box values of the two nodes that the input pointer points to, and if both input nodes have box value U then if one of the nodes has been visited this node is returned. The function `findB` will also be change such that it is not `b.v` that is updated for each node, but the equivalence class pointer `b.ecrp` of the node. The modified algorithm is shown in Figure 21. The definition of `lub` might seem a little strange. If one of the nodes has box value B then this node is returned. This is what one would expect, but if none of the nodes has box value B then if one of the nodes has been visited (`seenB4` field = Visited), but not updated (`seenB4` field = Updated) then this node is returned. The reason for this is that this node must lie on a part of the graph that we are at the moment computing `findB` for (meaning that there is a cycle in the graph) and that this node will eventually be assigned the right box value that will be given to all nodes on the cycle. A node that has either not been visited or is updated must belong to a part of the graph that has no influence on the part of the graph that contains the cycle.

7.1.2 Correctness of the algorithm

We will not formally prove the correctness of the algorithm in Figure 21 with respect to the specification given in Section 5.2.3, but present an informal argument for this:

1. **Termination of the algorithm.** Initially all nodes, except sources and sinks, have `seenB4` = `NotVisited`. Every time `findB` is called `seenB4` is either already different from `NotVisited` or will be set to `Visited`. This can only happen as many time as there are nodes in the graph. The algorithm will therefore terminate.

2. **Correctness.** By the specification, if a node b lies on the path between a source and a sink with box value B then $\varsigma(b) = B$, otherwise $\varsigma(b) = U$. The algorithm starts from

```
generate;
worklist := $A_B$;
while worklist <> [] do
 remove b from worklist;
 for b' in b.succ do findB b';
end while

function findB b =
 let b = find b in
  if b.seenB4 = NotVisited and b.succ <> [] then
    b.seenB4 := Visited;
    let (b1::bs) = b.succ in
     b.ecrp := foldr lub (findB b1) (map findB bs);
     b.seenB4 := Updated
    end
  end if;
  find b
  end
end function

function find b =
 if b.ecrp = b then b
 else
   b.ecrp = find(b.ecrp);b.ecrp
 end if;
end function

function lub b b' =
 let b1 = find b and b1' = find b' in
   case (b1.value,b1'.value) of
     (B,_) => b1
   | (_,B) => b1'
   | _ => if b1.seenB4 = Visited then b1 else b1' end if
 end
end function
```

Figure 21: Efficient algorithm for finding ψ-free optimal completions with primitives

a worklist A_B containing all the sources with box value B. It calls `findB` on all the immediate successor nodes of these (contained in the `succ` field) and `findB` in turns calls itself on each immediate successor node of these nodes unless the node has been visited (or updated). This means that when `findB` is called on a node then this node must lie on a path leading from a source with box value B. Four things can happen when `findB` is called on a node.

(a) The node has `seenB4` = `NotVisited`, but no immediate successors, in which case the node is either a sink node or a node corresponding to a value that is discarded. In both cases the nodes equivalence class representative is returned (ecr), which should be the node itself, as a representation of its box value.

(b) The node has `seenB4` = `Visited`, in which case the node has been visited before on the path from the source node who's successors are currently being treated. In plain English: there is a cycle in the graph. Since the `seenB4` field is `Visited` and not `Updated` it means that at some later point it will be updated with the box value that all the node on the cycle should have (all nodes on a cycle have the same successors and predecessors) and also here we return the ecr of the node.

(c) The node has `seenB4` = `Updated`, in which case the node has either been visited before on the path from some other source node or is a sink node. In both case we return the ecr of the node since it represents the final box value of the node.

(d) The node has `seenB4` = `NotVisited` and has one or several immediate successors. Since we know that the node lies on the path from a source with box value B, its own box values should be set to B, if just one of its immediate successors have box value B, because if one of the nodes immediate successors has box value B this can only be because of two things. Either the node is a sink node with box value B or its box values has been set to B because it lies on the path between a source and a sink with box value B. In both cases the node we are considering must also lie on such a path too and should therefore have its box value set to B. This is done by setting the ecr of the node to one of its immediate successors that have box value B. If no immediate successor nodes have box value B it could be that one of them has been visited on the way to the current node and could later be updated to B (if there are more than one they will all be updated with the same value). We should therefore choose this node as the ecr for the current node and not a node which has box value U. All this is ensured by the definition of `lub`.

This shows that when the algorithm terminates all the node that lies on a path from a source node with box value B will have its box value updated (indirectly through its ecr) to B if it has among its successors a sink with a box values B.

7.1.3 The prototype implementation

The prototype implementation is essentially an ML implementation of the algorithm describe in Section 7.1.1. The ML source code of the implementations is shown in Appendix B. The implementation consists of a parser, a type checker, and an implementation of the algorithm in Figure 21. The type checker is combined with the function `generate` from Figure 21, so when it checks that the F_2-expression is well formed (have correct type annotations) it also generates the graph.

7.2 Benchmarks

The implementation can analyze arbitrary F_2-expressions and output optimal completions for these, but we have no way to run these. We could have written an instrumented interpreter that records statistics while interpreting completions, but what we have done instead is to only run performance tests on F_2-expressions that corresponds to SML-expressions. This is what one would do in an ML compiler that uses F_2 as an intermediate language. When run on an F_2-expression our implementation will output an SML-expression corresponding to the completion where type application and type abstraction has been ignored. The SML-expression will contain operations corresponding to the box and unbox operations of the completion, but these operations will not actually box and unbox values, all they will do when they are executed is to record the fact that a box or an unbox operation has been performed and return the argument unchanged. The SML-expression will also contain operations that will count the stub code operations performed by the completion, e.g. extra closures, pairs, etc. created, extra applications, projections on pairs, etc. performed. The result of running the SML-expression will be a pair of the actual result of running the completion and the statistical information.

Eight programs, shown in Appendix D, were selected for the experiments. These were:

- *mi-sort*, a insert sorting program where the insert function is monomorphic.
- *insert-sort*, a insert sorting program where the insert function is polymorphic.
- *flip-list*, which "flips" the elements of a list of pairs of integers.
- *leroy*, which is an almost "pathological" program for which Leroy's benchmark shows a major slow-down compared to a fully boxed implementation.
- *poulsen*, a program for which Poulsen [Pou93] reports that his algorithm gives very poor results (606/903 box/unbox-operations).
- *sieve*, which computes the prime numbers between 1 and 100.
- *horner*, evaluates a polynomial by constructing a list of functions (from Thiemann [Thi95]).
- *mogensen*, a program that contains a lot of polymorphism which causes Leroy's completions to do a lot of extra boxing and unboxing.

The example programs *mi-sort*, *insert-sort*, *flip-list*, *leroy*, *poulsen* were all taken from [Pou93]. The example *mogensen* is based on an idea by Torben Mogensen.

We have performed several different tests describe in the following.

7.2.1 $\mathcal{C}_{**}$-completions with polymorphic data structures

For all eight programs we have generated two kinds of $\mathcal{C}_{**}$-completions, the optimal ψ-free normal form and the optimal ϕ-free normal form. We have run the two completions and counted the number of box and unbox operations performed and counted the stub code performed as explained above. All data structures are polymorphic, that is, `cons`, `hd`, `tl`, `pair`, `fst`, `snd`, etc. were given polymorphic types and treated as primitives.

Figure 22 shows the results. Since we require that the world is completely unboxed the examples include unbox operations to insure that this is the case. The operations corresponds to the operations that a print routine in an interpretive implementation of a language like ML will have to perform. We only show performance results for the stub code for closures (cl) and applications (app). Some of the examples also performed stub code for pairs and lists, but this code was only involved in unboxing the output, so is not really essential in judging the quality of the completions.

	opt. ψ-free norm.				opt. ϕ-free norm.			
	coercions		stub code		coercions		stub code	
Program	box	unbox	cl.	app.	box	unbox	cl.	app.
mi-sort	17	171	188	188	17	171	308	308
insert-sort	17	171	196	348	17	171	308	308
flip-list	20	20	54	86	20	25	60	60
leroy	269	269	14	542	446	446	22	880
poulsen	6	6	14	14	6	6	4	8
sieve	99	847	302	1074	99	847	1644	1644
horner	41	41	234	270	50	50	126	144
mogensen	6	7	58	58	6	7	16	16

Figure 22: Performance: benchmarks for $\mathcal{C}_{**}$ with polymorphic data structures

The results indicate, as one would expect, that there is no clear winner of the two kinds of completions and that it depends on the given program which is best. Also the two kinds of completions produce by the algorithm are formally optimal and represent an equivalence class of completions, which formally are considered equally good. This means that the choice made by the algorithm will not always be the best, in fact, since the algorithm in a sense chooses extremes in the equivalence classes, there could often be a better completion which puts the coercions at just the right places (compare the test results for *horner* with the one obtained for the $\overline{\mathcal{C}}_{p*}$-completions in Figure 24).

7.2.2 $\mathcal{C}_{**}$-completions with in-lined primitives

The next test is similar to the one with polymorphic data structures, except that we now allow data structures to be also monomorphic, e.g. lists of unboxed integers. We do this by making `cons`, `hd`, `tl`, `pair`, `fst`, `snd`, etc. in-lined (language) primitives. We performed the same tests and the results are shown in Figure 23.

An interesting observation is that, even though data structures are allowed to be monomorphic, none of the generated completions had coercions with recursive coercion constructors (`map`) in them, except for a an outermost application of `map(unbox)` which corresponds to the unboxing that will be performed by a print routine. This indicates that it is advantageous to allow recursive data structures with unboxed components and not exclude these as Leroy[Ler90] does. Notice that the monomorphic program *mi-sort* now performs no coercions at all. This will be the case for all monomorphic programs.

7.2.3 $\overline{\mathcal{C}}_{p*}$-completions

The last test shows the quality of the variant of our algorithm that produces $\overline{\mathcal{C}}_{p*}$-completions. We have tested this with the same programs as were used in the two previous tests, that is, with polymorphic data structures and with "free" data structures. The results are presented in Figure 24. Since there is no internal stub code in $\overline{\mathcal{C}}_{p*}$-completions (which the test also showed) there are no performance figures for this.

	opt. ψ-free norm.				opt. ϕ-free norm.			
	coercions		stub code		coercions		stub code	
Program	box	unbox	cl.	app.	box	unbox	cl.	app.
mi-sort	0	0	0	0	0	0	0	0
insert-sort	17	171	162	314	17	171	308	308
flip-list	20	20	6	14	30	35	64	72
leroy	269	269	14	542	446	446	22	880
poulsen	3	3	6	6	3	3	4	8
sieve	99	1283	104	876	872	848	878	926
horner	41	41	198	234	50	50	126	144
mogensen	6	7	58	58	6	7	16	16

Figure 23: Performance: benchmarks for $\mathcal{C}_{**}$ with in-lined primitives

	Polymorphic data structures				In-lined data structures			
	ψ-$\overline{\mathcal{C}}_{p*}$ norm.		ϕ-$\overline{\mathcal{C}}_{p*}$ norm.		ψ-$\overline{\mathcal{C}}_{p*}$ norm.		ϕ-$\overline{\mathcal{C}}_{p*}$ norm.	
Program	box	unbox	box	unbox	box	unbox	box	unbox
mi-sort	17	171	17	171	0	0	0	0
insert-sort	17	171	17	171	17	171	17	171
flip-list	20	25	20	25	20	25	20	25
leroy	270	786	270	786	270	786	270	786
poulsen	6	6	6	6	3	3	3	3
sieve	99	847	99	847	99	436	99	847
horner	41	41	50	50	41	41	50	50
mogensen	6	7	6	7	6	7	6	7

Figure 24: Performance: benchmarks for $\overline{\mathcal{C}}_{p*}$

It is interesting to notice, first of all that the performance figures for $\overline{\mathcal{C}}_{p*}$-completions is not much worse than the performance figures for $\mathcal{C}_{**}$-completions, and in some cases even better. If one takes into account that no stub code is inserted into the $\overline{\mathcal{C}}_{p*}$-completions it makes the difference even smaller. In some cases $\overline{\mathcal{C}}_{p*}$-completion is in fact E-equivalent to the corresponding $\mathcal{C}_{**}$-completion (seen by inspecting the completions).

7.2.4 Comparison with other approaches

We conclude by comparing our approach with other approaches. We compare with Leroy's, Poulsen's and Thiemann's approach. The figures are shown in Figure 25.

Since Leroy does not present his performance results in a similar way to ours (he presents real performance figures from a compiler) the results presented here were produced by generating completions the way Leroy's approach specifies, and translating them into SML as with our completions and running them. This does not include the kind of simplifications that Leroy's system does, but since these mostly have effect on the amount of stub code inserted, we believe that it should not have much influence on the number of representation coercions performed.

Poulsen gives several performance figures in [Pou93], but these are not very transparent. The problem is that his framework allows creation of boxed values (e.g. integers) directly which is then counted as a boxed value construction and not as a performed `box` coercion as it does in our framework. Furthermore, he counts the access to a boxed value as a boxed value reference and not as a performance of an `unbox` coercion. He only inserts coercions when the representation is changed and not referenced immediately, but saved for later use. So just counting the number of coercions performed will give a wrong picture of the quality of his completion. The performance figures that we present for his approach is of the form c/r where c is the number of box/unbox coercions performed, and r is either found by adding up the number of constructions of different boxed values (con.), or found by adding up the number of references to different boxed values (ref.). Since Poulsen does not specify precisely what he means by a reference to a value it is not clear how many references to boxed values corresponds to an unbox operation, but clearly some references must, as may be seen from the *flip-list* example which did not perform any box/unbox coercions at all. We also suspect that Poulsen's numbers do not include the number of unbox operations needed to produce an unboxed result.

Thiemann's numbers are taken almost directly from [Thi95] except that a number of unbox operations have been added that correspond to the number of unbox operations needed to produce an unboxed result.

	Leroy		Poulsen		Thiemann	
Program	box	unbox	box/con.	unbox/ref.	box	unbox
mi-sort	17	171	0/0	0/0	0	0
insert-sort	17	171	0/17	154/778	17	169
flip-list	20	20	0/20	0/35	20	20
leroy	445	445	256/270	256/1300	269	269
poulsen	6	6	300/303	300/603	3	3
sieve	99	847	-	-	99	436
horner	-	-	-	-	31	31
mogensen	12	13	-	-	-	-

Figure 25: Performance: compared benchmarks

7.2.5 Conclusion from tests

The tests show that our formally optimal completions are generally not better than Leroy and Thiemann. This is however not unexpected, since these corresponds to some fixed choice of element in an equivalence class and in some sense is an extreme (box/unbox as early/late as possible). There will of course be lots of cases where choosing something in between, like a $\overline{\mathcal{C}}_{p*}$-completion, is better. Comparing with Poulsen is hard, but at least for the example *poulsen* where Poulsen reports that his algorithm gives a bad result we get a much better result. At least one test shows the ability of our algorithm to optimize completions. The *mogensen* test shows an example where our algorithm gives a considerably better result than Leroy's, because repeated boxing and unboxing of data that is not referenced is eliminated.

The test also shows what the optimal completions perform a considerable amount of stub, although this could be considerably reduced by using E-equivalence to remove a lot of the induced coercions similar to what Thiemann [Thi95] does.

Also our $\overline{\mathcal{C}}_{p*}$-completions with in-lined primitives compare well with Leroy's and Thiemann's completions even if we do not consider that their completions may perform some amount of internal stub code where our $\overline{\mathcal{C}}_{p*}$-completions perform none.

A thing that cannot be seen directly from the test results is that in several of the tests the $\overline{\mathcal{C}}_{p*}$-completions were E-equivalent to the optimal completions.

Finally, we may conclude that the algorithm that finds $\overline{\mathcal{C}}_{p*}$-completions seems to be a good choice for a representation analysis in a real implementation, since it has the following advantages:

1. It inserts no stub internal code and thus produces smaller completions.
2. It seems to produce completions of good quality not much worse than the formally optimal ones.
3. It runs faster and uses less space than the one that produces formally optimal completions.

Chapter 8

Related work

There is essentially two different areas of related work. One area is representation analysis in general, but not necessarily with boxing and explicit coercions. The other area is proof theory for languages with explicit coercions, that is, coercion calculus, equational axiomatization of coherence, etc. In this chapter we will discuss some related work in these two areas.

8.1 Representation analysis

The substantial cost of manipulating data in boxed representation, especially for numeric programs, has been observed in both dynamically typed high-level languages like LISP and statically typed polymorphic languages such as Standard ML, and Haskell.

Most of the efforts in LISP implementation have focused on optimizing number representations by keeping them in unboxed form [Ste77, BGS82, KKR$^+$86]. Peterson [Pet89] uses an elaborate execution-frequency based criterion for the cost of representation coercions. In this setting he shows how the optimal placement of coercions can be reduced to a well-known network flow problem. Common to all these efforts is the intent to optimize representations of atomic data, particularly numbers. Indeed in Peterson's framework operations on pairs and lists simply require boxed arguments. Steenkiste and Hennessy, however, have found that in a suite of ten LISP programs up to 80% of the representation coercions are list tagging/untagging operations.

Peyton Jones and Launchbury [PJL91] and Leroy [Ler92] suggested making representation types and boxing and unboxing operations explicit in programs. Even though there are some technical differences, the languages they use are at the core similar: a functional language with explicit boxing/unboxing coercions.

Peyton Jones and Launchbury do not provide a method of inferring a completion, but concentrate on the semantics of their explicitly boxed language and on optimization of boxing by program transformation. Those optimizations are, for example, a form of common subexpression elimination that cannot be formalized in our framework as the transformations may change the underlying program.

Leroy describes a translation of Core-ML expressions to explicitly boxed Core-ML expressions. This translation is not deterministic as it depends on the specific typing derivation of the underlying Core-ML expression, but every translation of such a source Core-ML expression is a completion in our sense (not the other way round, however). The experimental results of incorporating his boxing analysis in the Gallium compiler for CAML Light show that the resulting performance can be drastically different from the original compiler that uses canonically boxed representations. The results, though, are not uniformly better. The canonically boxed completions are ψ-free whereas Leroy's is ϕ-free in our terminology. The results are in line with our general considerations that indicate that monomorphic programs should generally fare better with a ϕ-free completion whereas highly polymorphic programs are likely to be better off with a ψ-free completion. Our rewriting system for ϕ-free completions will improve the result of Leroy's completion by eliminating all ψ-redexes, and our rewriting system for ψ-free completions will improve the canonically boxed completion by eliminating all ϕ-redexes. A problem with Leroy's approach is that since he only boxes at points where variables with polymorphic types are instantiated (type applications in our terminology) he gets many induced coercions which means that there are many additional abstractions and applications (stub code) in his completions as explained in 5.3.2.

Using Leroy's framework Poulsen [Pou93] presents a more involved translation, but draws on "constraint solving" to eliminate more boxing/unboxing operations than Leroy's translation in many, but not all cases. The interesting aspect of Poulsen's completions is that they, just like our optimal completions, are not required to have a canonically defined representation type for the types occurring in type applications as in Leroy's work, but determines an appropriate representation type as part of the constraint solving process. On the other hand it appears that some boxing/unboxing operations are built into the constructors of the language and are not accounted for in the question of optimizing the boxing in the program.

Henglein and the author [HJ94] presented a boxing analysis for a small ML like language called core-XML. The approach was the same as in the present work, except that the language is different.

Given an explicitly typed expression for an ML-like language with type τ the result of a boxing analysis depends on the particular typing derivation chosen. Leroy's completion uses implicitly the derivation obtained by Algorithm W [Mil78] since his translation performs type inference and boxing simultaneously where let-bound variables receive the principal type of the bound expression. Peyton-Jones/Launchbury and Poulsen also appear to assume that type inference is performed in the style of Algorithm W. The principal type of a function is the "most" polymorphic type and thus imposes the most boxing demands on the arguments to the function. A "more" monomorphic derivation, on the other hand, could still yield the same type for the whole expression, but using more monomorphic types for the local variables. Bjørner gives an algorithm called M for finding a *minimally* polymorphic typing derivation [Bjø92, Bjø94]. Minimally polymorphic derivations do not always exist, but his algorithm generally lowers the local degree of polymorphism in comparison to Algorithm W. Note that our boxing analysis, if restricted to an ML-typable subset, does not presuppose a specific typing derivation for expression, but interfaces with *any* of its type derivations, which is represented by an explicitly typed expression.

Shao and Appel describes a type-based compiler for Standard ML in [SA94] and their approach to boxing is also based on Leroy's. They use the *minimal typing derivations* of Bjørner to eliminate unnecessary boxing. The compiler converts programs into CPS style before doing optimizations and one of the optimizations that they perform is the elimination of `box;unbox` and `unbox;box` pairs. To what degree they can eliminate unnecessary boxing and unboxing is hard to say, but they present interesting performance results showing a reduction in heap allocation by 36% and an improvement of 19% compared to a old non-type-based version of the compiler.

Harper and Morrisett [HM95] propose a different technique for implementing polymorphism than the traditional one where data is represented uniformly. By passing types, that are unknown at compile-time, at link-time or run-time, primitive operations can analyze types and select the appropriate specialized operations for the types. This approach allow monomorphic code to use efficient data representations, just as ours, Leroy's and Poulsen's, but without the need for representation shifts (through coercions) when data is passed to a polymorphic function. This may probably lead to considerable space savings due to the better representation of data and time savings due to the lack of coercions, but this may be out weighed by the time and space costs due to the passing of type arguments at run-time. The practical implications of this is not clear. Their approach can also handle mutable objects like arrays and references, which is something which may be problematic with a coercion approach.

Recent work by Thiemann [Thi95] also uses an explicitly typed core ML language (like Core-ML) for his representation analysis. He defines a translation from Core-ML expressions into Core-XML, explicitly boxed Core-ML. His translation uses fixed places where coercions can be placed, but the places are different from Leroy's, since it allows only coercions on the operator expression in applications, as in $\langle \mathbf{c} \rangle e\, e'$. He uses unification to optimize these coercions. Expressions that we in our work demand to be unboxed, will in his work get a type, written τ^U, that can have any representation, and generic parts of polymorphic functions get a type, written τ^B, that must be boxed. Two types τ^U and τ^B are unified to τ^B. This means that the completions that he generates are ψ-free. He follows his generation phase with a simplification phase that makes sure that the resulting expression contains only primitive coercions. This phase essentially uses Equation 13 from Figure 12, the two rules in Figure 11, and the following two rules:

$$\begin{aligned} \langle \mathbf{c} \to \mathbf{d} \rangle \lambda x{:}\rho.e &= \lambda x{:}\rho'.\mathtt{let}\ x{:}\rho = \langle \mathbf{c} \rangle x\ \mathtt{in}\ \langle \mathbf{d} \rangle e && (12') \\ \langle \mathbf{c} \to \mathbf{d} \rangle e &= (\lambda f{:}\rho'_x \to \rho.\lambda x{:}\rho_x.\langle \mathbf{d} \rangle (f\ (\langle \mathbf{c} \rangle x)))\ e && (17) \end{aligned}$$

where precedence is given to rules in the following order: (13) before (12′) before (17) to get rid of induced coercions. This method will, exactly like Leroy's method, not always produce completions completely free of internal stub code, but it does this in a much more limited way, and the experimental result presented by Thiemann (see Section 7.2) indicate that the method is quite useful. An interesting point in Thiemann's work is his use of order-sorted types to distinguish between types that have to be either always boxed (argument to polymorphic functions) or unboxed and types that may be free to be either boxed or unboxed (arguments in language primitives). Thiemann also introduces some further optimizations. For example, an optimization which may be used to eliminate some coercions in case the underlying implementation does tail call elimination.

Tofte's and Talpin's region inference [TT94] is an alternative approach to implementing polymorphism. At run time data is placed in regions and the store consists of a stack of regions. Region inference determines statically which regions to put data in. In the implementation an analysis is used to determine the size of regions at compile-time. If this is possible for a region it can be placed directly on the stack and the data in the region can the be accessed directly and not through a pointer. In this way regions for which the size may be determined at compile-time can be represented "unboxed" on the stack, while regions for which this is not possible have to be represented "boxed" on the stack (by a reference). Experiments presented in [TT94] indicate that this implementation scheme in many cases may lead to much less use of memory that traditional methods.

8.2 Coherence and equivalence

The notion of coherence appeared first in computer science literature in the work of Breazu-Tannen, Coquand, Gunter, Scedrov [BTCGS91, BTGS90] and Curien, Ghelli [CG90, Ghe90]. They use it to give interpretations of subtype based systems, where application of the subtyping rule is interpreted by an (explicit) coercion. Since a given program with subtyping may have many different translations it is integral to prove that all of them are coherent for the given semantics (via arbitrary translation and interpretation of the target program) to be well-defined.

Thatte describes a method for inferring very powerful implicit coercions between isomorphic types in a type inference system enriched with coercions between arbitrary isomorphic types. Our application can be viewed as simple variant of this problem as arbitrary representation types of the same standard type — and only those — can be coerced to each other. On the other hand Thatte does not deal with optimizing the coercions required in this fashion.

8.3 Dynamic typing

The notion of completion and its congruence theory is inspired and closely related to the work on dynamic typing by Henglein reported in [Hen93], which explicates type tagging and untagging (type checking) operations in dynamically typed languages. The purpose of doing so is completely analogous to boxing analysis: to eliminate most statically type tagging and untagging operations and to implement only the remaining ones while still obtaining "safe" program execution; i.e., well-defined program behavior. See also the work by Cartwright, Fagan and Wright on soft typing [CF91, WF92] or the work of Henglein and Rehof [HR95] on safe polymorphic type inference for Scheme. Rehof [Reh95] has in his work on polymorphic dynamic typing also studied the class of completions which he calls $\overline{\mathcal{C}}_{p*}$.

Chapter 9

Conclusion and Future Work

We have presented a calculus, formal optimality criteria and rewriting-based algorithms for finding good representations of data as boxed or unboxed data in a polymorphically typed programming language. The word "good" here is to be understand in a very general and broad sense. What has been left out is a detailed analysis of specific language properties and implementation considerations that have an effect on the actual performance. This has been done to make the results "universal" and applicable in different settings, even different semantics of the same language.

We have furthermore presented a simpler and more direct algorithm based on a different idea than the rewriting-based algorithms. Judging by experimentation with some small F_2-expressions corresponding to simple Standard ML programs our "formally" optimal completions also tend to be consistently better in practice than previously described boxing analyses, if we count only the number of boxing/unboxing operations executed. Since no implementation decisions are made at the time the boxing analysis is conducted its output should combine well and without much interference with later implementation phases.

The general framework of treating boxing analysis as a translation of a program to a language with explicit boxing and unboxing operations, due to Leroy [Ler92] and implicit also in Peyton Jones and Launchbury [PJL91] encapsulates boxing analysis as a single phase. The representation type of an explicitly boxed program specifies its interface and should thus in principle allow separate compilation of program modules.

Our work has several advantages over related work presented in the literature, these are:

- It has a general framework and robust criterion for the quality of boxing completions, which accounts for the costs of boxing/unboxing operations, but abstracts from other language properties and implementation concerns, and contains notion of (formal) optimality. Neither Leroy's nor Poulsen's nor Thiemann's work contains such a notion, and only Thiemann's completions are optimal (ψ-free) in our sense.

- Our framework admits recursive data structures with unboxed components by including methods to ensure that costly recursive coercions are not inserted in completions (in $\overline{\mathcal{C}}_{p*}$-completions). This is specially noticeable in monomorphic programs. Leroy's framework does not allow this.

- By using $\overline{\mathcal{C}}_{p*}$-completions we ensure that no internal stub code will be inserted without readmitting too many representation coercions. This could save both space and time at run-time as well as it make target programs smaller. All the related work will insert internal stub code in some completions.

- We present an efficient and flexible algorithm that can be used to find both optimal ($\mathcal{C}_{**}$) completions and $\overline{\mathcal{C}}_{p*}$-completions. It also seems to be easy to incorporate Peterson's ideas in the algorithm.

There are several problems with making full use of boxing-optimized programs:

1. Garbage collection often requires "tagging" of heap-allocated data with explicit type and size information. Thus an unboxed representation may well have to be tagged (= boxed) anyway to accommodate the garbage collector. This would not be a problem in the ML-kit compiler, but the implementation of polymorphic equality might be.

2. A boxed representation is the result of an evaluation. In lazy languages often boxed representations are required since the evaluation of an expression is not statically known to terminate or to be advantageous. Thus an expression determined to be best kept in unboxed form by our boxing analysis may still have to be boxed at run-time.

9.1 Future work

Future could include a plan to extend the efficient algorithm for computing optimal boxing completions, to also take account of control dependencies and carefully place coercions at points where they get executed with minimum run-time frequency. As we have pointed out before we expect that one could use analyses such as Peterson's [Pet89] for this purpose.

Otherwise, the work presented in this thesis can be extended in several directions:

1. Work out a proof that E-equality for completions only "moves" coercions around. One might assume certain properties about coercions like totality and strictness and show that the equations of Figure 12 are sound with respect to all interpretations of the primitive coercions in any parametric model.

2. Work out the relationship between Equation 17 of our equational system and parametricity.

3. Work out the relationship between the ψ/ϕ-free optimal completions found by reduction and the ones found by the graph based algorithm.

4. Write an efficient implementation of the reduction based algorithm and investigate how it performs compared to the graph based algorithm.

5. Work out an efficient algorithm that takes the semantics into account by using control dependencies to carefully place coercions at points where they get executed with minimum run-time frequency.

6. Work out how to integrate our boxing analysis into the region-based implementation of Standard ML currently underway at DIKU (see [TT94]). The use of region-based memory management also obviates the need for global garbage collection and thus the first of the two restrictions above.

7. Investigate what kind of results holds for $\overline{\mathcal{C}}_{p*}$-completions, especially, whether there exist optimal $\overline{\mathcal{C}}_{p*}$-completions.

Appendix A

Notation and terminology

It has been our primary goal, when ever possible, to use notation that is, if not standard, then widely used. This is of course not an easy task, especially in computer science where the standards are being set as we write this, but the notation that we feel is not completely standard and that we will use without further explanation will be described in this section.

The notation $t[t_i/x^i]$ means substitute t_i for the i'th occurrence of x in t", for some fixed ordering of occurrences of x in t. We will also use the term $t[t_i/x^i]$ as a pattern. If a term t' matches this pattern then the part of t' that matches the i'th occurrence of x in t will have to match t_i. Ordinary substitution $t[t'/x]$ will also be used as a pattern in a similar way, except that then all the occurrences of x will have to match the same expression.

If e is an expression, the notation $\Sigma(e)$ means the set of all subexpression in e.

Free indices are always assumed universally quantified, i.e. if we write $x_i{=}t_i$ this means for all i: $x_i{=}t_i$ and the range of i is assumed given by the context.

If t is a tuple or a term evaluating to a tuple $t.f$ is the n'th element of the tuple. If t is a record or a term evaluating to a record $t.f$ is the field with name f.

We use $\equiv$ for syntactic equality to distinguish it from provable equality $=$.

Often we will define an operator on some domain and we then extend this in the natural way to an operation (homomorphism) on a structural domain that contains the original domain. We will then not distinguish between the operator on the original domain and the operator on the structured domain if it is clear from the context, usually the domain of the argument, which operator we are using. If o is an operation on types, then we write $o(\tau)$ when we use o on a type τ, we may similarly write $o(x{:}\tau)$ when we use o on a type assumptions $x{:}\tau$ where $o(x{:}\tau) = x{:}o(\tau)$, and $o(\Gamma)$ when we use o on a type assignment $\Gamma = \{x_1{:}\tau_1,\ldots\}$ where $o(\Gamma) = \{o(x_1{:}\tau_1),\ldots\}$.

Definition A.1 *R-reduction(R-reduction) Let R be a set of rewrite rules and E be a set of equations.*

1. *We say that a term t* R-reduces in one step to t' under E-equality, *written $E \vdash t \Rightarrow_R t'$, if there exist terms r and r' such that $E \vdash t = r$, $E \vdash t' = r'$ and r reduces to r' under R.*

2. *We say that a term t* R-reduces to t' under E-equality, *written $E \vdash t \Rightarrow_R^* t'$, if t R-reduces in N steps, where $N{\geq}0$, to t'.*

If E is empty we will simply say that a term t *R-reduces to t'* if that a term t R-reduces to t' under E-equality. If R is a set of rewrite rules and E is a set of equations then the process of rewriting defined in 2 above is called R-reduction modulo E-equality. Often R will be a set of equations E oriented uniformly from left to right or from right to left, in which case we shall write $E^{\rightarrow}$ or $E^{\leftarrow}$, respectively.

Appendix B

Implementation of Boxing Analysis

This appendix contains the complete code of an experimental implementation of the algorithm presented in Chapter 7. The program is written in SML and has been run using Standard ML of New Jersey with the Edinburgh Library.

B.1 Signature: `EXPRESSION`

The file below contains the signature `EXPRESSION`. This is the signature of the functor `Expression` (shown in the next section) which contains definitions of the most important data structures:

- `node` : the node of the representation graph
- `f2type` : F_2 types
- `bf2type` : F_2 representation types
- `coercion` : coercions
- `f2exp` : F_2 expressions
- `bf2exp` : explicitly boxed F_2 expressions (completions)

and operations on these data structures.

```
signature EXPRESSION =
  sig
    eqtype id       (* identifiers *)
    eqtype tyvar    (* type variables *)
    eqtype nodeid   (* used for debugging *)
    eqtype tc       (* type constructor name *)

    datatype boxval = B | U  (* box values *)

    datatype mode = V   (* visited *)
                  | NV  (* not visited *)
                  | VU  (* visited and updated *)

    datatype node = Node of nodeid *        (* node id (for debuging) *)
```

```
                        node ref *      (* next node (for debuging) *)
                        node ref *      (* ecr pointer *)
                        boxval ref *    (* box value *)
                        node list ref * (* successors *)
                        node list ref * (* predecessors *)
                        mode ref
              | Nil

datatype f2type =
   VARf2type of tyvar |
   CONf2type of node * tc * f2type list |
   FUNf2type of node * f2type * f2type |
   FORALLf2type of node * tyvar * f2type

datatype bf2type =
   VARbf2type of tyvar |
   CONbf2type of tc * bf2type list |
   FUNbf2type of bf2type * bf2type |
   FORALLbf2type of tyvar * bf2type |
   BOXbf2type of bf2type

datatype coercion =
   IDcoer of bf2type |
   CONcoer of tc * coercion list |
   BOXcoer of bf2type |
   UNBOXcoer of bf2type |
   INBOXcoer of coercion |
   FUNcoer of coercion * coercion |
   FORALLcoer of tyvar * coercion |
   COMPcoer of coercion * coercion

datatype pmode = P | O

datatype 'a f2exp =
   IDENTf2exp of 'a * id * 'a |
   PRIMf2exp of 'a * id * f2type * f2type list * pmode |
   IFf2exp of 'a * 'a f2exp * 'a f2exp * 'a f2exp |
   ABSf2exp of 'a * id * f2type * 'a f2exp |
   APPf2exp of 'a * 'a f2exp * 'a f2exp |
   TABSf2exp of 'a * tyvar * 'a f2exp |
   TAPPf2exp of 'a * 'a f2exp * f2type

datatype 'a bf2exp =
   IDENTbf2exp of 'a * id |
   PRIMbf2exp of 'a * id * f2type * f2type list * pmode |
   IFbf2exp of 'a * 'a bf2exp * 'a bf2exp * 'a bf2exp |
   ABSbf2exp of 'a * id * bf2type * 'a bf2exp |
   APPbf2exp of 'a * 'a bf2exp * 'a bf2exp |
   TABSbf2exp of 'a * tyvar * 'a bf2exp |
   TAPPbf2exp of 'a * 'a bf2exp * bf2type |
   CAPPbf2exp of 'a * coercion * 'a bf2exp

val mkid : string -> id
val id2string : id -> string
val mktyvar : string -> tyvar
val changetyvar : tyvar * string -> unit
val tyvar2string : tyvar -> string
```

```
    val mktc : string -> tc
    val tc2string : tc -> string
    val mknodeid : int -> nodeid
    val nodeid2int : nodeid -> int

    val replacef2type : f2type * f2type f2exp -> f2type f2exp
    val getf2type : f2type f2exp -> f2type
    val getbf2type : bf2type bf2exp -> bf2type

    val printtypes : bool ref
    val printtypesOn : unit -> unit
    val printtypesOff : unit -> unit
    val layout_boxval : boxval -> string
    val getnodeid : node -> int

    val equaltyvar : tyvar -> tyvar -> bool
    val equaltype : f2type -> f2type -> bool
    val equalbtype : bf2type -> bf2type -> bool

    val screen : string->unit

    val pp_SML    : (string->unit) * bf2type bf2exp * string -> unit
    val pp_source : (string->unit) * f2type f2exp -> unit
    val pp_target : (string->unit) * bf2type bf2exp -> unit
    val pp_type   : (string->unit) * f2type -> unit
    val pp_btype  : (string->unit) * bf2type -> unit

  end;
```

B.2 Functor: `Expression`

The file below contains the functor `Expression`. This contains contains definitions of the most important data structures used in the program. It also contains different kinds of operation on these data structures. The most important of these are:

- `pp_SML` which prints a completion out as a ML program with operation to perform statistics. This was used to obtain the performance results presented in Section 7.2.
- `pp_source`, `pp_type`, `pp_target` and `pp_btype` that pretty-prints expressions and types for debug purposes. The pretty-printer use is the one written by Tofte and is essentially identical to the one used in the ML-Kit [BRTT93].
- `printtypesOn` and `printtypesOff` that switches printing of types during debuging on and off.

```
functor Expression() : EXPRESSION =
  struct
    structure PP = PrettyPrint();
    open PP

    type id = string          (* identifiers *)
type tyvar = string ref   (* type variables *)
type nodeid = int         (* used for debugging *)
type tc = string          (* type constructor name *)

exception ThisCannotHappen (* for situation I know for sure cannot happen *)
exception ThisShouldNotHappen (* for situation that should not happen *)

datatype boxval = B | U  (* box values *)

datatype mode = V   (* visited *)
              | NV  (* not visited *)
              | VU  (* visited and updated *)

datatype node = Node of nodeid *        (* node id (for debuging) *)
                        node ref *      (* next node (for debuging) *)
                        node ref *      (* ecr pointer *)
                        boxval ref *    (* box value *)
                        node list ref * (* successors *)
                        node list ref * (* predecessors *)
                        mode ref
              | Nil

datatype f2type =
   VARf2type of tyvar |
   CONf2type of node * tc * f2type list |
   FUNf2type of node * f2type * f2type |
   FORALLf2type of node * tyvar * f2type

datatype bf2type =
   VARbf2type of tyvar |
   CONbf2type of tc * bf2type list |
   FUNbf2type of bf2type * bf2type |
   FORALLbf2type of tyvar * bf2type |
   BOXbf2type of bf2type

datatype coercion =
   IDcoer of bf2type |
   CONcoer of tc * coercion list |
   BOXcoer of bf2type |
   UNBOXcoer of bf2type |
   INBOXcoer of coercion |
   FUNcoer of coercion * coercion |
   FORALLcoer of tyvar * coercion |
   COMPcoer of coercion * coercion

datatype pmode = P | O

datatype 'a f2exp =
   IDENTf2exp of 'a * id * 'a |
   PRIMf2exp of 'a * id * f2type * f2type list * pmode |
   IFf2exp of 'a * 'a f2exp * 'a f2exp * 'a f2exp |
   ABSf2exp of 'a * id * f2type * 'a f2exp |
   APPf2exp of 'a * 'a f2exp * 'a f2exp |
   TABSf2exp of 'a * tyvar * 'a f2exp |
   TAPPf2exp of 'a * 'a f2exp * f2type

datatype 'a bf2exp =
     IDENTbf2exp of 'a * id |
     PRIMbf2exp of 'a * id * f2type * f2type list * pmode |
     IFbf2exp of 'a * 'a bf2exp * 'a bf2exp * 'a bf2exp |
     ABSbf2exp of 'a * id * bf2type * 'a bf2exp |
     APPbf2exp of 'a * 'a bf2exp * 'a bf2exp |
     TABSbf2exp of 'a * tyvar * 'a bf2exp |
     TAPPbf2exp of 'a * 'a bf2exp * bf2type |
     CAPPbf2exp of 'a * coercion * 'a bf2exp

fun mkid s = s
fun id2string s = s
fun mktyvar s = ref s
fun changetyvar(tv,s) = tv:=s
fun tyvar2string (ref s) = s
fun mktc s = s
fun tc2string s = s
fun mknodeid i = i
fun nodeid2int i = i

(*******************************************************************)
(* Operations on expressions:                                      *)
(*******************************************************************)
fun replacef2type(t,IDENTf2exp(_,x,tx))      = IDENTf2exp(t,x,tx)
  | replacef2type(t,PRIMf2exp(_,n,t',ts,m)) = PRIMf2exp(t,n,t',ts,m)
  | replacef2type(t,IFf2exp(_,e1,e2,e3))    = IFf2exp(t,e1,e2,e3)
  | replacef2type(t,ABSf2exp(_,x,tx,e))     = ABSf2exp(t,x,tx,e)
  | replacef2type(t,APPf2exp(_,e,e'))       = APPf2exp(t,e,e')
  | replacef2type(t,TABSf2exp(_,tv,e))      = TABSf2exp(t,tv,e)
  | replacef2type(t,TAPPf2exp(_,e,t'))      = TAPPf2exp(t,e,t')

fun getf2type(IDENTf2exp(t,_,_)) = t
  | getf2type(PRIMf2exp(t,_,_,_,_)) = t
  | getf2type(IFf2exp(t,_,_,_)) = t
  | getf2type(ABSf2exp(t,_,_,_)) = t
  | getf2type(APPf2exp(t,_,_)) = t
  | getf2type(TABSf2exp(t,_,_)) = t
  | getf2type(TAPPf2exp(t,_,_)) = t

fun getbf2type(IDENTbf2exp(t,_)) = t
  | getbf2type(PRIMbf2exp(t,_,_,_,_)) = t
  | getbf2type(IFbf2exp(t,_,_,_)) = t
  | getbf2type(ABSbf2exp(t,_,_,_)) = t
  | getbf2type(APPbf2exp(t,_,_)) = t
  | getbf2type(TABSbf2exp(t,_,_)) = t
  | getbf2type(TAPPbf2exp(t,_,_)) = t
  | getbf2type(CAPPbf2exp(t,_,_)) = t

(*******************************************************************)
(* Equality of types:                                              *)
(*******************************************************************)
fun equaltyvar (x:tyvar) (y:tyvar) = tyvar2string x=tyvar2string y

fun equaltypelist [] [] _ = true
  | equaltypelist (t::ts) (t'::ts') tveq =
    equaltype' t t' tveq andalso equaltypelist ts ts' tveq
  | equaltypelist _ _ _ = false
```

```
and equaltype' (VARf2type tv) (VARf2type tv') tveq =
    equaltyvar tv tv' orelse
    List.member (tyvar2string tv,tyvar2string tv') tveq
  | equaltype' (CONf2type(_,c,ts)) (CONf2type(_,c',ts')) tveq =
    c=c' andalso equaltypelist ts ts' tveq
  | equaltype' (FUNf2type(_,t,t')) (FUNf2type(_,t1,t1')) tveq =
      equaltype' t t1 tveq andalso equaltype' t' t1' tveq
  | equaltype' (FORALLf2type(_,tv,t)) (FORALLf2type(_,tv1,t1)) tveq =
      equaltype' t t1 ((tyvar2string tv,tyvar2string tv1)::tveq)
  | equaltype' _ _ _ = false

fun equaltype t t' = equaltype' t t' []

fun equalbtypelist [] [] _ = true
  | equalbtypelist (t::ts) (t'::ts') tveq =
    equalbtype' t t' tveq andalso equalbtypelist ts ts' tveq
  | equalbtypelist _ _ _ = false

and equalbtype' (VARbf2type tv) (VARbf2type tv') tveq =
    equaltyvar tv tv' orelse
    List.member (tyvar2string tv,tyvar2string tv') tveq
  | equalbtype' (CONbf2type(c,ts)) (CONbf2type(c',ts')) tveq =
    c=c' andalso equalbtypelist ts ts' tveq
  | equalbtype' (FUNbf2type(t,t')) (FUNbf2type(t1,t1')) tveq =
      equalbtype' t t1 tveq andalso equalbtype' t' t1' tveq
  | equalbtype' (FORALLbf2type(tv,t)) (FORALLbf2type(tv1,t1)) tveq =
      equalbtype' t t1 ((tyvar2string tv,tyvar2string tv1)::tveq)
  | equalbtype' (BOXbf2type t) (BOXbf2type t') tveq =
      equalbtype' t t' tveq
  | equalbtype' _ _ _ = false
fun equalbtype t t' = equalbtype' t t' []

(*******************************************************************)
(* Pretty printing:                                                *)
(*******************************************************************)
val printtypes = ref false
fun printtypesOn()  = printtypes := true
fun printtypesOff() = printtypes := false

fun layout_boxval B = "B"
  | layout_boxval U = "U"

fun getnodeid n = case n of
                    Node(id,_,_,_,_,_) => id
                  | _ => ~1 (* default value negative *)

fun empty_block(i,ch) =
    NODE{start="",finish="",indent=i,childsep=NONE,children=ch}

fun layout_typeass(c,tt) =
    NODE{start="",finish="",indent=0,childsep=RIGHT ":",children=[c,tt]}

and layout_def(x,tx,e) =
    NODE{start=x ^ ": ",finish="",indent=0,childsep=RIGHT " = ",
         children=[layout_f2type tx,layout_f2exp e]}

and layout_f2type(VARf2type tv) = LEAF (tyvar2string tv)
```

```
    | layout_f2type(CONf2type(bv,cn,ts)) =
      if ts=[] then
         LEAF (cn ^ "#" ^ Int.string (getnodeid bv))
      else
         NODE{start=cn ^ "#" ^ Int.string (getnodeid bv) ^ "(",
              finish=")",indent=2,
              childsep=RIGHT ",",children=map layout_f2type ts}
    | layout_f2type(FUNf2type(bv,t,t')) =
      NODE{start="(",indent=1,
           childsep=RIGHT (" ->#" ^ Int.string (getnodeid bv) ^ " "),
           children=[layout_f2type t,layout_f2type t'],finish=")"}
    | layout_f2type(FORALLf2type(bv,tv,t)) =
      NODE{start="(A ",indent=2,
           childsep=RIGHT (".#" ^ Int.string (getnodeid bv) ^ " "),
           children=[LEAF (tyvar2string tv),layout_f2type t],finish=")"}

  and addf2type(t,c) =
      if !printtypes then
         layout_typeass(c,layout_f2type t)
      else
         c

  and layout_f2exp(IDENTf2exp(t',x,_)) = addf2type(t',LEAF x)
    | layout_f2exp(PRIMf2exp(t',name,t,_,_)) = addf2type(t',LEAF name)
    | layout_f2exp(IFf2exp(t',e1,e2,e3)) =
        addf2type(t',empty_block
                    (0,[NODE{start="if ",finish="",
                             indent=3,childsep=LEFT " then ",
                             children=[layout_f2exp e1,layout_f2exp e2]},
                        NODE{start=" else ",finish="",
                             indent=3,childsep=NONE,
                             children=[layout_f2exp e3]}]))
    | layout_f2exp(ABSf2exp(t',x,t,e)) =
      addf2type(t',NODE{start="(fn ",finish=")",indent=4,childsep=RIGHT "=>",
                        children=[layout_typeass(LEAF x,layout_f2type t),
                                  layout_f2exp e]})
    | layout_f2exp(APPf2exp(t',e,e')) =
      (case e of
         ABSf2exp(t'',x,tx,e'') =>
           addf2type(t',NODE{start="let ",
                             finish=" end",
                             indent=4,
                             childsep=LEFT " in ",
                             children=[layout_def(x,tx,e'),
                                       layout_f2exp e'']})
        | _ =>
         addf2type(t',NODE{start="(",finish=")",indent=1,childsep=RIGHT " ",
                           children=[layout_f2exp e,layout_f2exp e']}))
    | layout_f2exp(TABSf2exp(t',tv,e)) =
      addf2type(t',NODE{start="(FN ",finish=")",indent=4,childsep=RIGHT "=>",
                        children=[LEAF (tyvar2string tv),layout_f2exp e]})
    | layout_f2exp(TAPPf2exp(t',e,t)) =
      addf2type(t',NODE{start="",finish="",indent=0,childsep=NONE,
                        children=[layout_f2exp e,
                                  NODE{start="{",finish="}",indent=1,
                                       childsep=NONE,
                                       children=[layout_f2type t]}]})
```

```
fun layout_bdef(x,tx,e) =
    NODE{start=x ^ ": ",finish="",indent=0,childsep=RIGHT " = ",
         children=[layout_bf2type tx,layout_bf2exp e]}

and layout_bf2type(VARbf2type tv) = LEAF (tyvar2string tv)
  | layout_bf2type(CONbf2type(cn,ts)) =
    if ts=[] then
       LEAF cn
    else
       NODE{start=cn ^ "(",finish=")",indent=2,
            childsep=RIGHT ",",children=map layout_bf2type ts}
  | layout_bf2type(FUNbf2type(t,t')) =
    NODE{start="(",indent=1,childsep=RIGHT "->",
         children=[layout_bf2type t,layout_bf2type t'],finish=")"}
  | layout_bf2type(FORALLbf2type(tv,t)) =
    NODE{start="(A ",indent=2,childsep=RIGHT ".",
         children=[LEAF (tyvar2string tv),layout_bf2type t],finish=")"}
  | layout_bf2type(BOXbf2type t) =
    NODE{start="[",indent=0,childsep=NONE,
         children=[layout_bf2type t],finish="]"}

and layout_coer(IDcoer t) =
    (if !printtypes then
        NODE{start="",finish=")",indent=1,childsep=RIGHT "(",
             children=[LEAF "i",layout_bf2type t]}
     else
        LEAF "i")
  | layout_coer(CONcoer(c,cs)) =
    NODE{start="_" ^ c ^ "_(",finish=")",indent=2,childsep=RIGHT ",",
         children=map layout_coer cs}
  | layout_coer(BOXcoer t) =
    (if !printtypes then
        NODE{start="",finish=")",indent=1,childsep=RIGHT "(",
             children=[LEAF "box",layout_bf2type t]}
     else
        LEAF "box")
  | layout_coer(UNBOXcoer t) =
    (if !printtypes then
        NODE{start="",finish=")",indent=1,childsep=RIGHT "(",
             children=[LEAF "unbox",layout_bf2type t]}
     else
        LEAF "unbox")
  | layout_coer(INBOXcoer c) =
    NODE{start="[",finish="]",indent=1,
         childsep=NONE,children=[layout_coer c]}
  | layout_coer(FUNcoer(c,c')) =
    NODE{start="(",finish=")",indent=1,childsep=RIGHT "->",
         children=[layout_coer c,layout_coer c']}
  | layout_coer(FORALLcoer(tv,c)) =
    NODE{start="(A",finish=")",indent=2,childsep=RIGHT ".",
         children=[LEAF (tyvar2string tv),layout_coer c]}
  | layout_coer(COMPcoer(c,c')) =
    NODE{start="(",finish=")",indent=1,childsep=RIGHT ";",
         children=[layout_coer c,layout_coer c']}

and addbf2type(t,c) =
        if !printtypes then
           layout_typeass(c,layout_bf2type t)
        else
           c

and layout_bf2exp(IDENTbf2exp(t',x)) = addbf2type(t',LEAF x)
  | layout_bf2exp(PRIMbf2exp(t',name,t,_,_)) = addbf2type(t',LEAF name)
  | layout_bf2exp(IFbf2exp(t',e1,e2,e3)) =
      addbf2type(t',empty_block
                   (0,[NODE{start="if ",finish="",
                            indent=3,childsep=LEFT " then ",
                            children=[layout_bf2exp e1,layout_bf2exp e2]},
                       NODE{start=" else ",finish="",
                            indent=3,childsep=NONE,
                            children=[layout_bf2exp e3]}]))
  | layout_bf2exp(ABSbf2exp(t',x,t,e)) =
    addbf2type(t',NODE{start="(fn ",finish=")",indent=4,childsep=RIGHT "=>",
                       children=[layout_typeass(LEAF x,layout_bf2type t),
                                 layout_bf2exp e]})
  | layout_bf2exp(APPbf2exp(t',e,e')) =
    (case e of
       ABSbf2exp(t'',x,tx,e'') =>
         addbf2type(t',NODE{start="let ",
                            finish=" end",
                            indent=4,
                            childsep=LEFT " in ",
                            children=[layout_bdef(x,tx,e'),
                                      layout_bf2exp e'']})
     | _ =>
      addbf2type(t',NODE{start="(",finish=")",indent=1,childsep=RIGHT " ",
                         children=[layout_bf2exp e,layout_bf2exp e']}))
  | layout_bf2exp(TABSbf2exp(t',tv,e)) =
    addbf2type(t',NODE{start="(FN ",finish=")",indent=4,childsep=RIGHT "=>",
                       children=[LEAF (tyvar2string tv),layout_bf2exp e]})
  | layout_bf2exp(TAPPbf2exp(t',e,t)) =
    addbf2type(t',NODE{start="(",finish="}",indent=0,childsep=RIGHT "){",
                       children=[layout_bf2exp e,layout_bf2type t]})
  | layout_bf2exp(CAPPbf2exp(t',c,e)) =
    addbf2type(t',NODE{start="<",finish="",indent=1,childsep=RIGHT ">",
                       children=[layout_coer c,layout_bf2exp e]})

(************************************************************)
(* Translation into SML                                     *)
(************************************************************)
fun ctosig(IDcoer t) = (t,t)
  | ctosig(CONcoer(tc,cs)) =
    let val sigs = map ctosig cs in
        (CONbf2type(tc,map #1 sigs),CONbf2type(tc,map #2 sigs))
    end
  | ctosig(BOXcoer t) = (t,BOXbf2type t)
  | ctosig(UNBOXcoer t) = (BOXbf2type t,t)
  | ctosig(INBOXcoer c) =
    let val (t,t') = ctosig c in
        (BOXbf2type t,BOXbf2type t')
    end
  | ctosig(FUNcoer(c,d)) =
    let val (tc,tc') = ctosig c
```

```
        val (td,td') = ctosig d
      in
        (FUNbf2type(tc',td),FUNbf2type(tc,td'))
      end
    | ctosig(FORALLcoer(tv,c)) =
      let val (t,t') = ctosig c in
          (FORALLbf2type(tv,t),FORALLbf2type(tv,t'))
      end
    | ctosig(COMPcoer(c,d)) = (#1(ctosig c),#2(ctosig d))

  fun toptypecon(VARbf2type _) = "tyvar"
    | toptypecon(CONbf2type(tc,_)) = tc
    | toptypecon(FUNbf2type _) = "fun"
    | toptypecon(FORALLbf2type _) = "all" (* should not occur *)
    | toptypecon(BOXbf2type _) = "box"    (* should not occur *)

  fun AppCoer(IDcoer _,e) = e
    | AppCoer(CONcoer(tc,cs),e) =
        NODE{start="(" ^ "(insstat \"make:" ^ tc ^ "\";" ^ tc,
             finish="))",indent=1,childsep=RIGHT " ",
             children=(map (fn c =>
                             NODE{start="(fn x => (insstat \"access:" ^ tc ^ "\";",
                                  finish="))",
                                  indent=1,
                                  childsep=NONE,
                                  children=[AppCoer(c,LEAF "x")]})
                        cs)
                       @ [e]}
    | AppCoer(BOXcoer (FORALLbf2type _),e) = e
    | AppCoer(BOXcoer t,e) =
        NODE{start="(box(\"" ^ toptypecon t ^ "\",",
             finish="))",indent=6,childsep=RIGHT " ",
             children=[e]}
    | AppCoer(UNBOXcoer (FORALLbf2type _),e) = e
    | AppCoer(UNBOXcoer t,e) =
        NODE{start="(unbox(\"" ^ toptypecon t ^ "\",",
             finish="))",indent=1,childsep=RIGHT " ",
             children=[e]}
    | AppCoer(INBOXcoer c,e) =
        let val (t,t') = ctosig c
        in
            AppCoer(BOXcoer t',AppCoer(c,AppCoer(UNBOXcoer t,e)))
        end
    | AppCoer(FUNcoer(c,d),e) =
        NODE{start="(",finish=")",indent=1,childsep=RIGHT " ",
             children=
               [NODE{start="(fn f => (insstat \"cl\";" ^
                            "(fn x => (insstat \"app\";(",
                     finish=")))))",indent=1,
                     childsep=NONE,
                     children=
                       [AppCoer(d,NODE{start="(f ",
                                       finish=")",
                                       indent=1,
                                       childsep=RIGHT " ",
                                       children=[AppCoer(c,LEAF "x")]})]},
                e]}
  | AppCoer(FORALLcoer(_,c),e) = AppCoer(c,e)
  | AppCoer(COMPcoer(c,d),e) = AppCoer(d,AppCoer(c,e))

fun bf2exp2SML(IDENTbf2exp(_,x)) = LEAF x
  | bf2exp2SML(PRIMbf2exp(_,name,_,_,_)) = LEAF name
  | bf2exp2SML(IFbf2exp(t,e1,e2,e3)) =
    empty_block(0,[NODE{start="if ",finish="",
                        indent=3,childsep=LEFT " then ",
                        children=[bf2exp2SML e1,bf2exp2SML e2]},
                   NODE{start=" else ",finish="",
                        indent=3,childsep=NONE,
                        children=[bf2exp2SML e3]}])
  | bf2exp2SML(ABSbf2exp(_,x,_,e)) =
    NODE{start="(fn ",finish=")",indent=4,childsep=RIGHT "=>",
         children=[LEAF x,bf2exp2SML e]}
  | bf2exp2SML(APPbf2exp(t,e,e')) =
    (case e of
       ABSbf2exp(_,x,_,e'') =>
         NODE{start="let val ",
              finish=" end",
              indent=8,
              childsep=LEFT " in ",
              children=[NODE{start="",finish="",
                             indent=0,childsep=RIGHT " = ",
                             children=[LEAF x,bf2exp2SML e']},
                        bf2exp2SML e'']}
     | _ =>
      NODE{start="(",finish=")",indent=1,childsep=RIGHT " ",
           children=[bf2exp2SML e,bf2exp2SML e']})
  | bf2exp2SML(TABSbf2exp(_,_,e)) = bf2exp2SML e
  | bf2exp2SML(TAPPbf2exp(_,e,_)) = bf2exp2SML e
  | bf2exp2SML(CAPPbf2exp(_,c,e)) = AppCoer(c,bf2exp2SML e)

val screen = (print:string->unit)

fun pp_SML(device,e,com) =
    (device ("(* " ^ com ^ " *)\n");
     device "exception hderror;\n";
     device "exception tlerror;\n";
     device "let val stat = ref([]:(string * int) list)\n";
     device "    fun insstat tc =\n";
     device "        let fun f []          = [(tc,1)]\n";
     device "              | f ((tc',n)::s) =\n";
     device "                  if tc=tc' then\n";
     device "                     (tc,n+1)::s\n";
     device "                  else\n";
     device "                     (tc',n)::f s\n";
     device "        in\n";
     device "            stat := f (!stat)\n";
     device "        end\n";
     device "    fun fix f x = f (fix f) x\n";
     device "    fun box(tc,x) = (insstat (\"box:\" ^ tc); x)\n";
     device "    fun unbox(tc,x) = (insstat (\"unbox:\" ^ tc); x)\n";
     device "    fun mkpair x y = (x,y)\n";
     device "    fun fst (x,y) = x\n";
     device "    fun snd (x,y) = y\n";
     device "    fun cons x y = x :: y\n";
```

```
        device "    fun hd (x::_) = x\n";
        device "      | hd xs     = raise hderror\n";
        device "    fun hd (x::_) = x\n";
        device "      | hd xs     = raise hderror\n";
        device "    fun tl (_::t) = t\n";
        device "      | tl xs     = raise tlerror\n";
        device "    fun null []   = true\n";
        device "      | null _    = false\n";
        device "    fun plus   (x:int) y = x + y\n";
        device "    fun sub    (x:int) y = x - y\n";
        device "    fun mult   (x:int) y = x * y\n";
        device "    fun gt     (x:int) y = x > y\n";
        device "    fun eq     (x:int) y = x = y\n";
        device "    fun noteq  (x:int) y = x <> y\n";
        device "    fun modulo (x:int) y = x mod y\n";
        device "    fun print x = x\n";
        device "    (* coercion constructors *)\n";
        device "    val list = map\n";
        device "    fun pair c d x = mkpair (c (fst x)) (d (snd x))\n";
        device "in\n(";
        outputTree 200 device (bf2exp2SML e);
        device ",!stat)\nend")

    fun pp_source(device,e) = outputTree 80 device (layout_f2exp e)
    fun pp_target(device,e) = outputTree 80 device (layout_bf2exp e)
    fun pp_type(device,t)   = outputTree 80 device (layout_f2type t)
    fun pp_btype(device,t)  = outputTree 80 device (layout_bf2type t)

  end;
```

B.3 Signature: PARSE

The next file shown below contains the signature PARSE of the parser functor Parse.

```
signature PARSER =
  sig
    structure Expression : EXPRESSION
    open Expression

    datatype 'a res = Some of 'a | None

    val parse : string -> ((id * f2type) list * unit f2exp) res
  end;
```

B.4 Functor: Parse

The file below contains the functor Parse. This is a hand-written top-down recursive decent parser. The important function in this is parse which parses a string containing a program and returns either None if parse error have occured or Some of a pair of a type assignment and a `f2exp` data structure containing the parsed F_2 expression. The type assignment contains the free variables (primitives) of the F_2 expression and the types of these. These free variables must be declared in the source file with an extern declaration before the F_2 expression according to the syntax shown in the comment first in the file.

```
(*
   Syntax:

     Programs:

       p ::= decls e

     Declarations:

       decls ::=
              |  extern x : t ; decls

     Expressions:

       e ::= n                       (integers (of type int()))
          |  x                       (variables)
          |  fn x:t => e             (abstractions)
          |  e e                     (applications)
          |  Fn a => e               (type abstractions)
          |  e{t}                    (type application/instantiation)
          |  x*{ts}                  (in-line primitives of type t)
          |  let x:t = e in e end    (let-expression)
          |  if e then e else e      (conditional)
          |  (e)                     (parenthesized expression)

     Types:

       t ::= a                       (type variable)
          |  t -> t                  (function type)
          |  A a.t                   (type quantification)
          |  c(ts)                   (type constructor application)

       ts ::=                        (empty sequence of types)
           |  t ts                   (sequence of types)

*)
functor Parser(structure Expression : EXPRESSION) : PARSER =
  struct
    structure Expression = Expression
    open Expression

    datatype 'a res = Some of 'a | None

    exception TokenError

    val input_list = ref ([]:string list)
    val next_char = ref ""
    fun read_char() =
        (case !input_list of
           [] => next_char := ""
         | (c::ip) => (next_char:=c; input_list:= ip))
```

```
datatype Token =
    EndOfInput
  | ColonSym     (* : *)
  | SemiColonSym (* ; *)
  | AbsSym       (* fn *)
  | FatArrowSym  (* => *)
  | TypeAbsSym   (* Fn *)
  | CLeftParen   (* { *)
  | CRightParen  (* } *)
  | LeftParen    (* ( *)
  | RightParen   (* ) *)
  | ArrowSym     (* -> *)
  | ForallSym    (* A *)
  | DotSym       (* . *)
  | LetSym       (* let *)
  | InSym        (* in  *)
  | EndSym       (* end *)
  | ExtSym       (* extern *)
  | IfSym        (* if *)
  | ThenSym      (* then *)
  | ElseSym      (* else *)
  | EqSym        (* = *)
  | StarSym      (* * *)
  | StartComSym
  | EndComSym
  | Number of string
  | Identifier of string
  | UnknownChar of string

fun pp_token t =
    (case t of
       EndOfInput => "EOF"
     | ColonSym   => ":"
     | SemiColonSym => ";"
     | AbsSym     => "fn"
     | FatArrowSym  => "=>"
     | TypeAbsSym   => "Fn"
     | CLeftParen   => "{"
     | CRightParen  => "}"
     | LeftParen    => "("
     | RightParen   => ")"
     | ArrowSym     => "->"
     | ForallSym    => "A"
     | DotSym       => "."
     | LetSym       => "let"
     | InSym        => "in"
     | EndSym       => "end"
     | ExtSym       => "ext"
     | IfSym        => "if"
     | ThenSym      => "then"
     | ElseSym      => "else"
     | EqSym        => "="
     | StarSym      => "*"
     | StartComSym  => "(*"
     | EndComSym    => "*)"
     | Number  n    => n
```

```
      | Identifier s => s
      | UnknownChar s => s)

   fun keyword("fn")     = Some AbsSym
     | keyword("Fn")     = Some TypeAbsSym
     | keyword("A")      = Some ForallSym
     | keyword("let")    = Some LetSym
     | keyword("in")     = Some InSym
     | keyword("end")    = Some EndSym
     | keyword("if")     = Some IfSym
     | keyword("then")   = Some ThenSym
     | keyword("else")   = Some ElseSym
     | keyword("extern") = Some ExtSym
     | keyword _         = None

   val idfstsymbols =
       explode "abcdefghijklmnopqrstuvwxyzABDXEFGHIJKLMNOPQRSTUVWXYZ_'"
   val digits = explode "1234567890"
   val idsymbols = idfstsymbols @ digits

   fun read_identifier() =
       let val c = !next_char
       in
          if List.member c idsymbols then
             (read_char();
              c ^ read_identifier())
          else
             ""
       end

   fun read_number() =
       let val c = !next_char
       in
          if List.member c digits then
             (read_char();
              c ^ read_number())
          else
             ""
       end

   fun lex() =
       (case !next_char of
          "" =>   EndOfInput
        | " " =>  (read_char(); lex())
        | "\t" => (read_char(); lex())
        | "\n" => (read_char(); lex())
        | ":" =>  (read_char();ColonSym)
        | ";" =>  (read_char();SemiColonSym)
        | "=" =>  (read_char();
                   case !next_char of
                     ">" => (read_char();FatArrowSym)
                   | _ => EqSym)
        | "(" =>  (read_char();
                   case !next_char of
                     "*" => (read_char();StartComSym)
                   | _ => LeftParen)
        | ")" =>  (read_char();RightParen)
      | "{" =>  (read_char();CLeftParen)
      | "}" =>  (read_char();CRightParen)
      | "." =>  (read_char();DotSym)
      | "*" =>  (read_char();
                 case !next_char of
                   ")" => (read_char();EndComSym)
                 | _ => StarSym)
      | "-" =>
        (read_char();
          case !next_char of
            ">" => (read_char();ArrowSym)
          | c => (read_char();UnknownChar c))
      | c =>
        if List.member c idfstsymbols then
           let val s = read_identifier()
           in
               case keyword s of
                 Some token => token
               | None => Identifier s
           end
        else (if List.member c digits then
                 let val n = read_number()
                 in
                    Number n
                 end
              else
                 (read_char();UnknownChar c)))

exception ParseError

val next_token = ref EndOfInput

fun skipComment level =
    let val nt = lex()
    in
       case nt of
         StartComSym => skipComment(level+1)
       | EndComSym =>
         if level = 1 then read_token() else skipComment(level-1)
       | _ => skipComment(level)
    end

and read_token() =
    let val nt = lex()
    in
       case nt of
         StartComSym => skipComment(1)
       | UnknownChar c =>
         (print ("Parse error: unknown character: " ^ c ^
                 " found immediately before: \n\n" ^
                 (implode (!input_list)));
          raise TokenError)
       | _ => next_token := nt
    end

fun expect(tok) =
    if tok = !next_token then
```

```
        read_token()
     else
        (print ("Parse error: token " ^ pp_token tok ^ " expected but " ^
                pp_token (!next_token) ^ " found immediately before: \n\n" ^
                (!next_char) ^ implode (!input_list));
         raise ParseError)

val gte = ref([]:(id * f2type) list) (* global type assignment *)
fun lookup_in_te(i,(i',t)::te) = if i=i' then t else lookup_in_te(i,te)
  | lookup_in_te(i,_) =
    (print("Parse error: unknown variable or primitive: " ^ id2string i ^ "\n\n");
     raise ParseError)
val init_tv_env = fn i => mktyvar i
fun lookup(i,r) = r i
fun update(i,tv,r) = fn i' => if i=i' then tv else r i'

fun mkApp []  = (print "Parse error: error in mkApp\n\n";raise ParseError)
  | mkApp [e] = e
  | mkApp (e::e'::es) = mkApp ((APPf2exp((),e,e'))::es)

fun parse_ident() =
    (case !next_token of
       Identifier s => (read_token(); s)
     | _ =>
       (print ("Parse error: identifier expected immediately before: \n\n" ^
               (!next_char) ^ implode (!input_list));
             raise ParseError))

fun parse_type(r) =
  (case !next_token of
     ForallSym =>
       (read_token();
        let val i = parse_ident()
            val tv = mktyvar i
        in
           expect(DotSym);
           FORALLf2type(Nil,tv,parse_type(update(i,tv,r)))
        end)
   | Identifier s =>
       let val i = parse_ident()
           val t =
             case !next_token of
               LeftParen =>
                 (read_token();
                  let val t = CONf2type(Nil,mktc i,parse_type_list(r))
                  in
                     expect(RightParen);
                     t
                  end)
             | _ => let val tv = lookup(i,r)
                    in
                       VARf2type tv
                    end
       in
           case !next_token of
             ArrowSym =>
               (read_token();
                FUNf2type(Nil,t,parse_type(r)))
            | _ => t
        end
    | LeftParen =>
        (read_token();
         let val t = parse_type(r)
         in
            expect(RightParen);
            case !next_token of
              ArrowSym =>
                (read_token();
                 FUNf2type(Nil,t,parse_type(r)))
            | _ => t
         end)
    | _ => (print ("Parse error: syntax error in type immediately before: \n\n" ^
                   (!next_char) ^ implode (!input_list));
            raise ParseError))

and parse_type_list(r) =
    (case !next_token of
       ForallSym => parse_type(r)::parse_type_list(r)
     | Identifier s => parse_type(r)::parse_type_list(r)
     | LeftParen => parse_type(r)::parse_type_list(r)
     | _ => [])

fun parse_appexp(r) = mkApp (parse_appexp'(r))

and parse_appexp'(r) =
    (case !next_token of
       LeftParen =>
         (read_token();
          let val e = parse_exp(r)
          in
             expect(RightParen);
             parse_appexp''(e,r)
          end)
     | Identifier s =>
         let val i = mkid(parse_ident())
         in
            case !next_token of
               StarSym =>
                 (read_token();
                  let val ts = case !next_token of
                                  CLeftParen =>
                                    (read_token();
                                     let val ts = parse_type_list(r)
                                     in
                                        expect(CRightParen);
                                        ts
                                     end)
                                | _ => []
                  in
                     parse_appexp''(PRIMf2exp((),i,lookup_in_te(i,(!gte)),ts,0),r)
                  end)
             | _ => parse_appexp''(IDENTf2exp((),i,()),r)
         end
     | Number s =>
```

```
          (read_token();
           parse_appexp''(PRIMf2exp((),mkid s,
                                   CONf2type(Nil,mktc "int",[]),[],P),r))
      | _ => [])

  and parse_appexp''(e,r) =
      (case !next_token of
         CLeftParen =>
          (read_token();
           let val t = parse_type(r)
           in
              expect(CRightParen);
              (TAPPf2exp((),e,t))::parse_appexp'(r)
           end)
       | _ => e::parse_appexp'(r))

  and parse_exp(r) =
      (case !next_token of
         AbsSym =>
          (read_token();
           let val i = mkid(parse_ident())
           in
              expect(ColonSym);
              let val t = parse_type(r)
              in
                 expect(FatArrowSym);
                 ABSf2exp((),i,t,parse_exp(r))
              end
           end)
       | TypeAbsSym =>
          (read_token();
           let val i = parse_ident()
               val tv = mktyvar i
           in
              expect(FatArrowSym);
              TABSf2exp((),tv,parse_exp(update(i,tv,r)))
           end)
       | LetSym =>
          (read_token();
           let val i = mkid(parse_ident())
               val _ = expect(ColonSym)
               val t = parse_type(r)
               val _ = expect(EqSym)
               val e1 = parse_exp(r)
               val _ = expect(InSym)
               val e2 = parse_exp(r)
           in
               expect(EndSym);
               APPf2exp((),ABSf2exp((),i,t,e2),e1)
           end)
       | IfSym =>
          (read_token();
           let val e1 = parse_exp(r)
               val _  = expect(ThenSym)
               val e2 = parse_exp(r)
               val _  = expect(ElseSym)
               val e3 = parse_exp(r)
              in
                  IFf2exp((),e1,e2,e3)
              end)
         | LeftParen => parse_appexp(r)
         | Identifier s => parse_appexp(r)
         | Number s => parse_appexp(r)
         | _ =>
           (print ("Parse error: syntax error in expression immediately before: \n\n" ^
                       (!next_char) ^ implode (!input_list));
                raise ParseError))

    fun parse_prog te =
        (case !next_token of
           ExtSym =>
             (read_token();
              let val i = mkid(parse_ident())
                  val _ = expect(ColonSym)
                  val t = parse_type(init_tv_env)
                  val _ = expect(SemiColonSym)
              in
                   parse_prog ((i,t)::te)
              end)
         | _ => (gte := te;(te,parse_exp(init_tv_env))))

    fun parse s =
        (input_list := explode s;
         read_char();
         read_token();
         let val e = parse_prog([])
         in
            expect(EndOfInput);
            Some e
         end)
        handle ParseError =>
          (print "\n\nParse Error(s) detected\n\n";
           None)
  end;
```

B.5 Signature: BOXINGANALYSIS

The next file contains the signature BOXINGANALYSIS of the functor BoxingAnalysis which contains the actual analysis.

```
signature BOXINGANALYSIS =
  sig
    structure Expression : EXPRESSION
    open Expression

    val isources : node list ref
    val edges    : (node * node) list ref
    val sources  : node list ref
    val sinks    : node list ref
    val debug    : bool ref
    val debugOn  : unit -> unit
    val debugOff : unit -> unit
    val cpstar    : bool ref
    val cpstarOn  : unit -> unit
    val cpstarOff : unit -> unit

    val nodecount : int ref
    val lastnode  : node ref

    val print_edges : (node * node) list -> unit
    val print_nodes : node list -> unit
    val debug_print_edges_sources_and_nodes: unit -> unit
    val debug_print_sol : unit -> unit

    val txt : string ref

    val f2exp2bf2exp : f2type f2exp -> f2type-> bf2type bf2exp

    exception Typecheck

    val setpsi : unit -> unit
    val setphi : unit -> unit

    val init : unit -> unit
    val generate : (id * f2type) list * 'a f2exp -> f2type f2exp * f2type

    val propagate : node list -> unit
  end;
```

B.6 Functor: BoxingAnalysis

The functor `BoxingAnalysis`, shown below, contains the actual implementation of the algorithm presented in Chapter 7. It contains the following important functions:

- `generate` that calls `typecheck` to type check the source expression and generate the reprsentation graph. It also add sources and sinks for the type of the expression and for the types of the free variables in the type assignment.
- `propagate` that performs the solving. This corresponds to the code shown in Figure 21.
- `init` that initialized the internal data structures use by the program.
- `f2exp2bf2exp` that converts the annotated F_2 expressions (principal completions) into optimal completions when the analysis has been performed.

It also contains several function for use when debugging and the following function that may be used to change the behaviour of the program:

- `setpsi` set the algorithm to produce ψ-free completions.
- `setphi` set the algorithm to produce ϕ-free completions.

- `debugOn` switches the debug mode on.
- `debugOff` switches the debug mode off.

```
functor BoxingAnalysis(structure Expression : EXPRESSION) : BOXINGANALYSIS =
  struct
    structure Expression = Expression;
    open Expression

    exception ThisCannotHappen (* for situation I know for sure cannot happen *)
    exception ThisShouldNotHappen (* for situation that should not happen *)

    (*******************************************************************)
    (* Some elementary functions:                                      *)
    (*******************************************************************)
    fun zip [] []  = []
      | zip(x::xs) (y::ys) = (x,y)::zip xs ys
      | zip _ _ = raise ThisShouldNotHappen

    fun unzip(ps:('a * 'b) list) = (map #1 ps,map #2 ps)

    fun concat [] = []
      | concat (xs::xss) = xs @ (concat xss)

    (*******************************************************************)
    (* Options:                                                        *)
    (*******************************************************************)
    val debug         = ref false
    fun debugOn()     = debug := true
    fun debugOff()    = debug := false

    val cpstar        = ref false
    fun cpstarOn()    = cpstar := true
    fun cpstarOff()   = cpstar := false
    (*******************************************************************)
    (* Sources and sinks:                                              *)
    (*******************************************************************)
    val isources = ref ([]:node list)
    val edges    = ref ([]:(node * node) list) (* for debug *)
    val sources  = ref ([]:node list)
    val sinks    = ref ([]:node list)

    (*******************************************************************)
    (* Function on nodes :                                             *)
    (*******************************************************************)
    fun find n =
        (case n of
           Node(_,_,ecrp,_,_,_,_) =>
             (if !ecrp = n then
                 n
              else
                 let val n' = find(!ecrp)
                 in
                    ecrp := n';
                    n'
                 end)
```

```
     | _ => (print "internal reference 10";
             raise ThisCannotHappen))

fun lub n1 n2 =
    let val n1 = find n1
        val n2 = find n2
    in
       case (n1,n2) of
         (Node(_,_,_,ref B,_,_,_),_) => n1
       | (_,Node(_,_,_,ref B,_,_,_)) => n2
       | (Node(_,_,_,_,_,_,ref V),_) => n1
       | _ => n2
    end

fun glb n1 n2 =
    let val n1 = find n1
        val n2 = find n2
    in
       case (n1,n2) of
         (Node(_,_,_,ref U,_,_,_),_) => n1
       | (_,Node(_,_,_,ref U,_,_,_)) => n2
       | (Node(_,_,_,_,_,_,ref V),_) => n1
       | _ => n2
    end

fun mark m n = case n of
                 Node(_,_,_,_,_,_,rm) => rm:=m
               | _ => ()

fun setboxval n b = case n of
                      Node(_,_,_,br,_,_,_) => br:=b
                    | _ => ()

fun getboxval n = case find n of
                    Node(_,_,_,b,_,_,_) => !b
                  | Nil => (print "internal reference 1";
                            raise ThisCannotHappen)

fun getnode(CONf2type(bv,_,_))    = bv
  | getnode(FUNf2type(bv,_,_))    = bv
  | getnode(FORALLf2type(bv,_,_)) = bv
  | getnode _ = Nil

(********************************************************************)
(* Default values :                                                 *)
(********************************************************************)
val dbv = ref U            (* the default box value *)
val sbv = ref B            (* the box value of the initial sources *)
val dop = ref lub          (* the default operation used by findB *)
val txt = ref "Psi-free mode"

fun set(ndbv,nsbv,ndop,ntxt) =
    (dbv:=ndbv; sbv:=nsbv; dop:=ndop; txt:=ntxt)
fun setpsi() = (set(U,B,lub,"Psi-free mode");print(!txt))
fun setphi() = (set(B,U,glb,"Phi-free mode");print(!txt))

(********************************************************************)
```

```
(* Generation of nodes :                                              *)
(*********************************************************************)
(* Generation of numbers *)
val nodecount = ref 0
fun get_new_no() = let val n = !nodecount in nodecount:= n + 1; n end

(* Generation of nodes *)
val lastnode   = ref Nil (* for debugging: keeps all nodes
                            in one linked list *)
fun new_node() =
    let val n' = ref Nil
        val n = Node(mknodeid(get_new_no()),ref (!lastnode),n',ref(!dbv),
                     ref [],ref [],ref NV)
in
    n' := n;
    lastnode := n;
    n
end

(*********************************************************************)
(* Debug functions:                                                   *)
(*********************************************************************)
fun print_edges eds =
    (map (fn (n,m) => print (Int.string (getnodeid n) ^ "=>" ^
                            Int.string (getnodeid m) ^ ","))
         eds;
     ())

fun print_nodes ns =
    (map (fn n =>
            print (Int.string (getnodeid n) ^ "=" ^
                   layout_boxval (getboxval n) ^ ","))
          ns;
     ())

fun debug_print_edges_sources_and_nodes() =
    (print "\nEdges:\n\n";
     print_edges (!edges);
     print "\n\nSources:\n\n";
     print_nodes (!sources);
     print "\n\nISources:\n\n";
     print_nodes (!isources);
     print "\n\nSinks:\n\n";
     print_nodes (!sinks))

fun debug_print_sol' n =
    case !n of
      Node(id,next,_,_,_,_,_) =>
        let val Node(id',_,_,_,_,_,_) = find(!n)
        in
           (print (nodeid2int id); print "=(";
            print (nodeid2int id'); print ")";
            print (layout_boxval (getboxval(!n))); print ",";
            debug_print_sol' next)
        end
    | NIL => ()
fun debug_print_sol() = debug_print_sol' lastnode
```

```
(********************************************************************)
(* Graph generation:                                                *)
(********************************************************************)
fun assym_union(b,b') = (* make the first one the ecr *)
    let val Node(_,_,ecrp,_,succ,pred,_) = b
        val Node(_,_,ecrp',_,succ',pred',_) = b'
    in
       (* print (Int.string(getnodeid b') ^ "->" ^
              Int.string(getnodeid b) ^ ","); *)
       ecrp' := b;
       succ :=
         List.dropFirst (fn x => x = b') (!succ) @ (* We don't know *)
         List.dropFirst (fn x => x = b) (!succ');  (* which comes   *)
       pred :=                                     (* first!        *)
         List.dropFirst (fn x => x = b') (!pred) @
         List.dropFirst (fn x => x = b) (!pred')
    end

fun union(b,b') =
    let val Node(_,_,_,ref bv,_,_,ref m) = find b
        val Node(_,_,_,ref bv',_,_,ref m') = find b'
    in
        case (m,m') of
          (VU,VU) =>
            (if bv = !sbv then
                assym_union(b,b')
             else
                assym_union(b',b))
        | (VU,_) => assym_union(b,b')
        | (_,VU) => assym_union(b',b)
        | (V,_) => assym_union(b,b')
        | _ => assym_union(b',b)
    end

fun c' f fr (CONf2type(n,_,ts),CONf2type(n',_,ts')) =
    let val n = find n
        val n' = find n'
    in
       (case (n,n') of
          (Node(_,_,_,_,succ,_,_),Node(_,_,_,_,_,pred,_)) =>
             (succ := n' :: (!succ);
              pred := n  :: (!pred);
              edges := (n,n') :: !edges;
              (map (c' fr fr) (zip ts ts'));
              ())
        | _ => ());
       f(n,n')
    end
  | c' f fr (FUNf2type(n,t,t1),FUNf2type(n',t',t1')) =
    let val n = find n
        val n' = find n'
    in
       (case (n,n') of
          (Node(_,_,_,_,succ,_,_),Node(_,_,_,_,_,pred,_)) =>
             (succ := n' :: (!succ);
              pred := n  :: (!pred);
             edges := (n,n') :: !edges;
             c' fr fr (t',t);
             c' fr fr (t1,t1'))
         | _ => ());
        f(n,n')
    end
  | c' f fr (FORALLf2type(n,_,t),FORALLf2type(n',_,t')) =
    let val n = find n
        val n' = find n'
    in
       (case (n,n') of
          (Node(_,_,_,_,succ,_,_),Node(_,_,_,_,_,pred,_)) =>
             (succ := n' :: (!succ);
              pred := n  :: (!pred);
              edges := (n,n') :: !edges;
              c' fr fr (t,t'))
        | _ => ());
       f(n,n')
    end
  | c' _ _ (_,_) = ()
fun c(t,t') =
    c' (fn _ => ()) (if !cpstar then union else (fn _ => ())) (t,t')
fun conecttypes(t,t') = c' (fn _ => ()) (fn _ => ()) (t,t')
fun uniontypes(t,t') = c' union union (t,t')

fun ioSourceAndSinks(CONf2type(bv,_,ts)) =
    let val (so,si) = let val (x,y) = unzip(map ioSourceAndSinks ts)
                      in
                         (concat x,concat y)
                      end
    in
       (bv::so,si)
    end
  | ioSourceAndSinks(FUNf2type(bv,t,t')) =
    let val (si,so) = ioSourceAndSinks t
        val (so',si') = ioSourceAndSinks t'
    in
        (bv :: so @ so', si @ si')
    end
  | ioSourceAndSinks(FORALLf2type(bv,_,t)) =
    let val (so,si) = ioSourceAndSinks t
    in
        (bv :: so, si)
    end
  | ioSourceAndSinks _ = ([],[])

fun addtosources(n,b) =
    let val n = find n
    in
       (mark VU n; setboxval n b; sources := n :: !sources;
        if b = !sbv then
           isources := n :: !isources
        else
           ())
    end

fun addtosinks(n,b) =
    let val n = find n
    in
       (mark VU n; setboxval n b; sinks := n :: !sinks)
    end

(********************************************************************)
(* Substitution and instantiation:                                  *)
(********************************************************************)
val tvcount = ref 0
fun get_new_tv(tv) =
      let val n = !tvcount
      in
         tvcount:= n + 1;
         tyvar2string tv ^ "$" ^ Int.string n
      end

(* find free type variable in f2 types *)
fun ftvf2type(VARf2type tv) = Set.singleton tv
  | ftvf2type(FUNf2type(_,t1,t2)) =
      Set.union equaltyvar (ftvf2type t1) (ftvf2type t2)
  | ftvf2type(FORALLf2type(_,tv,t)) =
      Set.remove equaltyvar tv (ftvf2type t)
  | ftvf2type _ = Set.empty

(* substitution on f2 types *)
fun subst(tv,t,VARf2type tv') =
      if equaltyvar tv tv' then t else VARf2type tv'
  | subst(tv,t,CONf2type(bv,c,ts)) =
      CONf2type(bv,c,map (fn t' => subst(tv,t,t')) ts)
  | subst(tv,t,FUNf2type(bv,t1,t2)) =
      FUNf2type(bv,subst(tv,t,t1),subst(tv,t,t2))
  | subst(tv,t,FORALLf2type(bv,tv',t')) =
      if equaltyvar tv tv' then
         FORALLf2type(bv,tv',t')
      else
         (if Set.member equaltyvar tv' (ftvf2type t) then
             (changetyvar(tv',get_new_tv(tv'));
              FORALLf2type(bv,tv',subst(tv,t,t')))
          else
             FORALLf2type(bv,tv',subst(tv,t,t')))

exception Instantiate
fun instantiate(FORALLf2type(_,tv,t),t1::ts) =
    instantiate(subst(tv,t1,t),ts)
  | instantiate(t,[]) = t
  | instantiate(_,_) = raise Instantiate

(* find free type variable in representation types *)
fun ftvbf2type(VARbf2type tv) = Set.singleton tv
  | ftvbf2type(FUNbf2type(t1,t2)) =
      Set.union equaltyvar (ftvbf2type t1) (ftvbf2type t2)
  | ftvbf2type(FORALLbf2type(tv,t)) =
      Set.remove equaltyvar tv (ftvbf2type t)
  | ftvbf2type(BOXbf2type t) =
      ftvbf2type t
  | ftvbf2type _ = Set.empty
```

```
(* substitution on representation types *)
fun substb(tv,t,VARbf2type tv') =
      if equaltyvar tv tv' then t else VARbf2type tv'
  | substb(tv,t,CONbf2type(c,ts)) =
      CONbf2type(c,map (fn t' => substb(tv,t,t')) ts)
  | substb(tv,t,FUNbf2type(t1,t2)) =
      FUNbf2type(substb(tv,t,t1),substb(tv,t,t2))
  | substb(tv,t,FORALLbf2type(tv',t')) =
      if equaltyvar tv tv' then
         FORALLbf2type(tv',t')
      else
         (if Set.member equaltyvar tv' (ftvbf2type t) then
             (changetyvar(tv',get_new_tv(tv'));
              FORALLbf2type(tv',substb(tv,t,t')))
          else
             FORALLbf2type(tv',substb(tv,t,t')))
  | substb(tv,t,BOXbf2type t') =
      BOXbf2type(substb(tv,t,t'))

(* substitute on coercions *)
fun substoncoer(tv,t',IDcoer t) = IDcoer(substb(tv,t',t))
  | substoncoer(tv,t',CONcoer(tc,cs)) =
      CONcoer(tc,map (fn c => substoncoer(tv,t',c)) cs)
  | substoncoer(tv,t',BOXcoer t) = BOXcoer(substb(tv,t',t))
  | substoncoer(tv,t',UNBOXcoer t) = UNBOXcoer(substb(tv,t',t))
  | substoncoer(tv,t',INBOXcoer c) = INBOXcoer(substoncoer(tv,t',c))
  | substoncoer(tv,t',FUNcoer(c,d)) =
      FUNcoer(substoncoer(tv,t',c),substoncoer(tv,t',d))
  | substoncoer(tv,t',FORALLcoer(tv',c)) =
      if equaltyvar tv tv' then
         FORALLcoer(tv',c)
      else
         (if Set.member equaltyvar tv' (ftvbf2type t') then
             (changetyvar(tv',get_new_tv(tv'));
              FORALLcoer(tv',substoncoer(tv,t',c)))
          else
             FORALLcoer(tv',substoncoer(tv,t',c)))
  | substoncoer(tv,t',COMPcoer(c,d)) =
      COMPcoer(substoncoer(tv,t',c),substoncoer(tv,t',d))

(************************************************************)
(* Function related to the generation of completions        *)
(************************************************************)
(* Generate canonical coercions *)
exception CCError
fun cc(t,t') =
    if equalbtype t t' then
       IDcoer t
    else
      case (t,t') of
        (BOXbf2type t,BOXbf2type t') =>
           INBOXcoer(cc(t,t'))
      | (t,BOXbf2type t') =>
           if t=t' then
              BOXcoer(t)
           else
              COMPcoer(cc(t,t'),BOXcoer(t'))
      | (BOXbf2type t,t') =>
          if t=t' then
             UNBOXcoer(t)
          else
             COMPcoer(UNBOXcoer(t),cc(t,t'))
      | (CONbf2type(c,ts),CONbf2type(_,ts')) =>
          CONcoer(c,map cc (zip ts ts'))
      | (FUNbf2type(t1,t2),FUNbf2type(t1',t2')) =>
          FUNcoer(cc(t1',t1),cc(t2,t2'))
      | (FORALLbf2type(tv,t),FORALLbf2type(tv',t')) =>
          FORALLcoer(tv,cc(t,t'))
      | _ => raise CCError

(* Translate annotated f2-expressions into explicitly *)
(* boxed f2-expressions                               *)
fun f2type2bf2type(VARf2type tv) = VARbf2type tv
  | f2type2bf2type(CONf2type(n,cn,ts)) =
    if getboxval(n) = B then
       BOXbf2type(CONbf2type(cn,map f2type2bf2type ts))
    else
       CONbf2type(cn,map f2type2bf2type ts)
  | f2type2bf2type(FUNf2type(n,t,t')) =
    let val tf = FUNbf2type(f2type2bf2type t,f2type2bf2type t')
    in
       if getboxval(n) = B then BOXbf2type tf else tf
    end
  | f2type2bf2type(FORALLf2type(n,tv,t)) =
    let val t' = FORALLbf2type(tv,f2type2bf2type t)
    in
       if getboxval(n) = B then BOXbf2type t' else t'
    end

(***********************************************************)
(* Generate completions from annotated expressions          *)
(***********************************************************)
fun insertcoercion(tbx,tb,be) =
    (case cc(tbx,tb) of
       IDcoer _ => be
     | c => CAPPbf2exp(tb,c,be))

fun f2exp2bf2exp1(IDENTf2exp(t,x,tx)) =
    let val tbx = f2type2bf2type tx
    in
        insertcoercion(tbx,f2type2bf2type t,IDENTbf2exp(tbx,x))
    end
  | f2exp2bf2exp1(PRIMf2exp(t,name,t',ts,m)) =
    let val tb' = f2type2bf2type (instantiate(t',ts))
    in
        insertcoercion(tb',f2type2bf2type t,PRIMbf2exp(tb',name,t',ts,m))
    end
  | f2exp2bf2exp1(IFf2exp(t,e1,e2,e3)) =
    let val be1 = f2exp2bf2exp1 e1
        val be2 = f2exp2bf2exp1 e2
        val be3 = f2exp2bf2exp1 e3
        val tb = f2type2bf2type t
        val t1 = getbf2type be1
        val t2 = getbf2type be2
        val t3 = getbf2type be3
    in
        IFbf2exp(tb,insertcoercion(t1,CONbf2type(mktc "bool",[]),be1),
                    insertcoercion(t2,tb,be2),
                    insertcoercion(t3,tb,be3))
    end
  | f2exp2bf2exp1(ABSf2exp(t,x,tx,e)) =
    let val be  = f2exp2bf2exp1 e
        val tbe = getbf2type be
        val tbx = f2type2bf2type tx
        val tba = FUNbf2type(tbx,tbe)
    in
       insertcoercion(tba,f2type2bf2type t,ABSbf2exp(tba,x,tbx,be))
    end
  | f2exp2bf2exp1(APPf2exp(t,e,e')) =
    let val be = f2exp2bf2exp1 e
        val te = getf2type e
    in
       case te of
         FUNf2type(_,ta,_) =>
           let val tbe  = getbf2type be
               val be'  = f2exp2bf2exp1 (replacef2type(ta,e'))
               val tbe' = getbf2type be'
           in
              case tbe of
                FUNbf2type(_,tbr) =>
                  insertcoercion(tbr,f2type2bf2type t,
                                 APPbf2exp(tbr,be,be'))
              | _ => (print "internal reference 6";
                      pp_target(screen,be);
                      raise ThisCannotHappen)
           end
       | _ => (print "internal reference 7";
                      raise ThisCannotHappen)
    end
  | f2exp2bf2exp1(TABSf2exp(t,tv,e)) =
    let val be = f2exp2bf2exp1 e
        val tbe = getbf2type be
        val tbta = FORALLbf2type(tv,tbe)
    in
       insertcoercion(tbta,f2type2bf2type t,TABSbf2exp(tbta,tv,be))
    end
  | f2exp2bf2exp1(TAPPf2exp(t,e,t')) =
    let val be = f2exp2bf2exp1 e
        val tbe = getbf2type be
        val tb' = f2type2bf2type t'
        val (tv,tbb) = case tbe of
                         FORALLbf2type(tv,tbb) => (tv,tbb)
                       | _ => (print "internal reference 8";
                               raise ThisCannotHappen)
        val tbta = substb(tv,tb',tbb)
    in
       insertcoercion(tbta,f2type2bf2type t,TAPPbf2exp(tbta,be,tb'))
    end

fun f2exp2bf2exp e t =
    let val e' = f2exp2bf2exp1 e
```

```
        val te' = getbf2type e'
    in
        insertcoercion(te',f2type2bf2type t,e')
    end

(***********************************************************)
(* Type checker that also generates the representation graph *)
(***********************************************************)
exception Typecheck

fun lookup(x,r)     = r x
fun update(x,t,r)   = fn x' => if x=x' then t else lookup(x',r)

fun copytype(VARf2type tv) = VARf2type tv
  | copytype(CONf2type(_,c,ts)) = CONf2type(new_node(),c,map copytype ts)
  | copytype(FUNf2type(_,t,t')) =
      FUNf2type(new_node(),copytype t,copytype t')
  | copytype(FORALLf2type(_,tv,t)) =
      FORALLf2type(new_node(),tv,copytype t)

fun typecheck(IDENTf2exp(_,x,_),r) =
    let val t  = lookup(x,r)
        val tc = copytype t
    in
       c(t,tc);
       (IDENTf2exp(tc,x,t),tc)
    end
  | typecheck(PRIMf2exp(_,name,t,ts,m),_) =
    let val tc = copytype t
        val tbs = map copytype ts
        val (so,si) = ioSourceAndSinks tc
        val t' = instantiate(tc,tbs)
        val tc' = copytype t'
    in
       (if m=P then
           map (fn t =>
                  let val bv = getnode t
                  in
                     case bv of
                       Nil => ()
                     | _   => (addtosources(bv,B);addtosinks(bv,B))
                  end)
               tbs
        else
           []);
       c(t',tc');
       map (fn bv => addtosources(bv,U)) so;
       map (fn bv => addtosinks(bv,U)) si;
       (PRIMf2exp(tc',name,tc,tbs,m),tc')
    end
  | typecheck(IFf2exp(_,e1,e2,e3),r) =
    let val (e1',t1) = typecheck(e1,r)
        val (e2',t2) = typecheck(e2,r)
        val (e3',t3) = typecheck(e3,r)
    in
        if equaltype t1 (CONf2type(Nil,mktc "bool",[]))
        then
```

```
          (if equaltype t2 t3
           then
              let val tc = copytype t2
              in
                 uniontypes(t2,t3); (* make the demand on the boxing
                                       of the branches the same *)
                 c(t2,tc);
                 addtosinks(getnode t1,U);
                 (IFf2exp(tc,e1',e2',e3'),tc)
              end
           else
              (print "*** Type error: types:\n";
               pp_type(screen,t2);
               print "\nand\n";
               pp_type(screen,t3);
               print "\nof branches don't agree.";
               raise Typecheck))
        else
           (print "*** Type error: type:\n";
            pp_type(screen,t1);
            print "\nof test is not bool.";
            raise Typecheck)
   end
 | typecheck(ABSf2exp(_,x,tx,e),r) =
   let val txc = copytype tx
       val (e1,t1) = typecheck(e,update(x,txc,r))
       val bv = new_node()
       val t = FUNf2type(bv,txc,t1)
       val tc = copytype t
   in
      c(t,tc);
      addtosources(bv,U);
      (ABSf2exp(tc,x,txc,e1),tc)
   end
 | typecheck (APPf2exp(_,e,e'),r)  =
   let val (e1,t)   = typecheck(e,r)
       val (e1',t') = typecheck(e',r)
   in
       case t of
         FUNf2type(bv,t1,t2) =>
          (if equaltype t1 t' then
              let val t'' = copytype(t2)
              in
                 c(t2,t'');
                 c(t',t1);
                 addtosinks(bv,U);
                 (APPf2exp(t'',e1,e1'),t'')
              end
           else
              (print "*** Type error: type:\n";
               pp_type(screen,t');
               print "\nof actual argument:\n";
               pp_source(screen,e1');
               print "\ndoes not match type:\n";
               pp_type(screen,t1);
               print "\nof formal argument in:\n";
               pp_source(screen,e1);
```

```
                  raise Typecheck))
          | _ =>
              (print "*** Type error: type:\n";
               pp_type(screen,t);
               print "\nis not a function type.";
               raise Typecheck)
      end
    | typecheck (TABSf2exp(_,tv,e),r) =
      let val (e1,t) = typecheck(e,r)
          val bv = new_node()
          val t' = FORALLf2type(bv,tv,t)
          val tc = copytype t'
      in
         c(t',tc);
         addtosources(bv,U);
         (TABSf2exp(tc,tv,e1),tc)
      end
    | typecheck (TAPPf2exp(_,e,t),r) =
      let val (e',t') = typecheck(e,r)
      in
          case t' of
            FORALLf2type(bv,tv1,t1) =>
              let val pi    = copytype t
                  val bvpi  = getnode pi
                  val t''   = subst(tv1,pi,t1)
                  val tc    = copytype t''
              in
                 c(t'',tc);
                 case bvpi of
                     Nil => ()
                   | _   =>
                       (addtosources(bvpi,B);
                        addtosinks(bvpi,B));
                 addtosinks(bv,U);
                 (TAPPf2exp(tc,e',pi),tc)
              end
          | _ =>
             (print "*** Type error: type:\n";
              pp_type(screen,t');
              print "\nis not forall type.";
              raise Typecheck)
      end

  fun init_env x = (print("*** Error: unknown variable: " ^
                          (id2string x) ^ "\n\n");
                    raise Typecheck)

  fun mk_env [] = init_env
    | mk_env ((i,t)::a) =
      let val ct = copytype t
          val (so,si) = ioSourceAndSinks ct
      in
          map (fn bv => addtosources(bv,U)) so;
          map (fn bv => addtosinks(bv,U)) si;
          (if !debug then
              (print ("External " ^ id2string i ^ " of type:");
               pp_type(screen,ct);
             print "\n")
           else
             ());
          update(i,ct,mk_env a)
      end

fun init () = (isources := []; edges := []; sources := []; sinks := [];
               lastnode := Nil; nodecount := 0; tvcount := 0)

fun succ(Node(_,_,_,_,ref s,_,_)) = s
  | succ _ = []
fun pred(Node(_,_,_,_,_,ref p,_)) = p
  | pred _ = []

(***********************************************************)
(* Remove paths to and from terminal nodes that are not     *)
(* sources or sinks.                                         *)
(***********************************************************)
fun compress(n,nn,od) =
    case nn(n) of
      [n'] => let val n' = find(n')
              in
                 assym_union(n',n);
                 if od(n')=[] then
                    compress(n',nn,od)
                 else
                    ()
              end
    | _ => ()

fun cleanUp n =
    case n of
      Nil => ()
    | Node(_,next,_,_,_,_,_) =>
      let val n = find n
      in
        case n of
          Node(_,_,_,_,_,ref [],ref NV) =>
            (* no pred. and not a source node *)
            (compress(n,succ,pred);cleanUp(!next))
        | Node(_,_,_,_,ref [],_,ref NV) =>
            (* no succ. and not a sink node *)
            (compress(n,pred,succ);cleanUp(!next))
        | _ => cleanUp(!next)
      end
(***********************************************************)
(* Type check program and generate graph, etc.              *)
(***********************************************************)
fun generate (a,e) =
   let val r = mk_env a
       val (e',t') = typecheck(e,r)
       val tc' = copytype t'
       val (si,so) = ioSourceAndSinks tc'
   in
      conecttypes(t',tc');
      map (fn bv => addtosources(bv,U)) so;
      map (fn bv => addtosinks(bv,U)) si;
```

```
          cleanUp(!lastnode);
          (e',tc')
       end

    (**********************************************************)
    (* The core of the algorithm                              *)
    (**********************************************************)
    (* propagate from sources *)
    fun propagate(is) =
        let val dop = !dop
            fun findB n =
                let val n = find n
                in
                   (case n of
                      Node(_,_,ecrp,_,_,_,ref NV) =>
                        (mark V n;
                         case (succ n) of
                           [] =>()
                         | (n1::ns) =>
                           assym_union(List.foldR dop
                                                  (findB n1)
                                                  (map findB ns),
                                       n);
                         mark VU n)
                    | _ => ());
                    find n
                end
            fun foreachnode [] = ()
              | foreachnode (n::ns) =
                  (findB n;foreachnode ns)
            fun foreachsource [] = ()
              | foreachsource (n::ns) =
                  (foreachnode(succ (find n));
                   foreachsource ns)
        in
            foreachsource is
        end
end;
```

B.7 Main Program

The last file contains the functions the puts the whole thing together and defines functions to run examples. It contains the following two function to do this:

- `run` that take a string containing a source program as inputs and run the analysis on it.
- `runOnFile` that reads a source program from a file and runs the analysis on it outputting a ML program.

The following shows an example of a session with the program:

```
- runOnFile "test/fromthesis";
Processing file: test/fromthesis.f2.
```

```
Initializing...done!

Parsing...done!

Run-time: 0.010000

Solving...Run-time: 0.0

done!

Writing to file:
use "test/fromthesis.sml";

val it = () : unit
```

The program expect the file "fromthesis" to have extension "f2" and it outputs the result on a file "fromthesis.sml". One can run the program "fromthesis.sml" as follows:

```
- use "test/fromthesis.sml";
[opening test/fromthesis.sml]
exception hderror
exception tlerror
val it = (4,[("box:int",1),("unbox:int",2),("cl",1),("app",1)])
  : int * (string * int) list
val it = () : unit
-
```

The output from this is a tuple consisting of first, the actual output of the program. The second is a list containing the number of box and unbox operations that were performed sorted by the topmost type constructor of the type of the box operations, in this case only `"int"`. The list also contains information on how many "stub code" closures were created and how many extra applications these closures gave rise to. A more complete example of a session with the program is shown in Appendix C.

```
(* main.sml *)
structure Expression = Expression();
structure Parser = Parser(structure Expression = Expression);
structure BoxingAnalysis = BoxingAnalysis(structure Expression = Expression);
open Expression Parser BoxingAnalysis

val analyse = ref true
fun analysisOn() = analyse:=true
fun analysisOff() = analyse:=false

fun input_file fileName =
    let val file = open_in fileName
        fun getstring() =
            case input(file,1) of
              "" => ""
            | c  => c ^ getstring()
        val s = getstring()
    in
       close_in file;
       s
    end

(* Run small examples *)
fun run s =
    (print "Initializing...";
     init();
     print "done!\n\n";
     print "Parsing...";
     case parse s of
       Some p =>
         (print "done!\n\n";
          let val (e',t') = generate p
          in
             print "Source expression (with node no.):\n";
             pp_source(screen,e');
             pp_type(screen,t');

             (if !debug then debug_print_edges_sources_and_nodes() else ());

             print "\n\nSolving...";
             propagate(!isources);
             print "done!\n\n";

             (if !debug then debug_print_sol() else ());

             print("\n\nResult(" ^ !txt ^ ")\n");

             let val c = f2exp2bf2exp e' t'
             in
                pp_target(screen,c);
                print "\n\n"
             end
          end
          handle Typecheck => ())
     | None => ())

fun timeit f =
    let open SML_NJ.Timer
        val t = start_timer()
        val r = f()
        val t' = check_timer t
    in
        print "Run-time: ";
        print (makestring t');
        print "\n\n";
        r
    end

(* Run examples from a file *)
fun runOnFile fileName =
    let val f2FileName = fileName ^ ".f2"
        val s = input_file f2FileName
    in
        print ("Processing file: " ^ f2FileName ^ ".\n\n");
        print "Initializing...";
        init();
        print "done!\n\n";
    print "Parsing...";
    (case parse s of
       Some p =>
         (print "done!\n\n";
          let val (e',t') = timeit (fn () => generate p)
          in
             (if !debug then
                 (print "Source expression (with node no.):\n";
                  pp_source(screen,e');
                  pp_type(screen,t');
                  debug_print_edges_sources_and_nodes();
                  print "\n\n")
              else
                 ());

             (if !analyse then
                 (print "Solving...";
                  timeit (fn () => propagate(!isources));
                  print "done!\n\n")
              else
                 (* do nothing = canonical completion *)
                 ());

             (if !debug then
                 debug_print_sol()
              else
                 ());

             let val c = f2exp2bf2exp e' t'
             in
                (if !debug then
                    (print("\n\nResult(" ^ !txt ^ ")\n");
                     pp_target(screen,c);
                     print "\n\n")
                 else
                    let val SMLFilename = fileName ^ ".sml"
                        val file = open_out SMLFilename
                        fun device s = output(file,s)
                    in
                        print("Writing to file: \nuse \"" ^
                               SMLFilename ^ "\";\n\n");
                        pp_SML(device,c,!txt);
                        close_out file
                    end)
             end
          end
          handle Typecheck => ())
     | None => ())
end
```

Appendix C

An Example

This appendix shows a session with the system. The example program is the one that we used (in Chapter 4) to show that $\phi\psi$-reduction on E-congruence classes is not Church-Rosser. The program is here shown in the source syntax of our system:

```
extern true:bool();
extern plus:int()->int()->int();

let id:A a.a->a = Fn a => fn x:a => x
in
   (fn x:int() =>plus x (id{int()} x))
   (if true then 2 else id{int()} 5)
end
```

The `extern` definition corresponds to adding primitives to the initial type assignment Γ like explained in Section 6.2. In this example the program is set to produce $\mathcal{C}_{p*}$-completions (`cpstarOn()`), and print debug information.

```
- debugOn();
val it = () : unit
- setpsi();
Psi-free modeval it = () : unit
- cpstarOn();
val it = () : unit
- runOnFile "test/fromthesis";
Processing file: test/fromthesis.f2.

Initializing...done!

Parsing...done!

External plus of type:
(int#1 ->#0 (int#3 ->#2 int#4))

External true of type:
bool#5

Run-time: 0.010000

Source expression (with node no.):
```

```
let id: (A a.#6 (a ->#7 a)) = (FN a=>(fn x:a=>x))
in  let x: int#8 = if true then 2 else (id{int#36} 5)
    in  ((plus x) (id{int#20} x))
    end
end

int#56

Edges:

55=>56,54=>35,53=>6,49=>55,51=>54,52=>53,50=>51,44=>49,48=>35,47=>6,45=>46,
43=>8,30=>44,42=>43,33=>42,41=>39,39=>42,40=>41,36=>39,38=>36,35=>37,19=>35,
6=>34,32=>33,5=>31,26=>30,29=>8,27=>28,25=>3,4=>26,24=>23,23=>25,8=>24,
20=>23,22=>20,19=>21,7=>19,6=>18,14=>1,4=>17,16=>3,2=>15,8=>14,4=>13,12=>3,
2=>11,10=>1,0=>9,

Sources:

52=U,50=U,45=U,40=U,39=B,32=U,27=U,23=B,5=U,4=U,2=U,0=U,

ISources:

39=B,23=B,

Sinks:

56=U,46=U,28=U,31=U,37=U,34=U,39=B,15=U,21=U,18=U,23=B,9=U,3=U,1=U,

Solving...Run-time: 0.0

done!

56=(56)U,55=(55)U,54=(35)U,53=(53)U,52=(52)U,51=(35)U,50=(50)U,49=(49)U,
48=(35)U,47=(6)U,46=(46)U,45=(45)U,44=(49)U,43=(23)B,42=(23)B,41=(41)U,
40=(40)U,39=(39)B,38=(39)B,37=(37)U,36=(39)B,35=(35)U,34=(34)U,33=(23)B,
32=(32)U,31=(31)U,30=(30)U,29=(23)B,28=(28)U,27=(27)U,26=(30)U,25=(3)U,
24=(23)B,23=(23)B,22=(23)B,21=(21)U,20=(23)B,19=(35)U,18=(18)U,17=(4)U,
16=(3)U,15=(15)U,14=(1)U,13=(4)U,12=(3)U,11=(2)U,10=(1)U,9=(9)U,8=(23)B,
7=(35)U,6=(6)U,5=(5)U,4=(4)U,3=(3)U,2=(2)U,1=(1)U,0=(0)U,

Result(Psi-free mode)

let id: (A a.(a->a)) = (FN a=>(fn x:a=>x))
in  let x: [int] = if true then <box>2 else ((id){[int]} <box>5)
    in  ((plus <unbox>x) <unbox>((id){[int]} x))
    end
end


val it = () : unit
- setphi();
Phi-free modeval it = () : unit
- runOnFile "test/fromthesis";
Processing file: test/fromthesis.f2.

Initializing...done!
```

```
Parsing...done!

External plus of type:
(int#1 ->#0 (int#3 ->#2 int#4))

External true of type:
bool#5

Run-time: 0.010000

Source expression (with node no.):

let id: (A a.#6 (a ->#7 a)) = (FN a=>(fn x:a=>x))
in  let x: int#8 = if true then 2 else (id{int#36} 5)
    in  ((plus x) (id{int#20} x))
    end
end

int#56

Edges:

55=>56,54=>35,53=>6,49=>55,51=>54,52=>53,50=>51,44=>49,48=>35,47=>6,45=>46,
43=>8,30=>44,42=>43,33=>42,41=>39,39=>42,40=>41,36=>39,38=>36,35=>37,19=>35,
6=>34,32=>33,5=>31,26=>30,29=>8,27=>28,25=>3,4=>26,24=>23,23=>25,8=>24,
20=>23,22=>20,19=>21,7=>19,6=>18,14=>1,4=>17,16=>3,2=>15,8=>14,4=>13,12=>3,
2=>11,10=>1,0=>9,

Sources:

52=U,50=U,45=U,40=U,39=B,32=U,27=U,23=B,5=U,4=U,2=U,0=U,

ISources:

52=U,50=U,45=U,40=U,32=U,27=U,5=U,4=U,2=U,0=U,

Sinks:

56=U,46=U,28=U,31=U,37=U,34=U,39=B,15=U,21=U,18=U,23=B,9=U,3=U,1=U,

Solving...Run-time: 0.0

done!

56=(56)U,55=(56)U,54=(21)U,53=(18)U,52=(52)U,51=(21)U,50=(50)U,49=(56)U,
48=(21)U,47=(18)U,46=(46)U,45=(45)U,44=(56)U,43=(1)U,42=(1)U,41=(39)B,
40=(40)U,39=(39)B,38=(39)B,37=(37)U,36=(39)B,35=(21)U,34=(34)U,33=(1)U,
32=(32)U,31=(31)U,30=(56)U,29=(1)U,28=(28)U,27=(27)U,26=(56)U,25=(25)B,
24=(23)B,23=(23)B,22=(23)B,21=(21)U,20=(23)B,19=(21)U,18=(18)U,17=(4)U,
16=(3)U,15=(15)U,14=(1)U,13=(4)U,12=(3)U,11=(2)U,10=(1)U,9=(9)U,8=(1)U,
7=(21)U,6=(18)U,5=(5)U,4=(4)U,3=(3)U,2=(2)U,1=(1)U,0=(0)U,

Result(Phi-free mode)

let id: (A a.(a->a)) = (FN a=>(fn x:a=>x))
in  let x: int = if true then 2 else <unbox>((id){[int]} <box>5)
    in  ((plus x) <unbox>((id){[int]} <box>x))
    end
end


val it = () : unit
-
```

Appendix D

Test Programs

This appendix contains the test programs described in Section 7.2.

D.1 Monomorphic insert-sort example

```
(* Monomorphic insert-sort example
   Corresponds to Poulsen's example 1.
*)
extern cons   : A b.b->list(b)->list(b);
extern hd     : A a.list(a)->a;
extern tl     : A a.list(a)->list(a);
extern nil    : A b.list(b);
extern pair   : A a.A b.a->b->pair(a b);
extern null   : A a.list(a) -> bool();
extern fst    : A a.A b.pair(a b)->a;
extern snd    : A a.A b.pair(a b)->b;
extern plus   : int()->int()->int();
extern gt     : int()->int()->bool();
extern fix    : A a.(a->a)->a;

let insert:int()->(list(int())->list(int())) =
  fix*{int()->(list(int())->list(int()))}
   (fn f:int()->(list(int())->list(int())) =>
    (fn e:int() =>
     (fn l:list(int()) =>
       if (null{int()} l)
       then
          cons{int()} e nil{int()}
       else
          (if (gt e (hd{int()} l))
           then
              cons{int()} (hd{int()} l) ((f e) (tl{int()} l))
           else
              cons{int()} e l))))
in
let sort:list(int())->(list(int())->list(int())) =
  fix*{list(int())->(list(int())->list(int()))}
   (fn f:list(int())->(list(int())->list(int())) =>
    (fn l:list(int()) =>
     (fn a:list(int()) =>
      (if (null{int()} l)
       then
          a
       else
          (f (tl{int()} l) (insert (hd{int()} l) a))))))
in
    sort (cons{int()} 2
         (cons{int()} 4
         (cons{int()} 1
         (cons{int()} 8
         (cons{int()} 5
         (cons{int()} 3
         (cons{int()} 0
         (cons{int()} 4
         (cons{int()} 2
         (cons{int()} 1
         (cons{int()} 345
         (cons{int()} 12
         (cons{int()} 2
         (cons{int()} 4
         (cons{int()} 6
         (cons{int()} 3
         (cons{int()} 1 nil{int()}))))))))))))))))
         nil{int()}
end
end
```

D.2 Polymorphic insert-sort example

```
(* Polymorphic insert-sort example
   Corresponds to Poulsen's example 2
*)
extern cons   : A b.b->list(b)->list(b);
extern hd     : A a.list(a)->a;
extern tl     : A a.list(a)->list(a);
extern nil    : A b.list(b);
extern pair   : A a.A b.a->b->pair(a b);
extern null   : A a.list(a) -> bool();
extern fst    : A a.A b.pair(a b)->a;
extern snd    : A a.A b.pair(a b)->b;
extern plus   : int()->int()->int();
extern gt     : int()->int()->bool();
extern fix    : A a.(a->a)->a;

let insert:A a.a->(list(a)->((a->(a->bool()))->list(a))) =
  Fn a =>
   fix*{a->(list(a)->((a->(a->bool()))->list(a)))}
    (fn f:a->(list(a)->((a->(a->bool()))->list(a))) =>
     (fn e:a =>
      (fn l:list(a) =>
       (fn gt:a -> (a -> bool()) =>
         if (null{a} l)
         then
            cons{a} e nil{a}
         else
            (if (gt e (hd{a} l))
             then
```

```
                cons{a} (hd{a} l) ((f e) (tl{a} l) gt)
             else
                cons{a} e l)))))
in
let sort:A a.list(a)->(list(a)->((a->(a->bool()))->list(a))) =
  Fn a =>
   fix*{list(a)->(list(a)->((a->(a->bool()))->list(a)))}
    (fn f:list(a)->(list(a)->((a->(a->bool()))->list(a))) =>
     (fn l:list(a) =>
      (fn a:list(a) =>
       (fn gt:a->(a->bool()) =>
         (if (null{a} l)
          then
             a
          else
             (f (tl{a} l) (insert{a} (hd{a} l) a gt) gt))))))
in
let gt:int()->int()->bool() = fn x:int() => fn y:int() => gt x y
in
    sort{int()}
        (cons{int()} 2
        (cons{int()} 4
        (cons{int()} 1
        (cons{int()} 8
        (cons{int()} 5
        (cons{int()} 3
        (cons{int()} 0
        (cons{int()} 4
        (cons{int()} 2
        (cons{int()} 1
        (cons{int()} 345
        (cons{int()} 12
        (cons{int()} 2
        (cons{int()} 4
        (cons{int()} 6
        (cons{int()} 3
        (cons{int()} 1 nil{int()}))))))))))))))))
        nil{int()}
        gt
end
end
end
```

D.3 Flip-list example

```
(* Flip-list example
   Correspods to Poulsen's example 4
*)
extern cons   : A b.b->list(b)->list(b);
extern hd     : A a.list(a)->a;
extern tl     : A a.list(a)->list(a);
extern nil    : A b.list(b);
extern mkpair : A a.A b.a->b->pair(a b);
extern null   : A a.list(a) -> bool();
extern fst    : A a.A b.pair(a b)->a;
extern snd    : A a.A b.pair(a b)->b;
extern plus   : int()->int()->int();
extern gt     : int()->int()->bool();
extern fix    : A a.(a->a)->a;

let map:A a.A b.(a->b)->list(a)->list(b) =
  Fn a =>
    Fn b =>
      fix*{(a->b)->list(a)->list(b)}
       (fn map:(a->b)->list(a)->list(b) =>
         fn f:a->b =>
          fn l:list(a) =>
           if (null{a} l) then
              nil{b}
           else
              cons{b} (f (hd{a} l)) (map f (tl{a} l)))
in
let flip:A a.A b.pair(a b)->pair(b a) =
      Fn a =>
        Fn b =>
          fn p:pair(a b) =>
            ((mkpair{b}){a}) (((snd{a}){b}) p) (((fst{a}){b}) p)
in
    (map{pair(int() int())}){pair(int() int())}
      (flip{int()}){int()}
      (cons{pair(int() int())} ((mkpair{int()}){int()} 1 2)
      (cons{pair(int() int())} ((mkpair{int()}){int()} 3 4)
      (cons{pair(int() int())} ((mkpair{int()}){int()} 5 6)
      (cons{pair(int() int())} ((mkpair{int()}){int()} 7 8)
      (cons{pair(int() int())} ((mkpair{int()}){int()} 9 10)
                                nil{pair(int() int())})))))
end
end
```

D.4 Leroy's example

```
(* Leroy's example
   Corresponds to Poulsen's example 5
*)
extern plus:int()->int()->int();

let double:A a.(a->a)->a->a = Fn a => fn f:a->a => fn x:a => (f (f x)) in
let quad:A a.(a -> a) -> a -> a = Fn a => double{a->a} double{a} in
let f:int()->int() =
    quad{int()->int()} quad{int()} (fn x:int() => plus x 1)
in
    f 1
end
end
end
```

D.5 Poulsen's example

```
(* Poulsen's example
   Corresponds to Poulsen's example 6.
*)
```

```
extern cons : A b.b->list(b)->list(b);
extern hd   : A a.list(a)->a;
extern tl   : A a.list(a)->list(a);
extern nil  : A b.list(b);
extern sub  : int()->int()->int();
extern plus : int()->int()->int();
extern eq   : int()->int()->bool();
extern fix  : A a.(a->a)->a;

let id:A a.a->a = Fn a => fn x:a => x in
let f:int()->int() =
  fix*{int()->int()}
   (fn f:int()->int() =>
     (fn x:int() =>
       if (eq x 0) then
          0
       else
         plus (f (sub x 1)) 2))
in
    (cons{int()} (f (id{int()} 100))
    (cons{int()} (f (id{int()} 100))
    (cons{int()} (f (id{int()} 100))
                 nil{int()})))
end
end
```

D.6 Sieve example

```
(* Sieve example *)
extern cons   : A b.b->list(b)->list(b);
extern hd     : A a.list(a)->a;
extern tl     : A a.list(a)->list(a);
extern nil    : A b.list(b);
extern pair   : A a.A b.a->b->pair(a b);
extern null   : A a.list(a) -> bool();
extern fst    : A a.A b.pair(a b)->a;
extern snd    : A a.A b.pair(a b)->b;
extern plus   : int()->int()->int();
extern gt     : int()->int()->bool();
extern noteq  : int()->int()->bool();
extern modulo : int()->int()->int();
extern fix    : A a.(a->a)->a;

let filter:A a.(a->bool())->list(a)->list(a) =
  Fn a =>
   fix*{(a->bool())->list(a)->list(a)}
    (fn filter:(a->bool())->list(a)->list(a) =>
      fn t:a->bool() =>
       fn l:list(a) =>
        if null{a} l then
           nil{a}
        else
           let e:a = hd{a} l in
             if t e then
                cons{a} e (filter t (tl{a} l))
             else
```

```
                filter t (tl{a} l)
           end)
in
let fromto:int()->int()->list(int()) =
  fix*{int()->int()->list(int())}
   (fn fromto:int()->int()->list(int()) =>
     fn f:int() =>
      fn t:int() =>
       if gt f t then
          nil{int()}
       else
          cons{int()} f (fromto (plus f 1) t))
in
let sieve:list(int())->list(int()) =
  fix*{list(int())->list(int())}
   (fn sieve:list(int())->list(int()) =>
     fn l:list(int()) =>
      if null{int()} l then
         nil{int()}
      else
        let h:int() = (hd{int()} l)
        in
           cons{int()}
            h
            (sieve
             (filter{int()}
              (fn x:int() =>
               noteq (modulo x h) 0)
              (tl{int()} l)))
        end)
in
    sieve (fromto 2 100)
end
end
end
```

D.7 Horner example

```
(* Horner example
   Example from Thiemann
*)
extern cons   : A b.b->list(b)->list(b);
extern hd     : A a.list(a)->a;
extern tl     : A a.list(a)->list(a);
extern nil    : A b.list(b);
extern pair   : A a.A b.a->b->pair(a b);
extern null   : A a.list(a) -> bool();
extern fst    : A a.A b.pair(a b)->a;
extern snd    : A a.A b.pair(a b)->b;
extern plus   : int()->int()->int();
extern mult   : int()->int()->int();
extern gt     : int()->int()->bool();
extern fix    : A a.(a->a)->a;

let map:A a.A b.(a->b)->list(a)->list(b) =
  Fn a =>
```

```
   Fn b =>
    fix*{(a->b)->list(a)->list(b)}
     (fn map:(a->b)->list(a)->list(b) =>
       fn f:a->b =>
        fn l:list(a) =>
          if (null{a} l) then
             nil{b}
          else
             cons{b} (f (hd{a} l)) (map f (tl{a} l)))
in
let b:A a.A b.A c.(a->b)->(c->a)->c->b =
    Fn a =>
     Fn b =>
      Fn c =>
       fn f:a->b => fn g:c->a => fn x:c => (f (g x))
in
let horner:list(int())->int()->int() =
    fn coeffs:list(int()) =>
      fn x0:int() =>
        let funs:list(int()->int()) =
            (map{int()}){int()->int()}
               (fn c:int() =>
                 ((((b{int()}){int()}){int()})
                    (fn y:int() => (plus y c))
                    (fn y:int() => (mult y x0))))
               coeffs
        in
           ((fix*{list(int()->int())->int()->int()}
              (fn eval:list(int()->int())->int()->int() =>
                fn fs:list(int()->int()) => fn sofar:int() =>
                 if (null{int()->int()} fs) then
                    sofar
                 else
                    (eval (tl{int()->int()} fs)
                          ((hd{int()->int()} fs) sofar))))
            funs 0)
        end
in
   (horner
     (cons{int()} 1
     (cons{int()} 2
     (cons{int()} 3
     (cons{int()} 4
     (cons{int()} 5
     (cons{int()} 6
     (cons{int()} 7
     (cons{int()} 8
     (cons{int()} 9
     (cons{int()} 10 nil{int()}))))))))))
     1)
end
end
end
```

D.8 Mogensen example

```
(* Example constructed from an original
   idea from Torben Mogensen.
*)
extern mkpair   : A a.A b.a->b->pair(a b);
extern fst      : A a.A b.pair(a b)->a;
extern snd      : A a.A b.pair(a b)->b;
extern plus     : int()->int()->int();
extern sub      : int()->int()->int();
extern real2int : real()->int();
extern int2real : int()->real();

let cpair:A a.A b.A c.A d.(a->b)->(c->d)->pair(a c)->pair(b d) =
    Fn a =>
     Fn b =>
      Fn c =>
       Fn d =>
        fn f:a->b =>
         fn g:c->d =>
          fn p:pair(a c) =>
           ((mkpair{b}){d} (f ((fst{a}){c} p)) (g ((snd{a}){c} p)))
in
let fpair:A a.A b.A c.(a->b)->(a->c)->a->pair(b c) =
    Fn a =>
     Fn b =>
      Fn c =>
       fn f:a->b =>
        fn g:a->c =>
         fn x:a => ((mkpair{b}){c} (f x) (g x))
in
let flip:A a.A b.pair(a b)->pair(b a) =
      Fn a =>
        Fn b =>
          fn p:pair(a b) =>
            ((mkpair{b}){a}) (((snd{a}){b}) p) (((fst{a}){b}) p)
in
let sub1:int()->int() = fn x:int() => sub x 1
in
let add1:int()->int() = fn x:int() => plus x 1
in
let uncurry:A a.A b.A c.(a->b->c)->pair(a b)->c =
    Fn a =>
     Fn b =>
      Fn c =>
       fn f:a->b->c =>
        fn p:pair(a b) =>
         f ((fst{a}){b} p) ((snd{a}){b} p)
in
((uncurry{int()}){int()}){int()}
   plus
   ((((cpair{int()}){int()}){real()}){int()}
    sub1
    real2int
    ((flip{real()}){int()}
     (((fpair{int()}){real()}){int()} int2real add1 5)))
end
```

```
end
end
end
end
end
```

Appendix E

Dansk sammenfatning

En vigtig design beslutning ved implementation af programmeringssprog med polymorfe typer, så som Standard ML eller Haskell, er hvordan data skal repræsenteres. En polymorf funktion skal opføre sig uniformed (ens) for inddata værdier af uendelig mange forskellige typer. En polymorf funktion der arbejder på lister, såsom *reverse*, der vender en liste om, skal kunne virke for alle typer af lister, f. eks. lister af heltal (eng: integers), lister af reelle tal (eng: floating point numbers), lister af lister af heltal, o.s.v. Da den maskinnære repræsentation af disse kan være meget forskellig, f. eks. kan et heltal fylde et maskinord, mens et reellet tal måske fylder to maskinord, må man i en implementation derfor sikre sig at sådanne funktioner virker korrekt for alle korrekte argument typer. Den traditionelle metode til at sikre dette, er at repræsentere alle værdier på en uniform måde f. eks. som hægter (eng: pointers) til objekter i den maskinnære repræsentation. Denne metode betyder, at der introduceres ekstra køretids omkostninger ved udførelse af simple basale operationer, som f. eks. addition af tal. Ved udførelsen af en addition af to tal skal et program først "følge" to hægter for at få fat i de to tal i deres maskinnære repræsentation, hvorefter den egentlige addition kan finde sted, og til sidst skal endnu en hægte, der peger på resultatet, oprettes. Uniform repræsentation af værdier betyder også ekstra forbrug af lager, da f. eks. repræsentationen af tal både består af den egentlige (maskinnære) repræsentation af tallet såvel som hægten til denne. Værdier, der er repræsenteret uniformt, kaldes *boksede* (eng: *boxed*) og værdier, der er repræsenteret ved deres maskinnære repræsentation, kaldes *uboksede* (eng: *unboxed*).

En anden metode til implementation af polymorfi giver afkald på kravet om at data skal repræsenteres uniformt, men den pris man så må betale er at polymorfe funktioner på køretid må bruge typer til at kunne behandle data korrekt. Dette betyder så at disse typer skal være tilstæde på køretid og overføres til polymorfe funktioner som argumenter. Dette giver derfor en forøgelse af både køretid og lagerforbrug i forhold til implementationer der ikke bruger typer på køretid.

E.1 Boxanalyse

Denne afhandling beskriver en alternativ metode til implementation af polymorfi, kaldet box-analyse (eng: *boxing analysis*), hvor det statisk, d.v.s. på oversættelsestid, bestemmes hvordan værdier skal repræsenteres på køretid og hvornår værdier skal ændres fra en repræsentation til en anden. Denne metode blev først brugt af Leroy [Ler92], men resultaterne i denne afhandling forbedre metoden i forhold til Leroy's og giver et fast teoretisk grundlag for metoden, som bl. a. inkludere et formelt optimalitets begreb og viser at Leroy's resultater ikke er optimale med hensyn til dette begreb. Operationer der ændre repræsentation af værdier kaldes (repræsentations) coercions. Operationer der ændre værdier fra deres boksede repræsentation til deres uboksede repræsentation kaldes box coercions og operationer der ændre værdier fra deres uboksede repræsentation til deres boksede repræsentation kaldes unbox coercions.

Ved at bruge et sprog med eksplicite repræsentations typer og box/unbox coercions har vi fundet en axiomatisering af en lighedsteori over mængden af programmer med eksplicit box/unbox coercions, også kaldet *completions*. Denne lighedsteori har vi kaldt E-ekvivalens. Ved at give axiomerne i denne teori forskellige fortolkninger som udtryksomskrivningsregler (eng: term rewriting rules) har vi udviklet fire kanoniske (eng: canonical) omskrivningssystemer kaldet: $\psi/E\phi$, ψ/E, $\phi/E\psi$ og ϕ/E, som opererer på ekvivalence klasses of completions. Disse muligør elimination of unødige par af box/unbox coercions i completions, og dermed unødige skift af repræsentation. Dette gøres ved først at anvende $\psi/E\phi$ indtil en normalform for dette system er fundet og derefter anvende ϕ/E indtil en også en normalform for dette system er fundet. Den herved fundne normalform er en E-ekvivalens klasses og denne indeholder ingen completions med eliminerbare par af box/unbox coercions og elementerne kaldes *ψ-free optimale completions*. Ligeledes kan man, ved først at anvende $\phi/E\psi$ indtil en normalform for dette system er fundet og derefter anvende ψ/E indtil en også en normalform for dette system er fundet, finde en anden E-ekvivalens klasses af completions der heller ikke indeholder nogen completions med eliminerbare par af box/unbox coercions. Denne klasses af complecions kaldes *ψ-free optimale completions*.

Vi har vist at et hvert kildeprogram både har en ψ-free optimal completion og en ψ-free optimal completion og at disse hver især er unike modulo E-ekvivalens. Desuden har vi demonstreret hvordan man kan implementer de fire omskrivningssystemer $\psi/E\phi$, ψ/E, $\phi/E\psi$ og ϕ/E ved hjælp af fire andre omskrivningssystemer som virker direkte på ekvivalensklasserepræsentanter istedet for på ekvivalensklasser, og vi har udviklet en effektiv algoritme til at finde ψ-free og ϕ-free optimal completion.

Denne teory er udviklet for den polymorfe lambda kalkyle, også kaldet F_2 og udvidet tit også at kunne håndtere sprog med datatyper og (rekursive) primitiver, således at metoden skulle være anvendelig ved implementation af ML lignende sprog.

E.2 Nye resultater beskrevet i denne afhandling

Selvom det problem som er løst i denne afhandling skyldes tilstædeværelsen af polymorfi, så introducerede dette også nye og udfordrende problemer som også skulle løses. For eksemple betød dette at vi skulle vise "coherence" for "completions" under tilstædeværelsen af polymorfi. Vi har løst dette problem og vi tror at dette kan vise sig at være det vigtigste teoretiske resultat i denne afhandling, da det kan vise sig at være relevant også i andre sammenhænge, hvor man vil vise "coherence" under tilstædeværelsen af polymorfi, f. eks. dynamisk typning [Hen93] eller "subtypning".

Følgende opsummere nogle af de vigtigste resultater præsenteret i denne afhandling:

- Et generelt og robust kriterie for qualiteten of "boxing completions", som redegør for de enkelte box/unbox operations omkostninger, men abstrakthere fra andre sprog egenskaber og implementations hensyn.

- Et bevis for at der eksistere *formelle optimale (box) completions* og at disse er unike modulo med hensyn til en given equivalence teori.
- En metode til at integrere polymorfi og coercion kalkyle der kan vise sig brugbar i andre sammenhænge, som f. eks. dynamisk typning [Hen93] eller “subtypning”.
- En omskrivnings (eng: rewriting) baseret algoritme til at beregne formelle optimale completions, som er bedre en dem beskrevet i [PJL91, Ler92, Pou93] med hensyn til vores formelle optimalitets begreb.
- En effektiv graph baseret algoritme til at beregne formelle optimale completions.
- En eksperimentel implementation af den graph baserede algoritme for et “call-by-value” sprog.

Bibliography

[Bar84] H. P. Barendregt. *The lambda calculus, its syntax and semantics.* North-Holland, 2. edition, 1984.

[BGS82] R. Brooks, R. Gabriel, and G. Steele. An optimizing compiler for lexically scoped LISP. In *Proc. SIGPLAN '82 Symp. on Compiler Construction, Boston, Massachusetts*, pages 261–275, June 1982. SIGPLAN Notices, Vol. 17, No. 6.

[BJ93a] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming, special issue on partial evaluation*, 3(3):315–346, 1993.

[BJ93b] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation: extended version. Technical Report 93/4, DIKU, University of Copenhagen, Denmark, 1993.

[Bjø92] Nikolaj Bjørner. Minimal typing derivations. DIKU Student Report, July 1992.

[Bjø94] Nikolaj Bjørner. Minimal typing derivations. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 120–126. INRIA, 1994.

[Bon90] Anders Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, University of Copenhagen, Denmark, 1990.

[BRTT93] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML kit (version 1). Technical Report DIKU-report 93/14, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, 1993.

[BTCGS91] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, July 1991. Presented at LICS '89.

[BTGS90] V. Breazu-Tannen, C. Gunter, and A. Scedrov. Computing with coercions. In M. Wand, editor, *Proc. ACM Symp. on Lisp and Functional Programming (LFP), Nice, France*, pages 44–60, 1990.

[CF91] R. Cartwright and M. Fagan. Soft typing. In *Proc. ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation, Toronto, Ontario*, pages 278–292. ACM, ACM Press, June 1991.

[CG90] P. Curien and G. Ghelli. Coherence of subsumption. In A. Arnold, editor, *Proc. 15th Coll. on Trees in Algebra and Programming, Copenhagen, Denmark*, pages 132–146. Springer, May 1990.

[Ghe90] G. Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism.* PhD thesis, Universita di Pisa, Dipartimento di Informatica, March 1990.

[GJ94a] Robert Glück and Jesper Jørgensen. Generating optimizing specializers. In *IEEE International Conference on Computer Languages*, pages 183–194. IEEE Computer Society Press, 1994.

[GJ94b] Robert Glück and Jesper Jørgensen. Generating transformers for deforestation and supercompilation. In B. Le Charlier, editor, *Static Analysis. Proceedings*, volume 864 of *LNCS*, pages 432–448, Namur, Belgium, 1994. Springer-Verlag.

[Hen92] Fritz Henglein. Dynamic typing. In *Proc. European Symp. on Programming (ESOP), Rennes, France*, pages 233–253. Springer, Feb. 1992. LNCS, Vol. 582.

[Hen93] Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 1993.

[HJ94] Fritz Henglein and Jesper Jørgensen. Formally optimal boxing. In *21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Portland, Oregon*, pages 213–226, Jan 1994.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. 22nd Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California.* ACM, ACM Press, Jan. 1995.

[HR95] Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for Scheme: translating Scheme to ML. In *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA)*, 1995.

[Jeu95] Johan Jeuring. Polytypic pattern matching. In *FPCA'95, Conference on Functional Programming and Computer Architecture, La Jolla, California, USA*, pages 238–248. ACM, June 1995.

[KKR$^+$86] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. In *Proc. SIGPLAN '86 Symp. on Compiler Construction*, pages 219–233, 1986.

[Klo87] Jan Willem Klop. Term rewriting systems: a tutorial. *Bulletin of the European Association for Theoretical Computer Science*, 32:143–182, June 1987.

[Ler90] Xavier Leroy. Efficient data representation in polymorphic languages. Technical Report 1264, INRIA, August 1990.

[Ler92] Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Albuquerque, New Mexico*, pages 177–188, January 1992.

[LMS93] Giuseppe Longo, Kathleen Milsted, and Sergei Soloviev. The genericity theorem and parametricity in the polymorphic λ-calculus. *Bulletin of the European Association for Theoretical Computer Science*, 119:323–349, October 1993.

[Mil78] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[Pet89] J. Peterson. Untagged data in tagged environments: Choosing optimal representations at compile time. In *Proc. Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 89–99. ACM Press, Sept. 1989.

[PJL91] Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens. In *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA), Cambridge, Massachusetts*, pages 636–666. Springer, Aug. 1991. LNCS, Vol. 523.

[Pou93] Eigil Poulsen. Representation analysis for efficient implementation of polymorphism. Master's thesis, DIKU, University of Copenhagen, 1993.

[Reh95] Jacob Rehof. Polymorphic dynamic typing. Master's thesis, DIKU, University of Copenhagen, Denmark, March 1995.

[Rey83] J. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing*, pages 513–523, 1983.

[SA94] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. Technical Report CS-TR-477-94, Department of Computer Science, Princeton University, November 1994.

[Ste77] G. Steele. Fast arithmetic in MacLisp. In *Proc. 1977 MACSYMA Users' Conference, NASA Scientific and Technical Information Office, Washington, D.C.*, July 1977.

[Thi95] Peter J. Thiemann. Unboxed values and polymorphic typing revisited. In *FPCA '95 Conference on Functional Programming Languages and Computer Architecture.* ACM, ACM Press, June 1995.

[TT93] Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages. Technical Report DIKU-report 93/15, Department of Computer Science, University of Copenhagen, 1993.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Portland, Oregon.* ACM, ACM Press, Jan. 1994.

[Wad89] P. Wadler. Theorems for free! In *Proc. Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 347–359. ACM Press, Sept. 1989.

[Wel94] J.B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Proc. Logic in Computer Science (LICS)*, 1994.

[WF92] A. Wright and M. Fagan. Soft typing and global representation optimization. Manuscript, July 1992.

Index

ϕ-redex, 49, 60
ψ-redex, 49, 60
ψ-reduction, 72
$\psi^{\rightarrow}$ $E_p^{\leftarrow}$-reduction modulo ϕ, 65
$\mathcal{C}_{p*}$, 104
F_2, 2, 10
F_2-expression, well-formed, 10
F_2 types, 10

Appel, Andrew W., 129

Bjørner, Nikolaj, 129
booleans, 101
boxed representation, 1
boxing, 1, 128
boxing analysis, 2
box annotation, 77
box values, 77
box variables, 77
Breazu-Tannen, V., 131

CAML Light, 128
Cartwright, R., 131
case expressions, 97
category theory, 99
coercions
coercions, formation rules for, 12
  coherence of, 25
  composition of, 12
  congruence of, 25
  equality of, 17
  factoring of, 30
  formally optimal, 26
  head free, 40
  identity, 12
  implementing, 90
  induced, 12
  inverse, 26
  negative, 29
  normal form, 23
  optimal, 78
  polarity of, 50
  positive, 29
  primitive, 11
  proper, 17
  reduction, 19
  representation, 2, 11, 128
coherence, 131
completions, 2, 14, 131
completions, +/−-normalized, 72
  ϕ-free canonical, 66
  ψ-free canonical, 66, 68
  $\mathcal{C}_{**}$, 89
  $\mathcal{C}_{*f}$, 89
  $\mathcal{C}_{p*}$, 89
  $\mathcal{C}_{pf}$, 89
  canonical, 64
  coherence of, 2
  coherence of, 43
  congruence of, 14, 50
  equality of, 38
  maximally boxed, 64
  minimally boxed, 64
  optimal, 2, 82
  principal, 77
conditional expressions, 97
congruence, 14, 131
Coquand, T., 131
critical pairs, 50
Curien, P., 131

datatype declarations, 99

erasure, 14
explicitly boxed F_2, 11

Fagan, M., 131
fix-point operator, 97, 98
free theorems, 46
functor, 99

Ghelli, G., 131
Girard, J.-Y., 10
Gunter, C., 131

Harper, Robert, 129
Haskell, 1, 128
head coercion free, 72
Henglein, Fritz, viii, 129, 131

integers, 101

Jones, Neil, viii

Knuth-Bendix completion, 3

language primitive, 95
Launchbury, John, 110, 128
lazy evaluation, 110
Leroy, Xavier, 2, 66, 123, 125, 128
let-expressions, 98
LISP, 128
lists, 103

Mogensen, Torben, 123
Morrisett, Greg, 129

occurrences, 135

parametricity, 104
Peterson, J., 108, 128
Peyton Jones, Simon, 110, 128
polarized expression equations, 51
polymorphic constant, 95
polymorphic typed lambda calculus, 2, 10
polytypic function, 99
Poulsen, Eigil, 2, 123, 125, 129
pre-expressions, 10
primitives, 95
    recursive, 120
principal completion, 78, 79
products, 101

R-reduction, 135
Rehof, Jakob, 131
representation analysis, 1, 128
representation graph, 77, 80
Reynolds, John C., 10, 46

Scedrov, A., 131
second-order lambda calculus, 2, 10
semantics, of F_2, 14
Shao, Zhong, 129
sinks, 80
SML, 1, 105, 128
soft typing, 131
sources, 80
stub code, 90
substitution, 135
subtype hierarchy, 33
sumtypes, 101
syntactic equality, 135

Talpin, Jean-Pierre, 130
Thiemann, Peter, 123, 125, 130
Tofte, Mads, 130
types
    boolean, 101
    boxed, 11
    erasure of, 11
    integer, 101
    list, 103
    maximally boxed, 68
    principal representations, 77
    product, 101
    recursive data, 102
    reference, 105
    reflexive, 103
    representation, 11
    signature, 14
    standard, 11
    standard constructors, 11
    unboxed, 11
type constructor, 99
type assignment, 10
type erasure, 78
type signature, 72

unboxing, 1, 128
union/find, 120

Wadler, Philip, 46
Wright, A., 131