Polymorphic Binding Time Analysis

Masters Thesis

Christian Mossin

DIKU, Department of Computer Science
University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø
Denmark
e-mail: mossin@diku.dk

July 21, 1993

Abstract

Binding time analysis has proved to be a valuable pre-analysis for partial evaluation. Until now (almost) all binding time analyses have been *monovariant*, such that binding time analysis could assign only one binding time description to each function definition. This means that if a function $f(x)$ is called once with dynamic data, no reductions on x can be performed in the body of f even when f is called with static data.

There is currently great interest in *polyvariant* binding time analysis. Gengler and Rytz [Gengler & Rytz 1992a, Gengler & Rytz 1992b] use a kind of repeated abstract interpretation which is very slow. Consel [Consel 1992] avoids repeating the analysis by collapsing closure and binding time analysis and letting control flow information depend on binding time values. Both approaches copy (functions *resp.* binding time descriptions) in the "needed" number of variants.

Recently *type inference* has attracted much attention as a program analysis framework. This framework has been very successful in binding time analysis and has led to elegant and efficient algorithms [Henglein 1991, Bondorf & Jørgensen 1993a].

Instead of copying functions to achieve polyvariant binding time analysis, we note that polyvariancy corresponds to *polymorphism* over binding time values. We present a type system for inferring binding times for a higher order, monomorphically typed language with (partially static) data structures. The system is polymorphic in binding time values and since binding time values are also used for annotations we get polymorphism in unfold/residualize. We show how to extend the system to both polymorphically and dynamically typed languages.

We present a specializer for the language, and prove correctness of the binding time analysis *w.r.t.* this specializer: the mix equation holds for this combination of binding analysis and specializer, and specialization does not go wrong on well annotated terms.

A version of algorithm $\mathcal{W}$ generating constraints (corresponding to lifts) is used as the type inference mechanism. This leads to a large number of extra binding time parameters, which we show can be reduced by adding a *constraint set* reduction mechanism. Constraint set reduction also improves runtimes significantly. The work has led to a prototype implementation, showing the strength of the polymorphic idea in practice. The implementation can handle reasonably sized programs, but some work is needed to make the implementation run faster.

Contents

Preface

Empress of Art, for thee I twine
This wreath with all too slender skill
Forgive my Muse each halting line
and for the deed accept the will!

Lewis Carrol

Motivation

In recent years, partial evaluation has improved drastically and many larger applications have shown it to be a very promising program development tool. Nevertheless, obtaining good results with partial evaluation still requires both insight and skill from the user. If it is to be used as an automatic tool in line with compilers and interpreters, it should not be necessary rewriting programs to obtain good results, and no detailed knowledge of the specific partial evaluation machinery used should be required.

One step in this direction is the introduction of *polyvariancy.* Without this, the programmer always has to keep in mind the binding times of arguments to a given user defined function. If the programmer is using a function such as `append`, he will have to consider if it is always called with arguments of the same binding time. If this is not the case, he will have to write two (or more) identical versions of `append` to achieve evaluation when possible. Thus polyvariancy relieves the user of irrelevant (with respect to the program) considerations.

Current approaches to polyvariant binding time analysis have basically been automations of the above scheme: make as many copies of each function as necessary. How many then are necessary? In the first order case the answer is simple: since every parameter can have 2 binding times (*static* and *dynamic*), a function with n parameters should be copied in 2^n versions — leading to an explosive, but finite, blowup. In the higher order case it will generally not be finite.

Instead of doing copying in advance, one can copy during binding time analysis "on demand". This leads to less extensive copying, but every time a new copy has been made, binding time analysis (and possibly *closure analysis*) has to be redone, making such an algorithm very slow [Gengler & Rytz 1992a, Gengler & Rytz 1992b]. By combining closure and binding time analysis, and by letting control flow information depend on binding time information, Consel [Consel 1992] avoids redoing the analysis. Further, instead of copying whole functions, only binding time descriptions are copied.

Our purpose is to avoid any kind of copying during binding time analysis by parameterizing each function with binding times. In this way polyvariance can be expressed by a *polymorphic* type system. Thus copying is postponed to specialization time, and totally avoided in all the cases, where no residual function is generated.

Outline

Chapter 1 contains an introduction to type inference and binding time analysis. The chapter is not intended to be a complete introduction to all aspects of these subjects, but rather a summary of important issues. Thus the reader is assumed to have some knowledge of λ–calculus, type inference and partial evaluation.

Part I deals with theoretical sides of the problem: type systems and correctness. First we present syntax and semantics for a typed higher order language with data structures. This language is sufficiently powerful to show all aspects of our polymorphic binding time analysis and simple enough to avoid tedious detail. In chapter 3, we develop a type system for performing monovariant binding time analysis in the style of Nielson and Nielson [Nielson & Nielson 1988]. The system is formulated in a novel way so it can be extended to handle polymorphic binding times. This is done in chapter 4. We then present a specializer, consuming the information generated by the binding time analysis. The specializer together with the standard semantics works as the model, *w.r.t.* which we prove correctness (soundness) in chapter 6. Chapter 7 ends part I by presenting the extensions necessary to make the system handle polymorphically typed, or dynamically typed languages.

Part II describes how to turn the inference system of Part I into a working binding time analysis. Chapter 8 reformulates the type system slightly, making it better suited for execution. In chapter 9, we present an algorithm for inferring binding time values, and in chapter 10 we present constraint set reduction rules. These reductions reduce the number of binding time parameters to each function. In chapter 11, we describe our prototype implementation and present some runtimes of the implementation, and in the following chapter we show some examples of actual annotated programs.

In chapter 13 we suggest future work. Chapter 14 concludes.

Hints for the Reader

The essential chapters for understanding the analysis are 2, 3 and 4. The specializer of chapter 5 puts these thing in perspective. Chapters 8, 9 (and to some extent 10) are important for understanding the algorithm. The proof of chapter 6 can be skipped at first reading, and similarly the standard type system extensions of chapter 7. The reader is encouraged to refer to the examples of chapter 12 and appendix D for motivation. Appendix A contains a list of some of the frequently used symbols and names with an explanation, and appendix B contains the full type system.

Acknowledgements

Without the help of many people this thesis would not have been what it is. First of all I would like to thank my engaged supervisor Fritz Henglein, whose ideas and knowledge provided me with insight and inspiration. Also I have had fruitful discussions with Lars

Birkedal, Anders Bondorf, Dirk Dusart, Jesper Jørgensen and David Sands, all at DIKU (Dirk Dussart as guest from KU Leuven). Torben Amtoft, Flemming Nielson and Jens Palsberg, all at DAIMI, helped me out with an obstacle in the proof of correctness. Charles Consel at Oregon State University explained the intuition of the polyvariance in the Schism partial evaluator. Finally I would like to thank the rest of the TOPPS-group for a friendly and inspirering atmosphere.

Chapter 1

Introduction

This chapter gives an introduction to type inference and binding time analysis – the two basic concepts of this work. The chapter is organized as follows. In section 1.1, we give a general introduction to type inference, especially the two core concepts of the system(s) we will present later: polymorphism and subtyping. Section 1.3 discusses the suitability of type inference as a program analysis principle. Section 1.4 presents binding time analysis and introduces the concept of *polyvariant* binding time analysis.

1.1 Type Systems

Type systems and type inference are studied for a variety of reasons. Here we will focus on one: a type can be considered as a set to which (the value of) some expressions belong, or equivalently as a predicate over expressions. *E.g.* type Int corresponds to a predicate over expressions telling whether an expression will evaluate to an integer or not.

A *type assertion* has the form e:t, where e is an expression and t is a type. A type assertion is said to *hold* iff it is derivable *w.r.t.* a given type system. A type system is said to be sound *w.r.t.* a given model $[\![\cdot]\!]$ iff whenever e:t is derivable in the type system, then $[\![\mathrm{e}]\!] \in [\![t]\!]$ holds in the model.

Usually an assertion holds under some assumptions on the types of free variables in e. If A is a set of such assumptions [x:t], this is written $A \vdash$e:t. As a notational convinience, we will write A(x) for the type t of x, that is if $A = A' \bigcup \{\mathrm{x} : t\}$.

A substitution ζ is a function mapping type variables τ to types t. We use $[t/\tau]$ for the function $\lambda\tau'.\mathit{if}\ \tau'{=}\tau\ \mathit{then}\ t\ \mathit{else}\ \tau'$, and $\zeta[t/\tau]$ for $\lambda\tau'.\mathit{if}\ \tau'{=}\tau\ \mathit{then}\ t\ \mathit{else}\ \zeta\tau'$. Substitutions are trivially extended to be applicable to environments, expressions *etc.*. The same notation will, without further explanation, be used for substitutions with other domains (*e.g.* sustitutions mapping program variables to expressions).

1.1.1 Lambda Calculus

The typed lambda calculus is the basis of type theory and type inference. We will briefly sketch the simply typed lambda calculus and then show how this is extended to handle polymorphism. Both systems will be introduced in an *explicit* style (Church style). This is done to introduce the notation for explicit type abstraction which we will use later. General introductions can be found in [Girard, Lafont, & Taylor 1989, Barendregt &

Dekkers 199] (while waiting for the latter, we will have to settle for [Barendregt, Geuvers, & Dekkers 1991]).

Simply typed lambda calculus

In the simply typed lambda calculus $\lambda_{\rightarrow}$-Church, terms has the following syntax:

$$\mathcal{T} ::= \mathrm{x} \mid \mathcal{T}@\mathcal{T} \mid \lambda \mathrm{x}{:}t.\mathcal{T}$$

where $\lambda\mathrm{x}{:}t$ is abstraction over terms of type t and @ denotes application. We have the following syntax of types:

$$t ::= \tau \mid t \rightarrow t$$

where τ is a *type variable*, $t \rightarrow t'$ is the type of functions mapping terms of type t to terms of type t'. The operational semantics of the calculus can be expressed by the following reduction rules:

$$\begin{array}{lll} (\beta\text{-reduction}) & (\lambda \mathrm{x}{:}t.\mathcal{T})@\mathcal{T}' \longrightarrow_{\beta} [\mathcal{T}'/\mathrm{x}]\mathcal{T} & \\ (\text{comp}) & \dfrac{\mathcal{T} \longrightarrow \mathcal{T}'}{\mathcal{C}[\mathcal{T}] \longrightarrow \mathcal{C}[\mathcal{T}']} & \text{for any context } \mathcal{C} \end{array}$$

$$\begin{array}{ll} (\text{axiom}) & A \bigcup \{\mathrm{x}{:}t\} \vdash \mathrm{x}{:}t \\ (\text{appl}) & \dfrac{A \vdash \mathcal{T}{:}t \rightarrow t' \qquad A \vdash \mathcal{T}'{:}t}{A \vdash \mathcal{T}@\mathcal{T}'{:}t'} \\ (\text{abstr}) & \dfrac{A \bigcup \{\mathrm{x}{:}t\} \vdash \mathcal{T}{:}t'}{A \vdash \lambda \mathrm{x}{:}t.\mathcal{T} : t \rightarrow t'} \end{array}$$

Figure 1.1: Type inference for $\lambda_{\rightarrow}$-Church

The type system for inferring types for $\lambda_{\rightarrow}$-Church is given in figure 1.1. A term $\mathcal{T}$ is called *well-typed* iff for some A and type t, $A \vdash \mathrm{e}{:}t$.

Second Order Lambda Calculus

The second order polymorphic lambda calculus $\lambda 2$ ($\lambda 2$-Church to be exact), has the following types:

$$t ::= \tau \mid t \rightarrow t \mid \forall \tau.t$$

and the following terms:

$$\mathcal{T} ::= \mathrm{x} \mid \mathcal{T}@\mathcal{T} \mid \lambda \mathrm{x}{:}t.\mathcal{T} \mid \mathcal{T} \diamond t \mid \Lambda t.\mathcal{T}$$

We now have two kinds of abstraction: the usual λ abstracts over terms and Λ abstracts over types. @ denotes application of λ-terms to λ-terms, while $\diamond$ denotes application to of Λ-terms to types.

We can now express the polymorphic identity function by $id = \Lambda t.\lambda \mathrm{x}{:}t.\mathrm{x}$. We apply id to a term $\mathcal{T}$, by first applying id to the type of $\mathcal{T}$ then to $\mathcal{T}$ itself.

We have the following reduction rules:

$$\begin{array}{ll}
(\beta\text{-reduction}) & (\lambda \mathrm{x}.\mathcal{T})@\mathcal{T}' \longrightarrow_\beta \mathcal{T}[\mathcal{T}'/\mathrm{x}] \\
(\beta\text{-reduction}) & (\Lambda \tau.\mathcal{T})\diamond t \longrightarrow_\beta \mathcal{T}[t/\tau] \\
(\text{comp}) & \dfrac{\mathcal{T}\longrightarrow\mathcal{T}'}{\mathcal{C}[\mathcal{T}]\longrightarrow\mathcal{C}[\mathcal{T}']} \quad \text{for any context } \mathcal{C}
\end{array}$$

The type rules of $\lambda 2$ are given in figure 1.2. The side condition $(*)$ of rule (gen) is that τ must not appear free in A.

$$\begin{array}{cc}
(\text{axiom}) & A\bigcup\{\mathrm{x}{:}t\}\vdash \mathrm{x}{:}t \\[1ex]
(\text{appl}) & \dfrac{A\vdash\mathcal{T}{:}t\rightarrow t' \qquad A\vdash\mathcal{T}'{:}t}{A\vdash\mathcal{T}@\mathcal{T}'{:}t'} \\[1ex]
(\text{abstr}) & \dfrac{A\bigcup\{\mathrm{x}{:}t\}\vdash\mathcal{T}{:}t'}{A\vdash\lambda \mathrm{x}.\mathcal{T}{:}t\rightarrow t'} \\[1ex]
(\text{inst}) & \dfrac{A\vdash\mathcal{T}{:}\forall\tau.t}{A\vdash\mathcal{T}\diamond t'{:}[t'/\tau]t} \\[1ex]
(\text{gen}) & \dfrac{A\vdash\mathcal{T}{:}t}{A\vdash\Lambda\tau.\mathcal{T}{:}\forall\tau.t}(*)
\end{array}$$

Figure 1.2: Type inference for $\lambda 2$

1.1.2 ML Polymorphism

Above we have described a type type system for the polymorphic lambda calculus. Since ML and related languages have the lambda calculus as basis, the extension to these languages is pretty straight forward. However, due to decidability problems polymorphism is usually restricted to *let–polymorphism*. This means that only `let`–bound expressions can poses a polymorphic type; we consider global definitions as a big `let`(`rec`).

The idea is to divide the above types into types t and *typeschemes* σ:

$$\begin{array}{l}
\sigma ::= t \mid \forall\tau.\sigma \\
t ::= \tau \mid t\rightarrow t
\end{array}$$

A type system for a `let`-polymorphic lambda calculus is given in figure 1.3. The sidecondition $(*)$ is as before: τ must not appear free in A. The second order lambda calculus has been extended with a `let` construct with the following semantics:

$$(\texttt{let}\text{-reduction}) \quad \texttt{let x} = \mathcal{T} \texttt{ in } \mathcal{T}' \longrightarrow_{\texttt{let}} \mathcal{T}'[\mathcal{T}/\mathrm{x}]$$

Note that type abstraction is usually implicit (Curry-style) in ML-like languages.

$$\text{(axiom)}\quad A\bigcup\{\mathrm{x}{:}\sigma\}\vdash\mathrm{x}{:}\sigma$$

$$\text{(appl)}\quad \frac{A\vdash\mathcal{T}{:}t\to t' \qquad A\vdash\mathcal{T}'{:}t}{A\vdash\mathcal{T}@\mathcal{T}'{:}t'}$$

$$\text{(abstr)}\quad \frac{A\bigcup\{\mathrm{x}{:}t\}\vdash\mathcal{T}{:}t'}{A\vdash\lambda\mathrm{x}.\mathcal{T}{:}t\to t'}$$

$$\text{(inst)}\quad \frac{A\vdash\mathcal{T}{:}\forall\tau.\sigma}{A\vdash\mathcal{T}\diamond t'{:}[t'/\tau]\sigma}$$

$$\text{(gen)}\quad \frac{A\vdash\mathcal{T}{:}\sigma}{A\vdash\Lambda\tau.\mathcal{T}{:}\forall\tau.\sigma}(*)$$

$$\text{(let)}\quad \frac{A\vdash\mathcal{T}{:}\sigma \qquad A\bigcup\{\mathrm{x}{:}\sigma\}\vdash\mathcal{T}'{:}t}{A\vdash\texttt{let}\ \mathrm{x}=\mathcal{T}\,\texttt{in}\ \mathcal{T}'{:}t}$$

Figure 1.3: Type inference for a `let`-polymorphic lambda calculus

1.1.3 Subtypes

In many languages we wish to allow *subtypes*; *e.g.* it is customary to allow an integer to appear everywhere a real number is expected. We assume some ordering $\leq$ on types. Then, if a term $\mathcal{T}$ has some type t and $t \leq t'$, we can infer that $\mathcal{T}$ has type t'. A rule for such *implicit* subtyping is shown in figure 1.4

$$\text{(coerce)}\quad \frac{A\vdash\mathcal{T}{:}t \qquad t\leq t'}{A\vdash\mathcal{T}{:}t'}$$

Figure 1.4: Implicit subtyping

In addition to the (coerce) rule, we of course need a number of constant rules (*e.g.* Int$\leq$Real).

Later in this work, we are going to use *explicit* subtyping. Here we can view $t \leq t'$ as the type of a coercion c. The (coerce) rule now states, that if $\mathcal{T}$ has type t and c has type $t \leq t'$, then c applied to $\mathcal{T}$ written [c]e, has type t'. This rule is shown in figure 1.5.

$$\text{(coerce)}\quad \frac{A\vdash\mathcal{T}{:}t \qquad \mathrm{c}{:}t\leq t'}{A\vdash[\mathrm{c}]\mathcal{T}{:}t'}$$

Figure 1.5: Explicit subtyping

Instead of the the name c of type $t \leq t'$, we will often write $t{\leadsto}t'$. Though this is clearly redundant, it can be convenient to be able to extract the type from the name of the coercion.

In figure 1.6, we show two supplementary rules. Application of coercions is transitive, this is formulated in rule (trans). Coercions are often also allowed on function types. *E.g.* assume a function of type Real$\to$Int. This function is also applicable to integers, and the result can be used where a real number is expected. Thus the type is coercible to Int→Real. Note that the composition rule for creating such coercions on functions is contravariant. This is the (arrow) rule in figure 1.6.

$$\text{(trans)}\quad \frac{\mathrm{c}_1 : t \leq t' \quad \mathrm{c}_2 : t' \leq t''}{\mathrm{c}_1;\mathrm{c}_2 : t \leq t''}$$

$$\text{(arrow)}\quad \frac{\mathrm{c}_1 : {t_1}' \leq t_1 \quad \mathrm{c}_2 : t_2 \leq {t_2}'}{\rightarrow^{\mathrm{c}_1\mathrm{c}_2} : t_1 \rightarrow t_2 \leq {t_1}' \rightarrow {t_2}'}$$

Figure 1.6: Subtyping rules

Other rules might be added, such as reflexivity, but we assume all other rules to be captured by the constant coercion rules $t_i \leq t_k$ *resp.* c:$t_i \leq t_k$ for the implicit *resp.* explicit subtype systems.

It is customary to extend type assertions to include constraint assumptions. Such assertions are written $A,C \vdash \mathrm{e}:t$, where C is a set of constraints $t \leq t'$. The (coerce), (trans) and (arrow) rules are then written as in figure 1.7, where we also add a new rule (axiom) stating that under assumption that $t \leq t'$ holds $t \leq t'$ holds. We could choose to let C be an environment of named coercions, but in figure 1.7, we have chosen not to have names in C, but keep the explicit coercion, now with $t \rightsquigarrow t'$ as names. It is important to note, that $t \rightsquigarrow t'$ is merely a convenient name for a coercion of type $t \leq t'$.

$$\text{(coerce)}\quad \frac{A,C \vdash \mathcal{T}:t \qquad \mathrm{c}:t \leq t'}{A,C \vdash [t \rightsquigarrow t']\mathcal{T}:t'}$$

$$\text{(axiom)}\quad C \bigcup \{t \leq t'\} \vdash t \leq t'$$

$$\text{(trans)}\quad \frac{C \vdash t \leq t' \quad C \vdash t' \leq t''}{C \vdash t \leq t''}$$

$$\text{(arrow)}\quad \frac{C \vdash {t_1}' \leq t_1 \quad C \vdash t_2 \leq {t_2}'}{C \vdash t_1 \rightarrow t_2 \leq {t_1}' \rightarrow {t_2}'}$$

Figure 1.7: Subtyping with Constraint set

The (const) rules for this subtyping system are:

$$\text{(const)}\quad C \vdash t_i \leq t_k$$

1.2 Type Inference

Above we specified a number of type systems specifying when a term is well annotated. These systems consist of a number of equations or relations. From the relations we can easily devise algorithms answering the question: is it true, that term $\mathcal{T}$ has type t under type assumptions A? This problem we call the *type checking* problem. We are often more interested in the *type inference* problem: Given type assumptions A, what is the type of term $\mathcal{T}$? This problem is often much more difficult to answer: for many interesting type systems, the type inference problem is actually unsolvable. It has been shown, that ML typing is DEXP-time complete [Mairson 1990].

In an explicitly typed language with subtypes, in principle, the user should supply the coercions. This is done in languages such as C, but often we would expect the type inference to infer these.

If we allow assumptions on constraints, and we have proved assertion A,C⊢e:t to hold, we wish to check whether C is *solvable*. This amounts to checking if there exists a substitution ζ, such that every constraint in ζC is provable using only rule (const) (and (reflex) if this rule is not included in (const)).

[Jones 1987] contains an easy and pragmatic introduction to type inference, while [Cardelli 1985] is an introduction to the theoretical side. Classical articles include [Milner 1978, Damas & Milner 1982, Mycroft 1984].

Type inference for subtyping systems was first studied by Mitchell [Mitchell 1984]. His type inference leads to constraint sets of size proportional to the length of the program. Fuh and Mishra carried on this work [Fuh & Mishra 1988], and in [Fuh & Mishra 1989] algorithms was given for reducing the size of these sets.

1.3 Type Inference as a Program Analysis Principle

In the initial discussion of the last section, we introduced types as representing predicates over terms. This viewpoint makes it clear that type inference need not be restricted to standard types (Int, Bool, Int→Bool *etc.*), but that the principle can be used to infer every imaginable kind of information.

As a program analysis principle, type inference competes with abstract interpretation, projection analysis and others. I believe that it is generally agreed that there is nothing that one principle is capable of that the others cannot do. It should however not be a random choice when a programmer wants to specify a program analysis.

In many applications there are several advantages to type inference over other principles:

- In type inference, specification (a type system) and algorithm are separated. This gives elegant specifications, and makes reasoning about both specification and algorithms easier. In abstract interpretation specification and algorithm is the same.

- Many analyses need much the same flow information as standard type inference. This makes us able to draw from the vast knowledge gathered by research in standard type inference.

- Since there are strong similarities between standard and non-standard type systems, many theoretical results from standard type inference (and thereby the well studied typed lambda calculus) can be used almost immediately. This holds for *e.g.* *subject reduction*, *Church-Rosser theorem* and others. In addition, the concept of correctness is usually (depending on the type system) well understood.

- For subtype systems (which seem important in program analysis) recent years have led to very fast algorithms by dividing the analysis into type inference ($\approx$ *constraint set generation*) and *constraint set solving.* For constraint set solving, fast graph algorithms can often be employed.

Currently the type inference approach seem to be less practicable if an analysis needs to model a state/store, but ongoing research in this direction (*e.g.* handling references in ML) might improve this. Such problems have been studied more in abstract interpretation frameworks, but there is no reason, why this should be an inherently better approach. *Path analysis* seem like a "natural" approach to these problems.

Finally it is important to notice that while there is a connection between abstract interpretation and denotational semantics, there is a similar relationship between type inference and operational semantics.

1.4 Binding time analysis

To explain *binding time analysis*, we need to explain *partial evaluation* — the main area of application. Partial evaluation is the computer realization of Kleene's S-m-n–theorem stating that if function $p(x, y)$ is computable, then so is function $p_x(y)$ for all x (with the obvious interpretation).

A partial evaluator *mix* takes a program p and one of p's inputs x, and computes a *non–trivial* p_x, such that $\forall x, y : p(x, y) = p_x(y)$. Futamura [Futamura 1971] discovered that by self-application of the partial evaluator, a compiler generator — generating compilers from interpreters — is obtained.

In [Bondorf, Jones, Mogensen, & Sestoft 1988] the need for a separate *binding time analysis* in order to perform self-applicable partial evaluation efficiently is discussed. As a pre-phase to the *specializer* (which does the actual partial evaluation), binding time analysis annotates program expressions as being either evaluable at specialization time (*static* expressions) or as evaluable at run-time (*dynamic* expressions). The goal of the binding time analysis is to do this separation *safely*, such that every static expression depends solely on data known at partial evaluation time, and *optimally*, such that every expression that can be evaluated on the basis of such data is annotated as static. A "perfect" binding time separation is of course not generally possible, since the binding time analysis cannot catch the intent of the programmer (or even the semantics of the program). Throughout the report we will use the symbol S for static, and D for dynamic.

Binding time analysis has been used ever since the first self-applicable partial evaluator Mix [Jones, Sestoft, & Søndergaard 1985] where the analysis had to be performed manually. Later versions included automatic binding time analysis [Jones, Sestoft, & Søndergaard 1989]. The partial evaluator Similix treats a higher order subset of Scheme with primitive operators and global variables. [Bondorf 1991] describes the binding time analysis used in Similix.

[Mogensen 1988] shows how binding time analysis can handle *partially static* structures *e.g.* a list of static structure but with dynamic elements.

Since good binding time separation is crucial for good partial evaluation, there has been several attempt to improve binding time analyzers. *E.g.* it was suggested that cps–converting the source program before binding time analysis would improve binding times, but this was shown in [Bondorf 1992] to correspond to rewriting the specializer in cps-style.

In spite of these improvements almost all applications of partial evaluation use some amount of manual binding time improvement (manually rewriting the program in order to get better binding time separation). This holds for *e.g.* [Consel & Danvy 1989], [Jørgensen 1990], [Jørgensen 1992] and [Mossin 1993].

1.4.1 Polyvariant Binding Time analysis

Most current binding time analyses are *monovariant*. This means that every function definition $f(x, y)$ gets exactly one binding time description. So if f is called one place with a static first argument and dynamic second, and another place with dynamic first and static second argument, the binding time description of f becomes $(\mathsf{D},\mathsf{D})\rightarrow\mathsf{D}$, and no computation on the parameters can be performed at partial evaluation time.

Gengler and Rytz attempt to remedy this problem in [Gengler & Rytz 1992a, Gengler & Rytz 1992b] (the latter also handles partially static structures). Their approach is to do the usual abstract interpretation and note whenever a function is called with both static and dynamic argument(s). If this is the case, a twin version of the function is created — one being called with static and one with dynamic arguments — and the binding time analysis (and closure analysis) is redone. This is a very pragmatic approach to the problem, close to the way a user would do it manually. It is, however, not clear if all kinds of polymorphism can be handled, and their implementation is (according to themselves) very slow due to the redoing of binding time analysis. Consel [Consel 1992] does closure analysis and binding time analysis at once, so no redoing is necessary. Further only the binding time description of each function is copied. De Niel [Niel 1993] uses a projection based approach to binding time analysis. The language involved is first order (and typed in contrast to the former) so there is no need for closure analysis. Consequently binding time analysis does not need to be redone. Like Consel only descriptions are copied. These three systems are discussed further and compared to the system developed in this work in section 14.1.

1.4.2 Using Type Inference

Common to the binding time analyses mentioned above is, that they are based on variants of abstract interpretation (some analyses are not traditional abstract evaluation, such as the *Partial Equivalence Relation* approach of [Hunt & Sands 1991], and the projection based approach of [Launchbury 1990, Niel 1993, Davis 1993]).

Attempts to use type inference for binding time analysis originates in partial evaluators for the typed lambda calculus [Nielson & Nielson 1988], and later the untyped lambda calculus [Gomard 1989, Gomard 1991b, Jones *et al.* 1990]. The work of Gomard was extended to handle Scheme in [Andersen & Mossin 1990], but both the algorithm used and the implementation were quite slow. Gomard's monovariant system is shown in figure 1.8 (we use our S for his *base* and D for *code*).

The last rule (the `lift` rule) is interesting: it denotes an explicit coercion from S to D and is necessary in many applications of partial evaluation to get non–trivial specialization. The `lift` corresponds to emitting residual code representing the result of evaluating an expression.

$$\frac{A(\mathrm{x}) = T}{A \vdash \mathrm{x} : T}$$

$$\frac{A \cup \{\mathrm{x} : T_2\} \vdash \mathrm{e} : T_1}{A \vdash \lambda \mathrm{x}.\mathrm{e} : T_2 \rightarrow T_1} \qquad \frac{A \cup \{\mathrm{x} : \mathsf{D}\} \vdash \mathrm{e} : \mathsf{D}}{A \vdash \underline{\lambda} \mathrm{x}.\mathrm{e} : \mathsf{D}}$$

$$\frac{A \vdash \mathrm{e}_1 : T_2 \rightarrow T_1 \qquad A \vdash \mathrm{e}_2 : T_2}{A \vdash \mathrm{e}_1 @ \mathrm{e}_2 : T_1} \qquad \frac{A \vdash \mathrm{e}_1 : \mathsf{D} \qquad A \vdash \mathrm{e}_2 : \mathsf{D}}{A \vdash \mathrm{e}_1 \underline{@} \mathrm{e}_2 : \mathsf{D}}$$

$$\frac{A \vdash \mathrm{e}_1 : \mathsf{S} \quad A \vdash \mathrm{e}_2 : T \quad A \vdash \mathrm{e}_3 : T}{A \vdash \texttt{if } \mathrm{e}_1 \texttt{ then } \mathrm{e}_2 \texttt{ else } \mathrm{e}_3 : T} \qquad \frac{A \vdash \mathrm{e}_1 : \mathsf{D} \quad A \vdash \mathrm{e}_2 : \mathsf{D} \quad A \vdash \mathrm{e}_3 : \mathsf{D}}{A \vdash \underline{\texttt{if}}\ \mathrm{e}_1\ \underline{\texttt{then}}\ \mathrm{e}_2\ \underline{\texttt{else}}\ \mathrm{e}_3 : \mathsf{D}}$$

$$A \vdash \texttt{const}\ \mathrm{c} : \mathsf{S} \qquad A \vdash \underline{\texttt{const}}\ \mathrm{c} : \mathsf{D}$$

$$\frac{A \vdash \mathrm{e} : (T_1 \rightarrow T_2) \rightarrow (T_1 \rightarrow T_2)}{A \vdash \texttt{fix}\ \mathrm{e} : T_1 \rightarrow T_2} \qquad \frac{A \vdash \mathrm{e} : \mathsf{D}}{A \vdash \underline{\texttt{fix}}\ \mathrm{e} : \mathsf{D}}$$

$$\frac{A \vdash \mathrm{e} : \mathsf{S}}{A \vdash \texttt{lift}\ \mathrm{e} : \mathsf{D}}$$

Figure 1.8: Type Rules for Binding Time Analysis

A fast practical algorithm for doing such binding time analysis was devised by Henglein in [Henglein 1991] and applied in practice to Scheme in [Bondorf & Jørgensen 1993a, Bondorf & Jørgensen 1993b].

A nice approach to doing binding time analysis can be found in [Solberg, Nielson, & Nielson 1992] (also based on [Henglein 1991]).

We see that if we consider binding times as types, monovariance as defined above corresponds to *monomorphism*. Thus instead of following the scheme of Gengler and Rytz or Consel, polyvariance can be achieved by making the type system *polymorphic* in the binding times. We will hereafter use the idioms *polymorphic binding time analysis* and *polyvariant binding time analysis* interchangeably.

1.4.3 Correctness of Binding Time Analysis

Correctness of a type system for binding time analysis must be in terms of some model of the types. We can choose to follow the lines of *soundness* in type systems, and let the model be the semantics of the evaluator. The evaluator consuming binding time annotations is the specializer. To say that the residual program produced by specialization is correct is stated by the *mix-equation*

$$p(d_1, d_2) = \mathit{mix}(p, d_1)(d_2)$$

stating that when evaluating a specialized program, we get the same result as if we had evaluated the original program. This can be depicted as in figure 1.9.

We will actually formulate the main correctness theorem (soundness theorem) a little differently, but the above will follow as an immediate corollary. Further we will prove, that "specialization does not go wrong" on well-annotated program.

$$\begin{tikzcd}
p_{src} \arrow[r, "bta"] \arrow[drr, bend right, "\longrightarrow_{std}"'] & p_{ann} \arrow[r, "\longrightarrow_{spec}"] & p_{spec} \arrow[d, "\longrightarrow_{std}"] \\
 & & value
\end{tikzcd}$$

Figure 1.9: Correctness of Binding Time Analysis

In this formulation, correctness depends on (the semantics of) a specific specializer. This is reasonable, since our specializer will be “natural” in the sense that it will naively follow the binding time annotations. It is however difficult to formally argue that a specializer is “natural”, and correctness can be proved in other ways — for a further discussion see chapter 6.

1.5 Dynamic Typing

Gomard [Gomard 1990] shows, that there is a very strong relationship between the problem of *dynamic typing* and binding time analysis (Gomard coins dynamic typing *partial typing*), and we have in many ways been inspired by work in this field. Dynamic typing aims at typing an untyped program as much as possible, such that as many typechecks as possible can be done at compile time, and as few as possible have to be deferred to run time. This indeed has the same flavor as binding time analysis, where as many reductions as possible should be performed at partial evaluation time, and as few as possible deferred to run time. The works that have inspired us the most are [Thatte 1988, Henglein 1992].

Part I

Type System

Chapter 2

Language

This chapter describes the extended lambda calculus, with which we will work throughout the report. Section 2.1 describes the syntax of the language and section 2.2 describes the (standard) semantics. The presentation of the (standard) type system is deferred to chapter 3.

2.1 Syntax

The language we will use throughout this paper is an extended lambda calculus á la Curry with variables x, constants `true`, `false`, numbers n, abstractions, application (@), `if`, primitive binary operators (`op`), fix–point operator, `let`, and pairing (`pair`) with selectors `fst` and `snd`. The syntax of the language is shown in figure 2.1.

```
e  ::=  true | false | n | x
     |  if e then e else e
     |  λx.e | e@e
     |  e op e | fix f x.e | let x = e in e
     |  nil | pair(e,e) | fst e | snd e
```

Figure 2.1: Language

By choosing this language, we include the computational power of "real" functional programming languages, without having included the "syntactic sugar". By computational power, we think of our inclusion of data structures and an explicit fix-point operator. By syntactic sugar we mean global definitions (which can be mechanically replaced by `let` and `fix`), `case`–statements, user defined constructors *etc.* By choosing this language we thus have a foundation broad enough to ensure that our ideas can be generalized to any functional language, but simple enough to save us from unnecessary details and tedious work.

2.2 Semantics

The language has *call by value* semantics and static scoping. The semantics are presented as an operational semantics [Plotkin 1981, Kahn 1987]. We call this semantics *standard semantics*, since we will later (in chapter 5) give the semantics of a specializer. The sets of interest are defined by

$$\begin{array}{lcl}
\mathit{Val}_{std} & = & \mathit{Const} \bigcup \mathit{Funval}_{std} \bigcup \mathit{RecFunval}_{std} \bigcup (\mathit{Val}_{std} \times \mathit{Val}_{std}) \\
\mathit{Funval}_{std} & = & \mathit{Var} \times \mathit{Exp}_{std} \times \mathit{Varenv}_{std} \\
\mathit{RecFunval}_{std} & = & \mathit{Var} \times \mathit{Var} \times \mathit{Exp}_{std} \times \mathit{Varenv}_{std} \\
\mathit{Varenv}_{std} & = & \mathit{Var} \rightarrow \mathit{Val}_{std}
\end{array}$$

Closures in Funval_{std} are triples written as $[\![\lambda\mathrm{x.e}]\!]\rho$ [Kahn 1987]. Recursive closures in $\mathit{RecFunval}_{std}$ are written $[\![\mathtt{fix}\ \mathrm{f\ x.e}]\!]\rho$. We use v to range over values in Val_{std}, and $\rho \in \mathit{Varenv}_{std}$ for environments. $\rho[\mathrm{x}\mapsto v]$ is a shorthand for $\lambda\mathrm{id}\!:\mathit{Var}.\mathit{if}\ \mathrm{id}{=}\mathrm{x}\ \mathit{then}\ v\ \mathit{else}\ \rho(\mathrm{x})$. The construct $\sqcup\vdash\sqcup\longrightarrow_{std}\sqcup \in \mathit{Varenv}_{std} \times \mathit{Exp}_{std} \times \mathit{Val}_{std}$ is a relation over expression and values, such that $\rho\vdash\mathrm{e}\longrightarrow_{std}v$ states that given environment ρ, e evaluates to v. Exp_{std} is the set of unannotated expressions (given in figure 2.1).

The semantics of our language is given in figure 2.2.

2.2.1 The Semantic Rules

We will briefly describe the semantic rules of figure 2.2. (const) states that constant expressions evaluate to their value ($\in \mathit{Val}_{std}$). (var) states that the value of a variable is looked up in the environment. (if): depending on the value to which the conditional e evaluates to, the `then`- or `else`-branch is chosen. The result of evaluating `if` is the same as evaluating this branch.

A lambda-abstraction evaluates to a closure which is a triple of a the lambda bound variable x, the lambda bound expression e and the environment. The expression e is evaluated in this environment (plus a binding for x) when the closure is applied. At application time the argument e'' is evaluated to a value v. The function is evaluated to a closure, and the expression in the closure is evaluated in its own environment, where the lambda-bound parameter x is mapped to v (appl).

In rule (op), the argument of primitive operator `op` are evaluated. Operators only work on base values. We let $\otimes^{\mathtt{op}}$ denote the semantic meaning of `op`, and use this for evaluating the result. In rule (fix) `fix` f x.e evaluates to a recursive closure. Application of such closures (rule (rec-appl)) works by unfolding the closure once.

The rules (pair), (first) and (second) should contain no surprises — just note that `pair` is strict. (let) evaluates the `let`–bound expression and evaluates the expression in the new updated environment.

2.3 Types in the Language

Above the language is specified as being untyped. Through most of the report (all except chapter 7) we will impose a monomorphic type system on the language. The binding (const) $\rho\vdash\texttt{true}\longrightarrow_{std}\texttt{true}$ $\quad\rho\vdash\texttt{false}\longrightarrow_{std}\texttt{false}$ $\quad\rho\vdash n\longrightarrow_{std} n$

(var) $\rho[\mathrm{x}\mapsto v]\vdash\mathrm{x}\longrightarrow_{std} v$

(if-true) $$\frac{\rho\vdash\mathrm{e}\longrightarrow_{std}\texttt{true}\quad \rho\vdash\mathrm{e}'\longrightarrow_{std} v'}{\rho\vdash\texttt{if}\ \mathrm{e}\ \texttt{then}\ \mathrm{e}'\texttt{else}\ \mathrm{e}''\longrightarrow_{std} v'}$$

(if-false) $$\frac{\rho\vdash\mathrm{e}\longrightarrow_{std}\texttt{false}\quad \rho\vdash\mathrm{e}''\longrightarrow_{std} v''}{\rho\vdash\texttt{if}\ \mathrm{e}\ \texttt{then}\ \mathrm{e}'\texttt{else}\ \mathrm{e}''\longrightarrow_{std} v''}$$

(abstr) $$\frac{}{\rho\vdash\lambda\mathrm{x.e}\longrightarrow_{std}[\![\lambda\mathrm{x.e}]\!]\rho}$$

(fix) $$\frac{}{\rho\vdash\texttt{fix}\ \mathrm{f\ x.e}\longrightarrow_{std}[\![\texttt{fix}\ \mathrm{f\ x.e}]\!]\rho}$$

(appl) $$\frac{\rho\vdash\mathrm{e}\longrightarrow_{std}[\![\lambda\mathrm{x.e}']\!]\rho'\quad \rho\vdash\mathrm{e}''\longrightarrow_{std} v\quad \rho'[\mathrm{x}\mapsto v]\vdash\mathrm{e}'\longrightarrow_{std} v'}{\rho\vdash\mathrm{e@e}''\longrightarrow_{std} v'}$$

(rec-appl) $$\frac{\begin{array}{l}\rho\vdash\mathrm{e}\longrightarrow_{std}[\![\texttt{fix}\ \mathrm{f\ x.e}']\!]\rho'\\ \rho\vdash\mathrm{e}''\longrightarrow_{std} v\quad \rho'[\mathrm{f}\mapsto[\![\texttt{fix}\ \mathrm{f\ x.e}']\!]\rho',\mathrm{x}\mapsto v]\vdash\mathrm{e}'\longrightarrow_{std} v'\end{array}}{\rho\vdash\mathrm{e@e}''\longrightarrow_{std} v'}$$

(op) $$\frac{\rho\vdash\mathrm{e}\longrightarrow_{std} v\quad \rho\vdash\mathrm{e}'\longrightarrow_{std} v'\quad v\otimes^{\texttt{op}} v' = v''}{\rho\vdash\mathrm{e}\ \texttt{op}\ \mathrm{e}'\longrightarrow_{std} v''}$$

(pair) $$\frac{\rho\vdash\mathrm{e}\longrightarrow_{std} v\quad \rho\vdash\mathrm{e}'\longrightarrow_{std} v'}{\rho\vdash\texttt{pair}(\mathrm{e,e}')\longrightarrow_{std}(v,v')}$$

(first) $$\frac{\rho\vdash\mathrm{e}\longrightarrow_{std}(v,v')}{\rho\vdash\texttt{fst}\ \mathrm{e}\longrightarrow_{std} v}$$

(second) $$\frac{\rho\vdash\mathrm{e}\longrightarrow_{std}(v,v')}{\rho\vdash\texttt{snd}\ \mathrm{e}\longrightarrow_{std} v'}$$

(let) $$\frac{\rho\vdash\mathrm{e}'\longrightarrow_{std} v'\quad \rho[\mathrm{x}\mapsto v']\vdash\mathrm{e}\longrightarrow_{std} v}{\rho\vdash\texttt{let}\ \mathrm{x} = \mathrm{e}'\texttt{in}\ \mathrm{e}\longrightarrow_{std} v}$$

Figure 2.2: Standard Semantics

time system we are about to present, will rely on the presence of some standard type system. We choose a monomorphic system to avoid extra problems from more sophisticated standard type systems. Chapters 7.1 and 7.2 are devoted to showing that our binding time analysis is not restricted to monomorphically typed languages, but also works with polymorphically and dynamically typed languages. We will not give the standard type system for the language here, since it will be an integral part of the binding time analysis type system presented in chapters 3 and 4.

Chapter 3

Monomorphic Binding Time Analysis

In this chapter an inference system for doing binding time analysis in a monomorphically typed version of the language defined in chapter 2 is developed. Like in [Nielson & Nielson 1988] but in contrast to other type systems for binding time analysis, the standard types are an integral part of the binding time type system. Our system differs from other systems in two important ways: 1) Binding time values are attached to the standard type values and constructors (again like [Nielson & Nielson 1988]), 2) The "levels" of a two-level lambda calculus [Nielson & Nielson 1988, Gomard 1989] are replaced by annotations being binding time values themselves. This means that one rule is sufficient for each syntactical construct. That the annotations are binding time values (types), is important when we turn to polyvariancy, since it enables us to abstract over unfold/residualize.

It is convenient to assume that the standard types are already present (that standard type inference has been performed), and thus binding time analysis will be the problem of attaching binding time values to the standard types.

3.1 The Basic Type System

A *compound type* type κ has two components. The second component is always a binding time b. This can be either S (*static*) or D (*dynamic*). The first component of κ is either a standard base type (Bool or Int), a product of compound types κ or a function type from compound types κ to compound types κ:

$$b ::= \mathsf{S} \mid \mathsf{D}$$
$$\kappa ::= (\mathrm{Int}{,}b) \mid (\mathrm{Bool}{,}b) \mid (\kappa \to \kappa{,}b) \mid (\kappa \times \kappa{,}b)$$

We will use b as (meta) variable ranging over S and D (and later variables β), κ to range over compound types and t to range over Int, Bool, $\kappa \to \kappa$ and $\kappa \times \kappa$ (the first component of a compound type κ).

Compound types can be viewed as a refinement of a standard type, where binding time values are attached to every standard base type and every standard type constructor ($\to$ and $\times$).

We use binding time values b as annotations of expressions. These replace the usual notation of two-level expressions in a trivial way: non-underlined expressions in a two level syntax correspond to an expression annotated with S, and an underlined expression corresponds to an expression annotated with D. Thus so far it can be seen merely as a change in notation. The change, however, is important since it — together with our notation for types — enables us to express binding time analysis with just one rule for each operator, and later to introduce binding time variables as annotations and abstract over these. The syntax of two-level expressions (*Exp*$_{spec}$) is:

$$\begin{array}{lll} \mathrm{e} & ::= & \texttt{true} \mid \texttt{false} \mid n \mid \mathrm{x} \\ & \mid & \texttt{if}^b\ \mathrm{e}\ \texttt{then}\ \mathrm{e}\ \texttt{else}\ \mathrm{e} \\ & \mid & \lambda^b\mathrm{x.e} \mid \mathrm{e@}^b\mathrm{e} \\ & \mid & \mathrm{e}\ \texttt{op}^b\ \mathrm{e} \mid \texttt{fix}^b\ \mathrm{f\ x.e} \mid \mathrm{let\ x}\ \texttt{=}\ \mathrm{e}\ \texttt{in}\ \mathrm{e} \\ & \mid & \texttt{pair}^b\mathrm{(e,e)} \mid \texttt{fst}^b\ \mathrm{e} \mid \texttt{snd}^b\ \mathrm{e} \end{array}$$

Before presenting the inference system, we give a few examples:

Example 3.1

If an expression e has type (Int,S) it means that the specializer can evaluate e to an integer. If it has type (Int,D) it will evaluate to a residual expression of type integer. □(Ex.3.1)

Example 3.2

Suppose expression e has type ((Int,S)→(Bool,S),S). The second component belongs to the →-operator, and means that e can be applied at specialization time. (Int,S) means that it will take static integers as argument, and (Bool,S) means that it will return a static boolean. Similarly, if e has type ((Int,D)→(Bool,D),S), e can be applied (since the →-operator is static) to dynamic integers returning dynamic booleans. □(Ex.3.2)

In earlier type systems (such as [Gomard 1989, Andersen & Mossin 1990, Henglein 1991]), the last type of example 3.2 would have been written D → D. Such a type carries both the information that the expression is a function and that it can be applied at specialization time (otherwise the type would just have been D) besides the obvious information that the expression maps dynamic arguments to dynamic results. In our formulation all this information has been made explicit.

A *type environment* A will denote a set of type assumptions mapping variables x ∈ *Var* to compound types κ.

We take the assertion A⊢e:κ to mean that given type assumptions A, expression e ∈ *Exp*$_{spec}$ is well-annotated and of type κ.

We see that there exists compound types κ where no combination of type environment A and expression e exists such that A⊢e:κ. If the binding time value attached to a type constructor is D, then the binding time value attached to every type below the type constructor must be D. *E.g.* no expression exists with the type ((Bool,S)→(Bool,S),D).

Figure 3.1 shows the monovariant binding time system for our language.

$$\text{(const)}\quad A\vdash\texttt{true}:(\text{Bool},\mathsf{S})\qquad A\vdash\texttt{false}:(\text{Bool},\mathsf{S})\qquad A\vdash n:(\text{Int},\mathsf{S})$$

$$\text{(var)}\quad A\bigcup\{\text{x}:\sigma\}\vdash\text{x}:\sigma$$

$$\text{(if)}\quad \frac{A\vdash\text{e}:(\text{Bool},b)\quad A\vdash\text{e}':(t,b')\quad A\vdash\text{e}'':(t,b')}{A\vdash\texttt{if}^{b}\ \text{e}\ \texttt{then}\ \text{e}'\ \texttt{else}\ \text{e}'':\ (t,b')}(b\leq b')$$

$$\text{(abstr)}\quad \frac{A\bigcup\{\text{x}:(t,b)\}\vdash\text{e}:(t',b')}{A\vdash\lambda^{b''}\text{x.e}:\ ((t,b)\rightarrow(t',b'),b'')}\ (b''\leq b, b''\leq b')$$

$$\text{(appl)}\quad \frac{A\vdash\text{e}:((t,b)\rightarrow(t',b'),b'')\quad A\vdash\text{e}':(t,b)}{A\vdash\text{e}@^{b''}\text{e}':\ (t',b')}\ (b''\leq b, b''\leq b')$$

$$\text{(op)}\quad \frac{A\vdash\text{e}:(t,b)\quad A\vdash\text{e}':(t',b)\quad \mathcal{P}(\texttt{op})=t\times t'\rightarrow t''}{A\vdash\text{e}\ \texttt{op}^{b}\text{e}':\ (t'',b)}$$

$$\text{(fix)}\quad \frac{A\bigcup\{\text{x}:(t,b),\ \text{f}:((t,b)\rightarrow(t',b'),b'')\}\vdash\text{e}:(t',b')}{A\vdash\texttt{fix}^{b''}\text{f x.e}:\ ((t,b)\rightarrow(t',b'),b'')}\ (b''\leq b, b''\leq b')$$

$$\text{(pair)}\quad \frac{A\vdash\text{e}:(t,b)\quad A\vdash\text{e}':(t',b')}{A\vdash\texttt{pair}^{b''}(\text{e},\text{e}'):\ ((t,b)\times(t',b'),b'')}\ (b''\leq b\wedge b''\leq b')$$

$$\text{(first)}\quad \frac{A\vdash\text{e}:\ ((t,b)\times(t',b'),b'')}{A\vdash\texttt{fst}^{b''}\text{e}:\ (t,b)}\ (b''\leq b\wedge b''\leq b')$$

$$\text{(second)}\quad \frac{A\vdash\text{e}:\ ((t,b)\times(t',b'),b'')}{A\vdash\texttt{snd}^{b''}\text{e}:\ (t',b')}\ (b''\leq b\wedge b''\leq b')$$

$$\text{(let)}\quad \frac{A\vdash\text{e}':\kappa'\quad A\bigcup\{\text{x}:\kappa'\}\vdash\text{e}:\kappa}{A\vdash\texttt{let}\ \text{x}=\text{e}'\texttt{in}\ \text{e}:\kappa}$$

Figure 3.1: Type System

3.1.1 The Type Rules

The (const) rule states that every constant is static. Variables are looked up in the environment A. An `if`–expression is annotated as being static iff the conditional is static. The branches must have the same binding time greater than the binding time of the conditional and this binding time is also the binding time of the result.

An abstraction can only be annotated as dynamic (b'') if it appears in a dynamic context. In this case the sidecondition ensures that both the argument and results are dynamic. Otherwise the abstraction is given the expected binding time.

An application is annotated according to the binding time of the abstraction. The binding time types follow the standard types. The rule is guarded by the same sidecondition as the abstraction rule (see below).

Both operands of a primitive operator must have the same binding time, which is also the binding time of the result and the annotation of the operator. The standard type of operator `op` is given as $\mathcal{P}(\texttt{op})$. The binding times of `fix` follow the standard types in a way similar to abstraction.

The binding times of `pair` follow the standard types, unless the pair appears in a dynamic context. In this case the sidecondition ensures that both branches are dynamic.

Selectors `fst` and `snd` are annotated by the binding time of the $\times$–constructor and the result has the compound type of the selected branch. Note that the rules for `pair`, `fst` and `snd` permits binding times for partially static data structures. The same sidecondition as in rule (pair) ensures that if the pair is dynamic then so are the branches.

The reader might wonder why the same sidecondition is necessary in both abstraction and application *resp.* pairing and selectors. We will return with an example (example 3.3) when we have introduced *lift* in the next section — no reasonable examples can be done without it.

Notice in the (let) rule that `let` has no superscript. This is due to the fact, that `let`–expressions can always be safely unfolded. This can be seen easily as follows: Consider `let x =` e `in` e$'$ as an abbreviation for $(\lambda\texttt{x}.\mathrm{e}')$@e. Since the λ is immediately applied, it cannot get caught in a dynamic context and can thus never be raised. Unfolding a `let` can however lead to code duplication — this can be avoided using a memoizing specializer, more details will be discussed in section 13.3.1.

3.2 Coercion

As noted in [Gomard 1989] binding time analysis without a lift rule, is not very useful[1]. The lift (or coercion) rule for our system as presented so far is given in figure 3.2.

The rules (coerce) coerce first order values Int and Bool only. It is customary to restrict binding time subtyping to base values, but coercion of product and function types makes perfect sense. In the functional case we will need a rule similar to the usual known contravariant rule. This will be discussed in section 13.1.1.

[1]In our system constants cannot have type D. This implies, that many programs cannot be given a binding time type at all without the lift rule. This could easily be fixed by allowing any binding time value for constants.

$$\text{(coerce)}\quad \frac{A\vdash \mathrm{e}:(\mathrm{Bool},\mathsf{S})}{A\vdash \textit{lift}\ \mathrm{e}:(\mathrm{Bool},\mathsf{D})}$$

$$\text{(coerce)}\quad \frac{A\vdash \mathrm{e}:(\mathrm{Int},\mathsf{S})}{A\vdash \textit{lift}\ \mathrm{e}:(\mathrm{Int},\mathsf{D})}$$

Figure 3.2: Simple lifting

Example 3.3

As promised, we now show why the same sidecondition is needed in both abstraction and application *resp.* pairing and selectors. That the sideconditions are necessary in application *resp.* selectors can be seen by the following programs both being legal (the reader can verify) if sideconditions were omitted. First a program showing the need for side conditions on application:

$$(\texttt{if}\ \mathrm{d}^{\mathsf{D}}\ \texttt{then}\ (\texttt{fix}^{\mathsf{D}}\mathrm{f}\ \mathrm{x}.(\mathrm{f}@^{\mathsf{D}}\mathrm{x}))\ \texttt{else}\ (\texttt{fix}^{\mathsf{D}}\mathrm{g}\ \mathrm{y}.(\mathrm{g}@^{\mathsf{D}}\mathrm{y})))@^{\mathsf{D}}(\textit{lift}\ 1)@^{\mathsf{D}}2@^{\mathsf{S}}3$$

Then a program showing the need for side conditions on selectors:

$$\texttt{fst}^{\mathsf{S}}\texttt{fst}^{\mathsf{D}}((\texttt{if}\ \mathrm{d}^{\mathsf{D}}\ \texttt{then}\ (\texttt{fix}^{\mathsf{D}}\mathrm{f}\ \mathrm{x}.(\mathrm{f}@^{\mathsf{D}}\mathrm{x}))\ \texttt{else}\ (\texttt{fix}^{\mathsf{D}}\mathrm{g}\ \mathrm{y}.(\mathrm{g}@^{\mathsf{D}}\mathrm{y})))@^{\mathsf{D}}(\textit{lift}\ 1))$$

Any (reasonable) specializer will crash on these programs trying to do static computations on dynamic data. The problem is that $\vdash(\texttt{fix}^{\mathsf{D}}\mathrm{f}\ \mathrm{x}.(\mathrm{f}@^{\mathsf{D}}\mathrm{x}))@^{\mathsf{D}}(\textit{lift}\ 1){:}(\kappa,\mathsf{D})$ for any compound type κ.

We can not leave out the sidecondition in pair either, since this would allow dynamic pairs of static content (though never if selectors were applied to the pair), which is clearly not desirable since these can be returned by the program.

The same argument holds for leaving out the sidecondition in abstraction. The reader might argue that we should not allow programs resulting in higher order values anyway, so there is no problem. Still we keep the sidecondition for generality. □(Ex.3.3)

Chapter 4

Polymorphic Binding Time Analysis

This chapter extends the inference systems of the chapter 3 to deal with polymorphism. Section 4.1 introduces polymorphism over binding time variables, and since these are also used as annotations, we have the desired polyvariant effect. In section 4.2, we show that this kind of polymorphism is not enough, due to problems with sideconditions and coercions. We show how this can be dealt with by allowing to discard constraints from the assumptions and later re-introducing them (corresponding to a kind of polymorphism over constraints/coercions).

To motivate the extensions of the system of chapter 3, we begin by giving a simple example.

Example 4.1
Consider the following program where the (user defined) identity function is applied both static and dynamic data (`d` will here and in all examples denote a dynamic variable):

```
let f = λx.x
in f@d + (f@5 + 8)
```

In the monomorphic system of chapter 3, this program will be annotated as:

```
let f = λ^S x.x
in f@^S d +^D (f@^S (lift 5) +^D 8)
```

Since `f` is applied to dynamic data, the result of applying `f` is also dynamic — in other words `f` gets the type $((\text{Int}, \mathsf{D}) \rightarrow (\text{Int}, \mathsf{D}), \mathsf{S})$. This implies, that `5` has to be lifted before applying `f`, which in turn implies that the last `+` is annotated as dynamic.

If `f` also had appeared in a dynamic context (*e.g.* `if d=0 then f else λx.x+1`), the λ in the definition of `f` would be annotated as being dynamic and both applications above would have to be made residual.

In the polymorphic system, which we are about to present, we expect `f` to be given a polymorphic type such as $\forall\beta_x\forall\beta_f.((\text{Int}, \beta_x) \rightarrow (\text{Int}, \beta_x), \beta_f)$. Already here we notice, that using the type system of figure 3.1, β_x and β_f appear in the sidecondition of rule (abstr); this is the motivation for section 4.2.

We use an explicit (Church style) notation for abstraction over binding time values. The program can (except for the problem with sideconditions) be annotated as:

```
let f = $\Lambda\beta_x\Lambda\beta_f\lambda^{\beta_f}$x.x
in (f◇D◇S@$^{\mathsf{D}}$d) +$^{\mathsf{D}}$ (f◇S◇S@$^{\mathsf{S}}$5 +$^{\mathsf{S}}$ 8)
```

Here Λ denotes explicit abstraction over binding times and $\diamond$ denotes explicit application of such abstractions to binding time values. □(Ex.4.1)

4.1 Polymorhism over Binding Times

We will now make the ideas of example 4.1 precise. This is done in a way very similar to the way polymorphism was formulated in the `let`-polymorphic system of section 1.1.2. First we introduce typeschemes σ and binding time variables β:

$$\sigma ::= \kappa \mid \forall\beta.\sigma$$

$$b ::= \mathsf{S} \mid \mathsf{D} \mid \beta$$

$$\kappa ::= (\text{Int},b) \mid (\text{Bool},b) \mid (\kappa \rightarrow \kappa,b) \mid (\kappa \times \kappa,b)$$

The idea is — as in ML polymorphism — that typeschemes are only allowed for `let`-bound expressions. In contrast to ML polymorphism, we introduce an explicit extension á la Church to the syntax: a Λ-abstraction over binding times and an explicit application $\diamond$ of such abstractions:

$$\text{e} ::= \Lambda\beta_x.\text{e} \mid \text{e}\diamond b$$

The idea of having explicit abstraction and application is that these will have an operational meaning during specialization: the abstracted binding times will often decide whether an expression should be evaluated or residualized.

We denote the set of binding time variables *BtVar*. The type rules introducing polymorphic `let` are shown in figure 4.1. The rules ($\forall$–introduction) and ($\forall$–elimination) are similar to the rules of figure 1.3, and rule (let) restricts the use of type schemes to `let`–bound expressions.

$$(\forall\text{–introduction}) \quad \frac{A\vdash \text{e}:\sigma}{A\vdash\Lambda\beta.\text{e}:\forall\beta.\sigma}(\text{if } \beta \text{ not free in } A \text{ or any sidecondition})$$

$$(\forall\text{–elimination}) \quad \frac{A\vdash \text{e}:\forall\beta.\sigma}{A\vdash \text{e}\diamond b:[b/\beta]\sigma}$$

$$(\text{let}) \quad \frac{A\vdash \text{e}':\sigma \quad A\bigcup\{\text{x}:\sigma\}\vdash \text{e}:\kappa}{A\vdash \texttt{let}\ \text{x} = \text{e}'\ \texttt{in}\ \text{e}:\kappa}$$

Figure 4.1: Polymorphism over Binding Times

4.2 Lifting with Polymorphism

As foreshadowed in example 4.1, the polymorphism introduced in section 4.1 does not give the desired polyvariance. The first problem is that many variables are not free in all sideconditions. Another is that we need coercions in `let`-bound expressions to depend on the use of the `let`-bound variable (just as the binding time annotations in a `let`-bound expression depend on the use of the `let`-bound variable via binding time abstraction).

We now introduce the notion of a *constraint set* C containing constraints of the form $b \leq b'$. The constraints induces coercions $b \rightsquigarrow b'$ which are generalizations of the *lift* introduced in section 3.2. We will use the notation introduced in figure 1.7.

We see that the two problems mentioned above are actually two sides of the same coin. The solution to both problems is (intuitively) to allow quantification over coercions $\beta \rightsquigarrow \beta'$ and sideconditions (both of which are derived from constraints). The important effect of this is that it allows us to temporarily discard constraint assumption, and then abstract over binding time variables that would otherwise have appeared in the assumptions.

In figure 4.2 we present binding time analysis as presented so far (in figure 3.1 and 4.1) again, but now with a constraint set in the premise of each assertion. In rule ($\forall$-introduction) the condition is that β is not free in A or C. Otherwise the type system has not been changed. In figure 4.3, we present the new rules for lifting. To the lift rule $\mathsf{S} \rightsquigarrow \mathsf{D}$ we have added two extra constant coercion $\mathsf{S} \rightsquigarrow \mathsf{S}$ and $\mathsf{D} \rightsquigarrow \mathsf{D}$. We have also added the ability to use coercion induced by constraints in the constraint set.

The problem is the conflict between the need to quantify over binding time variables occurring in coercions and constraints, and the restriction disallowing quantification over variables occurring free in the constraint set.

We solve this problem by allowing abstraction over coercions, written $\Lambda b \rightsquigarrow b'.\mathrm{e}$. The type of this is $b \leq b' \Rightarrow \sigma$ if the type of e is σ — this type notation reflects the intuition: if $b \leq b'$ "holds", then we have type σ. Expression $\Lambda b \rightsquigarrow b'.\mathrm{e}$ must be applied to $b \rightsquigarrow b'$; we use $\Box$ to denote application of these abstractions to coercions. This corresponds to (re-)introducing discarded constraints.

Adding coercion abstraction gives us the following syntax of types and typeschemes:

$$\sigma ::= \kappa \mid \forall\beta.\sigma \mid b \leq b' \Rightarrow \sigma$$

$$b ::= \mathsf{S} \mid \mathsf{D} \mid \beta$$

$$\kappa ::= (\mathrm{Int}, b) \mid (\mathrm{Bool}, b) \mid (\kappa \to \kappa, b) \mid (\kappa \times \kappa, b)$$

and the following full syntax for expressions in Exp_{spec}:

$$\begin{array}{lll} \mathrm{e} & ::= & \mathtt{true} \mid \mathtt{false} \mid n \mid \mathrm{x} \\ & \mid & \mathtt{if}^b\ \mathrm{e}\ \mathtt{then}\ \mathrm{e}\ \mathtt{else}\ \mathrm{e} \\ & \mid & \lambda^b \mathrm{x.e} \mid \mathrm{e} @^b \mathrm{e} \\ & \mid & \mathrm{e}\ \mathtt{op}^b\ \mathrm{e} \mid \mathtt{fix}^b\ \mathrm{f\ x.e} \mid \mathrm{let\ x} = \mathrm{e}\ \mathtt{in}\ \mathrm{e} \\ & \mid & \mathtt{pair}^b(\mathrm{e,e}) \mid \mathtt{fst}^b\ \mathrm{e} \mid \mathtt{snd}^b\ \mathrm{e} \\ & \mid & \Lambda\beta_x.\mathrm{e} \mid \mathrm{e}\diamond b \mid \Lambda b \rightsquigarrow b'.\mathrm{e} \mid \mathrm{e}\Box b \rightsquigarrow b' \end{array}$$

That $b \leq b' \Rightarrow \sigma$ is a typescheme ensures, that $\Lambda b \rightsquigarrow b'.\mathrm{e}$ can only appear in `let`-bound expressions (just as $\Lambda\beta.\mathrm{e}$).

Figure 4.4 shows the rules for $\Rightarrow$-introduction and elimination — these rules correspond closely to other higher order abstraction rules.

(const) $A,C\vdash\texttt{true}:(\text{Bool},\mathsf{S})$ $\quad A,C\vdash\texttt{false}:(\text{Bool},\mathsf{S})$ $\quad A,C\vdash n:(\text{Int},\mathsf{S})$

(var) $A\bigcup\{\text{x}:\sigma\},C\vdash\text{x}:\sigma$

(if) $$\frac{A,C\vdash\text{e}:(\text{Bool},b)\quad A,C\vdash\text{e}':(t,b')\quad A,C\vdash\text{e}'':(t,b')\quad C\vdash b\leq b'}{A,C\vdash\texttt{if}^b\ \text{e}\ \texttt{then}\ \text{e}'\ \texttt{else}\ \text{e}'':\ (t,b')}$$

(abstr) $$\frac{A\bigcup\{\text{x}:(t,b)\},\ C\vdash\text{e}:(t',b')\quad C\vdash b''\leq b\quad C\vdash b''\leq b'}{A,C\vdash\lambda^{b''}\text{x.e}\ :\ ((t,b)\to(t',b'),b'')}$$

(appl) $$\frac{A,C\vdash\text{e}:((t,b)\to(t',b'),b'')\quad A,C\vdash\text{e}':(t,b)\quad C\vdash b''\leq b\quad C\vdash b''\leq b'}{A,C\vdash\text{e}@^{b''}\text{e}':\ (t',b')}$$

(op) $$\frac{A,C\vdash\text{e}:(t,b)\quad A,C\vdash\text{e}':(t',b)\quad \mathcal{P}(\texttt{op})=t\times t'\to t''}{A,C\vdash\text{e}\ \texttt{op}^b\text{e}':\ (t'',b)}$$

(fix) $$\frac{A\bigcup\{\text{x}:(t,b),\text{f}:((t,b)\to(t',b'),b'')\},C\vdash\text{e}:(t',b')\quad C\vdash b''\leq b\quad C\vdash b''\leq b'}{A,C\vdash\texttt{fix}^{b''}\text{f x.e}\ :\ ((t,b)\to(t',b'),b'')}$$

(pair) $$\frac{A,C\vdash\text{e}:(t,b)\quad A,C\vdash\text{e}':(t',b')\quad C\vdash b''\leq b\quad C\vdash b''\leq b'}{A,C\vdash\texttt{pair}^{b''}(\text{e},\text{e}')\ :\ ((t,b)\times(t',b'),b'')}$$

(first) $$\frac{A,C\vdash\text{e}:\ ((t,b)\times(t',b'),b'')\quad C\vdash b''\leq b\quad C\vdash b''\leq b'}{A,C\vdash\texttt{fst}^{b''}\text{e}\ :\ (t,b)}$$

(second) $$\frac{A,C\vdash\text{e}:\ ((t,b)\times(t',b'),b'')\quad C\vdash b''\leq b\quad C\vdash b''\leq b'}{A,C\vdash\texttt{snd}^{b''}\text{e}\ :\ (t',b')}$$

($\forall$–introduction) $$\frac{A,C\vdash\text{e}:\sigma}{A,C\vdash\Lambda\beta.\text{e}:\forall\beta.\sigma}(\text{if }\beta\text{ not free in }A,C)$$

($\forall$–elimination) $$\frac{A,C\vdash\text{e}:\forall\beta.\sigma}{A,C\vdash\text{e}\diamond b:[b/\beta]\sigma}$$

(let) $$\frac{A,C\vdash\text{e}':\sigma\quad A\bigcup\{\text{x}:\sigma\},C\vdash\text{e}:\kappa}{A,C\vdash\texttt{let}\ \text{x}=\text{e}'\ \texttt{in}\ \text{e}:\kappa}$$

Figure 4.2: Adding Constraint Sets

$$\text{(coerce)} \quad \frac{A,C\vdash \text{e:(Bool,}b) \quad C\vdash b \le b'}{A,C\vdash [b\rightsquigarrow b']\text{e:(Bool,}b')}$$

$$\text{(coerce)} \quad \frac{A,C\vdash \text{e:(Int,}b) \quad C\vdash b \le b'}{A,C\vdash [b\rightsquigarrow b']\text{e:(Int,}b')}$$

$$\text{(lookup-coerce)} \quad C\bigcup\{b \le b'\}\vdash b\rightsquigarrow b'$$

$$\text{(lift)} \quad C\vdash \mathsf{S} \le \mathsf{D}$$

$$\text{(id-dyn)} \quad C\vdash \mathsf{D} \le \mathsf{D}$$

$$\text{(id-stat)} \quad C\vdash \mathsf{S} \le \mathsf{S}$$

Figure 4.3: Coercion Rules

$$(\Rightarrow\text{–introduction}) \quad \frac{A,C\bigcup\{b \le b'\}\vdash \text{e:}\sigma}{A,C\vdash \Lambda b\rightsquigarrow b'.\text{e} : b{\le}b'{\Rightarrow}\sigma}$$

$$(\Rightarrow\text{–elimination}) \quad \frac{A,C\vdash \text{e} : b{\le}b'{\Rightarrow}\sigma \quad C\vdash b \le b'}{A,C\vdash \text{e}\Box b\rightsquigarrow b'{:}\sigma}$$

Figure 4.4: Abstraction over coercions & Lift Rule

Example 4.2
We are now able to give the correct typing of the program of example 4.1:

$$
\begin{array}{l}
\texttt{let id = }\Lambda\beta_\lambda.\Lambda\beta_{res}.\Lambda\beta_x.\Lambda\beta_\lambda\rightsquigarrow\beta_{res}.\Lambda\beta_\lambda\rightsquigarrow\beta_x.\Lambda\beta_x\rightsquigarrow\beta_{res}.\lambda^{\beta_\lambda}\texttt{x.}[\beta_x\rightsquigarrow\beta_{res}]\texttt{x} \\
\texttt{in (id}\diamond\mathsf{S}\diamond\mathsf{D}\diamond\mathsf{D}\Box\mathsf{S}\rightsquigarrow\mathsf{D}\Box\mathsf{S}\rightsquigarrow\mathsf{D}\Box\mathsf{D}\rightsquigarrow\mathsf{D}@^{\mathsf{S}}\texttt{ d)} \\
\quad +^{\mathsf{D}}\ [\mathsf{S}\rightsquigarrow\mathsf{D}]\texttt{(id}\diamond\mathsf{S}\diamond\mathsf{S}\diamond\mathsf{S}\Box\mathsf{S}\rightsquigarrow\mathsf{S}\Box\mathsf{S}\rightsquigarrow\mathsf{S}\Box\mathsf{S}\rightsquigarrow\mathsf{S}@^{\mathsf{S}}\texttt{5 }+^{\mathsf{S}}\texttt{ 8)}
\end{array}
$$

□(Ex.4.2)

While abstraction over binding time values has a clear operational meaning during specialization, this is not the case for abstraction over coercions. Rather, this construct ensures us that we are able to abstract over binding time variables that would otherwise have appeared free in the constraint set C. In chapter 8, the $\forall$– and $\Rightarrow$–introduction rules, are replaced by one (provably) equivalent rule allowing parallel abstraction over binding time values restricted by a set of constraints.

4.3 Well Annotatedness

The inference system presented above defines annotations on assumptions given by the type environment A and constraint set C. We are clearly not interested in typings such as $\{\},\{\mathsf{D} \leq \mathsf{S}\}\vdash\texttt{if}^{\mathsf{S}}\ [\mathsf{D}\rightsquigarrow\mathsf{S}]\mathrm{d}$ `then` 5 `else` 7:S. We therefore define the notion of *well annotatedness.*

While we think of $b \leq b' \in C$ syntactically, we introduce the notation $b \overset{!}{\leq} b'$ to denote that $b \leq b'$ is provable. Relation $\overset{!}{\leq}$ is defined by $\mathsf{S}\overset{!}{\leq}\mathsf{S}$, $\mathsf{S}\overset{!}{\leq}\mathsf{D}$ and $\mathsf{D}\overset{!}{\leq}\mathsf{D}$. This also means that $\overset{!}{\leq}$ is not defined on variables.

Definition 4.1
Let ζ be a substitution mapping binding time variables β to binding time values $\{\mathsf{S},\mathsf{D}\}$. Relation $\Vdash$ is defined by

$$\zeta\Vdash C \overset{def}{\Longleftrightarrow} \forall b \leq b' \in C : \zeta(b)\overset{!}{\leq}\zeta(b')$$

□(Def.4.1)

Definition 4.2
If A,$C\vdash$e:σ and $\zeta\Vdash C$ then ζe is called a well annotated expression. □(Def.4.2)

Chapter 5

The Specializer

This chapter presents the semantics of a *call-by-value* specializer consuming the information inferred by our binding time analysis. We do this using natural semantics ([Plotkin 1981], [Kahn 1987]) similar to the semantic definition in section 2.2.

An expression can evaluate to ($\longrightarrow_{spec}$) base values, pairs and closures as well as to residual expressions. The set of residual expression values is denoted $\underline{Val}$ — the set is isomorphic to Exp_{std}, underlines serve to distinguish objects in $\underline{Val}$ from objects in Exp_{std}; function φ which we will presented in chapter 6 is this isomorphism. Gomard (in [Gomard 1991b]) has special functions *build-λ*, *build-@*, *build-if*, *etc.* for emitting code; we simply write $\underline{\lambda}$, $\underline{@}$,$\underline{\texttt{if}}$ *etc.* Thus the underline is used building expressions in $\underline{Val}$. We also use the underline to map constants in *Const* to constant expressions (`true`, `false` and n) in $\underline{Val}$. This rule is used when lifting constants — note that this underline operation is only defined on values in *Const*.

An environment ρ maps a program variable `x` to either a value (if the variable is static) or to $\underline{\texttt{x}}$. The underline in $\underline{\texttt{x}}$ of course helps the reader to identify a variable as belonging to $\underline{Val}$, but is also a guide for the specializer meaning introduction of a new variable name (taken from an infinite list of variable names). The sets of interest are:

$$\begin{array}{lcl}
Val_{spec} & = & Const \bigcup Funval_{spec} \bigcup RecFunval_{spec} \\
 & & \bigcup BtFunval_{spec} \bigcup ClFunval_{spec} \\
 & & \bigcup (Val_{spec} \times Val_{spec}) \bigcup \underline{Val} \\
Funval_{spec} & = & Var \times Exp_{spec} \times Varenv_{spec} \\
RecFunval_{spec} & = & Var \times Var \times Exp_{spec} \times Varenv_{spec} \\
BtFunval_{spec} & = & BtVar \times Exp_{spec} \times Varenv_{spec} \\
ClFunval_{spec} & = & BtVal \times BtVal \times Exp_{spec} \times Varenv_{spec} \\
Varenv_{spec} & = & Var \rightarrow Val_{spec}
\end{array}$$

where $BtFunval_{spec}$ is the set of binding time abstraction closures written $[\![\Lambda\beta.\mathrm{e}]\!]\rho$ and $BtFunval_{spec}$ is the set of coercion abstraction closures written $[\![\Lambda b \rightsquigarrow b'.\mathrm{e}]\!]\rho$. Objects in $Funval_{spec}$ *resp.* $RecFunval_{spec}$ are written as closures $[\![\lambda\mathrm{x.e}]\!]\rho$ *resp.* $[\![\texttt{fix}\ \mathrm{f\ x.e}]\!]\rho$. By $\sqcup\vdash\sqcup\longrightarrow_{spec}\sqcup$ we define a relation over $Varenv_{spec} \times Exp_{spec} \times Val_{spec}$.

We use w to range over "real" values $Const \bigcup Funval_{spec} \bigcup RecFunval_{spec} \bigcup BtFunval_{spec} \bigcup ClFunval_{spec} \bigcup (Val_{spec} \times Val_{spec})$, and use $\underline{\mathrm{e}}$ as a meta variable ranging over $\underline{Val}$. We use v to range over all values in Val_{spec}.

$$\text{(const)}\quad \rho\vdash\texttt{true}\longrightarrow_{spec}\texttt{true}\qquad \rho\vdash\texttt{false}\longrightarrow_{spec}\texttt{false}\qquad \rho\vdash n\longrightarrow_{spec}n$$

$$\text{(var)}\quad \rho[\mathrm{x}\mapsto v]\vdash\mathrm{x}\longrightarrow_{spec}v$$

$$\text{(if-true)}\quad \frac{\rho\vdash\mathrm{e}\longrightarrow_{spec}\texttt{true}\qquad \rho\vdash\mathrm{e}'\longrightarrow_{spec}v'}{\rho\vdash\texttt{if}^{\mathsf{S}}\ \mathrm{e}\ \texttt{then}\ \mathrm{e}'\ \texttt{else}\ \mathrm{e}''\longrightarrow_{spec}v'}$$

$$\text{(if-false)}\quad \frac{\rho\vdash\mathrm{e}\longrightarrow_{spec}\texttt{false}\qquad \rho\vdash\mathrm{e}''\longrightarrow_{spec}v''}{\rho\vdash\texttt{if}^{\mathsf{S}}\ \mathrm{e}\ \texttt{then}\ \mathrm{e}'\ \texttt{else}\ \mathrm{e}''\longrightarrow_{spec}v''}$$

$$(\underline{\text{if}})\quad \frac{\rho\vdash\mathrm{e}\longrightarrow_{spec}\underline{\mathrm{e}}\qquad \rho\vdash\mathrm{e}'\longrightarrow_{spec}\underline{\mathrm{e}}'\qquad \rho\vdash\mathrm{e}''\longrightarrow_{spec}\underline{\mathrm{e}}''}{\rho\vdash\texttt{if}^{\mathsf{D}}\ \mathrm{e}\ \texttt{then}\ \mathrm{e}'\ \texttt{else}\ \mathrm{e}''\longrightarrow_{spec}\underline{\texttt{if}}\ \underline{\mathrm{e}}\ \underline{\texttt{then}}\ \underline{\mathrm{e}}'\ \underline{\texttt{else}}\ \underline{\mathrm{e}}''}$$

$$\text{(abstr)}\quad \frac{}{\rho\vdash\lambda^{\mathsf{S}}\mathrm{x.e}\longrightarrow_{spec}[\![\lambda\mathrm{x.e}]\!]\rho}$$

$$(\underline{\text{abstr}})\quad \frac{\rho[\mathrm{x}\mapsto\underline{\mathrm{x}}]\vdash\mathrm{e}\longrightarrow_{spec}\underline{\mathrm{e}}}{\rho\vdash\lambda^{\mathsf{D}}\mathrm{x.e}\longrightarrow_{spec}\underline{\lambda}\,\underline{\mathrm{x}}.\ \underline{\mathrm{e}}}\ \textit{where } \underline{\mathrm{x}} \textit{ is a fresh variable}$$

$$\text{(fix)}\quad \frac{}{\rho\vdash\texttt{fix}^{\mathsf{S}}\ \mathrm{f\ x.e}\longrightarrow_{spec}[\![\texttt{fix}\ \mathrm{f\ x.e}]\!]\rho}$$

$$(\underline{\text{fix}})\quad \frac{\rho[\mathrm{f}\mapsto\underline{\mathrm{f}},\mathrm{x}\mapsto\underline{\mathrm{x}}]\vdash\mathrm{e}\longrightarrow_{spec}\underline{\mathrm{e}}}{\rho\vdash\texttt{fix}^{\mathsf{D}}\mathrm{f\ x.e}\longrightarrow_{spec}\underline{\texttt{fix}}\ \underline{\mathrm{f}}\ \underline{\mathrm{x}}.\ \underline{\mathrm{e}}}\ \textit{where } \underline{\mathrm{x}} \textit{ and } \underline{\mathrm{f}} \textit{ are fresh variables}$$

$$\text{(appl)}\quad \frac{\rho\vdash\mathrm{e}\longrightarrow_{spec}[\![\lambda\mathrm{x.e}']\!]\rho'\qquad \rho\vdash\mathrm{e}''\longrightarrow_{spec}v\qquad \rho'[\mathrm{x}\mapsto v]\vdash\mathrm{e}'\longrightarrow_{spec}v'}{\rho\vdash\mathrm{e}@^{\mathsf{S}}\mathrm{e}''\longrightarrow_{spec}v'}$$

$$\text{(rec-appl)}\quad \frac{\begin{array}{l}\rho\vdash\mathrm{e}\longrightarrow_{spec}[\![\texttt{fix}\ \mathrm{f\ x.e}']\!]\rho'\\ \rho\vdash\mathrm{e}''\longrightarrow_{spec}v\qquad \rho'[\mathrm{f}\mapsto[\![\texttt{fix}\ \mathrm{f\ x.e}']\!]\rho',\mathrm{x}\mapsto v]\vdash\mathrm{e}'\longrightarrow_{spec}v'\end{array}}{\rho\vdash\mathrm{e}@^{\mathsf{S}}\mathrm{e}''\longrightarrow_{spec}v'}$$

$$(\underline{\text{appl}})\quad \frac{\rho\vdash\mathrm{e}\longrightarrow_{spec}\underline{\mathrm{e}}\qquad \rho\vdash\mathrm{e}'\longrightarrow_{spec}\underline{\mathrm{e}}'}{\rho\vdash\mathrm{e}@^{\mathsf{D}}\mathrm{e}'\longrightarrow_{spec}\underline{\mathrm{e}}\ \underline{@}\ \underline{\mathrm{e}}'}$$

$$\text{(op)}\quad \frac{\rho\vdash\mathrm{e}\longrightarrow_{spec}w\qquad \rho\vdash\mathrm{e}'\longrightarrow_{spec}w'\qquad w\otimes^{\mathsf{op}}w'=w''}{\rho\vdash\mathrm{e}\,\mathsf{op}^{\mathsf{S}}\mathrm{e}'\longrightarrow_{spec}w''}$$

$$(\underline{\text{op}})\quad \frac{\rho\vdash\mathrm{e}\longrightarrow_{spec}\underline{\mathrm{e}}\qquad \rho\vdash\mathrm{e}'\longrightarrow_{spec}\underline{\mathrm{e}}'}{\rho\vdash\mathrm{e}\,\mathsf{op}^{\mathsf{D}}\mathrm{e}'\longrightarrow_{spec}\underline{\mathrm{e}}\ \underline{\mathsf{op}}\ \underline{\mathrm{e}}'}$$

Figure 5.1: A Monomorphic Specializer

$$\text{(pair)}\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec} v \quad \rho\vdash \mathrm{e}'\longrightarrow_{spec} v'}{\rho\vdash \texttt{pair}^{\mathsf{S}}(\mathrm{e},\mathrm{e}')\longrightarrow_{spec}(v,v')}$$

$$\text{(}\underline{\text{pair}}\text{)}\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec} \underline{\mathrm{e}} \quad \rho\vdash \mathrm{e}'\longrightarrow_{spec} \underline{\mathrm{e}}'}{\rho\vdash \texttt{pair}^{\mathsf{D}}(\mathrm{e},\mathrm{e}')\longrightarrow_{spec}\underline{\texttt{pair}}(\underline{\mathrm{e}},\underline{\mathrm{e}}')}$$

$$\text{(first)}\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}(v,v')}{\rho\vdash \texttt{fst}^{\mathsf{S}}\mathrm{e}\longrightarrow_{spec} v}$$

$$\text{(}\underline{\text{first}}\text{)}\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec} \underline{\mathrm{e}}}{\rho\vdash \texttt{fst}^{\mathsf{D}}\mathrm{e}\longrightarrow_{spec}\underline{\texttt{fst}}\ \underline{\mathrm{e}}}$$

$$\text{(second)}\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}(v,v')}{\rho\vdash \texttt{snd}^{\mathsf{S}}\mathrm{e}\longrightarrow_{spec} v'}$$

$$\text{(}\underline{\text{second}}\text{)}\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec} \underline{\mathrm{e}}}{\rho\vdash \texttt{snd}^{\mathsf{D}}\mathrm{e}\longrightarrow_{spec}\underline{\texttt{snd}}\ \underline{\mathrm{e}}}$$

$$\text{(Coerce)}\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec} v}{\rho\vdash [b\rightsquigarrow b]\mathrm{e}\longrightarrow_{spec} v}$$

$$\text{(Coerce)}\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec} w}{\rho\vdash [\mathsf{S}\rightsquigarrow\mathsf{D}]\mathrm{e}\longrightarrow_{spec} \underline{w}}$$

Figure 5.2: A Monomorphic Specializer (continued)

$$\text{(bt-gen)}\quad \frac{}{\rho\vdash \Lambda\beta.\mathrm{e}\longrightarrow_{spec}[\![\Lambda\beta.\mathrm{e}]\!]\rho}$$

$$\text{(bt-inst)}\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}[\![\Lambda\beta.\mathrm{e}']\!]\rho' \quad \rho'\vdash [b/\beta]\mathrm{e}'\longrightarrow_{spec} v}{\rho\vdash \mathrm{e}\diamond b\longrightarrow_{spec} v}$$

$$\text{(c-gen)}\quad \frac{}{\rho\vdash \Lambda b\rightsquigarrow b'.\mathrm{e}\longrightarrow_{spec}[\![\Lambda b\rightsquigarrow b'.\mathrm{e}]\!]\rho}$$

$$\text{(c-inst)}\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}[\![\Lambda b\rightsquigarrow b'.\mathrm{e}']\!]\rho' \quad \rho'\vdash \mathrm{e}'\longrightarrow_{spec} v}{\rho\vdash \mathrm{e}\Box b\rightsquigarrow b'\longrightarrow_{spec} v}$$

$$\text{(let)}\quad \frac{\rho\vdash \mathrm{e}'\longrightarrow_{spec} v' \quad \rho[\mathrm{x}\mapsto v']\vdash \mathrm{e}\longrightarrow_{spec} v}{\rho\vdash \texttt{let}\ \mathrm{x} = \mathrm{e}'\ \texttt{in}\ \mathrm{e}\longrightarrow_{spec} v}$$

Figure 5.3: Adding polymorphism to the Specializer

5.1 The Semantic Rules

The semantic rules for specialization are shown in figure 5.1 and figure 5.2. The rules should contain no surprises; the static cases corresponds to the standard evaluation rules and the dynamic ones simply evaluates its subexpressions and emits code.

Since we have decided always to unfold `lets`, making the specializer able to handle polymorphism is pretty straightforward. This is shown in figure 5.3.

Notice that when applying an expression to a binding time value b, we actually substitute the value for the syntactic binding time parameter. This is necessary since binding time values have actual semantic meaning in the program guiding the evaluation.

Observation 5.1
The specializer is *natural* in the sense that every rule for a static expression is identical to the corresponding standard evaluation rule. □(Obs.5.1)

Observation 5.2
From observation 5.1 one would expect that for any expression e having a static type, termination of standard evaluation and specialization would coincide. This is not the case, as shown by the following well annotated expression:

$$\lambda^{\mathsf{S}}\mathtt{x}.5 \mathbin{@} \mathtt{if}^{\mathsf{D}}\ [\mathsf{S}\leadsto\mathsf{D}]\mathtt{true}\ \mathtt{then}\ 5\ \mathtt{else}\ \mathit{bomb}$$

where *bomb* is some nonterminating expression. Both standard evaluation and specialization will evaluate the `if` statement. Due to the dynamic test, the specializer will evaluate both branches.

Thus to get coincidence between evaluation of standard evaluation and specialization of an expression, every subexpression must be annotated as being static. □(Obs.5.2)

5.2 Specialization Errors

We want to be able to reason about errors in the specializer. As the specializer is given so far, we can end up in a stuck state — that is a state where no rule is applicable. Such a stuck state is not distinguishable from a non-terminating state, making it impossible to prove assertions like "*specialization does not go wrong*" (which we would like to prove in the next chapter). We make it possible to distinguish stuck states from non-termination by adding a special value ε.

$$Val^{error}_{spec} \quad = \quad Val_{spec} \bigcup \{\varepsilon\}$$

We let v,w,$\underline{e}$ range over the same sets as before — that is non of them can assume value ε.

If a standard type error occurs (*e.g.* the condition in an `if`-statement evaluates to an integer), we still leave the specializer stuck. These kinds of errors does not have our interest since we assume all programs to be well typed (in a standard type system).

$$\text{(if)}\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}\underline{\mathrm{e}}}{\rho\vdash \texttt{if}^{\mathsf{S}}\ \mathrm{e}\ \texttt{then}\ \mathrm{e}'\ \texttt{else}\ \mathrm{e}''\longrightarrow_{spec}\varepsilon}$$

$$(\underline{\text{if}})\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}w}{\rho\vdash \texttt{if}^{\mathsf{D}}\ \mathrm{e}\ \texttt{then}\ \mathrm{e}'\ \texttt{else}\ \mathrm{e}''\longrightarrow_{spec}\varepsilon}\qquad \frac{\rho\vdash \mathrm{e}'\longrightarrow_{spec}w}{\rho\vdash \texttt{if}^{\mathsf{D}}\ \mathrm{e}\ \texttt{then}\ \mathrm{e}'\ \texttt{else}\ \mathrm{e}''\longrightarrow_{spec}\varepsilon}$$

$$\frac{\rho\vdash \mathrm{e}''\longrightarrow_{spec}w}{\rho\vdash \texttt{if}^{\mathsf{D}}\ \mathrm{e}\ \texttt{then}\ \mathrm{e}'\ \texttt{else}\ \mathrm{e}''\longrightarrow_{spec}\varepsilon}$$

$$(\underline{\text{abstr}})\quad \frac{\rho[\mathrm{x}\mapsto\underline{\mathrm{x}}]\vdash \mathrm{e}\longrightarrow_{spec}w}{\rho\vdash\lambda^{\mathsf{D}}\mathrm{x.e}\longrightarrow_{spec}\varepsilon}\textit{where } \underline{\mathrm{x}} \textit{ is a fresh variable}$$

$$(\underline{\text{fix}})\quad \frac{\rho[\mathrm{f}\mapsto\underline{\mathrm{f}},\mathrm{x}\mapsto\underline{\mathrm{x}}]\vdash \mathrm{e}\longrightarrow_{spec}w}{\rho\vdash\texttt{fix}^{\mathsf{D}}\mathrm{f\ x.e}\longrightarrow_{spec}\varepsilon}\textit{where } \underline{\mathrm{f}} \textit{ and } \underline{\mathrm{x}} \textit{ are fresh variables}$$

$$\text{(appl)}\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}\underline{\mathrm{e}}}{\rho\vdash \mathrm{e}@^{\mathsf{S}}\mathrm{e}''\longrightarrow_{spec}\varepsilon}$$

$$\text{(rec-appl)}\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}\underline{\mathrm{e}}}{\rho\vdash \mathrm{e}@^{\mathsf{S}}\mathrm{e}''\longrightarrow_{spec}\varepsilon}$$

$$(\underline{\text{appl}})\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}w}{\rho\vdash \mathrm{e}@^{\mathsf{D}}\mathrm{e}'\longrightarrow_{spec}\varepsilon}\qquad \frac{\rho\vdash \mathrm{e}'\longrightarrow_{spec}w}{\rho\vdash \mathrm{e}@^{\mathsf{D}}\mathrm{e}'\longrightarrow_{spec}\varepsilon}$$

$$\text{(op)}\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}\underline{\mathrm{e}}}{\rho\vdash \mathrm{e}\texttt{op}^{\mathsf{S}}\mathrm{e}'\longrightarrow_{spec}\varepsilon}\qquad \frac{\rho\vdash \mathrm{e}'\longrightarrow_{spec}\underline{\mathrm{e}}}{\rho\vdash \mathrm{e}\texttt{op}^{\mathsf{S}}\mathrm{e}'\longrightarrow_{spec}\varepsilon}$$

$$(\underline{\text{op}})\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}w}{\rho\vdash \mathrm{e}\texttt{op}^{\mathsf{D}}\mathrm{e}'\longrightarrow_{spec}\varepsilon}\qquad \frac{\rho\vdash \mathrm{e}'\longrightarrow_{spec}w}{\rho\vdash \mathrm{e}\texttt{op}^{\mathsf{D}}\mathrm{e}'\longrightarrow_{spec}\varepsilon}$$

$$(\underline{\text{pair}})\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}w}{\rho\vdash \texttt{pair}^{\mathsf{D}}(\mathrm{e},\mathrm{e}')\longrightarrow_{spec}\varepsilon}\qquad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}w}{\rho\vdash \texttt{pair}^{\mathsf{D}}(\mathrm{e},\mathrm{e}')\longrightarrow_{spec}\varepsilon}$$

$$(\underline{\text{first}})\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}w}{\rho\vdash \texttt{fst}^{\mathsf{D}}\mathrm{e}\longrightarrow_{spec}\varepsilon}$$

$$(\underline{\text{second}})\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}w}{\rho\vdash \texttt{snd}^{\mathsf{D}}\mathrm{e}\longrightarrow_{spec}\varepsilon}$$

$$\text{(Coerce)}\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}\underline{\mathrm{e}}}{\rho\vdash [\mathsf{S}\rightsquigarrow\mathsf{D}]\mathrm{e}\longrightarrow_{spec}\varepsilon}$$

$$\text{(Coerce)}\quad \frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}w}{\rho\vdash [\mathsf{S}\rightsquigarrow\mathsf{D}]\mathrm{e}\longrightarrow_{spec}\varepsilon}$$

$$\textit{if } w\in \mathit{Funval}_{spec}\bigcup \mathit{RecFunval}_{spec}\bigcup \mathit{BtFunval}_{spec}\bigcup \mathit{ClFunval}_{spec}$$

Figure 5.4: Errors in the Specializer

Our interest is errors arising from wrong annotations — *e.g.* if the condition in a static `if`-statement results in a residual expression.

In figure 5.4, we present specialization rules resulting in an error value. They are presented with the same names as the rules of figures 5.1 to 5.3, in order to make it easy to check that the rules together cover all cases. This means that the error rules for application and recursive application are identical. We assume all constructors to be "error-strict" — that is, if any evaluation in the assumptions of a rule results in an error, then so does the conclusion — and we do not give explicit rules for this.

By going through all specialization rules, we can easily assure ourselves that for all standard welltyped programs the specializer will never be stuck.

Chapter 6

Correctness

In this chapter we will formulate a correctness criterion and prove our binding time analysis correct. Before going on to the actual proof, we will discuss the intuition of the correctness criterion and describe the proof in broad terms. After the proof we will discuss related correctness proofs for binding time analysis.

6.1 The Intuition of Correctness

Before introducing heavy notation and formalism, we find it important to give an intuition of the concepts, we are about to introduce, and to the structure of the proof.

6.1.1 What is Correctness

Correctness of binding time analysis should state that the value of dynamic data should not be needed for specialization and that the result of specialization is correct. We thus specify correctness relative to the semantics of specialization. We want to prove two properties:

1. The mix equation (see section 1.4) holds for our combination of binding time analysis and specializer.
2. Specialization does not go wrong.

The first involves the standard semantics as well as the specialization semantics. What we really prove here is that, when the specializer reduces an expression e (guided by the binding time analysis), it does so in accordance with the standard semantics. The other property states that, when the specializer reduces e, it does not need a value depending on dynamic data. With “does not go wrong”, we thus only think of specialization errors, and not of *e.g.* type errors.

We find that these properties together form a suitable form of correctness, but it is clear that this only gives any kind of correctness under the assumption that the specializer is “reasonable”. Consider a specializer that just inserts the static values, and dumps the code, regardless of the annotations — any analysis could be proven correct *w.r.t.* to this specializer.

6.1.2 Termination Properties

When we stated above that we want to prove the mix equation, this was only partly true. The problem is that specialization might fail to terminate when standard evaluation terminates and vice versa.

We will give two examples of this. The first stems from [Gomard 1991b]. Consider the program $(\lambda \mathrm{x}.2)$@`bomb` where `bomb` is a nonterminating dynamic expression. Clearly specialization will terminate since `bomb` will be discarded, while standard evaluation will fail to terminate due to our call by value semantics.

Similarly, the fact that both branches of a dynamic `if` are evaluated makes it easy to construct an example where standard evaluation terminates when specialization does not: `if true then` 5 `else bomb` where the `if` is dynamic.

Thus what we prove (instead of the mix equation) is that, if specialization of e results in an expression e$'$ and standard evaluation of e results in a value $v$then standard evaluation of e$'$ yields v.

As suggested above, the problem of specialization terminating when standard evaluation does not, is due to the call-by-value standard semantics *vs.* the inherent call-by-name semantics of the specializer. It would be interesting to look at how correctness could be stated if the standard semantics were call-by-name — we would expect something like: if specialization of e results in an expression e$'$ and standard evaluation of e$'$ results in a value v, then standard evaluation of e yields v.

That standard evaluation can terminate when specialization does not, is due to the "super-eagerness" of specialization, and can probably only be avoided by use of a termination analysis.

6.1.3 Central Concepts and Definitions

To state the mix equation more formally, we need to make the notion of equality precise. First we define equality $\equiv$ between standard values — the definition should be fairly straight forward, pairs are equal if their components are, closures (also recursive) are equal if they evaluate to equal values when applied to equal values. Thus $\equiv$ corresponds closely to usual β-equality (though, as we will see, not symmetric *w.r.t.* termination).

Using this definition, we define an equality relation $\mathcal{E}$ between residual expressions $\underline{\mathrm{e}}$ with a run-time environment and values v. The relation holds if $\underline{\mathrm{e}}$ in the run-time environment evaluates to a value $\equiv$-equal to v.

The mix-equation part of the correctness theorem is now stated as

If e specializes to $\underline{\mathrm{e}}$ and e evaluates to v, then $\underline{\mathrm{e}}$ and v are $\mathcal{E}$-equal.

This theorem should hold under the following conditions:

1. The full environment ρ (for standard evaluation), the compile-time environment ρ_s and the run-time environment ρ_d should *agree*, intuitively meaning that ρ should be "composed" of ρ_s and ρ_d.

2. The type environment A used for annotating e *suits* the compile time environment ρ_s, that is, the variables assumed static in A are available in ρ_s.

3. A substitution ζ exists, such that $\zeta \Vdash C$. In other words ζe is well annotated.

6.1.4 Intuition of the Proof

The proof is a simple induction proof over the binding time inference of the expression. The most “tricky” part is the proof of the case for `fix`. We deal with this problem, by (in subsection 6.3.1) annotating `fix`-expressions with a number n, being the number of necessary unfolding for evaluating the expression — applying expressions $\texttt{fix}_0$ f x.e loops. Now the `fix` case can be handled by a local induction proof over n.

6.2 Basic Properties

In this section we will prove a property of our typesystem and a property of the specializer.

Proposition 6.1 (*Substitution lemma for type assignment*)

$$A[\mathrm{x}{:}\sigma]\vdash \mathrm{e}{:}\sigma' \wedge A\vdash \mathrm{e}'{:}\sigma \Longrightarrow A\vdash \mathrm{e}[\mathrm{e}'/\mathrm{x}]{:}\sigma'$$

Proof
Induction over the derivation of $A[\mathrm{x}{:}\sigma]\vdash \mathrm{e}{:}\sigma'$. $\square$(Prop.6.1)

The substitution lemma for type assignment is usually used as a basis for proving subject reduction. Since we have not defined reduction rules (small step semantics), it would make no sense. We have chosen not to give reduction rules, since our prime objective is to prove correctness *w.r.t.* the (big step) semantics of chapters 2 and 5 (see however the discussion in section 13.2.1).

A similar property holds for reduction.

Proposition 6.2 (*Substitution lemma for specialization*)

$$\rho[\mathrm{x}{:}v]\vdash \mathrm{e}\longrightarrow_{spec} v' \wedge \rho\vdash \mathrm{e}'\longrightarrow_{spec} v \Longrightarrow A\vdash \mathrm{e}[\mathrm{e}'/\mathrm{x}]\longrightarrow_{spec} v'$$

Proof
Induction over the derivation of $\rho[\mathrm{x}{:}v]\vdash \mathrm{e}\longrightarrow_{spec} v'$. $\square$(Prop.6.2)

The same thing can of course be proved for the standard semantics, but we do not need this property.

6.3 Preliminaries

This section introduces the basic definitions.

(rec-appl) $$\frac{\rho\vdash \mathrm{e}\longrightarrow_{std}[\![\mathtt{fix}_0\ \mathrm{f}\ \mathrm{x.e}']\!]\rho'}{\rho\vdash \mathrm{e}@\mathrm{e}''\longrightarrow_{std}\bot}$$

(rec-appl) $$\frac{\begin{array}{l}\rho\vdash \mathrm{e}\longrightarrow_{std}[\![\mathtt{fix}_{n+1}\ \mathrm{f}\ \mathrm{x.e}']\!]\rho'\\ \rho\vdash \mathrm{e}''\longrightarrow_{std}v\quad \rho'[\mathrm{f}\mapsto[\![\mathtt{fix}_n\ \mathrm{f}\ \mathrm{x.e}']\!]\rho',\mathrm{x}\mapsto v]\vdash \mathrm{e}'\longrightarrow_{std}v'\end{array}}{\rho\vdash \mathrm{e}@\mathrm{e}''\longrightarrow_{std}v'}$$

(fix) $$\frac{}{\rho\vdash \mathtt{fix}_n\ \mathrm{f}\ \mathrm{x.e}\longrightarrow_{std}[\![\mathtt{fix}_n\ \mathrm{f}\ \mathrm{x.e}]\!]\rho}$$

Figure 6.1: Rewritten standard semantics for `fix`

(rec-appl) $$\frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}[\![\mathtt{fix}_0\ \mathrm{f}\ \mathrm{x.e}']\!]\rho'}{\rho\vdash \mathrm{e}@^{\mathsf{S}}\mathrm{e}''\longrightarrow_{spec}\bot}$$

(rec-appl) $$\frac{\begin{array}{l}\rho\vdash \mathrm{e}\longrightarrow_{spec}[\![\mathtt{fix}_{n+1}\ \mathrm{f}\ \mathrm{x.e}']\!]\rho'\\ \rho\vdash \mathrm{e}''\longrightarrow_{spec}v\quad \rho'[\mathrm{f}\mapsto[\![\mathtt{fix}_n\ \mathrm{f}\ \mathrm{x.e}']\!]\rho',\mathrm{x}\mapsto v]\vdash \mathrm{e}'\longrightarrow_{spec}v'\end{array}}{\rho\vdash \mathrm{e}@^{\mathsf{S}}\mathrm{e}''\longrightarrow_{spec}v'}$$

($\underline{\text{appl}}$) $$\frac{\rho\vdash \mathrm{e}\longrightarrow_{spec}\underline{\mathrm{e}}\qquad \rho\vdash \mathrm{e}'\longrightarrow_{spec}\underline{\mathrm{e}}'}{\rho\vdash \mathrm{e}@^{\mathsf{D}}\mathrm{e}'\longrightarrow_{spec}\underline{\mathrm{e}}\ \underline{@}\ \underline{\mathrm{e}}'}$$

(fix) $$\frac{}{\rho\vdash \mathtt{fix}^{\mathsf{S}}_n\ \mathrm{f}\ \mathrm{x.e}\longrightarrow_{spec}[\![\mathtt{fix}_n\ \mathrm{f}\ \mathrm{x.e}]\!]\rho}$$

($\underline{\text{fix}}$) $$\frac{\rho[\mathrm{f}\mapsto\underline{\mathrm{f}},\mathrm{x}\mapsto\underline{\mathrm{x}}]\vdash \mathrm{e}\longrightarrow_{spec}\underline{\mathrm{e}}}{\rho\vdash \mathtt{fix}^{\mathsf{D}}_n\mathrm{f}\ \mathrm{x.e}\longrightarrow_{spec}\underline{\mathtt{fix}}_n\ \underline{\mathrm{f}}\ \underline{\mathrm{x}}.\ \underline{\mathrm{e}}}\ \textit{where } \underline{\mathrm{x}} \textit{ and } \underline{\mathrm{f}} \textit{ are fresh variables}$$

Figure 6.2: Rewritten specializer semantics for `fix`

6.3.1 Annotating `fix`

We rewrite the standard semantics and the specializer as in figure 6.1 *resp.* 6.2. Every `fix`-combinator is annotated with a number n, being the number of necessary unfolding for evaluating the expression — applying expressions $\texttt{fix}_0$ f x.e loops

One can easily see, that every `fix` expression can be annotated with an n such that these semantics are equivalent to the original one: for every terminating `fix`-expression choose an appropriate n, for any non-terminating one any n will do.

6.3.2 Forgetful Functions

We need some functions "forgetting" unimportant information: 1) to standard evaluate an expression in Exp_{spec}, the annotations should be removed, 2) residual expression in $\underline{\mathit{Val}}$ should be made evaluable by mapping them to expressions in Exp_{std}, 3) the correctness theorem does not include any reference to the standard types, so these should be "forgotten" from compound types.

Definition 6.1 defines a function mapping annotated terms in Exp_{spec} to unannotated terms in Exp_{std},

Definition 6.1
The annotation forgetting function $\phi : \mathit{Exp}_{spec} \rightarrow \mathit{Exp}_{std}$ is defined by:

$$\begin{aligned}
\phi(\texttt{true}) &= \texttt{true}\\
\phi(\texttt{false}) &= \texttt{false}\\
\phi(n) &= n\\
\phi(\mathrm{x}) &= \mathrm{x}\\
\phi(\texttt{if}^b\ \mathrm{e}\ \texttt{then}\ \mathrm{e}'\ \texttt{else}\ \mathrm{e}'') &= \texttt{if}\ \phi(\mathrm{e})\ \texttt{then}\ \phi(\mathrm{e}')\ \texttt{else}\ \phi(\mathrm{e}'')\\
\phi(\lambda^b \mathrm{x.e}) &= \lambda \mathrm{x}.\phi(\mathrm{e})\\
\phi(\mathrm{e}@^b\mathrm{e}') &= \phi(\mathrm{e})@\phi(\mathrm{e}')\\
\phi(\mathrm{e}\texttt{op}^b\mathrm{e}') &= \phi(\mathrm{e})\texttt{op}\phi(\mathrm{e}')\\
\phi(\texttt{fix}^b\mathrm{f\ x.e}) &= \texttt{fix}\ \mathrm{f\ x}.\phi(\mathrm{e})\\
\phi(\texttt{pair}^b(\mathrm{e},\mathrm{e}')) &= \texttt{pair}(\phi(\mathrm{e}),\phi(\mathrm{e}'))\\
\phi(\texttt{fst}^b\mathrm{e}) &= \texttt{fst}\ \phi(\mathrm{e})\\
\phi(\texttt{snd}^b\mathrm{e}) &= \texttt{snd}\ \phi(\mathrm{e})\\
\phi(\Lambda\beta.\mathrm{e}) &= \phi(\mathrm{e})\\
\phi(\texttt{let}\ \mathrm{x} = \mathrm{e}'\texttt{in}\ \mathrm{e}) &= \texttt{let}\ \mathrm{x} = \phi(\mathrm{e}')\ \texttt{in}\ \phi(\mathrm{e})\\
\phi(\Lambda\beta\leadsto\beta'.\mathrm{e}) &= \phi(\mathrm{e})\\
\phi(\mathrm{e}\diamond b) &= \phi(\mathrm{e})\\
\phi([b\leadsto b']\mathrm{e}) &= \phi(\mathrm{e})\\
\phi(\mathrm{e}\Box b\leadsto b') &= \phi(\mathrm{e})
\end{aligned}$$

$\Box$(Def.6.1)

Definition 6.1 defines the isomorphism mentioned in chapter 5.

Definition 6.2

We use φ to denote the "underline forgetting" isomorphism $\varphi : \underline{Val} \rightarrow Exp_{std}$. □(Def.6.2)

Function φ is an isomorphism since the underlines merely serve as syntax distinguishing objects in $\underline{Val}$ from objects in Exp_{std} (remember though that the underlines on variables served to introduce new variable names).

Finally we have a standard type forgetting function ψ mapping closed compound type schemes to *pure* binding time type schemes:

Definition 6.3
The standard type forgetting function ψ is defined by:

$$\begin{array}{lcl}
\psi((\text{int},\mathsf{S})) & = & \mathsf{S} \\
\psi((\text{bool},\mathsf{S})) & = & \mathsf{S} \\
\psi((\kappa \times \kappa',\mathsf{S})) & = & \psi(\kappa)\times\psi(\kappa') \\
\psi((\kappa \rightarrow \kappa',\mathsf{S})) & = & \psi(\kappa)\rightarrow\psi(\kappa') \\
\psi(\forall\beta.\sigma) & = & \forall\beta.\psi(\sigma) \\
\psi(\beta \leq \beta' \Rightarrow \sigma) & = & \beta \leq \beta' \Rightarrow \psi(\sigma) \\
\psi((t,\mathsf{D})) & = & \mathsf{D}
\end{array}$$

□(Def.6.3)

By abuse of notation we will use b to range over pure binding time types S, D, $b \times b'$ and $b \rightarrow b'$, and use σ to range over pure binding time type schemes $\forall\beta.\sigma$, $\beta \leq \beta' \Rightarrow \sigma$ and b. The meaning should be clear from the context.

We can say that function ψ maps compound types from our Nielson and Nielson setting into a binding time types á la Gomard.

6.3.3 Types and Values

Now we define the concept of *suit*, stating a natural relationship between types and values.

Definition 6.4
We say that a pure binding time typescheme σ *suits* a value $v \in Val_{spec}$ iff one of the following conditions holds:

1. (a) $\sigma = \mathsf{S}$, and
   (b) $v \in Const$
2. (a) $\sigma = b_1 \times b_2$ and $v = (v_1, v_2) \in Val_{spec} \times Val_{spec}$, and
   (b) b_1 suits $v_1 \ \wedge\ b_2$ suits v_2.
3. (a) $\sigma = b' \rightarrow b''$ and $v = [\![\lambda\text{x.e}]\!]\rho \in Funval_{spec}$, and
   (b) $\forall v' \in Val_{spec} : b'$ suits $v' \wedge \rho\vdash[v'/\text{x}]\text{e}\longrightarrow_{spec}v'' \Longrightarrow b''$ suits v''
4. $\sigma = b' \rightarrow b''$ and $v = [\![\mathtt{fix}_0 \text{ f x.e}]\!]\rho \in RecFunval_{spec}$
5. (a) $\sigma = b' \rightarrow b''$ and $v = [\![\mathtt{fix}_{n+1} \text{ f x.e}]\!]\rho \in RecFunval_{spec}$, and

(b) $\forall v' \in \mathit{Val}_{spec} : b'$ suits $v' \land \rho \vdash [\mathtt{fix}^{\mathsf{S}}_{n}\ \mathrm{f\ x/f}, v'/\mathrm{x}]\mathrm{e} \longrightarrow_{spec} v'' \Longrightarrow b''$ suits v''

6. (a) $\sigma = \forall\beta.\sigma'$ and $v = [\![\Lambda\beta.\mathrm{e}]\!]\rho \in \mathit{BtFunval}_{spec}$ and

   (b) $\forall b \in \{\mathsf{S}, \mathsf{D}\} : \rho \vdash [b/\beta]e \longrightarrow_{spec} v' \Longrightarrow [b/\beta]\sigma'$ suits v'

7. (a) $\sigma = b \leq b' \Rightarrow \sigma'$ and $v = [\![\Lambda b \rightsquigarrow b'.\mathrm{e}]\!]\rho \in \mathit{ClFunval}_{spec}$ and

   (b) $\rho \vdash \mathrm{e} \longrightarrow_{spec} v' \Longrightarrow \sigma'$ suits v'

8. (a) $\sigma = \mathsf{D}$ and

   (b) $v \in \underline{\mathit{Val}}$

A type environment A suits an environment $\rho \in \mathit{Varenv}_{spec}$, iff for all variables x bound by ρ, $\psi(A(\mathrm{x}))$ suits $\rho(\mathrm{x})$. □(Def.6.4)

6.3.4 Equality

As a notational convenience we use $\rho \in \mathit{Varenv}_{std}$ to denote standard evaluation environments while $\rho_s \in \mathit{Varenv}_{spec}$ is used for compile-time environments (for specialization) and $\rho_d \in \mathit{Varenv}_{std}$ for run-time environments.

Definition 6.5
Equality between values v and v' in Val_{std} is defined by:

$$
v \equiv v' \quad \overset{def}{\Longleftrightarrow} \quad
\begin{cases}
\begin{array}{ll}
(1) & v, v' \in \mathit{Const} \land v = v', \text{ or} \\
(2) & v = (v_1, v_2) \in \mathit{Val}_{std} \times \mathit{Val}_{std} \land \\
 & v' = (v'_1, v'_2) \in \mathit{Val}_{std} \times \mathit{Val}_{std} \land \\
 & v_1 \equiv v'_1 \land v_2 \equiv v'_2, \text{ or} \\
(3) & v = [\![\lambda \mathrm{x}.e]\!]\rho \in \mathit{Funval}_{std} \land v' = [\![\lambda \mathrm{x}'.e']\!]\rho' \in \mathit{Funval}_{std} \land \\
 & \forall v_1, v_2 \in \mathit{Val}_{std} : v_1 \equiv v_1' \land \rho[\mathrm{x} \mapsto v_1] \vdash \mathrm{e} \longrightarrow_{std} v_2 \Longrightarrow \\
 & \qquad\qquad \rho_d'[\mathrm{x} \mapsto v_1'] \vdash \mathrm{e}' \longrightarrow_{std} v_2' \\
 & \qquad\qquad \land v_2 \equiv v_2', \text{ or} \\
(4) & v = [\![\mathtt{fix}_{n+1}\ \mathrm{f\ x.e}]\!]\rho \in \mathit{RecFunval}_{std} \land \\
 & v' = [\![\mathtt{fix}_{m+1}\ \mathrm{f\ x.e'}]\!]\rho' \in \mathit{RecFunval}_{std} \land \\
 & \forall v_1, v_2 \in \mathit{Val}_{std} : v_1 \equiv v_1' \land \\
 & \qquad\qquad \rho[\mathrm{f} \mapsto [\![\mathtt{fix}_{n}\ \mathrm{f\ x.e}]\!]\rho, \mathrm{x} \mapsto v_1] \vdash \mathrm{e} \longrightarrow_{std} v_2 \Longrightarrow \\
 & \qquad\qquad \rho_d'[\mathrm{f} \mapsto [\![\mathtt{fix}_{m}\ \mathrm{f\ x.e'}]\!]\rho, \mathrm{x} \mapsto v_1'] \vdash \mathrm{e}' \longrightarrow_{std} v_2' \\
 & \qquad\qquad \land v_2 \equiv v_2', \text{ or} \\
(5) & v = [\![\mathtt{fix}_{0}\ \mathrm{f\ x.e}]\!]\rho \in \mathit{RecFunval}_{std} \land \\
 & v' = [\![\mathtt{fix}_{0}\ \mathrm{f\ x.e'}]\!]\rho' \in \mathit{RecFunval}_{std}
\end{array}
\end{cases}
$$

□(Def.6.5)

Note that $v \equiv v'$ is not symmetric (reflexive): in cases (3) and (4) termination of an expression derived from v implies termination of an expression derived from v' but not vice versa. The reason for this can be found in the uses of $v \equiv v'$; v will be the result of standard evaluation of the original program while v' will be the result of standard evaluation of the residual program. The main theorem will state that termination of standard evaluation and of specialization shall imply termination of standard evaluation of the residual program.

Definition 6.6

Equality between values in Val_{std} and values in Val_{spec} (residual expressions) with environment ρ_d is defined by:

$$
\begin{array}{lcl}
\mathcal{E}_{\mathsf{S}}(v, v', \rho_d) & \overset{def}{\Longleftrightarrow} & v, v' \in Const \wedge v \equiv v' \\
\mathcal{E}_{b_1 \times b_2}(v, v', \rho_d) & \overset{def}{\Longleftrightarrow} & v = (v_1, v_2) \in Val_{std} \times Val_{std} \wedge \\
 & & v' = (v_1', v_2') \in Val_{spec} \times Val_{spec} \wedge \\
 & & \mathcal{E}_{b_1}(v_1, v_1', \rho_d) \wedge \mathcal{E}_{b_2}(v_2, v_2', \rho_d) \\
\mathcal{E}_{b_1 \to b_2}(v, v', \rho_d) & \overset{def}{\Longleftrightarrow} & v = [\![\lambda \mathrm{x.e}]\!]\rho \in Funval_{std} \wedge v' = [\![\lambda \mathrm{x.e}']\!]\rho' \in Funval_{spec} \wedge \\
 & & \forall v_1, v_2 \in Val_{std}, \forall v_1', v_2' \in Val_{spec} \\
 & & \mathcal{E}_{b_1}(v_1, v_1', \rho_d) \wedge \\
 & & \rho[\mathrm{x} \mapsto v_1] \vdash \mathrm{e} \longrightarrow_{std} v_2 \wedge \rho'[\mathrm{x} \mapsto v_1'] \vdash \mathrm{e}' \longrightarrow_{spec} v_2' \\
 & & \Longrightarrow \mathcal{E}_{b_2}(v_2, v_2', \rho_d) \\
 & \vee & v = [\![\mathtt{fix}_{n+1}\ \mathrm{f\ x.e}]\!]\rho \in RecFunval_{std} \wedge \\
 & & v' = [\![\mathtt{fix}_{n+1}\ \mathrm{f\ x.e}']\!]\rho' \in RecFunval_{spec} \wedge \\
 & & \forall v_1 \in Val_{std}, \forall v_1' \in Val_{spec} : \\
 & & \mathcal{E}_{b_1}(v_1, v_1', \rho_d) \wedge \\
 & & \rho[\mathrm{f} \mapsto [\![\mathtt{fix}_n\ \mathrm{f\ x.e}]\!]\rho, \mathrm{x} \mapsto v_1] \vdash \mathrm{e} \longrightarrow_{std} v_2 \wedge \\
 & & \rho'[\mathrm{f} \mapsto [\![\mathtt{fix}_n\ \mathrm{f\ x.e}']\!]\rho', \mathrm{x} \mapsto v_1'] \vdash \mathrm{e}' \longrightarrow_{spec} v_2' \\
 & & \Longrightarrow \mathcal{E}_{b_2}(v_2, v_2', \rho_d) \\
 & \vee & v = [\![\mathtt{fix}_0\ \mathrm{f\ x.e}]\!]\rho \in RecFunval_{std} \wedge \\
 & & v' = [\![\mathtt{fix}_0\ \mathrm{f\ x.e}']\!]\rho' \in RecFunval_{spec} \\
\mathcal{E}_{\forall \beta . \sigma}(v, v', \rho_d) & \overset{def}{\Longleftrightarrow} & v' = [\![\Lambda \beta . \mathrm{e}]\!]\rho_s \in BtFunval_{spec} \wedge \\
 & & \forall b \in \{\mathsf{S}, \mathsf{D}\} : \rho_s \vdash [b/\beta]e \longrightarrow_{spec} v'' \Longrightarrow \mathcal{E}_{[b/\beta]\sigma}(v, v'', \rho_d) \\
\mathcal{E}_{b \leq b' \Rightarrow \sigma}(v, v', \rho_d) & \overset{def}{\Longleftrightarrow} & v' = [\![\Lambda b \rightsquigarrow b' . \mathrm{e}]\!]\rho_s \in ClFunval_{spec} \wedge \\
 & & \rho_s \vdash e \longrightarrow_{spec} v'' \Longrightarrow \mathcal{E}_{\sigma}(v, v'', \rho_d) \\
\mathcal{E}_{\mathsf{D}}(v, v', \rho_d) & \overset{def}{\Longleftrightarrow} & v' \in \underline{Val} \wedge \rho_d \vdash \varphi(v') \longrightarrow_{std} v'' \wedge\ v \equiv v''
\end{array}
$$

□(Def.6.6)

Relations $\equiv$ and $\mathcal{E}_\sigma$ are well defined since they are defined inductively on σ (implicitly in the case of $\equiv$). There is more than an accidental similarity between the definition of $\mathcal{E}$ and the definition of *suit*. The relationship between the two is stated in proposition 6.3.

Proposition 6.3

$$\forall v \in Val_{spec} \forall \sigma : (\exists v' \in Val_{std} \exists \rho_d \in Varenv_{std} : \mathcal{E}_\sigma(v', v, \rho_d)) \Longrightarrow \sigma \text{ suits } v$$

Proof

Easy by induction over the structure of σ. Use proposition 6.2. $\square$(Prop.6.3)

6.3.5 Towards the Theorem

We can define well behavedness of ρ_s,ρ_d and ρ in terms of $\mathcal{E}$.

Definition 6.7
Given a set of identifiers *VarSet*, and three environments $\rho_s \in \mathit{Varenv}_{spec}$,$\rho_d \in \mathit{Varenv}_{std}$,$\rho \in \mathit{Varenv}_{std}$ and a type environment A that suits ρ_s, we say that ρ_s,ρ_d,ρ *agree* on *VarSet* iff $\forall$x$\in$ *VarSet*: $\mathcal{E}_{\psi(A(x))}(\rho(\mathrm{x}),\rho_s(\mathrm{x}),\rho_d)$ $\square$(Def.6.7)

That ρ_s,ρ_d,ρ *agree* on *VarSet* means what we expect; for base values we have: 1) If x is static, $\rho_s(\mathrm{x}) = \rho(\mathrm{x})$, 2) If x is dynamic, $\rho_d(\varphi(\rho_s(\mathrm{x}))) = \rho(\mathrm{x})$.

A map from binding time variables β to binding time values S and D, mapping all variables to a value is called a *ground substitution.*

Definition 6.8
We say A,$C$$\models$e:$\sigma$ iff $\forall\rho_s \in \mathit{Varenv}_{spec}$, ρ_d, $\rho \in \mathit{Varenv}_{std}$, ground substitutions ζ:

$$\underbrace{\zeta(A) \text{ suits } \rho_s \ \wedge \zeta \Vdash C \wedge \rho_s, \rho_d, \rho \text{ agree on } \mathit{FreeVars}(\mathrm{e})}$$
$$\Downarrow$$
$$\overbrace{\forall v \in \mathit{Val}_{std} \forall v' \in \mathit{Val}_{spec} : \rho_s \vdash \mathrm{e} \longrightarrow_{spec} v' \ \wedge \ \rho \vdash \phi(\mathrm{e}) \longrightarrow_{std} v \Longrightarrow \mathcal{E}_{\psi(\sigma)}(v, v', \rho_d)}$$

$\square$(Def.6.8)

6.4 The Correctness Theorem

In this section we state and prove the main correctness theorem.

6.4.1 Theorem and Corollaries

We are now able to state our main correctness theorem.

Theorem I (*Main Correctness Theorem – Soundness of BTA*)

$$\forall A, C, \sigma : A, C \vdash \mathrm{e}\colon \sigma \implies A, C \models \mathrm{e}\colon \sigma$$

Proof
By induction on the binding time inference of e. The proofs of the various cases can be found in lemmas 6.6 to 6.21. $\square$(Theorem I)

A variant of the Mix Equation follows as a corollary (where equality = is defined on pairs as equality of subparts).

Corollary 6.4 (*Mix Equation*)
For any well annotated dynamic program p resulting in a first order value, the following holds

$$\left.\begin{array}{l} \rho\vdash\phi(p)\longrightarrow_{std} v \\ \rho_s\vdash p\longrightarrow_{spec} \underline{e} \\ \rho_d\vdash\varphi(\,\underline{e}\,)\longrightarrow_{std} v' \end{array}\right\} \Longrightarrow v = v'$$

if ρ_s,ρ_d and ρ agree on *FreeVars*(p). For illustration see figure 1.9.
Proof
By definition of $\mathcal{E}_{\mathsf{D}}$ $\square$(Cor.6.4)

Also the desired property "Specialization does not go wrong" can be derived from theorem I.

Corollary 6.5 (*Specialization does not go wrong*)
$\forall A, C, \sigma, \rho_s, \rho_d, \rho, \zeta, v, v'$:

$$\left.\begin{array}{l} A, C\vdash \mathrm{e} : \sigma \\ \zeta(A) \text{ suits } \rho_s \\ \zeta\Vdash C \\ \rho_s, \rho_d, \rho \text{ agree on } \mathit{FreeVars}(\mathrm{e}) \\ \rho_s\vdash \mathrm{e}\longrightarrow_{spec} v' \\ \rho\vdash\phi(\mathrm{e})\longrightarrow_{std} v \end{array}\right\} \Longrightarrow v' \neq \varepsilon$$

Proof
Follows from the fact that $\neg\exists v, \rho_d, \sigma : \mathcal{E}_\sigma(v, \varepsilon, \rho_d)$. $\square$(Cor.6.5)

In corollary 6.5, "specialization does not go wrong" requires termination of standard evaluation. The corollary would not follow directly from theorem I without this assumption. It is however not a necessary assumption for the corollary to hold.

Theorem I is not a proof of soundness of the standard type system (though it might be extended to cover this as well). This can be seen from the definition of $\mathcal{E}_{\mathsf{D}}$ which covers all different standard types. Instead we take standard soundness for granted and use the fact that the program is well typed a couple of places in the proof.

6.4.2 Proof of correctness

This subsection contains the proof of theorem I. We define $\mathcal{H}$(e) to be the predicate $\forall A, C, \sigma : A, C\vdash \mathrm{e} : \sigma \Longrightarrow A, C\models \mathrm{e} : \sigma$. For every rule in the binding time inference (figure 4.2 to 4.4):

$$(\mathit{rule}) \quad \frac{\sqcup,\sqcup\vdash \mathrm{e}_1:\sqcup \quad \cdots \quad \sqcup,\sqcup\vdash \mathrm{e}_n:\sqcup}{\sqcup,\sqcup\vdash \mathrm{e}:\sqcup}$$

we show $\mathcal{H}(\mathrm{e}_1)\wedge\cdots\wedge\mathcal{H}(\mathrm{e}_n)\Rightarrow\mathcal{H}(\mathrm{e})$.

LEMMA 6.6
Rule (const):

$$\forall n : \mathcal{H}(n)$$
$$\forall \mathrm{b} \in \{\texttt{true}, \texttt{false}\} : \mathcal{H}(\mathrm{b})$$

Proof
Two (similar) things to prove:
$\boxed{\text{Int}}$ According to our binding time rules the only correct typing is $A,C\vdash n$:(int,S) for all A and C. By the evaluation rules $\rho_s\vdash n\longrightarrow_{spec}n$ and $\rho\vdash\phi(n)\longrightarrow_{std}n$. We have $n\equiv n$ and thus $\mathcal{E}_{\mathsf{S}}(n, n, \rho_d)$.
$\boxed{\text{Bool}}$ Similar to above. $\Box$(Lemma 6.6)

LEMMA 6.7
Rule (var):

$$\forall x \in \mathit{Var} : \mathcal{H}(x)$$

Proof
By the typing rule we have $A,C\vdash$x:A(x). Since ρ_s,ρ_d,ρ *agree* on *VarSet*, we have $\mathcal{E}_{\psi(A(x))}(\rho(\mathrm{x}),\rho_s(\mathrm{x}),\rho_d)$. The desired conclusion follows immediately.
$\Box$(Lemma 6.7)

LEMMA 6.8
Rule (if):

$$\forall \mathrm{e},\mathrm{e}',\mathrm{e}'' \colon \mathcal{H}(\mathrm{e})\wedge\mathcal{H}(\mathrm{e}')\wedge\mathcal{H}(\mathrm{e}'') \Rightarrow \mathcal{H}(\texttt{if}^b\ \mathrm{e}\ \texttt{then}\ \mathrm{e}'\ \texttt{else}\ \mathrm{e}'')$$

Proof
Assume A,C and σ such that $A,C\vdash\texttt{if}^b$ e $\texttt{then}$ e$'$ $\texttt{else}$ e$''$:σ. Assume ρ_s, ρ_d, ρ and ζ such that $\zeta(A)$ suits ρ_s, $\zeta\Vdash C$ and ρ_s,ρ_d,ρ agree on *FreeVars*($\texttt{if}^b$ e $\texttt{then}$ e$'$ $\texttt{else}$ e$''$).
$\boxed{\zeta b = \mathsf{S}}$ It follows from the binding time inference rule that $\zeta A,C\vdash\texttt{if}^{\mathsf{S}}$ ζe $\texttt{then}$ ζe$'$ $\texttt{else}$ ζe$''$:κ.
We have *FreeVars*($\texttt{if}^{\mathsf{S}}$ e $\texttt{then}$ e$'$ $\texttt{else}$ e$''$) = *FreeVars*(e)$\bigcup$*FreeVars*(e$'$)$\bigcup$*FreeVars*(e$''$). It now follows from the induction hypothesis and the binding time rules:

$$\rho_s\vdash\zeta\mathrm{e}\longrightarrow_{spec}v_1 \ \wedge\ \rho\vdash\phi(\zeta\mathrm{e})\longrightarrow_{std}v_2 \Longrightarrow \mathcal{E}_{\mathsf{S}}(v_1, v_2, \rho_d)$$
$$\rho_s\vdash\zeta\mathrm{e}'\longrightarrow_{spec}v_1{}' \ \wedge\ \rho\vdash\phi(\zeta\mathrm{e}')\longrightarrow_{std}v_2{}' \Longrightarrow \mathcal{E}_{\psi(\kappa)}(v_1{}', v_2{}', \rho_d)$$
$$\rho_s\vdash\zeta\mathrm{e}''\longrightarrow_{spec}v_1{}'' \ \wedge\ \rho\vdash\phi(\mathrm{e}'')\longrightarrow_{std}v_2{}'' \Longrightarrow \mathcal{E}_{\psi(\zeta\kappa)}(v_1{}'', v_2{}'', \rho_d)$$

Since $\mathcal{E}_{\mathsf{S}}(v_1, v_2, \rho_d) \Longrightarrow v_1\equiv v_2$, we can then deduce either

$$\left.\begin{array}{l}\rho_s\vdash\texttt{if}^{\mathsf{S}}\ \zeta\text{e}\ \texttt{then}\ \zeta\text{e}'\ \texttt{else}\ \zeta\text{e}''\longrightarrow_{spec}v_1{}'\ \wedge\\ \rho\vdash\phi(\texttt{if}^{\mathsf{S}}\ \text{e}\ \texttt{then}\ \text{e}'\ \texttt{else}\ \text{e}'')\longrightarrow_{std}v_2{}'\end{array}\right\}\Longrightarrow\mathcal{E}_{\psi(\kappa)}(v_1{}',v_2{}',\rho_d)$$

or

$$\left.\begin{array}{l}\rho_s\vdash\texttt{if}^{\mathsf{S}}\ \zeta\text{e}\ \texttt{then}\ \zeta\text{e}'\ \texttt{else}\ \zeta\text{e}''\longrightarrow_{spec}v_1{}''\ \wedge\\ \rho\vdash\phi(\texttt{if}^{\mathsf{S}}\ \text{e}\ \texttt{then}\ \text{e}'\ \texttt{else}\ \text{e}'')\longrightarrow_{std}v_2{}''\end{array}\right\}\Longrightarrow\mathcal{E}_{\psi(\kappa)}(v_1{}'',v_2{}'',\rho_d)$$

depending on the value of $v_1\equiv v_2$.

$\boxed{\zeta b=\mathsf{D}}$ It follows from the binding time inference rule and the fact $\zeta\Vdash C$ that $\zeta A{,}C\vdash\texttt{if}^{\mathsf{D}}\ \zeta\text{e}\ \texttt{then}\ \zeta\text{e}'\ \texttt{else}\ \zeta\text{e}''{:}(t',\mathsf{D})$.

Since $\mathit{FreeVars}(\texttt{if}^{\mathsf{D}}\ \text{e}\ \texttt{then}\ \text{e}'\ \texttt{else}\ \text{e}'') = \mathit{FreeVars}(\text{e})\bigcup\mathit{FreeVars}(\text{e}')\bigcup\mathit{FreeVars}(\text{e}'')$, it follows from the induction hypothesis and the binding time rules that

$$\begin{array}{l}\rho_s\vdash\zeta\text{e}\longrightarrow_{spec}v_1\ \wedge\ \rho\vdash\phi(\text{e})\longrightarrow_{std}v_2\Longrightarrow\mathcal{E}_{\mathsf{D}}(v_1,v_2,\rho_d)\\ \rho_s\vdash\zeta\text{e}'\longrightarrow_{spec}v_1{}'\ \wedge\ \rho\vdash\phi(\text{e}')\longrightarrow_{std}v_2{}'\Longrightarrow\mathcal{E}_{\mathsf{D}}(v_1{}',v_2{}',\rho_d)\\ \rho_s\vdash\zeta\text{e}''\longrightarrow_{spec}v_1{}''\ \wedge\ \rho\vdash\phi(\text{e}'')\longrightarrow_{std}v_2{}''\Longrightarrow\mathcal{E}_{\mathsf{D}}(v_1{}'',v_2{}'',\rho_d)\end{array}$$

The specializer rules tells us that $\rho_s\vdash\texttt{if}^{\mathsf{D}}\ \zeta\text{e}\ \texttt{then}\ \zeta\text{e}'\ \texttt{else}\ \zeta\text{e}''\longrightarrow_{spec}\underline{\texttt{if}}\ v_1\ \underline{\texttt{then}}\ v_2\ \underline{\texttt{else}}\ v_3$ when $\rho_s\vdash\zeta\text{e}\longrightarrow_{spec}v_1$, $\rho_s\vdash\zeta\text{e}'\longrightarrow_{spec}v_2$ and $\rho_s\vdash\zeta\text{e}''\longrightarrow_{spec}v_3$. If

$$\begin{array}{l}\rho\vdash\phi(\text{e})\longrightarrow_{std}v_1{}''\\ \rho\vdash\phi(\text{e}')\longrightarrow_{std}v_2{}''\\ \rho\vdash\phi(\text{e}'')\longrightarrow_{std}v_3{}''\end{array}$$

we have by the definition of $\mathcal{E}_{\mathsf{D}}$ that

$$\begin{array}{l}\rho_d\vdash\varphi(v_1)\longrightarrow_{std}v_1{}'\wedge v_1{}'\equiv v_1{}''\\ \rho_d\vdash\varphi(v_2)\longrightarrow_{std}v_2{}'\wedge v_2{}'\equiv v_2{}''\\ \rho_d\vdash\varphi(v_3)\longrightarrow_{std}v_3{}'\wedge v_3{}'\equiv v_3{}''\end{array}$$

Since e has type Bool by the type rule, $v_1{}'\equiv v_1{}''$ implies $v_1{}' = v_1{}''$. We now have $\rho\vdash\phi(\texttt{if}^{\mathsf{D}}\text{e}\ \texttt{then}\ \text{e}'\ \texttt{else}\ \text{e}'')\longrightarrow_{std}v'$ implies $\rho_d\vdash\varphi(\underline{\texttt{if}}\ v_1\ \underline{\texttt{then}}\ v_2\ \underline{\texttt{else}}\ v_3)\longrightarrow_{std}v\ \wedge$ $v\equiv v'$. □(Lemma 6.8)

Lemma 6.9
Rule (abstr)

$$\forall\text{e}{:}\mathcal{H}(\text{e})\Rightarrow\mathcal{H}(\lambda^b\text{x.e})$$

Proof

Assume A,C and σ such that $A{,}C\vdash\lambda^b\text{x.e}{:}\sigma$. Assume ρ_s, ρ_d, ρ and ζ such that $\zeta(A)$ suits ρ_s, $\zeta\Vdash C$ and ρ_s,ρ_d,ρ agree on $\mathit{FreeVars}(\lambda^b\text{x.e})$.

$\boxed{\zeta b=\mathsf{S}}$ From the inference rule we have that $\zeta A{,}C\vdash\lambda^{\mathsf{S}}\text{x.}\zeta\text{e}{:}(\kappa'\to\kappa'',\mathsf{S})$. We see that what we wish to prove is:

$$\left.\begin{array}{l}\rho_s \vdash \zeta(\lambda^{\mathsf{S}}\mathrm{x.e}) \longrightarrow_{spec} v' \wedge \\ \rho \vdash \phi(\lambda^{\mathsf{S}}\mathrm{x.e}) \longrightarrow_{std} v\end{array}\right\} \Longrightarrow \mathcal{E}_{\psi(\kappa')\to\psi(\kappa'')}(v, v', \rho_d)$$

By the evaluation rules for abstraction, we have

$$\rho \vdash \phi(\lambda^{\mathsf{S}}\mathrm{x.e}) \longrightarrow_{std} [\![\lambda \mathrm{x}.\phi(\mathrm{e})]\!]\rho \text{ and}$$
$$\rho_s \vdash \zeta(\lambda^{\mathsf{S}}\mathrm{x.e}) \longrightarrow_{spec} [\![\lambda \mathrm{x}.\zeta \mathrm{e}]\!]\rho_s$$

so $v' = [\![\lambda \mathrm{x}.\zeta \mathrm{e}]\!]\rho_s$ and $v = [\![\lambda \mathrm{x}.\phi(\mathrm{e})]\!]\rho$. We now want to prove

$$\mathcal{E}_{\psi(\kappa')\to\psi(\kappa'')}([\![\lambda \mathrm{x}.\phi(\mathrm{e})]\!]\rho, [\![\lambda \mathrm{x}.\zeta \mathrm{e}]\!]\rho_s, \rho_d)$$

Let $v_1 \in \mathit{Val}_{std}$ and $v_1{}' \in \mathit{Val}_{spec}$ be given, such that $\mathcal{E}_{\psi(\kappa')}(v_1, v_1{}', \rho_d)$. What we have to prove is:

$$\rho[\mathrm{x}\mapsto v_1] \vdash \phi(\mathrm{e}) \longrightarrow_{std} v_2 \wedge \rho_s[\mathrm{x}\mapsto v_1{}'] \vdash \zeta \mathrm{e} \longrightarrow_{spec} v_2{}' \Longrightarrow \mathcal{E}_{\psi(\kappa'')}(v_2, v_2{}', \rho_d)$$

According to the typerule we have $A \bigcup \{\mathrm{x}{:}\kappa'\}, C \vdash \mathrm{e}{:}\kappa''$. To use the induction hypothesis we just have to prove

1. $\zeta(A \bigcup \{\mathrm{x} : \kappa'\})$ suits $\rho_s[\mathrm{x}\mapsto v_1{}']$
2. $\rho_s[\mathrm{x}\mapsto v_1{}'], \rho_d, \rho[\mathrm{x}\mapsto v_1]$ agree on $\mathit{FreeVars}(\mathrm{e})$

To show the first, we have to show $\psi(\kappa')$ suits $v_1{}'$. By proposition 6.3, this follows from $\mathcal{E}_{\psi(\kappa')}(v_1, v_1{}', \rho_d)$. The second follows easily.

$\boxed{\zeta b = \mathsf{D}}$ From the binding time rule and the fact that $\zeta \Vdash C$, we have $A, C \vdash \lambda^{\mathsf{D}}\mathrm{x}.\zeta \mathrm{e}{:}((t', \mathsf{D}) \to (t'', \mathsf{D}), \mathsf{D})$.

The binding time inference rule tells us that $A[x : (t', \mathsf{D})] \vdash e : (t'', \mathsf{D})$. Let $\underline{\mathrm{x}}$, v_1, v_2, be given such that $v_1 \equiv v_2$ and $\underline{\mathrm{x}}$ is fresh. Since ρ_s, ρ_d, ρ agree on $\mathit{FreeVars}(\lambda^{\mathsf{D}}\mathrm{x.e})$ and D suits $\underline{\mathrm{x}}$, we have that $\rho[\mathrm{x}\mapsto v_1], \rho_s[\mathrm{x}\mapsto \underline{\mathrm{x}}], \rho_d[\underline{\mathrm{x}}\mapsto v_2]$ agree on $\mathit{FreeVars}(\mathrm{e})$.

From this, the induction hypothesis and the above observation concerning the type of e, we can conclude, that $\forall v \in \mathit{Val}_{std} \forall v' \in \mathit{Val}_{spec}$:

$$\left.\begin{array}{l}\rho[\mathrm{x}\mapsto v_1] \vdash \phi(\mathrm{e}) \longrightarrow_{std} v \wedge \\ \rho_s[\mathrm{x}\mapsto \underline{\mathrm{x}}] \vdash \zeta \mathrm{e} \longrightarrow_{spec} v'\end{array}\right\} \Longrightarrow \mathcal{E}_{\mathsf{D}}(v, v', \rho_d[\underline{\mathrm{x}}\mapsto v_2])$$

By the specialization rule we then have $\rho_s \vdash \lambda^{\mathsf{D}}\mathrm{x}.\zeta \mathrm{e} \longrightarrow_{spec} \underline{\lambda}\, \underline{\mathrm{x}}.v'$. By the standard evaluation rule $\rho_d \vdash \varphi(\underline{\lambda}\, \underline{\mathrm{x}}.v') \longrightarrow_{std} [\![\lambda \mathrm{x}.v']\!]\rho_d$. Similarly we have $\rho \vdash \phi(\lambda^{\mathsf{D}}\mathrm{x.e}) \longrightarrow [\![\lambda \mathrm{x}.\phi(\mathrm{e})]\!]\rho$.

What remains to be shown is $[\![\lambda \mathrm{x}.v']\!]\rho_d \equiv [\![\lambda \mathrm{x}.\phi(\mathrm{e})]\!]\rho$. But from the definition of $\mathcal{E}_{\mathsf{D}}(v, v', \rho_d[\underline{\mathrm{x}}\mapsto v_2])$, we have $\rho_d[\underline{\mathrm{x}}\mapsto v_2] \vdash \varphi(v') \longrightarrow_{std} v''$ and $v \equiv v''$, giving this result. $\square$(Lemma 6.9)

Lemma 6.10
Rule (fix):

$\forall$e:$\mathcal{H}$(e) $\Rightarrow\mathcal{H}(\mathtt{fix}_n^b$f x.e)

Proof

Assume A,C and σ such that $A,C\vdash\mathtt{fix}_n^b$ f x.e:σ. Assume ρ_s, ρ_d, ρ and ζ such that $\zeta(A)$ suits ρ_s, $\zeta\Vdash C$ and ρ_s,ρ_d,ρ agree on *FreeVars*($\mathtt{fix}_n^b$ f x.e).

$\boxed{\zeta b = \mathsf{S}}$ By the inference rule $\zeta\sigma = (\kappa' \to \kappa'', \mathsf{S})$. We have $\rho\vdash\phi(\mathtt{fix}_n^{\mathsf{S}}$ f x.e)$\longrightarrow_{std}[\![\mathtt{fix}_n$ f x.ϕ(e)$]\!]\rho$ and $\rho_s\vdash\mathtt{fix}_n^{\mathsf{S}}$ f x.ζe$\longrightarrow_{spec}[\![\mathtt{fix}_n$ f x.ζe$]\!]\rho_s$. We now proceed with an induction proof over n:

<u>$n = 0$</u>: We have $\mathcal{E}_{\psi(\kappa')\to\psi(\kappa'')}([\![\mathtt{fix}_0$ f x.ϕ(e)$]\!]\rho$,$[\![\mathtt{fix}_0$ f x.ζe$]\!]\rho_s, \rho_d)$ from the definition of $\mathcal{E}$. Then $\mathcal{H}(\mathtt{fix}_0^b$f x.e) follows immediately.

<u>$n > 0$</u>: Assume that $\mathcal{H}(\mathtt{fix}_m^{\mathsf{S}}$f x.e$'$) for all $m < n$, and for all e$'$. We want to prove $\mathcal{E}_{\psi(\kappa')\to\psi(\kappa'')}([\![\mathtt{fix}_n$f x.$\phi$(e)$]\!]\rho$, $[\![\mathtt{fix}_n$f x.ζe$]\!]\rho_s, \rho_d)$.

Let $v_1 \in Val_{std}$ and $v_1{}' \in Val_{spec}$ be given such that $\mathcal{E}_{\psi(\kappa')}(v_1, v_1{}', \rho_d)$ holds. Let $\rho' = \rho[$f$\mapsto[\![\mathtt{fix}_{n-1}$ f x.ζe$]\!]\rho$,x$\mapsto v_1]$ and $\rho_s{}' = \rho_s[$f$\mapsto[\![\mathtt{fix}_{n-1}$ f x.ζe$]\!]\rho_s$,x$\mapsto v_1{}']$. Let $A' = A[$f : $(\kappa' \to \kappa'', \mathsf{S})$, x : $\kappa']$. Now assume that $\rho'\vdash\phi$(e)$\longrightarrow_{std}v_2$ and $\rho_s{}'\vdash\zeta$e$\longrightarrow_{spec}v_2{}'$.

We have $\mathcal{E}_{\psi(\kappa')\to\psi(\kappa'')}([\![\mathtt{fix}_{n-1}$ f x.ϕ(e)$]\!]\rho$, $[\![\mathtt{fix}_{n-1}$ f x.ζe$]\!]\rho_s, \rho_d)$ by the local induction hypothesis. Thus by proposition 6.3 we have that $\psi(\kappa') \to \psi(\kappa'')$ suits $[\![\mathtt{fix}_{n-1}$ f x.ζe$]\!]\rho_s$. Similarly, we have by assumption that $\mathcal{E}_{\psi(\kappa')}(v_1, v_1{}', \rho_d)$ holds, so by proposition 6.3 $\psi(\kappa')$ suits $v_1{}'$. From this we can conclude, that $\zeta(A')$ suits $\rho_s{}'$.

To use the local induction hypothesis, we now just have to prove that $\rho_s{}'$,ρ_d,ρ' agree on *FreeVars*(e). This amounts to prove $\mathcal{E}_{\psi(A'(\mathrm{f}))}(\rho'(\mathrm{f}), \rho_s{}'(\mathrm{f}), \rho_d)$ and $\mathcal{E}_{\psi(A'(\mathrm{x}))}(\rho'(\mathrm{x}), \rho_s{}'(\mathrm{x}), \rho_d)$. Thus we want

$$\mathcal{E}_{\psi(\kappa')\to\psi(\kappa'')}([\![\mathtt{fix}_{n-1} \text{ f x.}\phi(\text{e})]\!]\rho, [\![\mathtt{fix}_{n-1} \text{ f x.}\zeta\text{e}]\!]\rho_s, \rho_d)$$

and

$$\mathcal{E}_{\psi(\kappa')}(v_1, v_1{}', \rho_d)$$

The first follows from our induction hypothesis $\mathcal{H}(\mathtt{fix}_{n-1}^{\mathsf{S}}$f x.e$'$). The second is our assumption on v_1 and $v_1{}'$. Now $\mathcal{H}(\mathtt{fix}_n^{\mathsf{S}}$ f x.e) follows from the global induction hypothesis.

$\boxed{\zeta b = \mathsf{D}}$ From the binding time rule and the fact that $\zeta\Vdash C$ we have $\zeta A,C\vdash\mathtt{fix}_n^{\mathsf{D}}$f x.$\zeta$e:$((t', \mathsf{D}) \to (t'', \mathsf{D}), \mathsf{D})$.

Assume that $\rho_s[$f$\mapsto\underline{\text{f}}$,x$\mapsto\underline{\text{x}}]\vdash$e$\longrightarrow_{spec}\underline{\text{e}}$, we then have $\rho_s\vdash\mathtt{fix}_n^{\mathsf{D}}$f x.$\zeta$e$\longrightarrow_{spec}\underline{\mathtt{fix}_n}$ $\underline{\text{f}}$ $\underline{\text{x}}$. $\underline{\text{e}}$. Also we have that $\rho\vdash\phi(\mathtt{fix}_n^{\mathsf{D}}$f x.e)$\longrightarrow_{std}[\![\mathtt{fix}_n$ f x.e$]\!]\rho$. Finally we have that $\rho_d\vdash\varphi(\underline{\mathtt{fix}}$ $\underline{\text{f}}$ $\underline{\text{x}}$. $\underline{\text{e}})\longrightarrow_{std}[\![\mathtt{fix}_n$ $\underline{\text{f}}$ $\underline{\text{x}}.\varphi(\underline{\text{e}})]\!]\rho_d$.

We wish to prove $[\![\mathtt{fix}_n$ f x.e$]\!]\rho\equiv[\![\mathtt{fix}_n$ $\underline{\text{f}}$ $\underline{\text{x}}.\varphi(\underline{\text{e}})]\!]\rho_d$. We do this by induction over n:

<u>$n = 0$</u>: The equation holds per definition.

<u>$n > 0$</u>: Let $v_1, v_2 \in Val_{std}$ be given such that $v_1\equiv v_2$. Further assume that $\rho[$f$\mapsto[\![\mathtt{fix}_{n-1}$ f x.e$]\!]\rho$,x$\mapsto v_1]\vdash$e$\longrightarrow_{std}v_1{}'$ and $\rho_d[\underline{\text{f}}\mapsto[\![\mathtt{fix}_{n-1}$ $\underline{\text{f}}$ $\underline{\text{x}}.\varphi(\underline{\text{e}})]\!]\rho$,$\underline{\text{x}}\mapsto v_2]\vdash\underline{\text{e}}\longrightarrow_{std}v_2{}'$.

By the (local) induction hypothesis, we have $[\![\mathtt{fix}_{n-1}$ f x.e$]\!]\rho\equiv[\![\mathtt{fix}_{n-1}$ $\underline{\text{f}}$ $\underline{\text{x}}.\varphi(\underline{\text{e}})]\!]\rho$. It is now easy to prove, that the environments behave well, so we can use the (global) induction hypothesis and we are through. $\square$(Lemma 6.10)

LEMMA 6.11
Rule (appl):

$\forall$e,e$'$:$\mathcal{H}$(e)$\wedge\mathcal{H}$(e$'$) $\Rightarrow$ $\mathcal{H}$(e@^{b}e$'$)

Proof

Assume A,C and σ such that $A,C\vdash$e@^{b}e$'$:σ. Assume ρ_s, ρ_d, ρ and ζ such that $\zeta(A)$ suits ρ_s, $\zeta\Vdash C$ and ρ_s,ρ_d,ρ agree on *FreeVars*(e@$^{\mathsf{S}}$e$'$).

$\boxed{\zeta b = \mathsf{S}}$ From *FreeVars*(e@$^{\mathsf{S}}$e$'$) = *FreeVars*(e) $\bigcup$ *FreeVars*(e$'$) and the inference rule we have $\zeta A,C\vdash\zeta$e:$(\kappa' \rightarrow \kappa'', \mathsf{S})$ and $\zeta A,C\vdash\zeta$e$'$:κ' (where $\zeta\sigma = \kappa''$). Now by the induction hypothesis:

$$\left.\begin{array}{l}\rho\vdash\phi(\mathrm{e})\longrightarrow_{std}v_1 \wedge\\ \rho_s\vdash\zeta\mathrm{e}\longrightarrow_{spec}{v_1}'\end{array}\right\} \Longrightarrow \mathcal{E}_{\psi(\kappa')\rightarrow\psi(\kappa'')}(v_1, {v_1}', \rho_d)$$

and

$$\left.\begin{array}{l}\rho\vdash\phi(\mathrm{e}')\longrightarrow_{std}v_2 \wedge\\ \rho_s\vdash\zeta\mathrm{e}'\longrightarrow_{spec}{v_2}'\end{array}\right\} \Longrightarrow \mathcal{E}_{\psi(\kappa')}(v_2, {v_2}', \rho_d)$$

By the definition of $\mathcal{E}_{\psi(\kappa')\rightarrow\psi(\kappa'')}$, we can either let ${v_1}' = [\![\lambda \mathrm{x.e}_1]\!]{\rho_s}'$ and $v_1 = [\![\lambda \mathrm{x.e}_2]\!]\rho'$, or let ${v_1}' = [\![\mathtt{fix}_n \text{ f x.e}_1]\!]{\rho_s}'$ and $v_1 = [\![\mathtt{fix}_n \text{ f x.e}_2]\!]\rho'$.

Standard application:
The above gives us

$$\mathcal{E}_{\psi(\kappa')\rightarrow\psi(\kappa'')}([\![\lambda \mathrm{x.e}_2]\!]\rho', [\![\lambda \mathrm{x.e}_1]\!]{\rho_s}', \rho_d)$$

Using the definition of $\mathcal{E}_{\psi(\kappa')\rightarrow\psi(\kappa'')}$ gives us $\rho'[\mathrm{x}\mapsto v_2]\vdash\mathrm{e}\longrightarrow_{std}v$ and ${\rho_s}'[\mathrm{x}\mapsto {v_2}']\vdash\mathrm{e}\longrightarrow_{std}v'$ implies $\mathcal{E}_{\psi(\kappa'')}(v, v', \rho_d)$. Hence

$$\left.\begin{array}{l}\rho\vdash\phi(\mathrm{e}@^{\mathsf{S}}\mathrm{e}')\longrightarrow_{std}v \wedge\\ \rho_s\vdash\zeta(\mathrm{e}@^{\mathsf{S}}\mathrm{e}')\longrightarrow_{spec}v'\end{array}\right\} \Longrightarrow \mathcal{E}_{\psi(\kappa'')}(v, v', \rho_d)$$

Recursive application:
In this case we get

$$\mathcal{E}_{\psi(\kappa')\rightarrow\psi(\kappa'')}([\![\mathtt{fix}_n \text{ f x.e}_2]\!]\rho', [\![\mathtt{fix}_n \text{ f x.e}_1]\!]{\rho_s}', \rho_d).$$

Assume

$$\rho'[\mathrm{f}\mapsto[\![\mathtt{fix}_{n-1} \text{ f x.e}_2]\!]\rho', \mathrm{x}\mapsto v_2]\vdash\mathrm{e}\longrightarrow_{std}v \text{ and}$$
$${\rho_s}'[\mathrm{f}\mapsto[\![\mathtt{fix}_{n-1} \text{ f x.e}_1]\!]{\rho_s}', \mathrm{x}\mapsto {v_2}']\vdash\mathrm{e}\longrightarrow_{std}v'$$

By the definition of $\mathcal{E}_{\psi(\kappa')\rightarrow\psi(\kappa'')}$, this implies $\mathcal{E}_{\psi(\kappa'')}(v, v', \rho_d)$. Hence

$$\left.\begin{array}{l}\rho\vdash\phi(\mathrm{e}@^{\mathsf{S}}\mathrm{e}')\longrightarrow_{std}v \wedge\\ \rho_s\vdash\zeta(\mathrm{e}@^{\mathsf{S}}\mathrm{e}')\longrightarrow_{spec}v'\end{array}\right\} \Longrightarrow \mathcal{E}_{\psi(\kappa'')}(v, v', \rho_d)$$

$\boxed{\zeta b = \mathsf{D}}$ From *FreeVars*(e@$^{\mathsf{D}}$e$'$) = *FreeVars*(e) $\bigcup$ *FreeVars*(e$'$), the induction hypothesis and the binding time rule for application we have

$$\left.\begin{array}{l}\rho\vdash\phi(\mathrm{e})\longrightarrow_{std} v_1 \wedge \\ \rho_s\vdash\zeta\mathrm{e}\longrightarrow_{spec} v_1{}'\end{array}\right\} \Longrightarrow \mathcal{E}_{\mathsf{D}}(v_1, v_1{}', \rho_d)$$

and

$$\left.\begin{array}{l}\rho\vdash\phi(\mathrm{e}')\longrightarrow_{std} v_2 \wedge \\ \rho_s\vdash\zeta\mathrm{e}'\longrightarrow_{spec} v_2{}'\end{array}\right\} \Longrightarrow \mathcal{E}_{\mathsf{D}}(v_2, v_2{}', \rho_d)$$

Standard application:
Let $v_1 = [\![\lambda\mathrm{x.e}_1]\!]\rho_1$. Then by definition of $\mathcal{E}_{\mathsf{D}}$, $\rho_d\vdash v_1{}'\longrightarrow_{std}[\![\lambda\mathrm{x.e}_1{}']\!]\rho_1{}'$ and $[\![\lambda\mathrm{x.e}_1]\!]\rho_1 \equiv [\![\lambda\mathrm{x.e}_1{}']\!]\rho_1{}'$. Similarly $\rho_d\vdash v_2{}'\longrightarrow_{std} v_2{}''$ and $v_2 \equiv v_2{}''$.

The definition of $\equiv$ gives that if $\rho_1[\mathrm{x}\mapsto v_2]\vdash\mathrm{e}_1\longrightarrow_{std} v_3$ then $\rho_1{}'[\mathrm{x}\mapsto v_2{}'']\vdash\mathrm{e}_1{}'\longrightarrow_{std} v_3{}'$ and $v_3 \equiv v_3{}'$.

Recursive application:
Let $v_1 = [\![\mathtt{fix}_n\ \mathrm{f\ x.e}_1]\!]\rho_1$. Then by definition of $\mathcal{E}_{\mathsf{D}}$, $\rho_d\vdash v_1{}'\longrightarrow_{std}[\![\mathtt{fix}_n\ \mathrm{f\ x.e}_1{}']\!]\rho_1{}'$ and $[\![\mathtt{fix}_n\ \mathrm{f\ x.e}_1]\!]\rho_1 \equiv [\![\mathtt{fix}_n\ \mathrm{f\ x.e}_1{}']\!]\rho_1{}'$. Similarly $\rho_d\vdash v_2{}'\longrightarrow_{std} v_2{}''$ and $v_2 \equiv v_2{}''$.

The definition of $\equiv$ gives that, if $\rho_1[\mathrm{f}\mapsto[\![\mathtt{fix}\ \mathrm{f\ x.e}_1]\!]\rho_1, \mathrm{x}\mapsto v_2]\vdash\mathrm{e}_1\longrightarrow_{std} v_3$ then $\rho_1{}'[\mathrm{f}\mapsto[\![\mathtt{fix}\ \mathrm{f\ x.e}_1{}']\!]\rho_1{}', \mathrm{x}\mapsto v_2{}'']\vdash\mathrm{e}_2\longrightarrow_{std} v_3{}'$ and $v_3 \equiv v_3{}'$. This concludes the proof for recursive application. □(Lemma 6.11)

Lemma 6.12
Rule (op):

$$\forall\mathrm{e},\mathrm{e}':\mathcal{H}(\mathrm{e})\wedge\mathcal{H}(\mathrm{e}') \Rightarrow \mathcal{H}(\mathrm{e}\mathsf{op}^b\mathrm{e}')$$

Proof
Assume A,C and σ such that $A,C\vdash\mathrm{e}\mathsf{op}^b\mathrm{e}':\sigma$. Assume ρ_s, ρ_d, ρ and ζ such that $\zeta(A)$ suits ρ_s, $\zeta\Vdash C$ and ρ_s,ρ_d,ρ agree on *FreeVars*(eop^be$'$).

$\boxed{\zeta b = \mathsf{S}}$ We have *FreeVars*(eop^{S}e$'$) = *FreeVars*(e) $\bigcup$ *FreeVars*(e$'$). Thus from the induction hypothesis and the binding time rule for `op` we have

$$\left.\begin{array}{l}\rho\vdash\phi(\mathrm{e})\longrightarrow_{std} v_1 \wedge \\ \rho_s\vdash\zeta\mathrm{e}\longrightarrow_{spec} v_1{}'\end{array}\right\} \Longrightarrow \mathcal{E}_{\mathsf{S}}(v_1, v_1{}', \rho_d)$$

$$\left.\begin{array}{l}\rho\vdash\phi(\mathrm{e}')\longrightarrow_{std} v_2 \wedge \\ \rho_s\vdash\zeta\mathrm{e}'\longrightarrow_{spec} v_2{}'\end{array}\right\} \Longrightarrow \mathcal{E}_{\mathsf{S}}(v_2, v_2{}', \rho_d)$$

Since the argument types of $\mathcal{P}(\mathsf{op})$ is assumed to be base types, we have by definition of $\mathcal{E}$ and $\equiv$ that $v_1 = v_2$ and $v_1{}' = v_2{}'$. We then have $v_1 \otimes^{\mathsf{op}} v_1{}' = v_2 \otimes^{\mathsf{op}} v_2{}'$, so the desired property follows immediately.

$\boxed{\zeta b = \mathsf{D}}$ We have *FreeVars*(eop^{D}e$'$) = *FreeVars*(e) $\bigcup$ *FreeVars*(e$'$). Thus from the induction hypothesis and the binding time rule for `op` we have

$$\left.\begin{array}{l}\rho\vdash\phi(\mathrm{e})\longrightarrow_{std}v_1\ \wedge\\ \rho_s\vdash\zeta\mathrm{e}\longrightarrow_{spec}v_1'\end{array}\right\}\Longrightarrow\mathcal{E}_{\mathsf{D}}(v_1,v_1',\rho_d)$$

$$\left.\begin{array}{l}\rho\vdash\phi(\mathrm{e}')\longrightarrow_{std}v_2\ \wedge\\ \rho_s\vdash\zeta\mathrm{e}'\longrightarrow_{spec}v_2'\end{array}\right\}\Longrightarrow\mathcal{E}_{\mathsf{D}}(v_2,v_2',\rho_d)$$

By the definition of $\mathcal{E}_{\mathsf{D}}$ we have $\rho_d\vdash\varphi(v_1')\longrightarrow_{std}v_1''\wedge v_1\equiv v_1''$ and $\rho_d\vdash\varphi(v_2')\longrightarrow_{std}v_2''\wedge v_2\equiv v_2''$. By the definition of $\equiv$ this implies $v_1=v_1''$ and $v_2=v_2''$.

By the specialization rule, we have $\rho_s\vdash\mathrm{e}\mathtt{op}^{\mathsf{D}}\mathrm{e}'\longrightarrow_{spec}v_1'\underline{\mathtt{op}}v_2'$. If $v_1\otimes^{\mathtt{op}}v_2=v_3$ then $\rho\vdash\phi(\mathrm{e}\mathtt{op}^{\mathsf{D}}\mathrm{e}')\longrightarrow_{std}v_3$. From this and the above, we can deduce $\rho_d\vdash\varphi(v_1'\underline{\mathtt{op}}v_2')\longrightarrow_{std}v_3$. □(Lemma 6.12)

Lemma 6.13
Rule (pair):

$$\forall\mathrm{e},\mathrm{e}':\mathcal{H}(\mathrm{e})\wedge\mathcal{H}(\mathrm{e}')\Rightarrow\mathcal{H}(\mathtt{pair}^b(\mathrm{e},\mathrm{e}'))$$

Proof

Assume A,C and σ such that $A,C\vdash\mathtt{pair}^b(\mathrm{e},\mathrm{e}'):\sigma$. Assume ρ_s, ρ_d, ρ and ζ such that $\zeta(A)$ suits ρ_s, $\zeta\Vdash C$ and ρ_s,ρ_d,ρ agree on *FreeVars*($\mathtt{pair}^b(\mathrm{e},\mathrm{e}')$).

$\boxed{\zeta b=\mathsf{S}}$ By the inference rule we have $\zeta A,C\vdash\mathtt{pair}^{\mathsf{S}}(\zeta\mathrm{e},\zeta\mathrm{e}'):(\kappa'\times\kappa'',\mathsf{S})$. From *FreeVars*($\mathtt{pair}^{\mathsf{S}}(\mathrm{e},\mathrm{e}')$) = *FreeVars*(e) $\bigcup$ *FreeVars*(e′), the induction hypothesis and the binding time rule for `pair` we have

$$\rho_s\vdash\zeta\mathrm{e}\longrightarrow_{spec}v_1'\wedge\rho\vdash\phi(\mathrm{e})\longrightarrow_{std}v_1\Longrightarrow\mathcal{E}_{\psi(\kappa')}(v_1,v_1',\rho_d)$$
$$\rho_s\vdash\zeta\mathrm{e}'\longrightarrow_{spec}v_2'\wedge\rho\vdash\phi(\mathrm{e}')\longrightarrow_{std}v_2\Longrightarrow\mathcal{E}_{\psi(\kappa'')}(v_2,v_2',\rho_d)$$

We now have

$$\left.\begin{array}{l}\rho_s\vdash\mathtt{pair}^{\mathsf{S}}(\zeta\mathrm{e},\zeta\mathrm{e}')\longrightarrow_{spec}(v_1',v_2')\\ \wedge\rho\vdash\phi(\mathtt{pair}^{\mathsf{S}}(\mathrm{e},\mathrm{e}'))\longrightarrow_{std}(v_1,v_2)\end{array}\right\}\Longrightarrow\mathcal{E}_{\psi(\kappa')\times\psi(\kappa'')}((v_1,v_2),(v_1',v_2'),\rho_d)$$

$\boxed{\zeta b=\mathsf{D}}$ By the inference rule we have $\zeta A,C\vdash\mathtt{pair}^{\mathsf{D}}(\zeta\mathrm{e},\zeta\mathrm{e}'):((t',\mathsf{D})\times(t'',\mathsf{D}),\mathsf{D})$. We have *FreeVars*($\mathtt{pair}^{\mathsf{S}}(\mathrm{e},\mathrm{e}')$) = *FreeVars*(e) $\bigcup$ *FreeVars*(e′). Thus by the induction hypothesis and the binding time rule for `pair`:

$$\rho_s\vdash\zeta\mathrm{e}\longrightarrow_{spec}v_1'\wedge\rho\vdash\phi(\mathrm{e})\longrightarrow_{std}v_1\Longrightarrow\mathcal{E}_{\mathsf{D}}(v_1,v_1',\rho_d)$$
$$\rho_s\vdash\zeta\mathrm{e}'\longrightarrow_{spec}v_2''\wedge\rho\vdash\phi(\mathrm{e}')\longrightarrow_{std}v_2\Longrightarrow\mathcal{E}_{\mathsf{D}}(v_2,v_2',\rho_d)$$

This means by the definition of $\mathcal{E}_{\mathsf{D}}$ that $\rho_d\vdash\varphi(v_1')\longrightarrow_{std}v_1''\ \wedge\ v_1\equiv v_1''$ and $\rho_d\vdash\varphi(v_2')\longrightarrow_{std}v_2''\wedge v_2\equiv v_2''$. We have

$$\rho_s\vdash\mathtt{pair}^{\mathsf{D}}(\zeta\mathrm{e},\zeta\mathrm{e}')\longrightarrow_{spec}\underline{\mathtt{pair}}(v_1',v_2')$$
$$\rho_d\vdash\varphi(\underline{\mathtt{pair}}(v_1',v_2'))\longrightarrow_{std}(v_1'',v_2'')$$
$$\rho\vdash\phi(\mathtt{pair}^{\mathsf{D}}(\mathrm{e},\mathrm{e}'))\longrightarrow_{std}(v_1,v_2)$$

Since $(v_1, v_2) \equiv (v_1'', v_2'')$, we are done. $\Box$(Lemma 6.13)

Lemma 6.14
Rule (first):

$$\forall e{:}\mathcal{H}(e) \Rightarrow \mathcal{H}(\texttt{fst}^b\ e)$$

Proof
Assume A,C and σ such that $A,C\vdash\texttt{fst}^b e{:}\sigma$. Assume ρ_s, ρ_d, ρ and ζ such that $\zeta(A)$ suits ρ_s, $\zeta\Vdash C$ and ρ_s,ρ_d,ρ agree on *FreeVars*($\texttt{fst}^b$e).

$\boxed{\zeta b = \mathsf{S}}$ The inference rule gives $\zeta A,C\vdash\zeta e{:}(\kappa' \times \kappa'', \mathsf{S})$. Since *FreeVars*($\texttt{fst}^{\mathsf{S}}$ e) $=$ *FreeVars*(e), it follows from the induction hypothesis and the binding time rule for `fst`, that

$$\rho_s\vdash\zeta e\longrightarrow_{spec}\mathit{v'} \wedge \rho\vdash\phi(e)\longrightarrow_{std}v \Longrightarrow \mathcal{E}_{\psi(\kappa')\times\psi(\kappa'')}(v, \mathit{v'}, \rho_d)$$

By definition of $\mathcal{E}_{\psi(\kappa')\times\psi(\kappa'')}$, we can let $(v_1, v_2) = v$ and $(v_1', v_2') = v_{res}$. Then $\mathcal{E}_{\psi(\kappa')}(v_1, v_1', \rho_d)$ follows from the definition of $\mathcal{E}_{\psi(\kappa')\times\psi(\kappa'')}$.

$\boxed{\zeta b = \mathsf{D}}$ The inference rule gives $\zeta A,C\vdash\zeta e{:}((t', \mathsf{D}) \times (t'', \mathsf{D}), \mathsf{D})$. Since *FreeVars*($\texttt{fst}^{\mathsf{D}}$ e) $=$ *FreeVars*(e), it follows from the induction hypothesis and the binding time rule for `fst`, that

$$\rho_s\vdash\zeta(e)\longrightarrow_{spec}\mathit{v'} \wedge \rho\vdash\phi(e)\longrightarrow_{std}v \Longrightarrow \mathcal{E}_{\mathsf{D}}(v, \mathit{v'}, \rho_d)$$

This implies $\rho_d\vdash\varphi(\mathit{v'})\longrightarrow_{std}\mathit{v''}$ and $v\equiv \mathit{v''}$. Let $(v_1, v_2) = v$, then, by definition of $\equiv$, we can let $(v_1'', v_2'') = \mathit{v''}$. It follows that $v_1 \equiv v_1''$.

Assume $\rho_s\vdash\texttt{fst}^{\mathsf{D}}\ \zeta e\longrightarrow_{spec}\underline{\texttt{fst}}\ \mathit{v'}$ and $\rho\vdash\phi(\texttt{fst}^{\mathsf{D}}\ e)\longrightarrow_{std}v_1$. Then by the above, we can deduce $\rho_d\vdash\varphi(\underline{\texttt{fst}}\ \mathit{v'})\longrightarrow_{std}v_1''$ and $v_1 \equiv v_1''$. But this is the definition of $\mathcal{E}_{\mathsf{D}}(v_1,\underline{\texttt{fst}}\mathit{v'}, \rho_d)$, so we are done. $\Box$(Lemma 6.14)

Lemma 6.15
Rule (second):

$$\forall e{:}\mathcal{H}(e) \Rightarrow \mathcal{H}(\texttt{snd}^b\ e)$$

Proof
Similar to the proof of Lemma 6.14. $\Box$(Lemma 6.15)

Lemma 6.16
Rule ($\forall$-introduction):

$$\forall e{:}\mathcal{H}(e) \Rightarrow \mathcal{H}(\Lambda\beta.e)$$

Proof

Assume A,C and σ such that $A,C\vdash\Lambda\beta.\mathrm{e}{:}\sigma$. Assume ρ_s, ρ_d, ρ and ζ such that $\zeta(A)$ suits ρ_s, $\zeta\Vdash C$ and ρ_s,ρ_d,ρ agree on *FreeVars*($\Lambda\beta$.e).

We have $\rho_s\vdash\zeta(\Lambda\beta.\mathrm{e})\longrightarrow_{spec}[\![\zeta(\Lambda\beta.\mathrm{e})]\!]\rho_s$ and that $\rho\vdash\phi(\Lambda\beta.\mathrm{e})\longrightarrow_{std}v$ whenever $\rho\vdash\phi(\mathrm{e})\longrightarrow_{std}v$. Further the inference rule gives us that $\zeta\sigma=\forall\beta.\sigma'$ for some σ'. We wish to prove $\mathcal{E}_{\forall\beta.\psi(\zeta\sigma')}(v,[\![\zeta(\Lambda\beta.\mathrm{e})]\!]\rho_s,\rho_d)$.

Let $b\in\{\mathsf{S},\mathsf{D}\}$ be given, and let ζ' be $\zeta[b/\beta]$. Then by definition of $\mathcal{E}$, what we want to prove is

$$\rho_s\vdash\zeta'\mathrm{e}\longrightarrow_{spec}\acute{v}\implies\mathcal{E}_{\psi(\zeta'\sigma')}(v,\acute{v},\rho_d)$$

From *FreeVars*($\Lambda\beta$.e) $=$ *FreeVars*(e), the induction hypothesis and the typerule, we clearly have that $\zeta'(A)$ suits ρ_s and $\zeta'\Vdash C$. From this we see that the above follows directly from the induction hypothesis. □(Lemma 6.16)

Lemma 6.17
Rule ($\forall$-elimination):

$$\forall\mathrm{e}{:}\mathcal{H}(\mathrm{e})\Rightarrow\mathcal{H}(\mathrm{e}\diamond b)$$

Proof

Assume A,C and σ such that $A,C\vdash\mathrm{e}\diamond b{:}\sigma$. Assume ρ_s, ρ_d, ρ and ζ such that $\zeta(A)$ suits ρ_s, $\zeta\Vdash C$ and ρ_s,ρ_d,ρ agree on *FreeVars*(e$\diamond b$).

By the inference rule $\zeta A,C\vdash\zeta\mathrm{e}{:}\forall\beta.\sigma$. We have that $\rho\vdash\phi(\mathrm{e}\diamond b)\longrightarrow_{std}v$ whenever $\rho\vdash\phi(\mathrm{e})\longrightarrow_{std}v$. Assume $\rho_s\vdash\zeta\mathrm{e}\longrightarrow_{spec}\acute{v}$.

The induction hypothesis and the fact that *FreeVars*(e$\diamond b$) $=$ *FreeVars*(e), gives us that $\mathcal{E}_{\forall\beta.\sigma}(v,\acute{v},\rho_d)$ which by definition means:

$$\acute{v}=[\![\Lambda\beta.\mathrm{e}']\!]\rho_s\in BtFunval_{spec}\wedge\ \forall b\in\{\mathsf{S},\mathsf{D}\}{:}\ \rho_s\vdash[b/\beta]\mathrm{e}'\longrightarrow_{spec}\acute{v}'\implies\mathcal{E}_{\sigma}(v,\acute{v}',\rho_d)$$

Since $\rho_s\vdash\zeta(\mathrm{e}\diamond b)\longrightarrow_{spec}\acute{v}''$ when $\rho_s\vdash[b/\beta]\mathrm{e}'\longrightarrow_{spec}\acute{v}''$ by the specialization rule, we are through. □(Lemma 6.17)

Lemma 6.18
Rule ($\Rightarrow$-introduction):

$$\forall\mathrm{e}{:}\mathcal{H}(\mathrm{e})\Rightarrow\mathcal{H}(\Lambda b\rightsquigarrow b'.\mathrm{e})$$

Proof

Assume A,C and σ such that $A,C\vdash\Lambda b\rightsquigarrow b'.\mathrm{e}{:}\sigma$. Assume ρ_s, ρ_d, ρ and ζ such that $\zeta(A)$ suits ρ_s, $\zeta\Vdash C$ and ρ_s,ρ_d,ρ agree on *FreeVars*($\Lambda b\rightsquigarrow b'$.e).

By the inference rule $\zeta\sigma=b{\leq}b'{\Rightarrow}\sigma'$ for some σ'. We have that $\rho\vdash\phi(\Lambda b\rightsquigarrow b'.\mathrm{e})\longrightarrow_{std}v$ whenever $\rho\vdash\phi(\mathrm{e})\longrightarrow_{std}v$. By the specialization rule, $\rho_s\vdash\Lambda b\rightsquigarrow b'.\zeta\mathrm{e}\longrightarrow_{spec}[\![\Lambda b\rightsquigarrow b'.\zeta\mathrm{e}]\!]\rho_s$. What we want to prove is $\mathcal{E}_{b\leq b'\Rightarrow\sigma'}(v,[\![\Lambda b\rightsquigarrow b'.\zeta\mathrm{e}]\!]\rho_s,\rho_d)$, which amount to showing

$$\rho_s \vdash \zeta \mathrm{e} \longrightarrow_{spec} v' \implies \mathcal{E}_{\zeta\sigma'}(v, v', \rho_d)$$

Since *FreeVars*($\Lambda b \rightsquigarrow b'$.e) = *FreeVars*(e), this follows from the induction hypothesis. □(Lemma 6.18)

Lemma 6.19
Rule (⇒-elimination):

$$\forall \mathrm{e}{:}\mathcal{H}(\mathrm{e}) \Rightarrow \mathcal{H}(\mathrm{e}\Box b \rightsquigarrow b')$$

Proof
Assume A,C and σ such that $A,C \vdash \mathrm{e}\Box b \rightsquigarrow b'{:}\sigma$. Assume ρ_s, ρ_d, ρ and ζ such that $\zeta(A)$ suits ρ_s, $\zeta \Vdash C$ and ρ_s,ρ_d,ρ agree on *FreeVars*($\mathrm{e}\Box b \rightsquigarrow b'$).

Assume $\rho_s \vdash \zeta \mathrm{e} \longrightarrow_{spec} v'$ and $\rho \vdash \phi(\mathrm{e}) \longrightarrow_{std} v$. We have *FreeVars*($\mathrm{e}\Box b \rightsquigarrow b'$) = *FreeVars*(e). The typerule gives us that $\zeta A, C \vdash \zeta \mathrm{e} : b \leq b' \Rightarrow \sigma'$. We can use the induction hypothesis implying $\mathcal{E}_{b \leq b' \Rightarrow \sigma'}(v, v', \rho_d)$, which per definition means

$$v' = [\![\Lambda b \rightsquigarrow b'.\mathrm{e}']\!]\rho_s{}' \wedge \rho_s{}' \vdash \zeta \mathrm{e}' \longrightarrow_{spec} v'' \implies \mathcal{E}_{\sigma'}(v, v'', \rho_d)$$

But since $\rho_s \vdash \zeta(\mathrm{e}\Box b \rightsquigarrow b') \longrightarrow_{spec} v''$ by the specialization rule, we are through. □(Lemma 6.19)

Lemma 6.20
Rule (let):

$$\forall \mathrm{e},\mathrm{e}'{:}\mathcal{H}(\mathrm{e}) \wedge \mathcal{H}(\mathrm{e}') \Rightarrow \mathcal{H}(\texttt{let}\ \mathrm{x} = \mathrm{e}'\ \texttt{in}\ \mathrm{e})$$

Proof
Assume A,C and σ such that $A,C \vdash \texttt{let}\ \mathrm{x} = \mathrm{e}'\ \texttt{in}\ \mathrm{e}{:}\sigma$. Assume ρ_s, ρ_d, ρ and ζ such that $\zeta(A)$ suits ρ_s, $\zeta \Vdash C$ and ρ_s,ρ_d,ρ agree on *FreeVars*($\texttt{let}\ \mathrm{x} = \mathrm{e}'\ \texttt{in}\ \mathrm{e}$). By the inference rule $\zeta\sigma = \kappa$ for some compound type κ.

We have *FreeVars*($\texttt{let}\ \mathrm{x} = \mathrm{e}'\ \texttt{in}\ \mathrm{e}$)⊇*FreeVars*(e′); thus by the inference rule there exist a σ' such that $\zeta A, C \vdash \zeta \mathrm{e}{:}\sigma'$. Using the induction hypothesis, we can deduce

$$\rho \vdash \phi(e') \longrightarrow_{std} v_1 \wedge \rho_s \vdash \zeta e' \longrightarrow_{spec} v_1{}' \implies \mathcal{E}_{\psi(\sigma')}(v_1, v_1{}', \rho_d)$$

Define

$$\rho_s{}' = \rho_s[\mathrm{x} \mapsto v_1{}']$$
$$\rho' = \rho[\mathrm{x} \mapsto v_1]$$
$$A' = \zeta A \bigcup \{\mathrm{x} : \sigma'\}$$

We have $\forall id \in$*FreeVars*(e):$\mathcal{E}_{\psi(A'(id))}(\rho'(id), \rho_s{}'(id), \rho_d)$. This means that $\rho_s{}'$, ρ_d, ρ' agree on *FreeVars*(e), since A' suits $\rho_s{}'$ by proposition 6.3.

From the induction hypothesis we now have

$$\rho'\vdash\phi(e)\longrightarrow_{std}v_2 \wedge \rho_s'\vdash\zeta e\longrightarrow_{spec}v_2' \implies \mathcal{E}_{\psi(\kappa)}(v_2, v_2', \rho_d)$$

By the evaluation rules for `let` we get

$$\rho\vdash\phi(\texttt{let}\ \mathrm{x} = \mathrm{e}'\,\texttt{in}\ \mathrm{e})\longrightarrow_{std}v_2$$
$$\rho_s\vdash\zeta(\texttt{let}\ \mathrm{x} = \mathrm{e}'\,\texttt{in}\ \mathrm{e})\longrightarrow_{spec}v_2'$$

So we have reached the desired conclusion. □(Lemma 6.20)

Lemma 6.21
Rule (coerce):

$$\forall \mathrm{e}{:}\mathcal{H}(\mathrm{e}) \Rightarrow \mathcal{H}([b\rightsquigarrow b']\mathrm{e})$$

Proof
Assume A,C and σ such that $A,C\vdash[b\rightsquigarrow b']\mathrm{e}{:}\sigma$. Assume ρ_s, ρ_d, ρ and ζ such that $\zeta(A)$ suits ρ_s, $\zeta\Vdash C$ and ρ_s,ρ_d,ρ agree on *FreeVars*($[b\rightsquigarrow b']$e).

Since $\zeta\Vdash C$, we must have $\zeta b\overset{!}{\leq}\zeta b'$ implying either $\zeta b = \zeta b'$ or $\zeta b = \mathsf{S}$ and $\zeta b' = \mathsf{D}$. By the inference rules $\sigma = (t, b)$ where $t \in \{\mathrm{Int}, \mathrm{Bool}\}$, so $\psi(\zeta\sigma) = \zeta b$.

From *FreeVars*($[b\rightsquigarrow b']$e) = *FreeVars*(e), the induction hypothesis and the binding time rule, we have

$$\rho\vdash\phi(e)\longrightarrow_{std}v \wedge \rho_s\vdash\zeta e\longrightarrow_{spec}v' \implies \mathcal{E}_{\zeta b}(v, v', \rho_d)$$

Let us look at the two cases:
$\boxed{\zeta b = \zeta b'}$ Since we have $\rho\vdash\phi([b\rightsquigarrow b']\mathrm{e})\longrightarrow_{std}v$ and $\rho_s\vdash\zeta([b\rightsquigarrow b']\mathrm{e})\longrightarrow_{spec}v'$, the conclusion follows immediately.
$\boxed{\zeta[b\rightsquigarrow b'] = [\mathsf{S}\rightsquigarrow\mathsf{D}]}$ By definition of $\mathcal{E}_{\mathsf{S}}(v, v', \rho_d)$, we get $v\equiv v'$ implying $v = v'$. But then $\rho_s\vdash[\mathsf{S}\rightsquigarrow\mathsf{D}]\mathrm{e}\longrightarrow_{spec}\underline{v}$. Since $\varphi(\underline{v}) = v$, we have $\rho_d\vdash\varphi(\underline{v})\longrightarrow_{std}v$ and thus we can conclude $\mathcal{E}_{\mathsf{D}}(v, \underline{v}, \rho_d)$ □(Lemma 6.21)

This concludes the proof of soundness of our binding time analysis.

6.5 Similar Proofs

Our correctness proof was inspired by proofs by Gomard in his Ph.D.thesis [Gomard 1991b] (based on [Gomard 1991a]) and by Wand in [Wand 1993].

Wand proves the correctness of a partial evaluator for the pure λ-calculus (originating from [Mogensen 1992]), and is in many ways a simplification and clarification of [Gomard 1991a]. Gomards language is also the λ-calculus, but extended with `if`, `fix` and constants.

Both Gomard and Wand define correctness in terms of (the semantics of) a specializer. Wand specifies his semantics directly as a self interpreter and a specializer written in the pure λ-calculus itself. Gomard specifies his semantics using denotational semantics.

Our choice of specifying the language and the specializer using operational semantics gives some technical and notational differences between our proof and the proofs by Gomard and Wand, but the basic structure (induction over the binding time derivation) is similar, though our bigger language and binding time polymorphism of course adds to the size of the proof and the complexity of the preliminary definitions.

While Gomard, Wand and the present work prove correctness of binding time analysis *w.r.t.* a specific specializer, Palsberg and Schwartzbach proves correctness for a class of specializers [Palsberg & Schwartzbach 1992, Palsberg 1993].

Other approaches to proving correctness of binding time analysis is possible. The concept of *congruence* introduced by Jones in [Jones 1988] serves as a basis for proving correctness independently of a specific specializer. This work has been carried on by Launchbury in his thesis [Launchbury 1990] — the projection based binding time analysis principle seems to suit congruence in a very natural way. Our intuition however is that the concept of congruence implicitly states a number of requirements, the specializer should meet. Thus congruence is no less a definition of a specializer than a denotational or operational semantics is.

Yet another approach to correctness would be to view binding time analysis as an abstraction of a *on-line specializer*. If an analysis is the abstraction of a semantic definition, we are on safe grounds, and can use well known techniques from abstract interpretation.

Chapter 7

Other Standard Type Systems

In this chapter we will show that our polyvariant binding time analysis is not restricted to monomorphically typed languages. In section 7.1, we show that the extension to a polymorphic standard type discipline is very straightforward, and in section 7.2 we present extensions to a dynamically typed language. In a dynamically typed language, some binding time information will be lost injecting and projecting to/from a common universal type and we discuss different choices to this. We also discuss how this binding time information loss might be avoided.

The main purpose of this chapter is to show the generality of the ideas presented in the previous chapters. We will thus neither give algorithms nor implement any of the systems presented in this chapter, and when we in part II turn to algorithms, it will be based on the system as presented in chapter 4.

7.1 Polymorphically Typed Languages

We need standard type quantification. We therefore introduce standard types t, which can be type variables τ, base types (integers or booleans), products of, or functional types between our compound type κ. Quantification can now be introduced in the standard ML manner.

$$\begin{aligned}
&\sigma ::= \kappa \mid \forall\beta.\sigma \mid b \leq b' \Rightarrow \sigma \mid \forall\tau.\sigma \\
&\kappa ::= (t,b) \\
&b ::= \mathsf{S} \mid \mathsf{D} \mid \beta \\
&t ::= \mathrm{Int} \mid \mathrm{Bool} \mid \kappa \rightarrow \kappa \mid \kappa \times \kappa \mid \tau
\end{aligned}$$

The syntax need not be changed since standard type abstractions will be implicit (á la Curry). The type system is extended by the rules shown in figure 7.1. Note, however, that the new standard type quantification is not solely over “pure” standard types — binding time values/variables can also be involved.

7.2 BTA for Dynamically Typed Languages

We wish to extend the polyvariant binding time analysis from chapter 4 to deal with dynamically typed languages. This will be of great interest if our binding time analysis is

$$\text{(Gen)}\quad \frac{A,C\vdash e{:}\sigma}{A,C\vdash e{:}\forall\tau.\sigma}(\text{if } \tau \text{ not free in } A)$$

$$\text{(Inst)}\quad \frac{A,C\vdash e{:}\forall\tau.\sigma}{A,C\vdash e{:}\sigma[t/\tau]}$$

Figure 7.1: Parametric Type Polymorphism

to be used with languages like Scheme (which is untyped, but onto which a dynamic type system can be imposed). Dynamic typing is described in *e.g.* [Henglein 1992, Gomard 1990].

In dynamic typing we have one universal type which every type can be coerced to and which coerces to every type. We will denote this type $\mathcal{U}$ (in [Henglein 1992] it is called *dyn* but we did not want it mixed up with our binding time value *dynamic* (D)). We then have the following types and typeschemes:

$$b ::= \mathsf{S} \mid \mathsf{D} \mid \beta$$
$$\kappa ::= (\kappa \to \kappa,b) \mid (\kappa \times \kappa,b) \mid (\text{Int},b) \mid (\text{Bool},b) \mid (\mathcal{U},b)$$
$$\sigma ::= \kappa \mid \forall\beta.\sigma \mid b{\le}b'{\Rightarrow}\sigma$$

The extensions to the type system are merely a number of constant coercion rules. The coercions are applied to compound types, so we have to rewrite our coercion rules slightly. This is done in figure 7.2. The symbol $\le$ might seem a bit awkward in this context coercing to/from the universal type.

7.2.1 Induced Coercions

We have so far not allowed induced coercions (on functions and product) and the dynamic type system presented here will work without them. The lack of induced coercions will however lead to a very "conservative" use of the universal type; intuitively, if the type of a pair or function at some point has to be coerced to $\mathcal{U}$, all values getting "caught" in the pair/function will have to have type $\mathcal{U}$ from "birth" (or rather, the type will have to be coerced to $\mathcal{U}$ before it is "caught"). It should thus be clear, that if this was to be used in practice, induced coercions should be allowed — introduction of induced coercions will be discussed further in subsection 13.1.1.

7.2.2 When to Lose Binding Time Information

Since we can coerce from a functional or product type to $\mathcal{U}$ and back again, we will inevitably lose binding time information. We have different choices as to how this can be done, and we will describe them in this and the following subsection and discuss the operational significance for the specializer.

The choice of how to lose binding time information, really comes down to a choice of interpretation of the type ($\mathcal{U}$,S). Coercing to/from this type from/to base types leave

$$\text{(coerce)}\quad \frac{A,C\vdash \mathrm{e}:\kappa \quad C\vdash \kappa \leq \kappa'}{A,C\vdash [\kappa \rightsquigarrow \kappa']\mathrm{e}:\kappa'}$$

$$\text{(lookup-coerce)}\quad C\bigcup\{\kappa \leq \kappa'\}\vdash \kappa \leq \kappa'$$

$$\text{(lift)}\quad C\vdash(\mathrm{Bool},\mathsf{S}) \leq (\mathrm{Bool},\mathsf{D})$$

$$\text{(lift)}\quad C\vdash(\mathrm{Int},\mathsf{S}) \leq (\mathrm{Int},\mathsf{D})$$

$$\text{(id-stat)}\quad C\vdash(\mathrm{Bool},\mathsf{S}) \leq (\mathrm{Bool},\mathsf{S})$$

$$\text{(id-stat)}\quad C\vdash(\mathrm{Int},\mathsf{S}) \leq (\mathrm{Int},\mathsf{S})$$

$$\text{(id-dyn)}\quad C\vdash(\mathrm{Bool},\mathsf{D}) \leq (\mathrm{Bool},\mathsf{D})$$

$$\text{(id-dyn)}\quad C\vdash(\mathrm{Int},\mathsf{D}) \leq (\mathrm{Int},\mathsf{D})$$

Figure 7.2: Rewritten Coercion Rules

no interesting alternatives. If an expression of type ($\mathcal{U}$,S) is used in a context where an integer (*resp.* boolean) is expected, we coerce the type to (Int,S) (*resp.* (Bool,S)).

If an expression of type ($\mathcal{U}$,S) is used in a context where a product is expected, we have a choice as to how to interpret this type: we can interpret it as being a fully static structure, or we can interpret it as being a partially static structure — that is a static pair of (potentially) dynamic components. It should be clear that whichever we choose, the process of coercing a product type to the universal type and back will potentially cause a loss of binding time information. With the first choice, the process will make partially static structures completely dynamic, the second choice will make completely static structures partially static. Since the second choice only gives us the additional power of retaining staticness of head pair cells, we are lead to believe, that the first choice is to be preferred. Other choices are possible, *e.g.* deciding that ($\mathcal{U}$,S) means static head and static second component; if products are used to form lists, this will will mean "spine-staticness".

With functional types the choice is: should ($\mathcal{U}$,S) mean completely static — that is $(\mathsf{S} \to \mathsf{S}, \mathsf{S})$, or should it mean applicable — that is including types such as $(\mathsf{D} \to \mathsf{D}, \mathsf{S})$. Again we have a choice of which information should be lost: applicability of functions of dynamic argument and result or the use of the result of applications. The choice might be less obvious than for product types, but we are lead to believe that the choice of interpreting ($\mathcal{U}$,S) as completely static is superior again. Other alternatives are of course possible (such as any function taking static arguments).

In figure 7.3, we present a number of coercion rules implementing dynamic typing with the choice that ($\mathcal{U}$,S) means everything static.

$$\text{(bool-embed)} \quad C \vdash (\text{Bool}, b) \leq (\mathcal{U}, b)$$

$$\text{(bool-proj)} \quad C \vdash (\mathcal{U}, b) \leq (\text{Bool}, b)$$

$$\text{(int-embed)} \quad C \vdash (\text{Int}, b) \leq (\mathcal{U}, b)$$

$$\text{(int-proj)} \quad C \vdash (\mathcal{U}, b) \leq (\text{Int}, b)$$

$$(\times\text{-embed}) \quad C \vdash ((\mathcal{U}, b') \times (\mathcal{U}, b''), b) \leq (\mathcal{U}, \mathsf{D}) \qquad \textit{if } b \neq \mathsf{S} \vee b' \neq \mathsf{S} \vee b'' \neq \mathsf{S}$$

$$(\times\text{-proj}) \quad C \vdash (\mathcal{U}, \mathsf{D}) \leq ((\mathcal{U}, \mathsf{D}) \times (\mathcal{U}, \mathsf{D}), \mathsf{D})$$

$$(\times\text{-embed}) \quad C \vdash ((\mathcal{U}, \mathsf{S}) \times (\mathcal{U}, \mathsf{S}), \mathsf{S}) \leq (\mathcal{U}, \mathsf{S})$$

$$(\times\text{-proj}) \quad C \vdash (\mathcal{U}, \mathsf{S}) \leq ((\mathcal{U}, \mathsf{S}) \times (\mathcal{U}, \mathsf{S}), \mathsf{S})$$

$$(\rightarrow\text{-embed}) \quad C \vdash ((\mathcal{U}, b') \rightarrow (\mathcal{U}, b''), b) \leq (\mathcal{U}, \mathsf{D}) \qquad \textit{if } b \neq \mathsf{S} \vee b' \neq \mathsf{S} \vee b'' \neq \mathsf{S}$$

$$(\rightarrow\text{-proj}) \quad C \vdash (\mathcal{U}, \mathsf{D}) \leq ((\mathcal{U}, \mathsf{D}) \rightarrow (\mathcal{U}, \mathsf{D}), \mathsf{D})$$

$$(\rightarrow\text{-embed}) \quad C \vdash ((\mathcal{U}, \mathsf{S}) \rightarrow (\mathcal{U}, \mathsf{S}), \mathsf{S}) \leq (\mathcal{U}, \mathsf{S})$$

$$(\rightarrow\text{-proj}) \quad C \vdash (\mathcal{U}, \mathsf{S}) \leq ((\mathcal{U}, \mathsf{S}) \rightarrow (\mathcal{U}, \mathsf{S}), \mathsf{S})$$

Figure 7.3: Dynamic Type System

7.2.3 Specialization must never type-err

In the above discussion, we have assumed that a dynamically typed language would impose dynamic typing during specialization. In other words, the systems presented in subsection 7.2.2, introduces the possibility of type errors at specialization time. This might not be desirable, since the "super-eagerness" of specialization can lead to run-time type-errors that would never occur during standard evaluation. *E.g.* the expression

```
if (integer? x) and b then x+1 else if x then ...
```

where x is static and b is dynamic will always lead to a specialization time type-error, but never (if everything else is well-typed) to a standard evaluation type-error.

Specialization time type errors can be avoided by making everything with a potential type error dynamic. If a type is coerced to $\mathcal{U}$, it is made dynamic (and this is of course preserved when projecting back). Thus no expression will have the type ($\mathcal{U}$,S); if an expression has universal type, it will be typed by ($\mathcal{U}$,D). This clarifies the well known correspondence between *partial type inference* (basically just another word for dynamic typing) and binding time analysis *e.g.* seen in the work by Gomard [Gomard 1990].

We will just show this for integers — all other cases are similar:

$$\text{(int-embed)} \quad C \vdash (\text{Int}, b) \rightsquigarrow (\mathcal{U}, \mathsf{D})$$

$$\text{(int-proj)} \quad C \vdash (\mathcal{U}, \mathsf{D}) \rightsquigarrow (\text{Int}, \mathsf{D})$$

The Partial Evaluator SIMILIX

SIMILIX [Bondorf 1991, Bondorf & Danvy 1991] raises (makes dynamic) some expressions to avoid specialization time type errors, while others are left (with potential risk of type errors). Expressions where first order and higher order types get mixed up, such as `5@5`, are raised in order to avoid type errors. Purely first order expression like `true + 5` are not. Such a type system can easily be expressed in our formalism.

7.2.4 Avoiding Binding Time Information Loss

The loss of binding time information seems to be closely linked with dynamic typing. It might be possible to avoid this by replacing dynamic typing by a sort of *union*-typing.

Dynamic typing works intuitively this way: whenever two non-unifiable types κ_1 and κ_2 are attempted unified, they are first coerced into type $\mathcal{U}$, and unification can succeed. Instead of coercing them to $\mathcal{U}$, one might consider coercing them to $\kappa_1 \bigcup \kappa_2$. This way no binding time information would be lost, since when projecting to either κ_1 or κ_2, the right information would follow.

It would be interesting to explore the possiblities and problems of such a system.

Part II

Algorithms

Chapter 8

Polymorphic BTA Revisited

The type system as presented in chapter 4 contains two kinds of second order types: $\forall\beta.\sigma$ and $b \rightsquigarrow b \Rightarrow \sigma$. This formulation was intentional in order to give simple inference rules thus making the correctness proof simpler.

A problem with the type rules is that they are not operationally *deterministic* (remember that insertion of second order abstraction and application has to be inferred by the binding time analysis). Rules ($\forall$–introduction) and ($\forall$–elimination) *resp.* ($\Rightarrow$–introduction) and ($\Rightarrow$–elimination) may be invoked at any time, so a strategy for constructing inference trees is not obvious. *Eg.* we have

$$(\forall\text{–introduction})\quad \dfrac{(\text{int})\quad A{,}C\vdash 5{:}(\text{Int},\mathsf{S})}{(\forall\text{–elimination})\quad \dfrac{A{,}C\vdash \Lambda\beta.5{:}\forall\beta.(\text{Int},\mathsf{S})}{A{,}C\vdash (\Lambda\beta.5)\diamond\mathsf{D}{:}(\text{Int},\mathsf{S})}}$$

which is not a minimal proof. There are two problems in this example: first we do superfluous quantification over a variable not occurring in the current expression or type, secondly we use ($\forall$–introduction) and ($\forall$–elimination) successively thus giving no effect.

We avoid such situations by making ($\forall$–introduction) and ($\Rightarrow$–introduction) an integral part of the rule for `let`, and ($\forall$–elimination) and ($\Rightarrow$–elimination) a part of the rule for variables. Further we introduce *simultaneously constrained typeschemes*, which makes it possible to do all necessary ($\forall$–introduction) and ($\Rightarrow$–introduction) at once. Finally we make explicit in the type rule for `let` which binding time variables we will quantify over.

This kind of reformulation is well known, and a similar modification can be found in [Clement, Despeyroux, Despeyroux, & Kahn 1986]. This reformulation will make it easier to formulate the algorithm for inferring types. The algorithm will be given in chapter 9.

8.1 Reformulation

The new formulation is very much inspired by Thatte [Thatte 1988]. The idea has two steps: First we allow *simultaneously* quantified binding time variables, secondly we *constrain* these. A *constrained typescheme* $\sigma = \forall(\beta_1 \ldots \beta_n)^C.\kappa$ denotes quantification of $(\beta_1 \ldots \beta_n)$ constrained by constraint set C. Note that κ cannot be a typescheme. The associated syntactical construct for expressions (*constrained type-abstraction*) is written $\Lambda(\beta_1 \ldots \beta_n)^C.$e. Application of constrained type-abstractions are denoted by $\diamond$. We thus get the following types and type schemes:

$$\sigma ::= \kappa \mid \forall(\beta_1 \ldots \beta_n)^C.\kappa$$
$$b ::= \mathsf{S} \mid \mathsf{D} \mid \beta$$
$$\kappa ::= (\mathrm{Int},b) \mid (\mathrm{Bool},b) \mid (\kappa \rightarrow \kappa,b) \mid (\kappa \times \kappa,b)$$

and syntax for expressions Exp_{spec}:

$$\begin{array}{lll}
\mathrm{e} & ::= & \texttt{true} \mid \texttt{false} \mid n \mid \mathrm{x} \\
& \mid & \texttt{if}^b\ \mathrm{e}\ \texttt{then}\ \mathrm{e}\ \texttt{else}\ \mathrm{e} \\
& \mid & \lambda^b\mathrm{x.e} \mid \mathrm{e}@^b\mathrm{e} \\
& \mid & \mathrm{e}\ \texttt{op}^b\ \mathrm{e} \mid \texttt{fix}^b\ \mathrm{f\ x.e} \mid \mathrm{let\ x}\ \texttt{=}\ \mathrm{e}\ \texttt{in}\ \mathrm{e} \\
& \mid & \texttt{pair}^b(\mathrm{e,e}) \mid \texttt{fst}^b\ \mathrm{e} \mid \texttt{snd}^b\ \mathrm{e} \\
& \mid & \Lambda(\beta_1 \ldots \beta_n)^C.\mathrm{e} \mid \mathrm{e}\diamond(b_1 \ldots b_n)
\end{array}$$

The type rules for introduction and elimination of constrained type schemes are given in figure 8.1.

$$(\mathrm{Var})\quad \frac{C' \vdash [b_i/\beta_i]c_j}{A \bigcup \{\mathrm{x}{:}\forall(\beta_1 \ldots \beta_n)^{\{c_1,\ldots,c_m\}}.\kappa\}, C' \vdash \mathrm{x}\diamond(b_1 \ldots b_n){:}[b_i/\beta_i]\kappa}$$

$$(\mathrm{let})\quad \frac{\begin{array}{c} A,C \vdash \mathrm{e}'{:}\kappa' \quad A \bigcup \{\mathrm{x}{:}\forall(\beta_1 \ldots \beta_n)^C.\kappa'\}, C' \vdash \mathrm{e}{:}\kappa \\ \{\beta_1 \ldots \beta_n\} = \mathit{FreeBTVars}(C) \bigcup \mathit{FreeBTVars}(\kappa') - \mathit{FreeBTVars}(A)\end{array}}{A,C' \vdash \texttt{let}\ \mathrm{x} = \Lambda(\beta_1 \ldots \beta_n)^C.\mathrm{e}'\ \texttt{in}\ \mathrm{e}{:}\kappa}$$

Figure 8.1: Type rules for system MT

In the new rule for `let`, $\mathit{FreeBTVars}(C) \bigcup \mathit{FreeBTVars}(\kappa') - \mathit{FreeBTVars}(A)$ is a generalization of the usual definition of *generification*. $\mathit{FreeBTVars}(C)$ is needed since variables in C can occur in e (in coercions or as annotations) without occurring in κ'. This implies that we will be abstracting over more variables than in usual ML-polymorphism — this is the motivation for the reductions on constraint sets presented in chapter 10. Since polymorphism in the new system corresponds more to ML-polymorphism, we will coin the old system DM (for Damas-Milner) and the new MT (for Milner-Tofte).

Another motivation for replacing the two second order constructs with one general one is that the $\Lambda b{\rightsquigarrow}b'.\mathrm{e}$ construct has no operational meaning (see the specialization rules in figure 5.3). The construct is only necessary at binding time analysis time, intuitively to make it possible to quantify over variables occurring in the coercion set. The $\Lambda\beta.\mathrm{e}$ construct, on the other hand, has a very important operational meaning, since it manages the instantiation of free binding time variables (in annotations and coercions) in `let`-bound expressions.

The specialization rules for the new (MT) constructs are similar to the rules for $\Lambda\beta.\mathrm{e}$ and $\mathrm{e}\diamond b$, and are given in figure 8.2

$$\text{(gen)}\quad \frac{}{\rho\vdash\Lambda(\beta_1\ldots\beta_n)^C.\mathrm{e}\longrightarrow_{spec}[\![\Lambda(\beta_1\ldots\beta_n)^C.\mathrm{e}]\!]\rho}$$

$$\text{(inst)}\quad \frac{\rho\vdash\mathrm{e}\longrightarrow_{spec}[\![\Lambda(\beta_1\ldots\beta_n)^C.\mathrm{e}']\!]\rho' \quad \rho'\vdash[b_i/\beta_i]\mathrm{e}'\longrightarrow_{spec}v}{\rho\vdash\mathrm{e}\diamond(b_1\ldots b_n)\longrightarrow_{spec}v}$$

Figure 8.2: Specialization of Constrained type-abstraction

8.2 Agreement with the Old System

Let $\vdash_{DM}$ denote the DM type system presented in figure 4.2 and 4.3 and let $\vdash_{MT}$ denote the MT system defined by replacing the rules (∀–introduction), (∀–elimination), (⇒–introduction), (⇒–elimination), (var) and (let) in DM by the rules (var) and (let) presented in figure 8.1.

We now want to show that MT is sound *w.r.t.* DM. First we define a relationship between types in DM and types in MT:

Definition 8.1
Function $\mathcal{M}$ mapping types in MT to types in DM is defined by:

1. $\mathcal{M}(\kappa) = \kappa$
2. $\mathcal{M}(\forall(b_1,\ldots b_n)^{(c_1,\ldots,c_m)}.\kappa) = \forall b_1\ldots\forall b_n.c_1 \Rightarrow \ldots c_m \Rightarrow \kappa$

$\square$(Def.8.1)

Then we define a relationship between expressions in DM and expressions in MT:

Definition 8.2
We define equality $\doteq$ between terms in MT and terms in DM by:

b	$\doteq$	b	, for b∈{`true`,`false`}
n	$\doteq$	n	, for n∈$\mathbb{N}$
x	$\doteq$	x	, if x not `let`-bound
x$\diamond(b_1,\ldots,b_n)$	$\doteq$	x$\diamond b_1\diamond\ldots\diamond b_n\square c_1\square\ldots\square c_m$	, for any m and c_i
`if`^{b} e$_1$ `then` e$_2$ `else` e$_3$	$\doteq$	`if`^{b} e$_1'$ `then` e$_2'$ `else` e$_3'$	, if e$_i\doteq$e$_i'$
λ^bx.e	$\doteq$	λ^bx.e$'$	, if e$\doteq$e$'$
e$_1$@^{b}e$_2$	$\doteq$	e$_1'$@^{b}e$_2'$	, if e$_i\doteq$e$_i'$
e$_1$`op`^{b}e$_2$	$\doteq$	e$_1'$`op`^{b}e$_2'$	, if e$_i\doteq$e$_i'$
`fix`^{b} f x.e	$\doteq$	`fix`^{b} f x.e$'$	, if e$\doteq$e$'$
`let` x = e$_1$ `in` e$_2$	$\doteq$	`let` x = e$_1'$ `in` e$_2'$	, if e$_i\doteq$e$_i'$
`pair`^{b}(e$_1$,e$_2$)	$\doteq$	`pair`^{b}(e$_1'$,e$_2'$)	, if e$_i\doteq$e$_i'$
`fst`^{b}e	$\doteq$	`fst`^{b}e$'$	, if e$\doteq$e$'$
`snd`^{b}e	$\doteq$	`snd`^{b}e$'$	, if e$\doteq$e$'$
$\Lambda(b_1,\ldots b_n)^{(c_1,\ldots,c_m)}$.e	$\doteq$	$\Lambda b_1\ldots\Lambda b_n,\Lambda c_1\ldots\Lambda c_m$.e$'$	, if e$\doteq$e$'$

$\square$(Def.8.2)

Function $\mathcal{M}$ is easily extended to handle type environments. We now turn to soundness of system MT. By slight abuse of notation we will use c to range over both constraints and coercions in system DM; *e.g.* we will write $\Lambda c.\mathrm{e} : c \Rightarrow \kappa$, where c stands for both $b \rightsquigarrow b'$ and $b \leq b'$ (c will always stand for corresponding coercions and constraints).

Proposition 8.1
We have soundness of the MT system:

$$\forall A, C, \mathrm{e}, \kappa : \exists \mathrm{e}' : \mathrm{e} \doteq \mathrm{e}' \wedge A,C \vdash_{MT} \mathrm{e}{:}\kappa \Longrightarrow \mathcal{M}(A),C \vdash_{DM} \mathrm{e}'{:}\kappa$$

if C is *minimal* in the sense that $\neg\exists C' \subsetneq C : A, C' \vdash_{MT} \mathrm{e}{:}\kappa$.
Proof

For each proof tree in MT, we must exhibit a proof tree in DM. We show only the cases for (var) and (let) since all other cases are trivial:

$\boxed{\text{Rule (var)}}$ We have in MT:

$$\text{(Var)}\quad \dfrac{C' \vdash [b_i/\beta_i]c_j}{A \bigcup \{\mathrm{x}{:}\forall(\beta_1 \ldots \beta_n)^{\{c_1,\ldots,c_m\}}.\kappa\}, C' \vdash \mathrm{x} \diamond (b_1 \ldots b_n){:}[b_i/\beta_i]\kappa}$$

Let $A' = \mathcal{M}(A \bigcup \{\mathrm{x} : \forall(\beta_1 \ldots \beta_n)^{\{c_1,\ldots,c_m\}}.\kappa\}) = \mathcal{M}(A) \bigcup \{\mathrm{x} : \forall b_1 \ldots \forall b_n.c_1 \Rightarrow \ldots c_m \Rightarrow \kappa\}$. So we can build the following proof tree in DM:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A', C' \vdash \mathrm{x} : \forall\beta_1 \ldots \forall\beta_n.c_1 \Rightarrow \ldots c_m \Rightarrow \kappa}{A', C' \vdash \mathrm{x} \diamond b_1 : [b_1/\beta_1](\forall\beta_2 \ldots \forall\beta_n.c_1 \Rightarrow \ldots c_m \Rightarrow \kappa)}}{\vdots}}{A', C' \vdash \mathrm{x} \diamond b_1 \ldots \diamond b_n : [b_i/\beta_i](c_1 \Rightarrow \ldots c_m \Rightarrow \kappa)} \quad C' \vdash [b_i/\beta_i]c_1}{A', C' \vdash \mathrm{x} \diamond b_1 \ldots \diamond b_n \Box c_1 : c_2 \Rightarrow \ldots c_m \Rightarrow \kappa} \quad C' \vdash [b_i/\beta_i]c_2}{\vdots}}{A', C' \vdash \mathrm{x} \diamond b_1 \ldots \diamond b_n \Box c_1 \ldots \Box c_m : \kappa}$$

The inference tree was constructed using the (var) rule, the ($\forall$–elimination) rule n times and the ($\Rightarrow$–elimination) m times.

$\boxed{\text{Rule (let)}}$ We have in MT:

$$\text{(let)}\quad \dfrac{\begin{array}{l} A,C \vdash \mathrm{e}'{:}\kappa' \quad A \bigcup \{\mathrm{x}{:}\forall(\beta_1 \ldots \beta_n)^C.\kappa'\}, C' \vdash \mathrm{e}{:}\kappa \\ \{\beta_1 \ldots \beta_n\} = \mathit{FreeBTVars}(C) \bigcup \mathit{FreeBTVars}(\kappa') - \mathit{FreeBTVars}(A) \end{array}}{A,C' \vdash \texttt{let}\ \mathrm{x} = \Lambda(\beta_1 \ldots \beta_n)^C.\mathrm{e}'\ \texttt{in}\ \mathrm{e}{:}\kappa}$$

Assume $C = \{c_1, \ldots, c_m\}$. Let $A' = \mathcal{M}(A) \bigcup \{\mathrm{x} : \forall b_1 \ldots \forall b_n.c_1 \Rightarrow \ldots c_m \Rightarrow \kappa\}$. We have that β_i does not occur free in C' since C' is minimal. We now have in DM:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{M}(A), C' \bigcup \{c_1, \ldots, c_m\} \vdash \mathrm{e}' : \kappa'}{\mathcal{M}(A), C' \bigcup \{c_1, \ldots, c_{m-1}\} \vdash \Lambda c_m.\mathrm{e}' : c_m \Rightarrow \kappa'}}{\vdots}}{\mathcal{M}(A), C' \vdash \Lambda c_1 \ldots \Lambda c_m \mathrm{e}' : c_1 \Rightarrow \ldots c_m \Rightarrow \kappa'}}{\mathcal{M}(A), C' \vdash \Lambda\beta_n \Lambda c_1 \ldots \Lambda c_m \mathrm{e}' : \forall\beta_n.c_1 \Rightarrow \ldots c_m \Rightarrow \kappa'}}{\vdots}}{\mathcal{M}(A), C' \vdash \Lambda\beta_1 \ldots \Lambda\beta_n \Lambda c_1 \ldots \Lambda c_m \mathrm{e}' : \forall\beta_1 \ldots \forall\beta_n.c_1 \Rightarrow \ldots c_m \Rightarrow \kappa'} \quad A', C' \vdash \mathrm{e} : \kappa}{\mathcal{M}(A), C' \vdash \texttt{let}\ \mathrm{x} = \Lambda\beta_1 \ldots \Lambda\beta_n.\Lambda c_1 \ldots \Lambda c_n.\mathrm{e}'\ \texttt{in}\ \mathrm{e} : \kappa}$$

The leaves $\mathcal{M}(A), \{c_1, \ldots, c_m\} \vdash \mathrm{e}' : \kappa')$ and $(A[\mathrm{x} : \forall b_1 \ldots \forall b_n.c_1 \Rightarrow \ldots c_m \Rightarrow \kappa], C \vdash \mathrm{e} : \kappa$ follow from the induction hypothesis. The left branch uses (from leave to root) m times ($\Rightarrow$-introduction) and n times ($\forall$-introduction). Finally the (let) rule is used.

$\square$(Prop.8.1)

We can define relation $\doteq$ on values in Val_{spec} similarly to above. Using proposition 8.1, it would then be easy to prove

$$\forall \rho, \rho', e, e' : \rho \doteq \rho' \wedge e \doteq e' \wedge \rho \vdash_{MT} e \longrightarrow_{spec} v \Longrightarrow \rho' \vdash_{DM} e' \longrightarrow_{spec} v' \wedge v \doteq v'$$

It should be clear that this implies, that the main correctness theorem (Theorem I) also holds for MT.

A similar proof can be found in [Clement, Despeyroux, Despeyroux, & Kahn 1986] (for a Curry-style language). Here the opposite implication is also proved, that is that MT is complete *w.r.t.* DM. Stating and proving this is slightly more complicated than proposition 8.1, but the proof will probably carry over from the work in [Clement, Despeyroux, Despeyroux, & Kahn 1986] as it did above. We are confident that a completeness proposition can be stated and proved correct, but we have chosen not to do so for lack of time.

Chapter 9

Algorithm

We are now prepared to present the algorithm for inferring binding times. During the report so far we have assumed well–typedness of the programs to be analyzed and assumed the types of every expression to be present in order to use them as "guidelines" for our analysis. In practice it is much more convenient to infer the standard types along with the binding times.

In general, type inference is the problem of finding a principal type of an expression e *w.r.t.* a set of type rules, given an environment A mapping the free variables of e to types. We will split this problem into two phases:

1. Find the most general typing κ for e along with a set of *verification conditions* C which must hold for the typing to be correct.

2. Find a substitution ζ such that $\zeta \Vdash C$ holds. Apply ζ to A, κ and e.

In usual type inference, we would need to check whether C was consistent before finding substitution ζ. In binding time analysis however, C *is* consistent (since making everything D is a legal typing) so this is not necessary. Also, in standard type inference, we are interested in the most general solution (defined in some appropriate way), while we are interested in a (somehow) minimal substitution so we can apply it to e and get the annotations right.

We will follow this division into phases: section 9.2 will describe the first phase, and the last will be presented in section 9.3. It is possible to reduce the constraint set during type inference; this will lead to fewer parameters in $\Lambda(\beta_1 \ldots \beta_n)^C$, fewer coercions in the final program and a faster algorithm. This will be described in chapter 10.

9.1 Preliminaries

The programmer, who will be using the binding time analysis, writes his programs e in Exp_{std}. The binding time analysis type system is defined on expressions in Exp_{spec}, so expressions e in Exp_{std} should be mapped to expressions e$'$ in Exp_{spec}, such that e$'$ is general enough to allow all typings. This is done by function α: insert annotations, allow coercions on every sub-expression, insert binding time abstractions on all `let`-bound expression, and insert binding time application on all `let`-bound variable.

Definition 9.1

Function $\alpha : \mathit{Exp}_{std} \rightarrow \mathit{Exp}_{spec}$ is defined by:

$$\begin{array}{lcll}
\alpha(\texttt{true}) & = & [\beta\rightsquigarrow\beta']\texttt{true} & \beta,\beta' \ \mathit{new} \\
\alpha(\texttt{false}) & = & [\beta\rightsquigarrow\beta']\texttt{false} & \beta,\beta' \ \mathit{new} \\
\alpha(n) & = & [\beta\rightsquigarrow\beta']n & \beta,\beta' \ \mathit{new} \\
\alpha(\mathrm{x}) & = & [\beta\rightsquigarrow\beta'](\mathrm{x}\diamond\mathcal{B}) \quad \mathit{,if\ x\ is}\ \texttt{let}\text{-}\mathit{bound} & \beta,\beta',\mathcal{B} \ \mathit{new} \\
\alpha(\mathrm{x}) & = & [\beta\rightsquigarrow\beta']\mathrm{x} \qquad \mathit{,if\ x\ is\ not}\ \texttt{let}\text{-}\mathit{bound} & \beta,\beta' \ \mathit{new} \\
\alpha(\texttt{if}\ \mathrm{e}\ \texttt{then}\ \mathrm{e}'\ \texttt{else}\ \mathrm{e}'') & = & [\beta\rightsquigarrow\beta']\texttt{if}^{\beta''}\alpha(\mathrm{e})\ \texttt{then}\ \alpha(\mathrm{e}')\ \texttt{else}\ \alpha(\mathrm{e}'') & \beta,\beta',\beta'' \ \mathit{new} \\
\alpha(\lambda\mathrm{x.e}) & = & [\beta\rightsquigarrow\beta']\lambda^{\beta''}\mathrm{x}.\alpha(\mathrm{e}) & \beta,\beta',\beta'' \ \mathit{new} \\
\alpha(\mathrm{e}@\mathrm{e}') & = & [\beta\rightsquigarrow\beta'](\alpha(\mathrm{e})@^{\beta''}\alpha(\mathrm{e}')) & \beta,\beta',\beta'' \ \mathit{new} \\
\alpha(\mathrm{e}\ \texttt{op}\ \mathrm{e}') & = & [\beta\rightsquigarrow\beta'](\alpha(\mathrm{e})\ \texttt{op}^{\beta''}_{t\times t'\rightarrow t''}\alpha(\mathrm{e}')) & \beta,\beta',\beta'' \ \mathit{new} \\
\alpha(\texttt{fix}\ \mathrm{f\ x.e}) & = & [\beta\rightsquigarrow\beta']\texttt{fix}^{\beta''}\ \mathrm{f\ x}.\alpha(\mathrm{e}) & \beta,\beta',\beta'' \ \mathit{new} \\
\alpha(\texttt{pair}(\mathrm{e},\mathrm{e}')) & = & [\beta\rightsquigarrow\beta']\texttt{pair}^{\beta''}(\alpha(\mathrm{e}),\alpha(\mathrm{e}')) & \beta,\beta',\beta'' \ \mathit{new} \\
\alpha(\texttt{fst}\ \mathrm{e}) & = & [\beta\rightsquigarrow\beta']\texttt{fst}^{\beta''}\ \alpha(\mathrm{e}) & \beta,\beta',\beta'' \ \mathit{new} \\
\alpha(\texttt{snd}\ \mathrm{e}) & = & [\beta\rightsquigarrow\beta']\texttt{snd}^{\beta''}\ \alpha(\mathrm{e}) & \beta,\beta',\beta'' \ \mathit{new} \\
\alpha(\texttt{let}\ \mathrm{x} = \mathrm{e}'\texttt{in}\ \mathrm{e}) & = & [\beta\rightsquigarrow\beta']\texttt{let}\ \mathrm{x} = \Lambda\mathcal{B}^{\mathcal{C}}\alpha(\mathrm{e}')\ \texttt{in}\ \alpha(\mathrm{e}) & \beta,\beta',\mathcal{B},\mathcal{C} \ \mathit{new}
\end{array}$$

$\mathcal{B}$ and $\mathcal{C}$ are special variables to be updated during algorithm `type`. □(Def.9.1)

We thus allow coercions everywhere, coercions are also inserted on expressions which does not have base type. We then have to change our type system slightly — we simply allow identity coercions on expressions of all types. It should be clear, that this does not change the legal typings:

$$\text{(coerce)} \quad \frac{A,C\vdash \mathrm{e}:(t,b)}{A,C\vdash [b\rightsquigarrow b]\mathrm{e}:(t,b)}$$

Since all identity coercions are removed after the application of algorithm $\mathcal{W}$, the final annotated program will be well typed according to the original type system.

9.2 Algorithm `type`

This section presents the algorithm performing phase 1 from above. The algorithm is a version of algorithm $\mathcal{W}$ [Damas & Milner 1982]. The algorithm takes an expression e and a type environment A, and returns a compound type κ, a substitution ζ and a set of constraints C. We use ζ_{id} to denote the identity substitution.

Algorithm $\mathcal{W}$ descends recursively through the syntax of expression e. When an expression $[b\rightsquigarrow b']$e is encountered, the standard type of e might not be inferable from e. An example could be:

$$(\lambda \mathrm{x}.[b\rightsquigarrow b']\mathrm{x})@5$$

If this expressions is analyzed from left to right, we would encounter $[b\rightsquigarrow b']$x without knowing the type of x.

The solution is to introduce *conditioned* constraints $(b \leq b' \mid t) \in C$. Since $b < b'$ only holds for base types, $(b \leq b' \mid t) \in C$ means: if t is a base type then $b \leq b'$ otherwise $b = b'$.

```
type(e,$A$) = (($t$,$b$), $\zeta$, $C$) where
(($t$,$b$), $\zeta$, $C$) =
case e of
  true: ((Bool,S), $\zeta_{\text{id}}$, {})

  false: ((Bool,S), $\zeta_{\text{id}}$, {})

  $n$: ((Int,S), $\zeta_{\text{id}}$, {})

  x: ($A$(x), $\zeta_{\text{id}}$, {})                *if x is not* let*-bound*

  x$\diamond\mathcal{B}$:
      let $\forall\mathcal{B}_1^{\mathcal{C}_1}(t_1, b_1) = A(\text{x})$
          $\zeta$ be a substitution such that for all $\beta_i$ in $\mathcal{B}_1$: $\zeta(\beta_i) = \beta'_i$ where $\beta'_i$ are fresh.
      in *update*($\mathcal{B}$, $\zeta(\mathcal{B}_1)$);
         $(\zeta((t_1, b_1)),\ \zeta_{\text{id}},\ \zeta(\mathcal{C}_1))$

  if$^b$ e$_1$ then e$_2$ else e$_3$:
      let $((t_1, b_1), \zeta_1, C_1)$ = type(e$_1$,$A$)
          $((t_2, b_2), \zeta_2, C_2)$ = type(e$_2$,$\zeta_1 A$)
          $((t_3, b_3), \zeta_3, C_3)$ = type(e$_3$,$\zeta_2\zeta_1 A$)
          $\zeta_4$ = *unify*((Bool,$b$),$\zeta_3\zeta_2(t_1, b_1)$)
          $\zeta_5$ = *unify*($\zeta_4\zeta_3(t_2, b_2)$,$\zeta_4(t_3, b_3)$)
      in $(\zeta_5\zeta_4(t_3, b_3),\ \zeta_5\zeta_4\zeta_3\zeta_2\zeta_1,\ \zeta_5\zeta_4(\{\zeta_3\zeta_2 b_1 \leq b_3\}\bigcup\zeta_3\zeta_2 C_1\bigcup\zeta_3 C_2\bigcup C_3))$

  $\lambda^b$x.e$_1$:
      let $(\tau_1, \beta_1)$ be fresh
          $((t_2, b_2), \zeta, C)$ = type(e$_1$,$A$[x:$(\tau_1, \beta_1)$])
      in $((\zeta(\tau_1, \beta_1) \rightarrow (t_2, b_2), b),\ \zeta, \{b \leq \zeta\beta_1,\ b \leq b_2\}\bigcup C)$

  e$_1$@$^b$ e$_2$:
      let $((t_1, b_1), \zeta_1, C_1)$ = type(e$_1$,$A$)
          $((t_2, b_2), \zeta_2, C_2)$ = type(e$_2$,$\zeta_1 A$)
          $(\tau_3, \beta_3)$ be fresh
          $\zeta_3$ = *unify*($((t_2, b_2) \rightarrow (\tau_3, \beta_3), b)$, $\zeta_2(t_1, b_1)$)
      in $(\zeta_3(\tau_3, \beta_3),\ \zeta_3\zeta_2\zeta_1,\ \zeta_3(\{b \leq b_2, b \leq \beta_3\}\bigcup\zeta_2 C_1\bigcup C_2))$

  e$_1$op$^b$e$_2$:
      let $(t \times t' \rightarrow t'') = \mathcal{P}$(op)
          $((t_1, b_1), \zeta_1, C_1)$ = type(e$_1$,$A$)
          $((t_2, b_2), \zeta_2, C_2)$ = type(e$_2$,$\zeta_1 A$)
          $\zeta_3$ = *unify*($(t, b)$,$\zeta_2(t_1, b_1)$)
          $\zeta_4$ = *unify*($\zeta_3(t', b)$,$\zeta_3(t_2, b_2)$)
      in $(\zeta_4\zeta_3(t'', b),\ \zeta_4\zeta_3\zeta_2\zeta_1,\ \zeta_4\zeta_3(\zeta_2 C_1\bigcup C_2))$
```

Figure 9.1: Algorithm type

```
fix$^b$f x.e$_1$:
    let $(\tau_1,\beta_1)$ and $(\tau_2,\beta_2)$ be fresh
        $((t_2,b_2),\zeta_1,C)$ = type(e$_1$,$A$[f:$((\tau_1,\beta_1)\to(\tau_2,\beta_2),b)$,x:$(\tau_1,\beta_1)$])
        $\zeta_2 = \mathit{unify}(\zeta_1((\tau_1,\beta_1)\to(\tau_2,\beta_2),b),(t_2,b_2))$
    in $(\zeta_2\zeta_1((\tau_1,\beta_1)\to(\tau_2,\beta_2),b),\ \zeta_2\zeta_1,\ \zeta_2 C)$

pair$^b$(e$_1$,e$_2$)
    let $((t_1,b_1),\zeta_1,C_1)$ = type(e$_1$,$A$)
        $((t_2,b_2),\zeta_2,C_2)$ = type(e$_2$,$\zeta_1 A$)
    in $((\zeta_2(t_1,b_1)\times(t_2,b_2),b),\ \zeta_2\zeta_1,\ \{b\le\zeta_2 b_1, b\le b_2\}\bigcup\zeta_2 C_1\bigcup C_2)$

fst$^b$e$_1$:
    let $((t_1,b_1),\zeta_1,C_1)$ = type(e$_1$,$A$)
        $(\tau_2,\beta_2)$ and $(\tau_3,\beta_3)$ be fresh
        $\zeta_2 = \mathit{unify}(((\tau_2,\beta_2)\times(\tau_3,\beta_3),b),(t_1,b_1))$
    in $(\zeta_2(\tau_2,\beta_2),\ \zeta_2\zeta_1,\ \zeta_2(\{b\le\beta_2, b\le\beta_3\}\bigcup C_1))$

snd$^b$e$_1$:
    let $((t_1,b_1),\zeta_1,C_1)$ = type(e$_1$,$A$)
        $(\tau_2,\beta_2)$ and $(\tau_3,\beta_3)$ be fresh
        $\zeta_2 = \mathit{unify}(((\tau_2,\beta_2)\times(\tau_3,\beta_3),b),(t_1,b_1))$
    in $(\zeta_2(\tau_3,\beta_3),\ \zeta_2\zeta_1,\ \zeta_2(\{b\le\beta_2, b\le\beta_3\}\bigcup C_1))$

let x = $\Lambda\mathcal{B}^{\mathcal{C}}$.e$_1$ in e$_2$:
    let $((t_1,b_1),\zeta_1,C_1)$ = type(e$_1$,$A$)
    in $\mathit{update}(\mathcal{B},\ \mathit{FreeBTVars}(t_1,b_1)\bigcup\mathit{FreeBTVars}(C_1)-\mathit{FreeBTVars}(\zeta_1 A))$;
        $\mathit{update}(\mathcal{C},\ C_1)$;
        let $((t_2,b_2),\zeta_2,C_2)$ = type(e$_2$,$\zeta_1 A\bigcup\{$x:$\forall\mathcal{B}^{\mathcal{C}}(t_1,b_1)\}$)
        in $((t_2,b_2),\ \zeta_2\zeta_1,\ \zeta_2 C_1\bigcup C_2)$

[$b\rightsquigarrow b'$]e$_1$:
    let $((t_1,b_1),\zeta_1,C_1)$ = type(e$_1$,$A$)
        $\zeta_2 = \mathit{unify}(b_1,b)$
    in $(\zeta_2(t_1,b'),\ \zeta_2\zeta_1,\ \zeta_2(\{(b\le b'\mid t_1)\}\bigcup C_1))$
```

Figure 9.2: Algorithm type (continued)

There are two reasons why we need conditioned constraints: first we have chosen to do standard type inference and binding time analysis at once, so the standard type information is not always at hand, secondly (and most important) the type system suffers from the deficit that not all types are treated equally; base types can be lifted while other types cannot. The solution to the second (making the first oblivious) is to allow induced coercions — this will be discussed in subsection 13.1.1.

Figure 9.1 and 9.2 presents algorithm `type` for the non-polyvariant part of our binding time analysis. The algorithm should contain no surprises for readers familiar with algorithm $\mathcal{W}$. In the algorithm segment for `let` x $= \Lambda\mathcal{B}^{\mathcal{C}}.\mathrm{e}_1$ `in` e_2, variable $\mathcal{B}$ is updated to $\mathit{FreeBTVars}(C) \bigcup \mathit{FreeBTVars}(\kappa) - \mathit{FreeBTVars}(\zeta_1 A)$ according to the type rule for `let` in figure 8.1.

When typing (x$\diamond\mathcal{B}$), we need to update the variable $\mathcal{B}$ to a list of binding time values b_i. The length of this is easily determined by the type of x. If $\forall\mathcal{B}^{\mathcal{C}}.(t_1, b_1)$ is the type of x, we return $(\zeta((t_1, b_1)), \zeta_{\mathrm{id}}, \zeta(\mathcal{C}))$, where ζ maps all variables in $\mathcal{B}$ to fresh variables.

9.3 Finding a Solution to the Set of Constraints

After applying algorithm `type` to a term (program) e, we have got A,C⊢e:κ. We then need to find a solution to C. First note that every standard type has been instantiated, thus every conditioned constraint either leads to a unification or to a non-conditioned constraint. Since constraints now are simple ($b \leq b'$), finding a solution of C is an easy task. The overall algorithm becomes:

- Let (κ,ζ,C) = `type`(e,A)
- Let ζ' be defined by: $\forall\beta \in \mathit{FreeBTVars}(e)$:
  If $\mathsf{D} \leq \beta$ is a member of the reflexive, transitive closure of C, substitute D for β.
  Otherwise substitute S for β.
- Remove identity coercions in $\zeta'\zeta$e, and return the result.

Chapter 10

Reducing the Constraint Set

Algorithm `type` as presented in the last chapter results in a lot of superfluous binding time parameters to every `let` bound expression and many unnecessary coercions. The reason for this is that we abstract over all variables in $\mathit{FreeBTVars}(C) \bigcup \mathit{FreeBTVars}(\kappa) - \mathit{FreeBTVars}(A)$ and many of these are not significant. By reducing the constraint set, most of the insignificant variables can be eliminated.

In the annotated program, the reductions will manifest itself in fewer binding time parameters in `let`-bound expressions and fewer lifts (some coercions will by unification be turned into identity coercions, which can be safely removed). The annotated programs will thus be made more accessible for manual inspection, and the specialization phase will be speeded up. Further we believe (and this belief will be justified in chapter 11) that the time spent reducing, will be more than saved during binding time analysis, due to the fewer parameters and constraints in `let`-bound expressions.

The idea of the reductions is that a variable β in the constraint set C can be substituted for by a value (or variable) b. The reduction is correct, if β is not *observable* (occur in the type or type environment) and replacing b for β does not effect the coercion consequences of C not involving β.

Our constraint set reductions will be based heavily on work by Fuh and Mishra [Fuh & Mishra 1989]. Their reduction are proved correct in the above sense. Our case is somewhat more complicated since, besides occurring in type and type environment, binding time variables can occur "significantly" as annotations. We are not interested in leaving those untouched — actually we are interested in reducing the number of different annotations — but the reductions should be "safe" such that we never get more residualization. We will not prove our reductions correct here, but we hope that the discussion and the fact that the reductions have proved to work in practice (appendix D shows some convincing results of reduction) will convince the reader of their correctness. Many of the problems encountered here are general of nature and a further investigation including formal proofs would be an interesting topic for future work.

We find it important to first explain the work by Fuh and Mishra [Fuh & Mishra 1989]. This is done in section 10.1, modified to take compound types (t, b) and product types *etc.* into account, but without considering the special problem related to our binding time analysis problem. We then (in section 10.2) discuss why their reduction scheme cannot immediately be adopted in our case, and in section 10.3 we present the modifications necessary. Section 10.4 discusses the quality of the reductions.

10.1 Fuh and Mishra

Fuh and Mishra [Fuh & Mishra 1989] observe, that not all type variables in a set of constraints are "*visible or observable, in the sense, that future coercions may refer to them*". The idea is then, that "*type variables, that are not observable, are useless, unless they constrain observable variables, by "connecting" them*".

First we define the notion of *observable* and *internal* variables:

DEFINITION 10.1

$$\begin{aligned} Obv(A,C \vdash \text{e}{:}\kappa) &= FreeBTVars(A) \bigcup FreeBTVars(\kappa) \\ Intv(A,C \vdash \text{e}{:}\kappa) &= FreeBTVars(C) - Obv(A, C \vdash \text{e}{:}\ \kappa)) \end{aligned}$$

□(Def.10.1)

Fuh and Mishra define *minimal typing* in terms of observable and internal variables, and gives a proof of the existence of minimal typing by giving an effective algorithm for computing a minimal typing. Fuh and Mishra argue that this algorithm will be very slow and propose two reduction mechanisms *G-reduction* and *S-reduction* instead, which can detect and remove most redundancy.

Before going on to specifying G-reduction, we illustrate in figure 10.1, what such a reduction can do. In this and subsequent figures, a type b_1, less than another type b_2, is depicted below b_2.

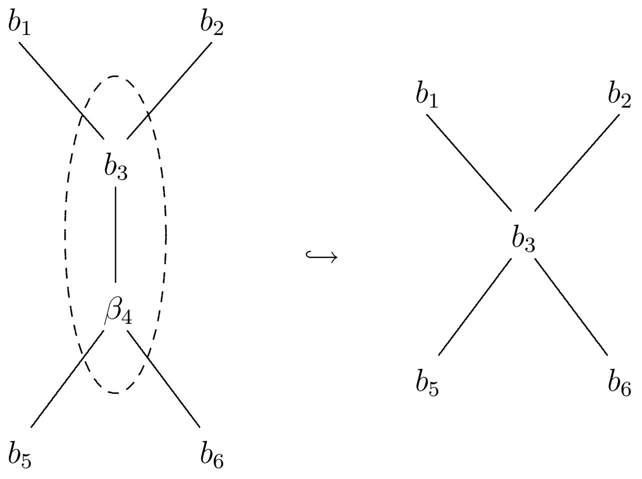

Figure 10.1: A reduction of internal variables

In figure 10.1 assume that β_4 is an internal variable. It then does not serve any function in this constraint set and we can thus apply $[b_3/\beta_4]$ to the constraint set.

The reason that we can do the reduction of figure 10.1 is that, the reduction does not alter the behavior of other variables. We also see, that in the reduced graph/constraint set, even if b_3 is an internal variable β, it serves an important function "connecting" b_1, b_2, b_5 and b_6 (if b_6 is S, we could of course substitute b_5 for β, but if b_6 is a variable, this would not be safe). The problem of when such a reduction applies, seems to depend on the binding times occurring above and below a given binding time variable. This leads to the following definition [Fuh & Mishra 1989]:

DEFINITION 10.2
Let C be a set of constraints, and b a type in C. Now define:

1. $\mathit{above}_C(b) = \{b' \mid b \leq b' \in C^*\}$
2. $\mathit{below}_C(b) = \{b' \mid b' \leq b \in C^*\}$

where C^* denotes the reflexive, transitive closure of C. □(Def.10.2)

10.1.1 G-reduction

Now define a relation $\leq_G$ between binding time values:

DEFINITION 10.3
If β is a variable in C and b is a binding time value (or variable) in C, we say that β G-subsumes b, written $\beta \leq_G b$ in C, iff

1. $\mathit{above}_C(\beta) - \{\beta\} \subseteq \mathit{above}_C(b)$
2. $\mathit{below}_C(\beta) - \{\beta\} \subseteq \mathit{below}_C(b)$

□(Def.10.3)

It is important to notice that the relation $\leq_G$, does not necessarily reflect the ordering $\leq$ between binding time values. Now we are able to define $\hookrightarrow_G$ (in [Fuh & Mishra 1989] it is denoted $\mapsto_G$), the reduction based on relation $\leq_G$.

DEFINITION 10.4
The relation $\hookrightarrow_G$ is defined as follows:

$$A,C\vdash \mathrm{e}:\sigma \hookrightarrow_G A,C'\vdash \mathrm{e}:\sigma \iff \begin{cases} (1) & \beta \leq_G b_1 \text{ in } C, \text{ and} \\ (2) & \beta \text{ not in } A \text{ or } \sigma, \text{ and} \\ (3) & C' = [b_1/\beta]C. \end{cases}$$

□(Def.10.4)

Notice that (2) says that G-reduction only can be applied to variables in *Intv*.

In figure 10.2, we show a number of situations, where G-reduction is applicable. In situation (a), we have $\beta_4 \leq_G b_3$ (and if b_3 is a variable β_3 then also $\beta_3 \leq_G \beta_4$). In situation (b) $\mathit{above}(\beta_3) - \{\beta_3\} \subseteq \mathit{above}(b_4)$ and $\mathit{below}(\beta_3) - \{\beta_3\} \subseteq \mathit{below}(b_4)$, and thus $\beta_3 \leq_G b_4$. In situation (c), we have $\mathit{above}(\beta_4) - \{\beta_4\} \subseteq \mathit{above}(b_3)$ and $\mathit{below}(\beta_4) - \{\beta_4\} \subseteq \mathit{below}(b_3)$, and thus $\beta_4 \leq_G b_3$. Situation (d) shows, that G-reduction can be performed even when b_2 and β_3 are unrelated.

The result of G-reducing the four cases of figure 10.2 is shown in figure 10.3. In situation (a) and (c) assume, that β_4 is in *Intv*, and in (b) and (d) that β_3 is.

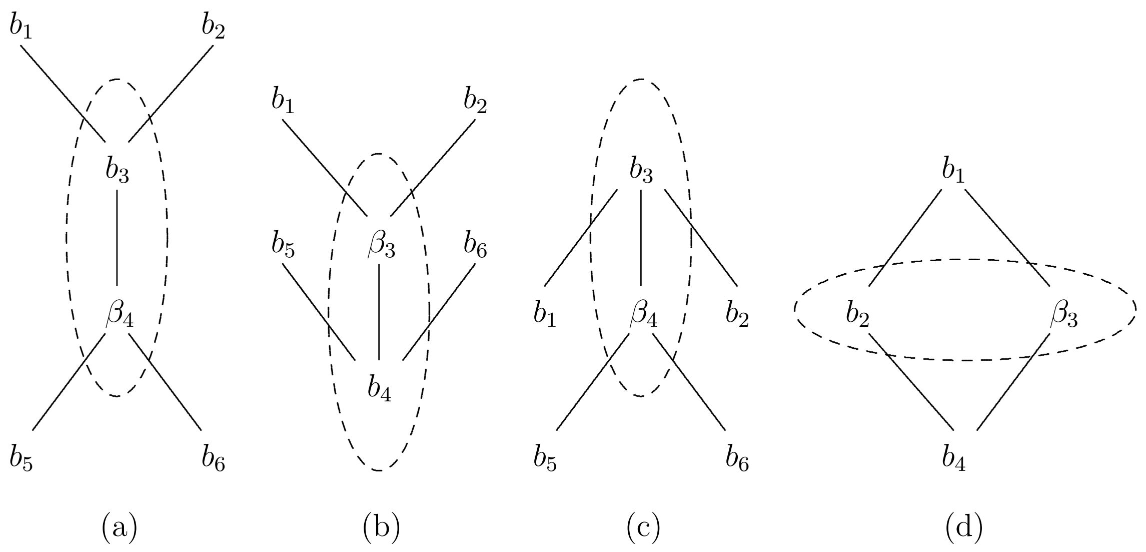

Figure 10.2: Possible to do apply $\hookrightarrow_G$

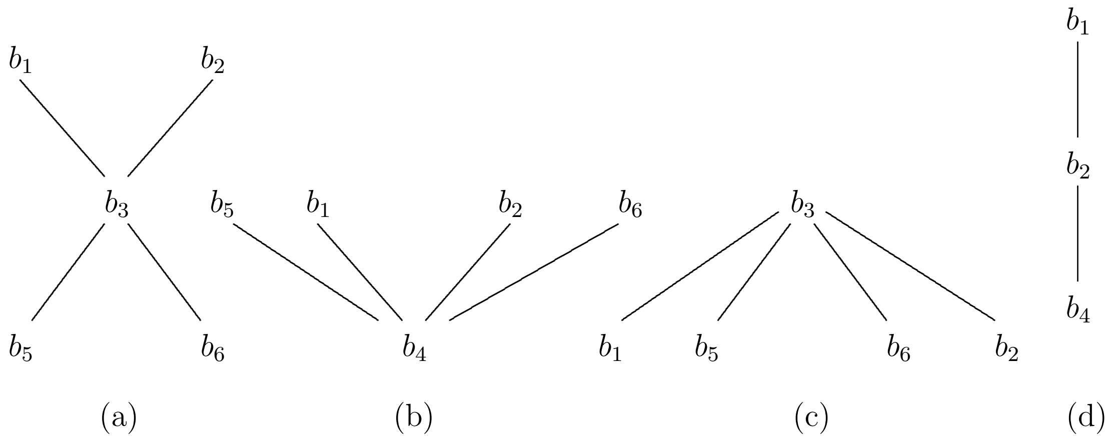

Figure 10.3: Result of applying $\hookrightarrow_G$

10.1.2 S-reduction

In [Fuh & Mishra 1989], Fuh and Mishra also presents a reduction called S-reduction, which we will denote $\hookrightarrow_S$ (Fuh and Mishra calls it $\mapsto_S$). S-reduction is a special case of G-reduction extended to handle variables in *Obv*. The idea of observable variables was exactly that they are important (occur in the type or environment) to the result so such variables cannot be "collapsed" with other values as freely as internal variables. We thus need a restriction on which observable variables we can reduce; for this we need functions *expand-polarity* and *contract-polarity* defined in definition 10.5. Occurrences of a variable β in a typescheme σ can be affected in four ways: no effect, expansion, contraction and a combination of expansion and contraction. These are represented by $\emptyset$, $\{\uparrow\}$, $\{\downarrow\}$ and $\{\uparrow,\downarrow\}$ respectively.

Definition 10.5
expand-polarity and *contract-polarity* is defined by simultaneous induction:

$$\textit{expand-polarity}(\beta,\sigma) = \begin{cases} \{\uparrow\} & \text{,if } \sigma = (\mathrm{Int},\beta) \vee \sigma = (\mathrm{Bool},\beta) \\ \textit{expand-polarity}(\beta,\sigma') & \text{,if } \sigma = \forall(\beta_1 \dots \beta_n)^C.\sigma' \\ \textit{contract-polarity}(\beta,(t_1,b_1)) \\ \bigcup \textit{expand-polarity}(\beta,(t_2,b_2)) & \text{,if } \sigma = ((t_1,b_1) \to (t_2,b_2),b) \wedge \beta \neq b \\ \textit{contract-polarity}(\beta,(t_1,b_1)) \\ \bigcup \textit{expand-polarity}(\beta,(t_2,b_2)) \bigcup \{\uparrow\} & \text{,if } \sigma = ((t_1,b_1) \to (t_2,b_2),\beta) \\ \textit{expand-polarity}(\beta,(t_1,b_1)) \\ \bigcup \textit{expand-polarity}(\beta,(t_2,b_2)) & \text{,if } \sigma = ((t_1,b_1) \times (t_2,b_2),b) \wedge \beta \neq b \\ \textit{expand-polarity}(\beta,(t_1,b_1)) \\ \bigcup \textit{expand-polarity}(\beta,(t_2,b_2)) \bigcup \{\uparrow\} & \text{,if } \sigma = ((t_1,b_1) \times (t_2,b_2),\beta) \\ \emptyset & \text{,otherwise} \end{cases}$$

$$\textit{contract-polarity}(\beta,\sigma) = \begin{cases} \{\downarrow\} & \text{,if } \sigma = (\mathrm{Int},\beta) \vee \sigma = (\mathrm{Bool},\beta) \\ \textit{contract-polarity}(\beta,\sigma') & \text{,if } \sigma = \forall(\beta_1 \dots \beta_n)^C.\sigma' \\ \textit{expand-polarity}(\beta,(t_1,b_1)) \\ \bigcup \textit{contract-polarity}(\beta,(t_2,b_2)) & \text{,if } \sigma = ((t_1,b_1) \to (t_2,b_2),b) \wedge \beta \neq b \\ \textit{expand-polarity}(\beta,(t_1,b_1)) \\ \bigcup \textit{contract-polarity}(\beta,(t_2,b_2)) \bigcup \{\downarrow\} & \text{,if } \sigma = ((t_1,b_1) \to (t_2,b_2),\beta) \\ \textit{contract-polarity}(\beta,(t_1,b_1)) \\ \bigcup \textit{contract-polarity}(\beta,(t_2,b_2)) & \text{,if } \sigma = ((t_1,b_1) \times (t_2,b_2),b) \wedge \beta \neq b \\ \textit{contract-polarity}(\beta,(t_1,b_1)) \\ \bigcup \textit{contract-polarity}(\beta,(t_2,b_2)) \bigcup \{\downarrow\} & \text{,if } \sigma = ((t_1,b_1) \times (t_2,b_2),\beta) \\ \emptyset & \text{,otherwise} \end{cases}$$

$\square_{\text{(Def.10.5)}}$

The *expand-polarity* of a variable β in a type σ indicates the effect on occurrences of β in σ when σ is coerced to a super type σ'.

Since Fuh and Mishra do not have compound types (t, b), they have no definition of the polarity of β in $((t_1, b_1) \rightarrow (t_2, b_2), \beta)$ (or in $((t_1, b_1) \times (t_2, b_2), \beta)$). The definition of definition 10.5 is the immediately obvious (though we will change it in the following sections).

Now we return to the definition of S-reduction. First we define the notion of *S-subsumes*:

Definition 10.6
Let $\beta \in \mathit{FreeBTVars}(C)$ and β occurs in σ. We say β is *S-subsumed* by b in C and σ written $\beta \leq_S b$ in C and σ, iff either

1. $C \vdash b \leq \beta$ and *expand-polarity*(β,σ) $= \{\uparrow\}$ and $\mathit{below}_C(\beta) - \{\beta\} \subseteq \mathit{below}_C(b)$, or
2. $C \vdash \beta \leq b$ and *expand-polarity*(β,σ) $= \{\downarrow\}$ and $\mathit{above}_C(\beta) - \{\beta\} \subseteq \mathit{above}_C(b)$.

□(Def.10.6)

If *expand-polarity*(β,σ) $= \{\}$, then β does not occur in σ. Thus there is no need for this case in the definition of S-subsume, since it would overlap with the definition of G-subsume. We can now define S-reduction:

Definition 10.7
Let A,$C \vdash$e:σ be a typing and $\mathit{domain}(A) = \{x_1, \ldots, x_n\}$. Now let *type-closure*(A,$C \vdash$e:σ) denote the type expression $A(x_1) \rightarrow \ldots \rightarrow A(x_n) \rightarrow \sigma$. The relation $\hookrightarrow_S$ is defined as follows:

$$A,C \vdash \mathrm{e}:\sigma \hookrightarrow_S A',C' \vdash \mathrm{e}:\sigma'$$
$$\Longleftrightarrow \begin{cases} (1) & \beta \leq_S b_1 \text{ in } C \text{ and } \mathit{type\text{-}closure}(C, A \vdash \mathrm{e} : \sigma), \text{ and} \\ (2) & C' = [b_1/\beta]C. \\ (3) & A' = [b_1/\beta]A. \\ (4) & \sigma' = [b_1/\beta]\sigma. \end{cases}$$

□(Def.10.7)

Before modifying G-reduction and S-reduction to suit our needs, we will discuss the fundamental differences between Fuh and Mishras setting and our own.

10.2 Differences to Fuh and Mishra

There are three fundamental differences between Fuh and Mishras type inference and our binding time analysis:

- In binding time analysis, types appear in the program as annotations.
- In our binding time analysis, coercion of function or product types are not allowed.

- Fuh and Mishras reductions are performed on the whole program — we perform them on (`let`-bound) subexpressions.

Types appearing as annotations of operators in the program means that neither G- nor S-reduction can be performed freely. Take the constraints shown in figure 10.2 situation (c). Assume, that b_3 is dynamic (or will be made so later) and β_4 appears in the program as an annotation. Then G-reduction will lead to the substitution $[b_3/\beta_4]$, which is too conservative (makes too much dynamic) since it will lead to residualization of the expression annotated by β_4 even though this might have been evaluated by the specializer.

Disallowing coercion of functional and product types affects our version of G- and S-reduction in two ways. First, we have conditioned constraints. A constraint $(b \leq b' \mid t)$, where t is a functional or product type, leads per definition to unification of the involved types. But we cannot allow b or b' to be substituted for another variable β by G- or S-reduction, since β would then be "caught" by the condition. This of course also applies if t is a variable, which potentially might be instantiated to a functional or product type. *E.g.* if we have the constraint $(b \leq b' \mid \tau)$, then $\beta \leq_G b$ and β not in A or σ is not enough to allow G-reduction — we need also require $\beta \leq_G b'$.

Secondly, functions *expand-polarity* and *contract-polarity* implicitly assume a (contravariant) coercion composition rule for functions and a coercion composition rule for products. In our case, intuitively, functions and product need to have the "right" binding time when they are "built".

G- and S-reduction as presented by Fuh and Mishra are intended to be applied after analysis of the whole program. That the reductions are *not* context independent is a general problem with Fuh and Mishras work, and not only a problem to binding time analysis. This can be seen easily by means of an example:

Example 10.1
Consider the following program:

λx.`let` y = `if` false `then` $[\beta_x \rightsquigarrow \mathsf{D}]$x `else` d
    `in` x+2
@5

If we do G- and S-reduction after analyzing the `let`-bound expression e (= `if` false...), *type-closure*(A,C⊢e:(Int, D)) will be $(\text{Int}, \beta_x) \rightarrow (\text{Int}, \mathsf{D})$. We see that *expand-polarity*(β_x,*type-closure*(A,C⊢e:(Int, D))) = $\{\uparrow\}$. This means that the constraint $\beta_x \leq \mathsf{D}$ (corresponding to $[\beta_x \rightsquigarrow \mathsf{D}]$) can be reduced by a substitution $[\mathsf{D}/\beta_x]$. But this is clearly wrong since `x` and therefore also operator + will be made dynamic. We thus get a residual program `5+2` instead of `7`.

The problem is that `x` occurs in the context of the `let`-bound expression. □(Ex.10.1)

10.3 Modifications

This section will deal with the modifications necessary in order to make Fuh and Mishras reductions work in our system.

10.3.1 Reduction of Conditioned Constraints

Before going on to G- and S-reduction, we can eliminate every conditioned constraint ($\beta \leq \beta' \mid t$), where t has been (partially) instantiated. If t is a base type, it can be discarded, if it is a functional or product type, β and β' should be unified. This is captured in a what we call *T-reduction*:

Definition 10.8
The relation $\hookrightarrow_T$ is defined as follows:

$$A,C \vdash \mathrm{e}:\sigma \hookrightarrow_T A',C' \vdash \mathrm{e}':\sigma'$$
$$\iff \begin{cases} (1) & (b \leq b_1 \mid t) \in C \text{ and} \\ & t = (\kappa \to \kappa', b) \text{ or } t = (\kappa \times \kappa', b) \\ (2) & \zeta = \mathit{unify}(b, b_1) \\ (3) & C' = \zeta C. \\ (4) & A' = \zeta A. \\ (5) & \mathrm{e}' = \zeta \mathrm{e}. \\ (6) & \sigma' = \zeta\sigma. \\ \text{or} & \\ (1) & (b \leq b_1 \mid t) \text{ in } C \text{ and } (t = \mathrm{Int} \text{ or } t = \mathrm{Bool}) \\ (2) & C' = (C - \{b \leq b_1 \mid t\}) \bigcup \{b \leq b_1\}. \\ (3) & A' = A. \\ (4) & \mathrm{e}' = \mathrm{e}. \\ (5) & \sigma' = \sigma. \end{cases}$$

□(Def.10.8)

After T-reduction all conditioned constraints in C have the form ($\beta \leq \beta' \mid \tau$), where τ is a variable. We have to be careful about binding time variables occurring in conditioned constraints, since τ might be instantiated to a functional or product type at some later time.

10.3.2 Eliminating Cycles

Due to the `fix` construct, our constraint sets can contain cycles. G-reduction is capable of removing cycles among internal variables, but we wish to eliminate every kind of cycle (involving observable variables, variables occurring in conditioned constraints *etc.*). We call this reduction *C-reduction*:

Definition 10.9
The relation $\hookrightarrow_C$ is defined as follows:

$$A,C \vdash \mathrm{e}:\sigma \hookrightarrow_C A',C' \vdash \mathrm{e}:\sigma''$$
$$\iff \begin{cases} (1) & C = \{\beta_1 \leq \beta_2 \ldots \beta_{n-1} \leq \beta_n \beta_n \leq \beta_1\} \bigcup C' \\ & \text{where all constraints } b_i \leq b_j \text{ can be conditioned} \\ (2) & \zeta = [\beta/\beta_i], \text{ where } \beta \text{ is fresh} \\ (3) & C'' = \zeta C'. \\ (4) & A' = \zeta A. \\ (5) & \mathrm{e}' = \zeta \mathrm{e}. \\ (6) & \sigma' = \zeta\sigma. \end{cases}$$

□(Def.10.9)

10.3.3 Modified S- and G-reduction

We will take the three points mentioned in section 10.2 into account one by one.

Types appear as Annotations

The immediate solution to this problem is to define all variables occurring free in the program to be observable. This is somewhat conservative and will only lead to few reductions.

Now reconsider the various cases in figure 10.2. First look at cases (b) and (c) which reduces to 10.3(b) and 10.3(c). In case (b) assume that β_3 is an annotation in the program (but otherwise internal) so b_4 is substituted for β_3 — intuitively a (possibly) smaller value is substituted for a larger one, so the resulting program is at least "as static". In case (c), if β_4 is an annotation, the opposite situation occurs: a (possibly) larger value (b_3) is substituted for smaller one, resulting in a possibly "more dynamic" program. Case (a) as presented in figures 10.2 and 10.3 can as (c) result in a "more dynamic" program, but if b_3 is a variable β_3, substituting β_4 for β_3 (which is also allowed as in (b)) will result in a program as least "as static". We thus wish to allow G-reduction involving annotations in cases such as (b), but disallow it in cases such as (c) and (d).

For another, more precise, intuition, consider the constraints associated with the vertices in figures 10.2 and 10.3. Case (b) is shown in figure 10.4 and case (c) in figure 10.5. Here $c_{i,j}$ is used as a name for coercion $b_i \rightsquigarrow b_j$ (corresponding to constraint $b_i \leq b_j$).

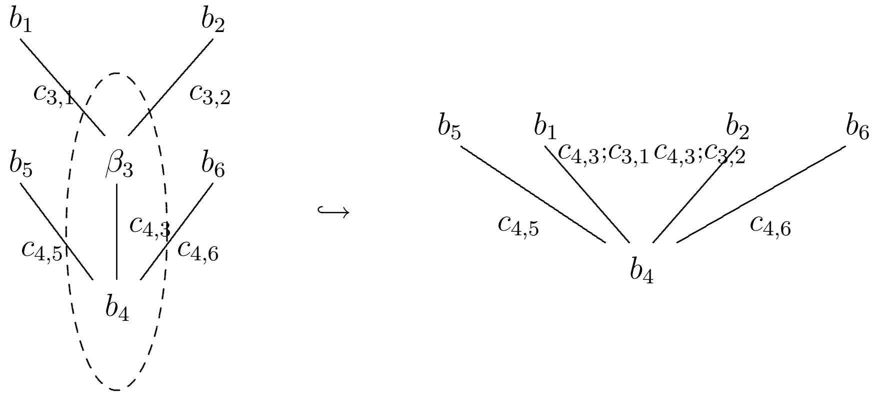

Figure 10.4: Applying $\hookrightarrow_G$, associated constraints shown

In figure 10.4 we see that coercion $b_4 \rightsquigarrow \beta_3$ is applied "later", justifying our intuition that the program is at least as static. Similar in figure 10.5 showing the (c) case of G-reduction, we see that coercion $\beta_4 \rightsquigarrow b_3$ is applied earlier.

Following the intuition above, a new version of G-reduction, taking into account that type variables may appear as annotations, is defined by adding an extra clause ($b \in$ *below*(β) or β not in e). Since G-reduction will be modified further below, the actual definition is deferred to definition 10.11.

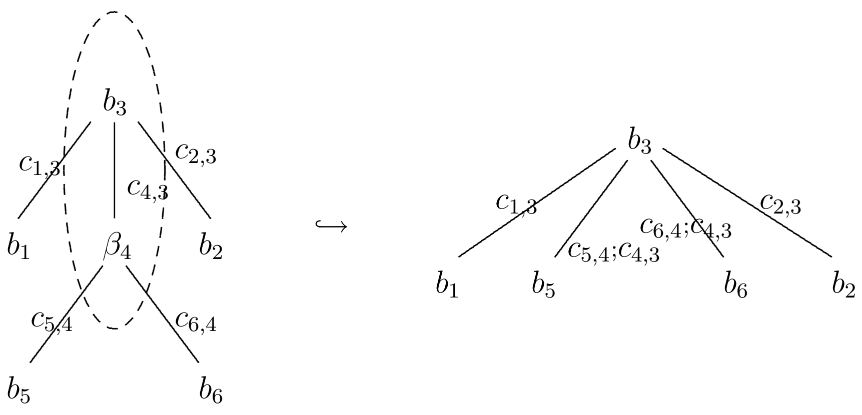

Figure 10.5: Applying $\hookrightarrow_G$, associated constraints shown

No Coercions on Functional or Product types

First we consider conditioned constraints. Having performed T-reductions, the only conditioned constraints left are conditioned by type variables.

Consider a set of constraints C containing the constraints $\beta \leq \beta'$ and $(\beta' \leq \beta'' \mid \tau)$. Now suppose that $\beta \leq_G \beta'$ as defined above. This is clearly not enough to allow G-reduction, in case τ is later instantiated to a functional or product type: in this case we should also require $\beta \leq_G \beta''$.

For β to G-subsume b, we require $\beta' \leq_G b'$ for all β', which because of conditioning might be unified with β, and all b' which by conditioning might be unified with b. Further we cannot allow β to (potentially) be unified with any constant S or D by conditioning.

Now consider a situation, where the set of constraints contain $(\beta \leq \beta' \mid \tau)$, and assume, that $\beta \leq_G \beta'$. Checking G-subsumedness in the above sense would also lead to checking $\beta' \leq_G \beta$. This is clearly not a necessary condition for doing G-reduction. The problem is that β and β' occur together in a conditioned constraint.

We therefore introduce a function *conditioned-with*(b,b'), calculating the values to which b is conditioned minus the values to which b is conditioned only via b'.

Definition 10.10
Define *conditioned-with*(b,b') to be the least set $\mathcal{S}$ satisfying the following equation:

$$\mathcal{S} = \{b\} \bigcup \{b'' \mid \beta \in \mathcal{S} \wedge ((b'' \leq \beta \mid \tau) \in C \vee (\beta \leq b'' \mid \tau) \in C) \wedge b' \neq b''\}$$

$\square$(Def.10.10)

In the graph representation of a constraint set C, the intuition is that *conditioned-with*(b,b') contains all binding time values connected to b via conditioned edges but not via b'. If b is not a variable then *conditioned-with*$(b, b') = \{b\}$.

Example 10.2
Let $C = \{(\beta_1 \leq \beta_2 \mid \tau_1)(\beta_2 \leq \beta_3 \mid \tau_2)(\beta_3 \leq \beta_4 \mid \tau_3)(\beta_3 \leq b \mid \tau_4)\}$. Then *conditioned-with*$(\beta_3, \beta_2) = \{\beta_3, \beta_4, b\}$. $\square$(Ex.10.2)

We are now able to give the new definition of G-reduction:

Definition 10.11
The relation $\hookrightarrow_G$ is defined as follows:

$$A{,}C{\vdash}\mathrm{e}{:}\sigma \hookrightarrow_G A{,}C'{\vdash}\mathrm{e}'{:}\sigma \iff$$
$$\forall b{\in}\textit{conditioned-with}(\beta{,}b''),\ b'{\in}\textit{conditioned-with}(b''{,}\beta){:}$$
$$\begin{cases} (1) & b \leq_G b' \text{ in } C \\ (2) & b \text{ is not in } A \text{ or } \sigma \\ (3) & b' \in \textit{below}(b) \text{ or } b \text{ not in e} \\ (4) & \mathrm{e}' = [b''/\beta]\mathrm{e}. \\ (5) & C' = [b''/\beta]C. \end{cases}$$

□(Def.10.11)

The definitions of G-subsume ($\leq_G$), *above* and *below* are not changed; they should treat conditioned and non-conditioned constraints equally. The first requirement $b \leq_G b'$ also implies that b is a variable. In other words *conditioned-with*(β,b'') is not allowed to contain any constants.

Now we turn to the other problem with not allowing coercions on function and product types. This concerns S-reduction, or rather the functions *extend-polarity* and *contract-polarity*.

The intuition behind S-reduction is, that any coercion, that may just as well be applied "outside" the expression, is redundant. *E.g.* the coercion in `let id =` λ`x.`$[\beta_x \rightsquigarrow \beta_{res}]$`x in` ... is redundant, since it may just as well be applied (if necessary) to the result of applying `id` in the body of the `let`. Similarly in `let silly =` λ`x.if` true `then` $[\beta_x \rightsquigarrow \mathsf{D}]$`x else d in` ..., where `d` is dynamic, the coercion is redundant, since it can be applied (if necessary) to arguments to function `silly`.

Now this works fine for Fuh and Mishra, but consider the following example (actually including `id` as above).

Example 10.3
Consider the following program, where a coercion has been removed from the body of function `id` as described above:

```
let id = λ^β1 x.x
in (if false
    then id
    else λ^β2 y.(y+1)+d)
   @5
```

The branches of the `if` are unified — the `then`-branch has type $(\beta_x \rightarrow \beta_x, \beta_1)$ while the `else`-branch has type $(\beta_y \rightarrow \mathsf{D}, \beta_2)$ — giving type $(\mathsf{D} \rightarrow \mathsf{D}, \beta_2)$. This means that `y` has suddenly become dynamic, and thus `(y+1)` is made residual.

With induced coercions, we would insert a coercion like $[(\beta_x \rightarrow \beta_x, \beta_1) \rightsquigarrow (\beta_x \rightarrow \mathsf{D}, \beta_1)]$ on `id`. Without induced coercions, we have to restrict the application of reductions on `id`.

The problem is that in Fuh and Mishras definition, β_{res} S-subsumes b_x in $\{\beta_x \leq \beta_{res}\}$ and $((\mathrm{Int}, \beta_x) \rightarrow (\mathrm{Int}, \beta_{res}), \beta_1)$. We cannot allow this since we want context independent reductions — unifying two variables occurring in the type can lead to undesired unification when the type is used in the context. □(Ex.10.3)

There are two problems to solve:

1. S-reduction cannot be performed on variables occurring in non-base argument or result types. This is because we do not allow induced coercions.

2. No two types appearing in the type may be unified. Solves the problem of example 10.3.

The first point is made clear by a new definition of *expand-polarity* and *contract-polarity* in definition 10.12. We introduce a new symbol ⤧, representing that an occurrence of a variable β is affected in a way disallowing any S-reduction.

Definition 10.12
expand-polarity is defined by:

$$\textit{expand-polarity}(\beta,\sigma) = \begin{cases} \{\uparrow\} & \text{,if } \sigma = (\text{Int},\beta) \vee \sigma = (\text{Bool},\beta) \\ \textit{expand-polarity}(\beta,\sigma') & \text{,if } \sigma = \forall(\beta_1 \ldots \beta_n)^C.\sigma' \\ \textit{contract-polarity}(\beta,(t_1,b_1)) & \\ \bigcup \textit{expand-polarity}(\beta,(t_2,b_2)) & \text{,if } \sigma = ((t_1,b_1) \to (t_2,b_2), b) \wedge \beta \neq b \\ \{\text{⤧}\} & \text{,if } \sigma = ((t_1,b_1) \to (t_2,b_2), \beta) \\ \{\text{⤧}\} & \text{,if } \sigma = ((t_1,b_1) \times (t_2,b_2), b) \\ & \quad \wedge\ \beta \text{ occurs in } (t_1,b_1) \text{ or } (t_2,b_2) \\ \{\text{⤧}\} & \text{,if } \sigma = ((t_1,b_1) \times (t_2,b_2), \beta) \\ \emptyset & \text{,otherwise} \end{cases}$$

and *contract-polarity* (which is no longer symmetric) by:

$$\textit{contract-polarity}(\beta,\sigma) = \begin{cases} \{\downarrow\} & \text{,if } \sigma = (\text{Int},\beta) \vee \sigma = (\text{Bool},\beta) \\ \{\text{⤧}\} & \text{,if } \sigma \neq (\text{Int},\beta) \wedge \sigma \neq (\text{Bool},\beta) \wedge \beta \text{ occurs in } \sigma \\ \emptyset & \text{,otherwise} \end{cases}$$

□(Def.10.12)

The second point consists simply of disallowing $\beta \leq_S \beta'$, if both β and β' occur in σ. The new definition of S-subsume becomes:

Definition 10.13
Let $\beta \in \textit{FreeBTVars}(C)$ and β occurs in σ. We say β is *S-subsumed* by b in C and σ written $\beta \leq_S b$ in C and σ, iff either

1. $C \vdash b' \leq \beta'$ and $\textit{expand-polarity}(\beta',\sigma) \subseteq \{\uparrow\}$ and $\textit{below}_C(\beta') - \{\beta'\} \subseteq \textit{below}_C(b')$, or

2. $C \vdash \beta' \leq b'$ and *expand-polarity*$(\beta', \sigma) \subseteq \{\downarrow\}$ and $above_C(\beta') - \{\beta'\} \subseteq above_C(b')$

and if b is a variable, then b does not occur in σ. $\square$(Def.10.13)

In the definition "$= \{\uparrow\}$" (*resp.* "$= \{\downarrow\}$") has been replaced by "$\subseteq \{\uparrow\}$" (*resp.* "$\subseteq \{\downarrow\}$"). This is done because to do S-reduction, we require $\leq_S$ to hold for all $\beta' \in$ *conditioned-with*(β, b). Note, that for β this makes no difference, since β occurs in σ implies, that *expand-polarity*$(\beta, \sigma) \neq \{\}$. If *expand-polarity*$(\beta', \sigma) = \{\}$ for some $\beta' \in$ *conditioned-with*(β, b), we just have a variant of G-reduction.

Context Dependency

The restrictions to S-reduction presented above can be seen as a reflection of the context dependency of Fuh and Mishras constraint reduction. Another form of context dependency was illustrated in example 10.1. The problem is that variables occurring in the environment can be referenced later.

We solve this simply by replacing *type-closure* by the type of the analyzed expression in the definition of G-reduction. Function *conditioned-with* is used in the same way as in the definition of $\hookrightarrow_G$ (definition 10.11).

Definition 10.14

Let $A, C \vdash \mathrm{e}:\sigma$ be a typing. The relation $\hookrightarrow_S$ is defined as follows:

$$A, C \vdash \mathrm{e}:\sigma \hookrightarrow_S A, C' \vdash \mathrm{e}':\sigma \Longleftrightarrow$$
$$\forall b \in \textit{conditioned-with}(\beta, b''),\ b' \in \textit{conditioned-with}(b'', \beta):$$
$$\begin{cases} (1) & b \leq_S b' \text{ in } C \text{ and } \sigma \\ (2) & b \text{ is not in } A \\ (3) & b' \in below(b) \text{ or } b \text{ not in e} \\ (4) & C' = [b''/\beta]C. \\ (5) & \mathrm{e}' = [b''/\beta]\mathrm{e}. \\ (6) & \sigma' = [b''/\beta]\sigma. \end{cases}$$

$\square$(Def.10.14)

This definition restricts $\hookrightarrow_S$ from doing any reductions on variables in A. Notice (as in the definition of $\hookrightarrow_G$), that the definition of $\leq_S$ ensures no binding time constants in *conditioned-with*(β, b'').

10.4 Quality of the Modified Reductions

As mentioned in the beginning of this chapter, we will not give any correctness proof for the reductions — we do however feel very confident of their correctness. We will briefly discuss the quality of the reductions developed in section 10.3.

First it is clear, that the occurrence of (non-instantiated) standard type variables prohibits some applications of G- and S-reduction, and leaves constraints $(b \leq b')$ which should have led to unification because b or b' occurs in a conditioned constraint.

The real problem implying the need for conditioned constraints is our (early) choice of prohibiting induced coercions. At this point we see that this — which was originally thought as a simplification — has led to several problems.

If we insist on only allowing coercions on base types, the above reduction rules can be simplified by doing standard type inference as a separate phase before binding time analysis (as suggested at the initial presentation). We have chosen not to do so, since in polymorphically typed languages this would not be possible (we would have constraints conditioned with polymorphic types).

In example 10.3, we saw that we cannot unify β_x and β_{res} in `let id =` λ`x.`$[\beta_x \rightsquigarrow \beta_{res}]$`x in` e. In cases where `id` is always immediately applied, such a unification is legal. Identifying such cases will, however, require a complicated analysis of e.

In many cases, however, we believe that our modified reductions removes most redundant parameters. For a justification of this claim, see chapter 12 and appendix D, showing annotated programs. For comparison, the number of binding time parameters with and without reductions is given here as well, showing significant improvement by the reductions in many cases. Also the number of variables in $\mathit{FreeBTVars}(\kappa)-\mathit{FreeBTVars}(A)$ is given to show a lower limit to the number of parameters.

It is not always possible to erach this lower limit, as can be seen by the following program:

$$\texttt{let}\ \mathrm{p} = \lambda \mathrm{x}.\lambda \mathrm{y}.\lambda \mathrm{z}.[b_x \rightsquigarrow b_{res}]\mathrm{x} +^{b_{res}} [b_{inter} \rightsquigarrow b_{res}]([b_y \rightsquigarrow b_{inter}]\mathrm{y} +^{b_{inter}} [b_z \rightsquigarrow b_{inter}]\mathrm{z})$$
$$\texttt{in}\ldots$$

where the binding time variable b_{inter} does not occur in $\mathit{FreeBTVars}(\kappa)$, but is necessary to get the full degree of polyvariance.

In appendix D, we see that the number of parameters is equal to or (in few cases) slightly greater than the number of variables in $\mathit{FreeBTVars}(\kappa)-\mathit{FreeBTVars}(A)$. From this we conclude that the reductions presented in this chapter indeed removes most superfluous parameters. In the next chapter, we will see that the reductions also speeds up the total binding time analysis significantly.

Chapter 11

Implementation

The monomorphically typed polyvariant binding time analysis has been implemented together with a non-selfapplicable specializer (it is written in Scheme). Together, they form a system we call PERPLEX (Partial Evaluation with Polyvariant Lets for an EXtended lambda calculus — the R has no meaning). Further an interpreter and a (very simple) post-reduction have been implemented. All programs can be found in appendix C: appendix C.1 contains the core binding time analysis together with unification procedures, and procedures for finding free variables, appendix C.2 contains procedures for constraint set reduction as presented in section 9.3 and chapter 10, appendix C.3 contains miscellaneous primitive functions (looking up in program syntax, types, making and applying substitutions *etc.*) and procedures for pretty-printing annotated programs as LaTeX documents, appendix C.4 contains the specializer, and appendix C.5 contains an interpreter for our language. The specializer trivially follows the definitions of figure 2.2 *resp.* figures 5.1, 5.2 and 8.2.

Since our specializer does not handle variable splitting (fully exploiting the partially static data structures), residual programs often contains segments of code first building and immediately destructuring datastructures. To make the residual programs more readable, we have a simple procedure for post-optimizing residual programs — it simply locally eliminates expressions where `fst` or `snd` are immediately applied to `pair`-expressions. This not semantically sound in general but can be used at wish. The procedure for this can be found in appendix C.6.

This chapter will not attempt to describe the implementation of the binding time analysis rigorously, since the programs largely follows the algorithms presented in chapters 9 and 10. We will only describe the major differences and discuss the efficiency of the implementation.

11.1 Implementation Issues

The implementation contains a few minor optimizations to the algorithm presented so far. *E.g.* analyzing coercions, we only produce conditioned constraints if the coerced type is a type variable. All optimizations however are local and should be understandable from the program text.

The implementation differs from the algorithm in a few other ways presented in this section.

11.1.1 Input

So far we have assumed that the input to a program is given as an environment. This implies that the user has to supply a full compound type (both the binding time and standard type) of the input. This is very inconvenient for the user, especially if the input is a program (which can have tremendous types).

Instead we let the user just supply an environment mapping the free variables (the input) to binding time values S and D. To obtain the initial environment, free variables τ_i are simply added to get compound types. If a datastructureis given binding time value S in the initial environment, this means completely static (the binding times of the subparts will be free). We disallow higher order input since it would require the user to know the internal representation of closures (PERPLEX has this restriction in common with other partial evaluators, *e.g.* SIMILIX [Bondorf 1991]).

11.1.2 Adding List-like Structures

In some of the examples we wanted to run, we needed list-like structures. In the implementation of the binding time analysis and the specializer, we therefore include the constant `nil` and the predicate `null?` (which cannot be handled as a primitive operator, since we need restrictions on the type like in `fst` and `snd`).

$$\text{(nil)} \quad \frac{C \vdash b \leq b' \qquad C \vdash b \leq b''}{A \vdash \texttt{nil}{:}((t',b') \times (t'',b''),b)}$$

$$\text{(null)} \quad \frac{A,C \vdash \text{e: } ((t',b') \times (t'',b''),b) \qquad C \vdash b \leq b' \qquad C \vdash b \leq b''}{A,C \vdash \texttt{null?}^b \text{ e : (Bool},b)}$$

Figure 11.1: Extensions to the Type System

```
type(e,$A$) = (($t$,$b$), $\zeta$, $C$) where
(($t$,$b$), $\zeta$, $C$) =
case e of
  nil: $(((t_1,\beta_1) \times (t_2,\beta_2),\beta_3),\ \zeta_{\text{id}},\ \{\beta_3 \leq \beta_1, \beta_3 \leq \beta_2\})$, where $t_i$, $\beta_i$ are new

  null?$^b$e$_1$:
      let $((t_1,b_1),\zeta_1,C_1)$ = type(e$_1$,$A$)
          $(\tau_2,b_2)$ and $(\tau_3,b_3)$ be fresh
          $\zeta_2 = \mathit{unify}(((\tau_2,b_2) \times (\tau_3,b_3),b),(t_1,b_1))$
      in $((\text{bool},\zeta_2 b),\ \zeta_2\zeta_1,\ \zeta_2(\{b \leq \beta_2, b \leq \beta_3\} \bigcup C_1))$
```

Figure 11.2: Extensions to Algorithm `type`

The extensions to the type system are given in figure 11.1 and the extensions to algorithm `type` in figure 11.2.

$$\text{(nil)} \quad \rho \vdash \texttt{nil} \longrightarrow_{std} \texttt{nil}$$

$$\text{(null)} \quad \frac{\rho \vdash \mathrm{e} \longrightarrow_{std} \texttt{nil}}{\rho \vdash \texttt{null?}\ \mathrm{e} \longrightarrow_{std} \texttt{true}}$$

$$\text{(null)} \quad \frac{\rho \vdash \mathrm{e} \longrightarrow_{std} \texttt{pair}(v,v')}{\rho \vdash \texttt{null?}\ \mathrm{e} \longrightarrow_{std} \texttt{false}}$$

Figure 11.3: Extensions to the Standard Semantics

$$\text{(nil)} \quad \rho \vdash \texttt{nil} \longrightarrow_{spec} \texttt{nil}$$

$$\text{(null)} \quad \frac{\rho \vdash \mathrm{e} \longrightarrow_{spec} \texttt{nil}}{\rho \vdash \texttt{null?}^{\mathsf{S}}\ \mathrm{e} \longrightarrow_{spec} \texttt{true}}$$

$$\text{(null)} \quad \frac{\rho \vdash \mathrm{e} \longrightarrow_{spec} \texttt{pair}(v,v')}{\rho \vdash \texttt{null?}^{\mathsf{S}}\ \mathrm{e} \longrightarrow_{spec} \texttt{false}}$$

$$(\underline{\text{n}}\text{ull}) \quad \frac{\rho \vdash \mathrm{e} \longrightarrow_{spec} \underline{\mathrm{e}}}{\rho \vdash \texttt{null?}^{\mathsf{D}}\ \mathrm{e} \longrightarrow_{spec} \underline{\texttt{null?}}\ \underline{\mathrm{e}}}$$

Figure 11.4: Extensions to the Specializer

The interpretation of these new constructors should be obvious. The standard semantics are given in figure 11.3, and the specialization rules are given in figure 11.4.

This way of introducing lists is inspired by Lisp-like languages, but in our typed language introducing "proper" lists with special list-types would have been more elegant though a little more complicated. It should thus be seen as no more than a convenient ad hoc extension.

11.1.3 Recursive Types

In the presentation of algorithm `type` in chapter 9, we did not give any algorithm for unification. Unification algorithms are well known and we see no reason to discuss the basic algorithm.

There is however one thing we have to consider: due to the `fix`-point operator, we will often unify a variable τ with a type t in which τ occurs. This of course has to do with the standard type system, but it nevertheless have an important impact on the binding time polyvariancy.

We have chosen the following solution: unifying τ with t where τ occurs in t, gives the substitution $\tau \mapsto \mu\tau'.[\tau'/\tau]t$, where τ' is a fresh variable. This can be seen as introducing recursive typing through the back door, but it should be seen as merely an implementation technique.

Note that types such as $\mu\tau.((\mathrm{Int},\beta) \times (\tau, \mathsf{S}))$ and $(\mathrm{Int},\beta') \times \mu\tau.((\mathrm{Int},\beta) \times (\tau, \mathsf{S}))$ both defining lists of integers are not equal *w.r.t.* binding times. The second is a "finer" type, and therefore potentially able to express more staticness. This is not necessarily desirable, since it might lead to extensive copying at specialization time, some of the copies being "uninteresting".

11.2 Implementation *vs.* Type System

The implementation was developed in parallel with the more theoretical aspects of this work. This has had the unfortunate effect that there is a small disagreement between the implementation and the type system. While we use the constructor `fix` in the type system, algorithms *etc.* with two variable names — one being the name of the recursive function, and the other its argument — the `fix`-constructor in the implementation is used with only one variable name — the name of the recursive function.

Changing the implementation to fit the system is easy, but we have found it unimportant.

11.3 Runtimes

The implementation of our polyvariant binding time analysis is not very fast. It is however capable of handling reasonable sized programs. To illustrate the speed, we have chosen two programs: the first "facs" computes the list $(n!, (n-1)!, \ldots, 1!)$. It contains one `let`-bound function containing 3 nested `let`-bound functions. The program and its annotation can be seen in appendix D.1.

The second example is an interpreter for a language MP (the interpreter has been used earlier as an example for other partial evaluators). The interpreter contains a total of 52 `let`-bound functions and is 324 lines long. The original and annotated program can be found in appendix D.2.

The cpu time needed to binding time analyze these programs can be seen in the first column of figure 11.5. The time in parenthesis is the time spent garbage collecting (included in the total time). All programs are run on a Sparc ELC using Chez Scheme 3.2.

	Total	Substitution	Reduction	$\hookrightarrow_T$	$\hookrightarrow_C$	$\hookrightarrow_G$	$\hookrightarrow_S$
Facs	5.38s(0.26s)	4.4s[†]	1.52s	0s	0.07s	0.67s	0.08s
MP-interpreter	1861s(192s)	1427s[*]	442s	0.33s	16.9s	160s	5.6s
[†] plus 1081 substitution taking <1ms							
[*] plus 24157 substitution taking <1ms							

Figure 11.5: Speed of BTA

In figure 11.5, we have factorized the total run time into the times spent performing subtasks. The complexity of the reductions presented in chapter 10, has lead us to believe, that most of the time of a binding time analysis was spent performing reductions. This is not true as the figures show; only 28% *resp.* 24% is spent reducing. Of the time spent reducing 0.51s out of 1.52s *resp.* 202s out of 442s was spent substituting.

Notice, that the total time spent reducing is much longer than the sum of the times spent on the individual reductions. The extra time is spent on applying substitutions and identifying free variables.

To speed up binding time analysis it is thus crucial to speed up the application of substitutions. This can be achieved by replacing our functional (copying) substitution application, with a destructive variant. Not only the time spent substituting, but also the time spent garbage collecting will be reduced.

Still more than 7 minutes (4 minutes without substitutions) is spent doing reductions for a reasonable sized but not large MP-interpreter. We believe that this figure can be reduced since no great attention has been paid to speed during implementation (we do believe to have avoided the worst pitfalls of repeated calculations), and no graph algorithms — which are well suited for such problems — have been used.

11.4 How to Speed Up the Analysis

We have experimented with improving the implementation (within the framework of algorithm $\mathcal{W}$), and achieved some speedup (see figure 11.6). We will not go into details concerning the changes, just mention the chief ones:

- Function α is modified, such that coercions are only inserted where they may be needed (*e.g.* knowing that b and b' in $[b\leadsto b']\lambda\text{x.e}$ will always be unified, there is no need to insert coercions on lambda's in the first place). Also, if an annotation and the "from" binding time in a coercion will always be unified, the same variable is used (*e.g.* $\alpha(\text{e } \texttt{op} \text{ e}') = [\beta\leadsto\beta']\alpha(\text{e})\texttt{op}^{\beta}\alpha(\text{e}')$).
- Unification of annotations with another binding time value is replaced by destructive updating of the annotation.

The main purpose was to reduce the size of the substitutions carried around (and thereby the speed of application of substitutions).

	Total	Substitution	Reduction	$\hookrightarrow_T$	$\hookrightarrow_C$	$\hookrightarrow_G$	$\hookrightarrow_S$
Facs	3.74s(0.18s)	3.18s†	1.32s	0s	0.26s	0.48s	0.06s
MP-interpreter	1550s(134s)	1185s*	305s	1.42s	12.2s	101s	4.29s
† plus 756 substitution taking <1ms							
* plus 22127 substitution taking <1ms							

Figure 11.6: Speed of BTA after improvements

An interesting perspective of this test is that the MP-interpreter (which has virtually remained unchanged since the early Mix-days) does not exploit the polyvariancy. Still a large effort is put into making each function polymorphic. It requires a global analysis, to detect when polyvariance is superfluous, and since it depends on the binding analysis itself, the "best" result can only be obtained by redoing the analysis. Still a simple heuristic analysis might be worth while considering to speed up the analysis.

In figure 11.6 we see that a notable speedup is obtained by the improvements but we are still far from reasonable run times. We believe that the slowness is due to the choice of algorithm $\mathcal{W}$ for implementing the analysis.

By choosing another algorithm than $\mathcal{W}$ we believe, that the analysis could be made to run efficiently. A constraint solving algorithm could be a choice, which we believe would be well suited since our constraints have a very simple structure. Another choice would be a semi-unification based algorithm.

We feel a further discussion is beyond the scope of this report, since our prime objective with the implementation was to show that our binding time analysis works and works well.

11.5 Effect of Reductions

To see what we gain by performing the reductions of chapter 10, we have tested a version of the binding time analysis identical to the one of section 11.4 but without reductions[1]. Figure 11.7 shows the run times for binding time analysis.

	Runtime	Runtime without reductions
Facs	3.74s(0.18s)	6.81s(0.75s)
MP-interpreter	1550s(134s)	8123s(1740s)

Figure 11.7: Speed of BTA without Reductions

We note, that doing the reductions speeds up binding time analysis from 6.06s to 3.56s *resp.* from 6383s to 1316s (which was what we hoped — the time spent reducing is more than saved by the fewer binding time parameters). Another good reason for doing reductions can be found in appendix D.2.2, where we show the decrease in number of binding

[1]C-reduction (eliminating cycles) is actually performed on the final set of constraints C, but this is only to ensure termination of the procedure computing the reflexive, transitive closure of C

time parameters — in the main functions `evalbcc` *resp.* `run` in the MP-interpreter the number of binding time parameters is reduced from 323 to 12 *resp.* 386 to 16. The power of the reductions is most evident when the number of parameters has “accumulated”.

Chapter 12

Examples

In this chapter some examples of analyzing programs will be given. The programs are direct output from the PERPLEX binding time analysis (the binding time analysis has an option for generating output in $\LaTeX$ format). In all examples d is a dynamic variable.

More complex examples can be found in appendix D, where we present the annotated "list of factorials" program and the MP-interpreter, both used for measuring run-times in chapter 11.

12.1 The Identity Function

To illustrate the notation, we begin with one of the simplest imaginable programs — the identity function applied to a static and a dynamic integer (5 *resp.* d):

```
let id = $\lambda$x.x
in pair(id@5,id@d)
```

Using the improved implementation discussed in section 11.4, analysis takes 0.11s (without garbage collection). The reduction presented in chapter 10 has no effect on the number of binding time parameters. The result of binding time analyzing this program is:

```
let id = $\Lambda\Big( \beta_4\beta_6\beta_7 \Big)$.
      $\lambda^{\beta_7}$x. [$\beta_6 \rightsquigarrow \beta_4$] x$^{\beta_6}$
in pair$^{\mathsf{D}}$([$\mathsf{S} \rightsquigarrow \mathsf{D}$] ((id$^{\mathsf{S}}\diamond\Big($ SSS $\Big)$)
                          @$^{\mathsf{S}}$5$^{\mathsf{S}}$)
              ,((id$^{\mathsf{S}}\diamond\Big($ DDS $\Big)$)
                 @$^{\mathsf{S}}$d$^{\mathsf{D}}$))
```

We see that id has three binding time parameters; one for annotating the λ, one for the argument x and one for the result. Another way of seeing that they are all needed is that the type of id ((Int,β_4)$\rightarrow$(Int,β_6),β_7) contains all three variables.

The result of specializing the annotated program is the expected:

```
pair(5,d)
```

12.2 The Factorial Function

The factorial function is a nice example of a simple recursive function. To avoid infinite specialization, the program contains the definition of `generalize`. The purpose of this function is to make a λ or a `fix` appear in a dynamic context (thus raising the λ *resp.* `fix`) — it is thus purely a (well known) binding time improvement needed here because the specializer is not memoizing (see section 13.3.1 for a further discussion of this). The definition of `generalize` is a bit more complicated than usual in order to make it type-check.

```
let fac = fix f.$\lambda$n.if (n=0)
                 then  1
                 else  n*(f@(n-1))
in let generalize = $\lambda$e.if true
                       then e
                       else if d=1
                            then $\lambda$x.x
                            else $\lambda$x.x
in pair(fac@5,(generalize@fac)@d)
```

Using the improved implementation the analysis takes 0.61s (without garbage collection). The result is

```
let fac = $\Lambda\left(\ \beta_6\beta_{34}\beta_{43}\ \right)$.
    fix$^{\beta_{43}}$f.
     $\lambda^{\beta_{43}}$n. if$^{\beta_{34}}$ (n$^{\beta_{34}}$
                  $=^{\beta_{34}}$[S$\rightsquigarrow\beta_{34}$] 0$^{\mathsf{S}}$)
            then  [S$\rightsquigarrow\beta_6$] 1$^{\mathsf{S}}$
            else  ([$\beta_{34}\rightsquigarrow\beta_6$] n$^{\beta_{34}}$
                   $*^{\beta_6}$(f$^{\beta_{43}}$
                      @$^{\beta_{43}}$(n$^{\beta_{34}}$
                           $-^{\beta_{34}}$[S$\rightsquigarrow\beta_{34}$] 1$^{\mathsf{S}}$)))
in let generalize = $\Lambda\left(\ \beta_{80}\ \right)$.
       $\lambda^{\beta_{80}}$e. if$^{\mathsf{S}}$ true$^{\mathsf{S}}$
             then  e$^{\mathsf{D}}$
             else  if$^{\mathsf{D}}$ (d$^{\mathsf{D}}$
                        $=^{\mathsf{D}}$[S$\rightsquigarrow$D] 1$^{\mathsf{S}}$)
                   then  $\lambda^{\mathsf{D}}$x. x$^{\mathsf{D}}$
                   else  $\lambda^{\mathsf{D}}$x. x$^{\mathsf{D}}$
in pair$^{\mathsf{D}}$([S$\rightsquigarrow$D] ((fac$^{\mathsf{S}}\diamond\left(\ \mathsf{SSS}\ \right)$)
                 @$^{\mathsf{S}}$5$^{\mathsf{S}}$)
         ,(((generalize$^{\mathsf{S}}\diamond\left(\ \mathsf{S}\ \right)$)
           @$^{\mathsf{S}}$(fac$^{\mathsf{S}}\diamond\left(\ \mathsf{DDD}\ \right)$))
          @$^{\mathsf{D}}$d$^{\mathsf{D}}$))
```

By specialization we obtain the following program:

```
pair(120,
     fix $f_1$.$\lambda n_2$.
         if $n_2 = 0$ then 1 else $n_2*(f_1@(n_2-1))$)
```

Notice that `generalize` needs only one binding time parameter: one for the head λ. The factorial function needs the three appearing in the type — just as the identity. The reductions presented in chapter 10 reduces the number of binding time parameters in fac from 7 to 3, and in generalize from 6 to 1.

12.3 The Map Function

With the map function, we show both recursivity and use of `pair`. We apply the map function to a static list of static integers, to a static list of dynamic elements and to a dynamic list. To make the last application terminate, we use function generalize in a way similar to above. Variables d1,d2 and d are dynamic

```
let map = λf.fix m.λl.if null? l
                      then  nil
                      else  pair(f@(fst l),
                                 m@(snd l))
in let generalize = λe.if true
                       then e
                       else if d=1
                            then λx.nil
                            else λx.nil
in pair((map@λx.x)@pair(1,pair(2,pair(3,pair(4,nil)))),
        pair((map@λx.x)@pair(d1,pair(d1,pair(2,pair(d1,nil)))),
             (generalize@(map@λx.x))@d2))
```

Using the improved implementation the analysis takes 2.27s (without garbage collection) and gives us the following annotated program:

```
let map = $\Lambda\begin{pmatrix} \beta_{21}\beta_{46}\beta_{31}\beta_{42}\beta_{48}\beta_{32}\beta_{26}\beta_{25} \\ \beta_{49} \end{pmatrix}$.
    $\lambda^{\beta_{49}}$f. fix$^{\beta_{48}}$m.
          $\lambda^{\beta_{48}}$l. if$^{\beta_{42}}$ null?$^{\beta_{42}}$(l$^{\beta_{42}}$)
               then  nil
               else  pair$^{\beta_{46}}$([$\beta_{32}\rightsquigarrow\beta_{21}$] (f$^{\beta_{25}}$
                                         @$^{\beta_{25}}$[$\beta_{31}\rightsquigarrow\beta_{26}$] fst$^{\beta_{42}}$(l$^{\beta_{42}}$))
                              ,(m$^{\beta_{48}}$
                                @$^{\beta_{48}}$snd$^{\beta_{42}}$(l$^{\beta_{42}}$)))
in let generalize = $\Lambda(\ \beta_{86}\ )$.
     $\lambda^{\beta_{86}}$e. if$^{S}$ true$^{S}$
          then  e$^{D}$
          else  if$^{D}$ (d1$^{D}$
                        =$^{D}$[S$\rightsquigarrow$D] 1$^{S}$)
                then  $\lambda^{D}$x. nil
```

```
                     else  λ^D x. nil
in pair^D((((map^S◇( DDSSSSSS ))
                   ( S        )
            @^S λ^S x. x^S)
           @^S pair^S(1^S
                      ,pair^S(2^S
                              ,pair^S(3^S
                                      ,pair^S(4^S
                                              ,nil)))))
          ,pair^D((((map^S◇( DDDSSDDS ))
                           ( S        )
                   @^S λ^S x. x^D)
                  @^S pair^S(d1^D
                             ,pair^S(d1^D
                                     ,pair^S([S⇝D] 2^S
                                             ,pair^S(d1^D
                                                     ,nil)))))
                 ,(((generalize^S◇( S ))
                   @^S((map^S◇( DDDDDDDS ))
                              ( S        )
                      @^S λ^S x. x^D))
                  @^D d2^D)))
```

As expected, we get more binding time parameters to the `map`-function than to the functions seen above. This is due to the more complicated type of map, and the more binding time values this involves. The reduction presented in chapter 10 only reduces the number of parameters in map from 11 to 9, while the number of parameters to generalize is reduced from 8 to 1. Specialization gives the following program:

```
pair(pair(1,pair(2,pair(3,pair(4,nil)))),
     pair(pair(d1,pair(d1,pair(2,pair(d1,nil)))),
          fix m_3.λ l_4.if (null? l_4)
                      nil
                      pair(fst l_4,
                           m_3@(snd l_4))
          @d2))
```

Chapter 13

Future Work

Though we believe to have been around many aspect of the subject, a lot of questions are left unanswered. This chapter attempts to discuss some of the many possible future directions of work implied by the present work.

13.1 Extensions of the System

The system proposed in this thesis can be extended to a more powerful system. Some directions are discussed.

13.1.1 Stronger Lifting

In [Andersen & Mossin 1990] it is noted that the usual contravariant rule (of figure 1.6) for lifting function also holds with binding time values as non standard types. The rule is however not made part of the type system. Similarly a rule for lifting pairs, can be added to the system.

Figure 13.1 shows the new rules concerning coercions. Rule (coerce) has to be extended to handle compound types instead of merely binding time types. Rules (→–composition) and (×–composition) are the new rules allowing lifts on products and functions. Rule (composition) allows sequential composition of coercions. This rule could have been added earlier without problems, but it only gives additional power together with the (→–composition) and (×–composition) rules.

It remains to be seen what additional strength the added rules will give to a binding time analysis and how it will affect the complexity of the analysis. It should however be clear, that the system in some sense will be more "natural" and that we would get rid of oddities like conditioned constraints.

One can imagine adding rules for lifting the "head" binding time of products and functions. The rules should be:

$$(\text{→-lift}))\quad C\vdash((t,\mathsf{D}) \to (t',\mathsf{D}),\mathsf{S}) \leq ((t,\mathsf{D}) \to (t',\mathsf{D}),\mathsf{D})$$

$$(\text{→-lift}))\quad C\vdash((t,\mathsf{D}) \times (t',\mathsf{D}),\mathsf{S}) \leq ((t,\mathsf{D}) \times (t',\mathsf{D}),\mathsf{D})$$

Allowing this requires the specializer to be able to create code from closures and internal representation of pairs.

$$\text{(coerce)} \quad \frac{A,C \vdash \mathrm{e}:\kappa \quad C \vdash \kappa \leq \kappa'}{A,C \vdash [\kappa \rightsquigarrow \kappa']\mathrm{e}:\kappa'}$$

$$\text{(lookup-coerce)} \quad C \bigcup \{\kappa \leq \kappa'\} \vdash \kappa \leq \kappa'$$

$$\text{(lift)} \quad C \vdash (\mathrm{Bool}, \mathsf{S}) \leq (\mathrm{Bool}, \mathsf{D})$$

$$\text{(lift)} \quad C \vdash (\mathrm{Int}, \mathsf{S}) \leq (\mathrm{Int}, \mathsf{D})$$

$$\text{(id-stat)} \quad C \vdash (t, \mathsf{S}) \leq (t, \mathsf{S})$$

$$\text{(id-dyn)} \quad C \vdash (t, \mathsf{D}) \leq (t, \mathsf{D})$$

$$\text{(composition)} \quad \frac{C \vdash \kappa \leq \kappa' \quad C \vdash \kappa' \leq \kappa''}{C \vdash \kappa \leq \kappa''}$$

$$\text{($\rightarrow$–composition)} \quad \frac{C \vdash {\kappa_1}' \leq \kappa_1 \quad C \vdash \kappa_2 \leq {\kappa_2}'}{C \vdash (\kappa_1 \rightarrow \kappa_2, \mathsf{S}) \leq ({\kappa_1}' \rightarrow {\kappa_2}', \mathsf{S})}$$

$$\text{($\times$–composition)} \quad \frac{C \vdash \kappa_1 \leq {\kappa_1}' \quad C \vdash \kappa_2 \leq {\kappa_2}'}{C \vdash (\kappa_1 \times \kappa_2, \mathsf{S}) \leq ({\kappa_1}' \times {\kappa_2}', \mathsf{S})}$$

Figure 13.1: Coercion Composition

13.1.2 Polymorphic Fixpoint Operator

The system presented is only polyvariant in `lets`, not in `fix`. In other words all recursive calls are bound to have the same binding time description. A `fix`–polymorphic type inference for languages like ML is known to be undecidable. Adding `fix`–polymorphic binding times, however, need not be. Actually we believe, that it might be added without too much difficulty. The intuitive reason for this is that binding time values always "stick" to a standard type, so the "degree" of binding time polymorphism is "bounded" by the standard types, making things decidable.

Adding recursive polymorphism in binding times, woud allow more evaluation at specialization time in functions such as Ackerman's function.

13.2 Proofs

We have proved, that our binding time analysis is correct in the sense that together with the presented specializer the mix-equation holds and "well annotated does not go wrong". Such a proof could however easily be conducted for a binding time analysis annotating everything as being dynamic.

The reductions presented in chapter 10, even though based on work by Fuh and Mishra, was presented in an "ad hoc" way.

13.2.1 Improving the proof of correctness

In chapter 6, we proved that certain properties — which we coined correctness — held for our binding time analysis. It is not clear, that these properties actually capture our intuition of correctness (though we hope, we have convinced the reader of this).

If we accept this, one might argue that any proof of these properties is equally good — only the existence of the proof is interesting. We do not agree with this standpoint. We often found ourselves struggling with complications, that we found did *not* arise from the properties we were trying to prove, but rather from the formalism in which they were stated.

Another observation was that large portions of the proof was really trivial — though tedious — and did not directly depend on the analysis, but rather on some connection between standard type system and binding time analysis.

We therefore believe that more work has to be put into the theory of correctness of inference based program analyses (in particular, but not restricted to, binding time analysis). One interesting aspect is the influence on the proof structure, the choice of semantic formulation has. There are both differences and similarities between our proof (using a big-step operational semantic model) and Gomard's (using a denotational model). We believe that some of the technical problems encountered concerning "moving things in/out of environments" might be avoided using a small-step operational semantic model, but it is not clear if this might introduce other problems.

13.2.2 Quality of Annotations

What is a good binding time analysis? How is the quality of binding time analysis measured? These are questions of current research and we will not attempt to dig deeper into them. We will just emphasize that some notion of quality is indeed needed.

One such very important aspect of our binding time analysis has not been proved: how does our binding time polymorphic version of polyvariant binding time analysis relate to other polyvariant analyses. First we should prove a kind of subject expansion theorem for `let`-reduction. This would state, that polymorphism is as good as actually copying `let`-bound expressions to every occurrence of the `let`-bound variable.

Further we would like our analysis to be superior to other polyvariant binding time analyses. We believe, that this is the case in the following sense: other attempts at polyvariant binding time analysis copies a function (or rather the binding time description of the function) in as many versions as needed — the number of versions can be exponential in the size of the program. Instead we add a number of extra (binding time) parameters — this number is potentially exponential as well. Even so, we believe that the exponential worst-case will appear more rarely with our analysis than with the standard approach. Further we do not know to what extent the reductions of chapter 10 decrease the number of parameters and whether it even results in a better worst-case behavior. Practical experiments have shown significant improvements, *e.g.* for the main functions `evalbcc` *resp.* `run` in the MP-interpreter the number of binding time parameters is reduced from 323 to 12 *resp.* 386 to 16 (see appendix D.2.2).

A formal investigation of these matters would be very important and could be of importance to other analyses than our binding time analysis.

13.2.3 Correctness and Quality of Reductions

The reductions $\hookrightarrow_T$, $\hookrightarrow_C$, $\hookrightarrow_G$ and $\hookrightarrow_S$ as presented in section 10.3 were developed on the basis of an investigation of the differences between Fuh and Mishras type inference and ours. Fuh and Mishra prove their reductions correct and claim that they capture almost all redundancy in constraint sets. We have got no similar correctness results, but we see that a formal investigation of both correctness and quality of the proposed reductions would be of great value. We do, however, believe that the proposed reductions are “good under the circumstances” (conditioned constraints *etc.*). This belief was supported by our practical experiments.

13.2.4 “Lub”-types

By introducing abstraction over binding time values, more work has to be performed by the specializer. This we believe is no problem and we might even consider imposing more work on the specializer by introducing *least upper bound* in the type system. This would be done by having binding time values:

$$b ::= \mathsf{S} \mid \mathsf{D} \mid \beta \mid b \sqcup b$$

We then introduce coercion rules such as

$$\begin{array}{ll} \text{(lift-lub)} & C \vdash b \leq b \sqcup b' \\ \text{(lift-lub)} & C \vdash b \leq b' \sqcup b \end{array}$$

The idea of introducing least upper bound in the types, would be to use them in the annotations. *E.g.* we would have the typing:

$$\texttt{let}\ \mathrm{p} = \Lambda(\beta_{\lambda_1}, \beta_{\lambda_2}, \beta_x, \beta_y)^C \lambda^{\beta_{\lambda_1}}\mathrm{x}.\lambda^{\beta_{\lambda_2}}\mathrm{y}.[\beta_x \rightsquigarrow \beta_x \sqcup \beta_y]\mathrm{x} +^{\beta_x \sqcup \beta_y} [\beta_y \rightsquigarrow \beta_x \sqcup \beta_y]\mathrm{y}$$
$$\texttt{in} \ldots$$

where $C = \{\beta_{\lambda_1} \leq \beta_x, \beta_{\lambda_1} \leq \beta_{\lambda_2}, \beta_{\lambda_2} \leq \beta_y, \beta_{\lambda_2} \leq \beta_x \sqcup \beta_y\}$.

We then only need to abstract over the binding time variables occurring in the type of the `let`-bound expression (minus the ones occurring in the environment). This will correspond to the usual definition of generification.

13.3 The Specializer

The purpose of the specializer developed in this work was only to cast light on the binding time analysis (*e.g.* in the proof of correctness). The implemented specializer served as test-bed for developing the binding time analysis implementation. Thus little attention was given to the specializer and it is therefore clear that for a practically useful system the specializer should be improved.

13.3.1 Memoization and Variable Splitting

The specializer presented in chapter 5 treated `lets` by always unfolding them. To create a practical system, we should introduce *specialization points* and make the specializer *memoizing*. The lack of memoization results in code duplication and non-terminating specialization.

A specialization point gives rise to a residual function in the specialized program (in our case that would be a `let`-bound function). The usual way of inserting specialization points is on dynamic conditionals and dynamic lambda's, and in our case we would also include dynamic `fix`-point operators.

To take full advantage of partially static data structures, the specializer should also do *variable splitting*. This technique splits a partially static parameter into new parameters, all being fully static or fully dynamic. The result of not having variable splitting was seen very clearly when specializing the MP-interpreter: in the residual program, the store, which is a partially static data structure, was constantly being constructed and destructed.

13.3.2 Self-application

Since binding time analysis and not specialization was our prime goal, and because our language is not very easy to program with, we chose to implement the specializer in Scheme. This, of course, makes *self-application* impossible. Self-application is essential in a partial evaluator if we want to use it to generate compilers (and other compiler-like program generators).

Two things kept us from writing the specializer in its own language. First the language is not very user friendly; secondly, it is strongly typed. The second problem is discussed in [Launchbury 1990]. We will not go into details here, but basically the coding problem is a problem of representing programs as data.

It would be interesting to see how our polymorphic scheme of polyvariance will manifest itself in *cogen* (= *mix*(*mix*,*mix*)). We expect *cogen* to contain code for duplicating functions in the necessary number of variants.

13.4 Extending the System to a Real Language

We would like to extend our system to a "real" partial evaluator but this requires some work.

13.4.1 Extending the Language

Extending the language should be quite simple since our language includes most central features. Global definitions can be modeled by a combination of `let` and `fix`. Since our binding time analysis assigns binding times in a local way, the analysis will be well suited for a language with modules. Thus after a change in one module, only this module (and its parents) need to analyzed again.

13.4.2 Speeding up the Analysis

To make our analysis part of a real system, it need to be speeded up. We have already discussed this in section 11.3 where we showed that most of the time is spent applying substitutions. We believe that algorithm $\mathcal{W}$ is inherently a slow algorithm, so no decisive conclusions can be drawn from the slowness of our implementation.

Many fast methods are known from standard type inference, which could be applied to our binding time analysis. These include constraint solving methods and semi-unification methods. We believe that with one of these methods, an algorithm could be obtained superior in speed to existing polyvariant binding time analyses.

13.4.3 Polymorphism When Needed

The MP-interpreter (see appendix D.2) was written in a monovariant style. Even so, many function came out with many binding time parameters (which is not surprising). This is not desirable, since it gives an increase in program size and thereby a slow-down in specialization time. It might thus be worth while considering to either do a simple pre-analysis determining which functions will definitely not be used polyvariantly, or to do a post-reduction (before specialization) unfolding unnecessary binding time abstractions. This, however, would imply that we lose the ability to do modular binding time analysis.

13.4.4 Presenting Annotated Programs to the User

Since partial evaluation does not always yield the desired results, it is often necessary for the user to look at binding time annotated programs. This can be difficult even for monomorphic binding time analysis, but inspecting the annotated programs in appendix D is almost impossible (even though the reductions of chapter 10 made them more readable). Thus presenting the polyvariantly annotated program in an easy to read manner poses a difficult problem.

Chapter 14

Conclusion

Section 14.1 will relate our polyvariant binding time analysis with other similar analyses, and in section 14.2 we will sum up the work and conclude.

14.1 Comparison with other Work

A number of polyvariant binding time analyses have appeared within the last year. These include the work by Gengler and Rytz [Gengler & Rytz 1992a, Gengler & Rytz 1992b] adding polyvariance to the Similix binding time analysis, the binding time analysis of Schism [Consel 1992] and the binding time analysis of Petrarca [Niel 1993].

The simplest of these is the analysis of Gengler and Rytz. The extension is built directly on top of the existing Similix binding time analysis (or rather on an old version using abstract interpretation). The polyvariance is achieved in the following way: in the monovariant Similix analysis, if two different binding time descriptions of a function collide, the new description is obtained as the least upper bound of the two; in this case Gengler and Rytz instead makes a new copy of the function (such that there is a function for each of the two descriptions). After the copy has been made, first the closure analysis and then the binding time analysis is redone (actually they let the binding time analysis finish before redoing the analysis, possibly detecting more than one collision), since the copying might have affected the result of these analyses. This scheme for polyvariance is very intuitive and corresponds closely to the way a programmer using a monovariant binding time analysis would achieve “polyvariance by hand”. Unfortunately it is very slow, since both closure and binding time analysis has to be redone.

Consel uses a more refined scheme where closure analysis and binding time analysis is integrated in one analysis, making the loop over closure analysis/binding time analysis superfluous. Also, the Schism binding time analysis does not copy the full functions, only the binding time descriptions are copied. The Schism partial evaluator with the polyvariant analysis has been successfully used in larger applications and the binding time analysis is reasonably fast. The binding time analysis has another feature: it is possible for the user to define the degree of polyvariance needed for a specific application (sometimes too much polyvariance can lead to useless duplicates of functions in the residual program).

The binding time analysis in Petrarca by De Niel works by projection analysis and here, like in Schism, the polyvariance is achieved by making copies of the binding time descriptions. Petrarca differs from the above in being for a typed, but only first order language. Selfapplication has not been achieved with Petrarca.

We will call these analyses *copying*. Our analysis differs from the above in two important ways:

- Higher order typed language.
- Based on type inference.

Our analysis is based on the monovariant binding time analysis of Nielson and Nielson [Nielson & Nielson 1988] which we reformulated to make it possible to do abstraction over binding times and thereby over annotations. We take advantage of the standard types in the sense that binding time values "stick" to standard types and standard type constructors.

We feel that the type inference approach, where polyvariance is expressed as polymorphism in binding time values, is a more elegant formulation than the copying abstract interpretation based analyses. It is not clear whether our method is computationally better: it is clear, that the copying analyses can lead to an exponential number of copies (exponential in the program size). Our analysis can potentially lead to an exponential number of parameters to `let`-bound functions. We do, however, believe that there exist cases that lead to an exponential number of copies without leading to an exponential number of parameters. It is not clear what the effect of the reductions of chapter 10 is on the program size increase.

Our analysis can lead to program size increase in cases where a copying binding time analysis will give no increase at all. This happens because the copying analyses are global, and only makes copies on demand. Our analysis is local and does always give the full degree of polyvariance. This can be seen as a drawback of our analysis, but it also gives rise to another improvement to partial evaluation as a "real life" tool: since the analysis is local, it is possible to do binding time analysis in a modular way (though it has to be done hierarchically) — that is binding time analysis need not be redone for all modules if only one module is changed.

14.2 Summary

We have reached our initial goal of achieving polyvariancy in binding time analysis using a polymorphic type system. This was done for a monomorphically typed language with partially static data structures. The type system was proved correct *w.r.t.* a given specializer in the sense that the mix-equation was true for this combination of binding time analysis and specializer, and that specialization does not go wrong on well annotated terms.

We then showed, that this formulation of polyvariant binding time analysis is not restricted to simply typed languages, by showing the extensions necessary to make the language either polymorphically typed or dynamically typed.

A version of the well known algorithm $\mathcal{W}$ was adapted for the analysis. Based on work by Fuh and Mishra, we developed a number of reductions reducing the number of binding time parameters given to each `let`-bound function. The developed reduction schemes were complicated but gave significant reductions in the number of binding time parameters; this made binding time analysis and specialization run faster and made annotated programs easier to read.

Our prototype implementation was not very fast (though not impractical) but we believe that by carefully devising fast algorithms for the problem and making the implementation speed oriented instead of readability oriented, fast implementations can be achieved.

Bibliography

[Andersen & Mossin 1990] Lars Ole Andersen and Christian Mossin. Binding Time Analysis via Type Inference. Student Project 90-10-12, DIKU, University of Copenhagen, Denmark, October 1990.

[Barendregt & Dekkers 199] Henk Barendregt and Will Dekkers. *Typed lambda calculi.* 199? To appear.

[Barendregt, Geuvers, & Dekkers 1991] Henk Barendregt, Herman Geuvers, and Will Dekkers. Lambda Calculus. Lecture notes for Summer School on the λ-calculus, July 1991.

[Bondorf & Danvy 1991] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, Vol. 16, pp. 151–195, 1991.

[Bondorf & Jørgensen 1993a] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming, special issue on partial evaluation*, Vol. 11, 1993.

[Bondorf & Jørgensen 1993b] 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.

[Bondorf 1991] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, Vol. 17 (Selected papers of ESOP '90, the 3rd European Symposium on Programming), No. 1-3, pp. 3–34, December 1991.

[Bondorf 1992] Anders Bondorf. Improving binding times without explicit cps-conversion. In *1992 ACM Conference on Lisp and Functional Programming. San Francisco, California*, pp. 1–10, June 1992.

[Bondorf, Jones, Mogensen, & Sestoft 1988] Anders Bondorf, Neil D. Jones, Torben Mogensen, and Peter Sestoft. Binding Time Analysis and the Taming of Self-Application. Draft, 18 pages, DIKU, University of Copenhagen, Denmark, August 1988.

[Cardelli 1985] L. Cardelli. Basic Polymorphic Typechecking. *Polymorphism*, Vol. 2, No. 1, Jan. 1985.

[Clement, Despeyroux, Despeyroux, & Kahn 1986] D. Clement, J. Despeyroux, T. Despeyroux, and G. Kahn. A Simple Applicative Language: Mini-ML. *INRIA Centre Sophia Antipolis*, RR No. 529, May 1986.

[Consel & Danvy 1989] Charles Consel and Olivier Danvy. Partial Evaluation of Pattern Matching in Strings. *Information Processing Letters*, Vol. 30, pp. 79–86, January 1989.

[Consel 1992] Charles Consel. Polyvariant Binding-Time Analysis for Applicative Languages. Technical Report, Oregon Graduate Institute of Science & Technology, November 1992.

[Damas & Milner 1982] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages*, pp. 207–212. ACM Press, 1982.

[Davis 1993] Kei Davis. Higher-order Binding-time Analysis. In David Schmidt, editor, *1993 ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, June 1993. To appear.

[Fuh & Mishra 1988] Y. Fuh and P. Mishra. Type Inference with Subtypes. In *Proc. 2nd European Symp. on Programming*, pp. 94–114. Springer-Verlag, 1988. Lecture Notes in Computer Science 300.

[Fuh & Mishra 1989] Y. Fuh and P. Mishra. Polymorphic Subtype Inference: Closing the Theory-Practice Gap. In *Proc. Int'l J't Conf. on Theory and Practice of Software Development*, pp. 167–183, Barcelona, Spain, March 1989. Springer-Verlag.

[Futamura 1971] Yoshihiko Futamura. Partial evaluation of computing process — an approach to a compiler-compiler. *Systems, Computers, Controls*, Vol. 2, No. 5, pp. 45–50, 1971.

[Gengler & Rytz 1992a] Marc Gengler and Bernhard Rytz. A Polyvariant Binding Time Analysis. In *1992 ACM Wokshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 21–28. Yale University, 1992.

[Gengler & Rytz 1992b] Marc Gengler and Bernhard Rytz. A Polyvariant Binding Time Analysis Handling Partially Known Values. In *WSA'92: Workshop on Static Analysis*, pp. 322–330, September 1992.

[Girard, Lafont, & Taylor 1989] J. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1989.

[Gomard 1989] Carsten K. Gomard. Higher Order Partial Evaluation – HOPE for the Lambda Calculus. Master's thesis, DIKU, University of Copenhagen, Denmark, September 1989.

[Gomard 1990] Carsten K. Gomard. Partial Type Inference for Untyped Functional Programs (extended abstract). In *Lisp and Functional Programming*, pp. 282–287, 1990.

[Gomard 1991a] Carsten K. Gomard. A Self-applicable Partial Evaluator for the Lambda Calculus: Correctness and Pragmatics. *TOPLAS*, 1991.

[Gomard 1991b] Carsten Krogh Gomard. *Program Analysis Matters*. PhD thesis, DIKU, University of Copenhagen, Denmark, November 1991. DIKU report 91/17.

[Henglein 1991] Fritz Henglein. Efficient Type Inference for Higher-Order Binding-Time Analysis. In J. Hughes, editor, *FPCA*, pp. 448–472. 5th ACM Conference, Cambridge, MA, USA, Springer-Verlag, August 1991. Lecture Notes in Computer Science, Vol. 523.

[Henglein 1992] Fritz Henglein. Dynamic Typing. In B. Krieg-Brückner, editor, *Proc. European Symp. on Programming (ESOP), Rennes, France*, pp. 233–253. Springer-Verlag, February 1992. Lecture Notes in Computer Science, Vol. 582.

[Hunt & Sands 1991] S. Hunt and D. Sands. Binding Time Analysis: A New PERspective. In *Proc. ACM/IFIP Symp. on Partial Evaluation and Semantics Based Program Manipulation (PEPM), New Haven, Connecticut*, June 1991.

[Jones 1987] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Jones 1988] Neil D. Jones. Automatic Program Specialization: A Re-Examination from Basic Principles. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pp. 225–282. North-Holland, 1988.

[Jones *et al.* 1990] Neil D. Jones, Carsten K. Gomard, Anders Bondorf, Olivier Danvy, and Torben Æ. Mogensen. A Self-Applicable Partial Evaluator for the Lambda Calculus. In *1990 International Conference on Computer Languages, New Orleans, Louisiana, March 1990*, pp. 49–58. IEEE Computer Society, March 1990.

[Jones, Sestoft, & Søndergaard 1985] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In Jean-Pierre Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. Lecture Notes in Computer Science 202*, pp. 124–140. Springer-Verlag, 1985.

[Jones, Sestoft, & Søndergaard 1989] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation. *Lisp and Symbolic Computation*, Vol. 2, No. 1, pp. 9–50, 1989. DIKU Report 91/12.

[Jørgensen 1990] Jesper Jørgensen. Generating a Pattern Matching Compiler by Partial Evaluation. In Professor C.J. van Rijsbergen, editor, *Glasgow Workshop on Functional Programming, Ullapool*, pp. 177–195, Glasgow University, July 1990. Springer-Verlag.

[Jørgensen 1992] Jesper Jørgensen. Generating a Compiler for a Lazy Language by Partial Evaluation. In *Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Albuquerque, New Mexico*, pp. 258–268, January 1992.

[Kahn 1987] G. Kahn. Natural Semantics. Technical Report 601, INRIA, Feb. 1987.

[Launchbury 1990] J. Launchbury. *Projection Factorisations in Partial Evaluation.* PhD thesis, University of Glasgow, Jan. 1990.

[Mairson 1990] H. Mairson. Deciding ML Typability is Complete for Deterministic Exponential Time. In *Proc. 17th ACM Symp. on Principles of Programming Languages (POPL).* ACM, Jan. 1990.

[Milner 1978] Rober Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, Vol. 17, pp. 348–375, 1978.

[Mitchell 1984] John C. Mitchell. Coercion and Type Inference (summary). In *Eleventh ACM Symposium on Principles of Programming Languages*, pp. 175–185. ACM Press, January 1984.

[Mogensen 1988] Torben Mogensen. Partially Static Structures in a Self-Applicable Partial Evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pp. 325–347. North-Holland, 1988.

[Mogensen 1992] Torben Æ. Mogensen. Efficient Self-Interpretation in Lambda Calculus. *Functional Programming*, Vol. 2, No. 3, pp. 345–364, July 1992.

[Mossin 1992] Christian Mossin. Partial evaluation of General Parsers. Student Report 92–8–1, DIKU, University of Copenhagen, Denmark, August 1992.

[Mossin 1993] Christian Mossin. Partial evaluation of General Parsers (Extended abstract). In David Schmidt, editor, *1993 ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, June 1993. Conference version of [Mossin 1992].

[Mycroft 1984] A. Mycroft. Polymorphic Type Schemes and Recursive Definitions. In *Proc. 6th Int. Conf. on Programming, LNCS 167*, 1984.

[Niel 1993] Anne De Niel. *Self-Applicable Partial Evaluation of Polymorphically Typed Functional Languages.* PhD thesis, Katholieke Universiteit Leuven, January 1993.

[Nielson & Nielson 1988] H. Nielson and F. Nielson. Automatic Binding Time Analysis for a Typed Lambda Calculus. *Science of Computer Programming*, Vol. 10, pp. 139–176, 1988.

[Palsberg & Schwartzbach 1992] Jens Palsberg and Michael I. Schwartzbach. Binding time analysis: Abstract interpretation versus type inference. 1992. Submitted for publication.

[Palsberg 1993] Jens Palsberg. Correctness of Binding Time Analysis. *Journal of Functional Programming*, 1993. To appear.

[Plotkin 1981] G. Plotkin. A Structural Approach to Operational Semantics. Technical Report FN-19, Aarhus University, DAIMI, Sept. 1981.

[Solberg, Nielson, & Nielson 1992] Kirsten Lackner Solberg, Hanne Riis Nielson, and Flemming Nielson. Inference Systems for Binding Time Analysis *(Extended Abstract)*. In *WSA'92: Workshop on Static Analysis*, pp. 247–254, September 1992.

[Thatte 1988] S. Thatte. Type Inference with Partial Types. In *Proc. Int'l Coll. on Automata, Languages and Programming (ICALP)*, pp. 615–629, 1988. Lecture Notes in Computer Science.

[Wand 1993] Mitchell Wand. Specifying the Correctness of Binding-Time Analysis. *Journal of Functional Programming*, 1993. To appear.

Appendix A

Symbols

Symbols used in the text, their meaning and a reference to the text.

A	*Type environment mapping variables (∈ Var) to typeschemes.* Section 3.1.
C	*Constraint set.* Section 4.2.
ρ,ρ_d	*Environment mapping variables (∈ Var) to values in Val_{std}.* Section 2.2.
ρ_s	*Environment mapping variables (∈ Var) to values in Exp_{spec}.* Chapter 5.
ζ	*Substitution.* Section 6.4.
t	*Meta-variable over standard types.* Section 3.1.
τ	*Syntactic variable over standard types.* Section 7.1.
b	*Meta-variable over binding time values.* Section 3.1.
β	*Syntactic variable over binding time values.* Section 4.1.
$BtVar$	*Set of binding time variables β.* Section 4.1.
Exp_{std}	*Unannotated expressions.* Section 2.2.
Exp_{spec}	*Annotated (two-level) expressions.* Sections 3.1 and 4.2
Val_{std}	*Standard values.* Section 2.2.
Val_{spec}	*Two level values (including $\underline{Val}$).* Chapter 5
Var	*Identifier names.* Section 2.2.
$\underline{Var}$	*Identifier names in ρ_s for dynamic variables. Subset of $\underline{Val}$.* Chapter 5.
$\underline{Val}$	*Residual expressions (isomorphic to Exp_{std}).* Chapter 5.
$Const$	*Constants.* Section 2.2.
$Funval_{std}$	*Closures in Val_{std}.* Section 2.2.
$Funval_{spec}$	*Closures in Val_{spec}.* Chapter 5.
$RecFunval_{std}$	*Recursive closures in Val_{std}.* Section 2.2.
$RecFunval_{spec}$	*Recursive closures in Val_{spec}.* Chapter 5.
$BtFunval_{spec}$	*Binding time abstraction closures.* Chapter 5.
$ClFunval_{spec}$	*Coercion abstraction closures.* Chapter 5.
$Varenv_{std}$	*Set of environments ρ.* Section 2.2.
$Varenv_{spec}$	*Set of environments ρ_s.* Chapter 5.

ϕ	*Annotation forgetting function* $Exp_{spec} \to Exp_{std}$. Definition 6.1.
φ	*“Underline forgetting” isomorphism* $\underline{Val} \to Exp_{std}$. Definition 6.2.
ψ	*Standard type forgetting function.* Definition 6.3.
$\mathcal{U}$	*Universal type in dynamic typing.* Section 7.2

Appendix B

BTA Typerules

B.1 The Basic Polymorphic System — System DM

$$\text{(const)}\quad A{,}C\vdash\texttt{true}{:}(\text{Bool},\mathsf{S})\qquad A{,}C\vdash\texttt{false}{:}(\text{Bool},\mathsf{S})\qquad A{,}C\vdash n{:}(\text{Int},\mathsf{S})$$

$$\text{(var)}\quad A\bigcup\{\text{x}{:}\sigma\}{,}C\vdash\text{x}{:}\sigma$$

$$\text{(if)}\quad \frac{A{,}C\vdash\text{e}{:}(\text{Bool},b)\quad A{,}C\vdash\text{e}'{:}(t{,}b')\quad A{,}C\vdash\text{e}''{:}(t{,}b')\quad C\vdash b\leq b'}{A{,}C\vdash\texttt{if}^{b}\ \text{e}\ \texttt{then}\ \text{e}'\ \texttt{else}\ \text{e}''{:}\ (t{,}b')}$$

$$\text{(abstr)}\quad \frac{A\bigcup\{\text{x}{:}(t{,}b)\},\ C\vdash\text{e}{:}(t',b')\quad C\vdash b''\leq b\quad C\vdash b''\leq b'}{A{,}C\vdash\lambda^{b''}\text{x.e}\ :\ ((t,b)\to(t',b'),b'')}$$

$$\text{(appl)}\quad \frac{A{,}C\vdash\text{e}{:}((t,b)\to(t',b'),b'')\quad A{,}C\vdash\text{e}'{:}(t{,}b)\quad C\vdash b''\leq b\quad C\vdash b''\leq b'}{A{,}C\vdash\text{e@}^{b''}\text{e}'{:}\ (t',b')}$$

$$\text{(op)}\quad \frac{A{,}C\vdash\text{e}{:}(t{,}b)\quad A{,}C\vdash\text{e}'{:}(t',b)\quad \mathcal{P}(\texttt{op})=t\times t'\to t''}{A{,}C\vdash\text{e}\ \texttt{op}^{b}\text{e}'{:}\ (t'',b)}$$

$$\text{(fix)}\quad \frac{A\bigcup\{\text{x}{:}(t{,}b){,}\text{f}{:}((t{,}b)\to(t'{,}b'){,}b'')\}{,}C\vdash\text{e}{:}(t'{,}b')\quad C\vdash b''\leq b\quad C\vdash b''\leq b'}{A{,}C\vdash\texttt{fix}^{b''}\text{f x.e}\ :\ ((t{,}b)\to(t'{,}b'){,}b'')}$$

$$\text{(pair)}\quad \frac{A{,}C\vdash\text{e}{:}(t{,}b)\quad A{,}C\vdash\text{e}'{:}(t',b')\quad C\vdash b''\leq b\quad C\vdash b''\leq b'}{A{,}C\vdash\texttt{pair}^{b''}(\text{e{,}e}')\ :\ ((t,b)\times(t',b'),b'')}$$

$$\text{(first)}\quad \frac{A{,}C\vdash\text{e}{:}\ ((t,b)\times(t',b'),b'')\quad C\vdash b''\leq b\quad C\vdash b''\leq b'}{A{,}C\vdash\texttt{fst}^{b''}\text{e}\ :\ (t{,}b)}$$

$$\text{(second)}\quad \frac{A{,}C\vdash\text{e}{:}\ ((t,b)\times(t',b'),b'')\quad C\vdash b''\leq b\quad C\vdash b''\leq b'}{A{,}C\vdash\texttt{snd}^{b''}\text{e}\ :\ (t',b')}$$

$$(\forall\text{–introduction})\quad \frac{A{,}C\vdash \mathrm{e}{:}\sigma}{A{,}C\vdash \Lambda\beta.\mathrm{e}{:}\forall\beta.\sigma}(\text{if } \beta \text{ not free in } A{,}C)$$

$$(\forall\text{–elimination})\quad \frac{A{,}C\vdash \mathrm{e}{:}\forall\beta.\sigma}{A{,}C\vdash \mathrm{e}\diamond b{:}[b/\beta]\sigma}$$

$$(\Rightarrow\text{–introduction})\quad \frac{A{,}C\bigcup\{b\leq b'\}\vdash \mathrm{e}{:}\sigma}{A{,}C\vdash \Lambda b\rightsquigarrow b'.\mathrm{e} : b\leq b'\Rightarrow\sigma}$$

$$(\Rightarrow\text{–elimination})\quad \frac{A{,}C\vdash \mathrm{e} : b\leq b'\Rightarrow\sigma \quad C\vdash b\leq b'}{A{,}C\vdash \mathrm{e}\Box b\rightsquigarrow b'{:}\sigma}$$

$$(\text{let})\quad \frac{A{,}C\vdash \mathrm{e}'{:}\sigma \quad A\bigcup\{\mathrm{x}{:}\sigma\}{,}C\vdash \mathrm{e}{:}\kappa}{A{,}C\vdash \mathtt{let}\ \mathrm{x}=\mathrm{e}'\ \mathtt{in}\ \mathrm{e}{:}\kappa}$$

$$(\text{coerce})\quad \frac{A{,}C\vdash \mathrm{e}{:}(\mathrm{Bool}{,}b) \quad C\vdash b\leq b'}{A{,}C\vdash [b\rightsquigarrow b']\mathrm{e}{:}(\mathrm{Bool}{,}b')}$$

$$(\text{coerce})\quad \frac{A{,}C\vdash \mathrm{e}{:}(\mathrm{Int}{,}b) \quad C\vdash b\leq b'}{A{,}C\vdash [b\rightsquigarrow b']\mathrm{e}{:}(\mathrm{Int}{,}b')}$$

$$(\text{lookup-coerce})\quad C\bigcup\{b\leq b'\}\vdash b\rightsquigarrow b'$$

$$(\text{lift})\quad C\vdash \mathsf{S}\leq\mathsf{D}$$

$$(\text{id-dyn})\quad C\vdash \mathsf{D}\leq\mathsf{D}$$

$$(\text{id-stat})\quad C\vdash \mathsf{S}\leq\mathsf{S}$$

B.2 Deterministic Abstraction — System MT

In Part II the following rules replace (var), (let), (∀–introduction), (⇒–introduction), (∀–elimination) and (⇒–elimination)

$$(\text{Var})\quad \frac{C'\vdash [b_i/\beta_i]c_j}{A\bigcup\{\mathrm{x}{:}\forall(\beta_1\ldots\beta_n)^{\{c_1,\ldots,c_m\}}.\kappa\}{,}C'\vdash \mathrm{x}\diamond(b_1\ldots b_n){:}[b_i/\beta_i]\kappa}$$

$$(\text{let})\quad \frac{\begin{array}{l}A{,}C\vdash \mathrm{e}'{:}\kappa' \quad A\bigcup\{\mathrm{x}{:}\forall(\beta_1\ldots\beta_n)^{C}.\kappa'\}{,}C'\vdash \mathrm{e}{:}\kappa\\ \{\beta_1\ldots\beta_n\} = \mathit{FreeBTVars}(C)\bigcup \mathit{FreeBTVars}(\kappa') - \mathit{FreeBTVars}(A)\end{array}}{A{,}C'\vdash \mathtt{let}\ \mathrm{x}=\Lambda(\beta_1\ldots\beta_n)^{C}.\mathrm{e}'\ \mathtt{in}\ \mathrm{e}{:}\kappa}$$

Appendix C

Programs

C.1 Main Program

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                            ;;
;; The Perplex System                                                         ;;
;; Partial Evaluation with Polyvariant Lets for the EXtended lambda calculus ;;
;;                                                                            ;;
;; Christian Mossin                                                           ;;
;;                                                                            ;;
;; This File:                                                                 ;;
;;   Polyvariant Binding Time Analysis for Higher Order Lambda Calculus      ;;
;;   with Partially Static Structures.                                        ;;
;;                                                                            ;;
;; Created: Mon Feb 8                                                         ;;
;; Last changed: Thu Jul 15                                                   ;;
;;                                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Concrete Syntax of the Language:
;;
;; e ::= (bool true) | (bool false) | (int n) | (nil) | (var x) |
;;       (if e then e else e) |
;;       (lambda x e) | (apply e e) |
;;       (primop e e) | (fix f e) | (let x e e) | (null? e)
;;       (pair e e) | (fst e) | (snd e)
;;

(define test -1)    ; <- level of test:
                    ;   test = -2 : is silent
                    ;   test = -1 : tells the name of the currently analyzed let
                    ;   test = 0 : tells what is going on
                    ;   test = 1 : Gives types and reductions
                    ;   test > 1 : is for debugging only
(define pp-with-constraints #f) ; Should constraints be pretty-printed
```

```
(define (primop-env) '((+ (integer integer integer))
                       (- (integer integer integer))
                       (* (integer integer integer))
                       (/ (integer integer integer))
                       (and (boolean boolean boolean))
                       (or (boolean boolean boolean))
                       (= (integer integer boolean))
                       (> (integer integer boolean))
                       (< (integer integer boolean))))

(define bta
  (lambda inp
    (let ((program (car inp))
          (goal-env (make-env (cadr inp)))
          (file (if (= (length inp) 3) (caddr inp) '())))
      (let* ((ann-program (annotate-program program))
             (text (if (> test -2) (begin (printf "Performing W ")(newline))))
             (bt-pgm (w ann-program goal-env))
             (text (if (> test -2)
                       (begin (printf "Annotating")
                              (newline))))
             (type (car bt-pgm))
             (subst (cadr bt-pgm))
             (constraints (caddr bt-pgm))
             (applenv (cadddr bt-pgm))
             (text (if (> test 2)
                       (begin
                         (printf
                          "In Main: unreduced coercions: ~s"
                          constraints)(newline))))
             (text (if (> test -1)
                       (begin (printf "  Finding least solution:")(newline))))
             (constraints-subst (find-least-solution constraints))
             (free-subst (make-free-static-subst
                          (apply-subst constraints-subst constraints)))
             (new-subst (compose-subst free-subst
                                       (compose-subst constraints-subst
                                                      subst)))
             (mid-pgm (apply-subst new-subst ann-program))
             (text (if (> test 9) (begin (pretty-print mid-pgm)(newline))))
             (new-appl-env (apply-subst constraints-subst applenv))
             (mid-pgm1 (apply-env new-appl-env mid-pgm))
             (mid-pgm2 (make-free-static mid-pgm1))
             (final-pgm (delift mid-pgm2)))
        (begin
          (if (and file (> test -2))
              (begin (printf "Pretty Printing")(newline)))
          (if (> test -2)
              (begin (printf "type: ~s" (apply-subst new-subst type))
                     (newline)))
          (if file
              (writef (pretty-print-latex final-pgm) file))
          (printf "Total:")(newline)
          final-pgm
          )))))

(define (w e env)
  (let ((wres
  (cond
   [(isBoolean? e)
    (let* ((ann (get-annotation e))
           (new-subst (make-subst ann type-static)))
      (list (make-compound-type type-boolean type-static)
            new-subst empty-constraint-set empty-appl-env))]
   [(isInteger? e)
    (let* ((ann (get-annotation e))
           (new-subst (make-subst ann type-static)))
      (list (make-compound-type type-integer type-static)
            new-subst empty-constraint-set empty-appl-env))]
   [(isNil? e)
    (let* ((ann (get-annotation e))
           (bt-fst-type (bt-make-var))
           (std-fst-type (std-make-var))
           (fst-type (make-compound-type std-fst-type bt-fst-type))
           (bt-snd-type (bt-make-var))
           (std-snd-type (std-make-var))
           (snd-type (make-compound-type std-snd-type bt-snd-type))
           (pair-type (make-pair-type fst-type snd-type ann))
           (new-constraints (list (list 'leq ann bt-fst-type)
                                  (list 'leq ann bt-snd-type))))
      (list pair-type
            identity new-constraints empty-appl-env))]
   [(isVar? e)
    (let* ((ann (get-annotation e))
           (vname (get-var e))
           (type (lookup-env vname env))
           )
      (if (type-forall? type)
          (let* ((arg-type (get-forall-arg type))
                 (constraints (get-forall-constraints type))
                 (result (get-forall-result type))
                 (subst (make-subst-to-new arg-type))
                 (new-constraints (apply-subst subst constraints))
                 (new-appl (update-env (get-var-number e)
                                       (apply-subst subst arg-type)
                                       '()))
                 (new-type (apply-subst subst result))
                 )
            (list new-type
                  identity
                  new-constraints
                  new-appl))
          (let* ((bt-type (get-bt-type type))
                 (new-subst (unify-bt ann bt-type)))
            (list (apply-subst new-subst type)
                  new-subst empty-constraint-set empty-appl-env))))]
   [(isIf? e)
    (let* ((ann (get-annotation e))
           (conditional (get-conditional e))
           (cond-then (get-cond-then e))
           (cond-else (get-cond-else e))
           (w1 (w conditional env))
           (t1 (car w1))
           (s1 (cadr w1))
           (w2 (w cond-then (apply-subst s1 env)))
           (t2 (car w2))
           (s2 (cadr w2))
           (w3 (w cond-else (apply-subst (compose-subst s2 s1) env)))
           (t3 (car w3))
           (s3 (cadr w3))
           (subst1 (unify (make-compound-type type-boolean ann) t1))
           (t2s (apply-subst subst1 t2))
           (t3s (apply-subst subst1 t3))
           (subst2 (unify t2s t3s))
           (subst (compose-subst subst2 subst1))
           (new-type (apply-subst subst2 t2s))
           (new-subst
            (compose-subst*
             (list subst2 subst1 s3 s2 s1)))
           (new-constraints (cons (list 'leq
                                 (apply-subst subst (get-bt-type t1))
                                 (apply-subst subst2 (get-bt-type t2s)))
                           (apply-subst
                            subst
                            (append (apply-subst (compose-subst s3 s2)
                                                 (caddr w1))
                                    (apply-subst s3 (caddr w2))
                                    (caddr w3)))))
           (new-applenv (apply-subst
                         subst
                         (append (apply-subst (compose-subst s3 s2)
                                              (cadddr w1))
                                 (apply-subst s3 (cadddr w2))
                                 (cadddr w3)))))
      (list new-type new-subst new-constraints new-applenv))]
   [(isLambda? e)
    (let* ((e1 (get-lambda-exp e))
           (var (get-lambda-var e))
           (bt-arg-type (bt-make-var))
           (std-arg-type (std-make-var))
```

```
           (arg-type (make-compound-type std-arg-type bt-arg-type))
           (new-env (update-env var arg-type env))
           (w1 (w e1 new-env))
           (t1 (car w1))
           (ann (get-annotation e))
           (new-subst (cadr w1))
           (new-type (apply-subst new-subst (make-fun-type arg-type t1 ann)))
           (new-constraints (cons (list 'leq ann (apply-subst new-subst
                                                              bt-arg-type))
                                  (cons (list 'leq ann (get-bt-type t1))
                                        (caddr w1))))
           (new-applenv (cadddr w1)))
      (list new-type new-subst new-constraints new-applenv))]
  [(isApply? e)
   (let* ((e1 (get-appl-funct e))
          (e2 (get-appl-arg e))
          (w1 (w e1 env))
          (t1 (car w1))
          (s1 (cadr w1))
          (w2 (w e2 (apply-subst s1 env)))
          (t2 (car w2))
          (s2 (cadr w2))
          (text (if (> test 2)
                    (begin
                      (printf "In apply: Applying something of type ~s" t1)
                      (printf " to something of type ~s" t2)
                      (newline))))
          (type (make-compound-type (std-make-var) (bt-make-var)))
          (ann (get-annotation e))
          (u (unify (apply-subst s2 t1) (make-fun-type t2 type ann)))
          (new-type (apply-subst u type))
          (new-subst (compose-subst u (compose-subst s2 s1)))
          (new-constraints
           (cons (apply-subst new-subst
                              (list 'leq ann (get-bt-type t2)))
                 (cons (apply-subst new-subst
                                    (list 'leq ann (get-bt-type type)))
                       (append (apply-subst u (apply-subst s2 (caddr w1)))
                               (apply-subst u (caddr w2))))))
          (new-applenv (apply-subst u (append (apply-subst s2 (cadddr w1))
                                              (cadddr w2)))))
      (list new-type new-subst new-constraints new-applenv))]
  [(isPrimop? e)
   (let* ((op (get-primop-op e))
          (e1 (get-primop-arg1 e))
          (e2 (get-primop-arg2 e))
          (prim-type (lookup-env op (primop-env)))
          (pt1 (car prim-type))
          (pt2 (cadr prim-type))
          (ptres (caddr prim-type))
```

```
         (ann (get-annotation e))
         (w1 (w e1 env))
         (t1 (car w1))
         (s1 (cadr w1))
         (w2 (w e2 (apply-subst s1 env)))
         (t2 (car w2))
         (s2 (cadr w2))
         (u1 (unify (make-compound-type pt1 ann) t1))
         (u2 (unify (apply-subst u1 (make-compound-type pt2 ann))
                    (apply-subst u1 t2)))
         (subst (compose-subst u2 u1))
         (new-type (apply-subst subst (make-compound-type ptres ann)))
         (new-subst (compose-subst
                     subst
                     (compose-subst s2 s1)))
         (new-constraints (apply-subst subst
                                       (append (apply-subst s2 (caddr w1))
                                               (caddr w2))))
         (new-applenv (apply-subst subst (append (apply-subst s2 (cadddr w1))
                                                 (cadddr w2)))))
    (list new-type new-subst new-constraints new-applenv))]
  [(isFix? e)
   (let* ((e1 (get-fix-exp e))
         (var (get-fix-var e))
         (ann (get-annotation e))
         (std-arg-type (std-make-var))
         (arg-type (make-compound-type std-arg-type ann))
         (new-env (update-env var arg-type env))
         (w1 (w e1 new-env))
         (t1 (car w1))
         (s1 (cadr w1))
         (text (if (> test 3)
                   (begin (printf "In fix: arg-type = ~s" arg-type)(newline)
                          (printf "        S arg-type = ~s"
                                  (apply-subst s1 arg-type))(newline)
                          (printf "Fix unification between S arg-type and ~s"
                                  t1)
                          (newline))))
         (u (unify (apply-subst s1 arg-type) t1))
         (new-type (apply-subst u t1))
         (new-subst (compose-subst u s1))
         (new-constraints (apply-subst u (caddr w1)))
         (new-applenv (apply-subst u (cadddr w1)))
         )
    (list new-type new-subst new-constraints new-applenv))]
  [(isPair? e)
   (let* ((e1 (get-pair-fst e))
         (e2 (get-pair-snd e))
         (ann (get-annotation e))
         (w1 (w e1 env))
```

```
          (t1 (car w1))
          (s1 (cadr w1))
          (w2 (w e2 (apply-subst s1 env)))
          (t2 (car w2))
          (s2 (cadr w2))
          (new-type (make-pair-type t1 t2 ann))
          (new-subst (compose-subst s2 s1))
          (new-constraints (cons (list 'leq ann (get-bt-type t1))
                                 (cons (list 'leq ann (get-bt-type t2))
                                       (append (apply-subst s2 (caddr w1))
                                               (caddr w2)))))
          (new-applenv (append (apply-subst s2 (cadddr w1))
                               (cadddr w2))))
     (list new-type new-subst new-constraints new-applenv))]
  [(isFst? e)
   (let* ((exp (get-fst-exp e))
          (w1 (w exp env))
          (t1 (car w1))
          (bt-fst-type (bt-make-var))
          (std-fst-type (std-make-var))
          (fst-type (make-compound-type std-fst-type bt-fst-type))
          (bt-snd-type (bt-make-var))
          (std-snd-type (std-make-var))
          (snd-type (make-compound-type std-snd-type bt-snd-type))
          (ann (get-annotation e))
          (pair-type (make-pair-type fst-type snd-type ann))
          (u (unify pair-type t1))
          (new-type (apply-subst u fst-type))
          (new-subst (compose-subst u (cadr w1)))
          (new-constraints
           (apply-subst u (cons (list 'leq ann bt-fst-type)
                                (cons (list 'leq ann bt-snd-type)
                                      (caddr w1)))))
          (new-applenv (apply-subst u (cadddr w1)))
          )
     (list new-type new-subst new-constraints new-applenv))]
  [(isSnd? e)
   (let* ((exp (get-snd-exp e))
          (w1 (w exp env))
          (t1 (car w1))
          (bt-fst-type (bt-make-var))
          (std-fst-type (std-make-var))
          (fst-type (make-compound-type std-fst-type bt-fst-type))
          (bt-snd-type (bt-make-var))
          (std-snd-type (std-make-var))
          (snd-type (make-compound-type std-snd-type bt-snd-type))
          (ann (get-annotation e))
          (pair-type (make-pair-type fst-type snd-type ann))
          (u (unify pair-type t1))
          (new-type (apply-subst u snd-type))
          (new-constraints
           (apply-subst u (cons (list 'leq ann bt-fst-type)
                                (cons (list 'leq ann bt-snd-type)
                                      (caddr w1)))))
          (new-subst (compose-subst u (cadr w1)))
          (new-applenv (apply-subst u (cadddr w1)))
          )
     (list new-type new-subst new-constraints new-applenv))]
   [(isNull? e)
    (let* ((exp (get-null-exp e))
          (w1 (w exp env))
          (t1 (car w1))
          (bt-fst-type (bt-make-var))
          (std-fst-type (std-make-var))
          (fst-type (make-compound-type std-fst-type bt-fst-type))
          (bt-snd-type (bt-make-var))
          (std-snd-type (std-make-var))
          (snd-type (make-compound-type std-snd-type bt-snd-type))
          (ann (get-annotation e))
          (pair-type (make-pair-type fst-type snd-type ann))
          (u (unify pair-type t1))
          (new-type (apply-subst u (make-compound-type type-boolean ann)))
          (new-subst (compose-subst u (cadr w1)))
          (new-constraints
           (apply-subst u (cons (list 'leq ann bt-fst-type)
                                (cons (list 'leq ann bt-snd-type)
                                      (caddr w1)))))
          (new-applenv (apply-subst u (cadddr w1))))
     (list new-type new-subst new-constraints new-applenv))]
   [(isLift? e)
    (let* ((exp (get-lift-exp e))
          (from-bt (get-lift-from e))
          (to-bt (get-lift-to e))
          (w1 (w exp env))
          (t (car w1))
          (text (if (> test 2)
                    (begin (printf "  Lifting something of type ~s" t)
                           (newline))))

          (t-std (get-std-type t))
          (u (unify-bt (get-bt-type t) from-bt))
          (new-subst (compose-subst u (cadr w1)))
          (newtype (make-compound-type t-std to-bt)))
     (cond
      [(or (type-boolean? t-std)
           (type-integer? t-std))
       (list (apply-subst u newtype)
             new-subst
             (apply-subst u
                          (cons (list 'leq from-bt to-bt) (caddr w1)))
```

```
              (apply-subst u (cadddr w1)))]
       [(std-type-var? t-std)
        (list (apply-subst u newtype)
              new-subst
              (apply-subst u
                           (cons (list 'leq from-bt to-bt t-std) (caddr w1)))
              (apply-subst u (cadddr w1)))]
       [else
        (let* ((s (unify-bt to-bt (apply-subst u from-bt)))
               (subst (compose-subst s u)))
          (list (apply-subst subst newtype)
                (compose-subst s new-subst)
                (apply-subst subst (caddr w1))
                (apply-subst subst (cadddr w1))))]))]
   [(isLet? e)
    (let* ((var (get-let-var e))
           (e1 (get-let-exp1 e))
           (e2 (get-let-exp2 e))
           (text (if (> test -2)
                     (begin (printf "Analyzing let ~s...:" var)(newline))))
           (w1 (w e1 env))
           (t1 (car w1))
           (s1 (cadr w1))

           (text (if (> test 0)
                     (begin
                       (printf "  Type of let ~s ...-bound exp: forall ~s."
                               var (get-forall-arg t1))
                       (newline)
                       (pretty-print (get-forall-result t1))
                       (if (> test 2)
                           (begin
                             (printf "  With constraints: ~s"
                                     (get-forall-constraints t1))
                             (newline)))
                       (newline))))
           (new-env (update-env var t1 (apply-subst s1 env)))
           (w2 (w e2 new-env))
           (t2 (car w2))
           (s2 (cadr w2))
           (new-subst (compose-subst s2 s1))
           (new-constraints (append (apply-subst s2 (caddr w1))
                                    (caddr w2)))
           (new-applenv (append (apply-subst s2 (cadddr w1))
                                (cadddr w2)))
           (text (if (> test 0)
                    (begin
                       (printf "  Finished let ~s ..." var)
                       (newline)))))
      (list t2 new-subst new-constraints new-applenv))]
```

```
[(isForall? e)
 (let* ((e1 (get-forall-exp e))
        (id (get-forall-id e))
        (w1 (w e1 env))
        (type (car w1))
        (subst (cadr w1))
        (text (if (> test 1)
                  (begin (printf "  type = ~s" type)
                         (newline))))
        (constraints (caddr w1))
        (applenv (cadddr w1))
        (text (if (> test 2) (begin (printf "  constraints = ~s"
                                            constraints)
                                    (newline))))
        (text (if (> test 2)
                  (begin (printf "  Applying substitution to environment")
                         (newline))))
        (text (begin (printf "Reducing:")(newline)))
        (reduced (time (reduce-constraints constraints
                                           type env
                                           e applenv
                                           subst)))
        (reduced-constraints (cadr reduced))
        (text (if (> test 2) (begin (printf "  Reduced constraints = ~s"
                                           reduced-constraints)
                                    (newline))))
        (constraints-subst (car reduced))
        (res-type (apply-subst constraints-subst type))

        (new-env (apply-subst constraints-subst
                              (apply-subst subst env)))
        (free-vars-in-env
         (sort-out-consts-and-duplicates
          (flatten
           (map (lambda (a)
                  (find-freevars-in-type (cadr a)))
                new-env))))
        (free-vars-in-constraints
         (find-free-vars-in-constraints reduced-constraints))
        (free-vars-in-type (find-freevars-in-type res-type))
        (abstr-vars
         (sort-out-consts-and-duplicates
          (set-minus (union free-vars-in-constraints
                            free-vars-in-type)
                     free-vars-in-env)))
        (new-type
         (make-forall-type abstr-vars reduced-constraints res-type))
        (new-subst (compose-subst constraints-subst subst))
        (new-applenv (update-env id
                                 (list abstr-vars reduced-constraints)
```

```
                                        (apply-subst constraints-subst applenv)))
                  )
              (list new-type
                    new-subst
                    empty-constraint-set
                    new-applenv))]
       )))
        (if
         (> test 4)
         (begin
           (printf "Evaluation of ~s" (apply-subst (cadr wres) e))(newline)
           (printf "  in environment ~s" env)(newline)
           (printf "Gives: Type is ~s" (car wres))(newline)
           (printf "       Constraints are ~s" (caddr wres))(newline)
           (printf "       Applenv is ~s" (cadddr wres))(newline)(newline)))
        wres))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Make room for annotations

(define (annotate-program e)
  (let* ((ann-e (annotate-program1 e))
         (res (get-lift-to ann-e))
         (subst (make-subst res type-dynamic)) ; Result is forced dynamic
         (ann-e-d (apply-subst subst ann-e)))
    ann-e-d))

(define annotate-program1
  (lambda (e)
    (if (not (null? e))
        (list 'lift
              (bt-make-var)
              (bt-make-var)
              (cond
               [(or (isBoolean? e)(isInteger? e))
                (list (car e) (bt-make-var) (cadr e))]
               [(isVar? e)
                (list (car e) (bt-make-var) (cadr e) (make-var-symbol))]
               [(or (isLambda? e)(isFix? e))
                (list (car e)
                      (bt-make-var)
                      (cadr e)
                      (annotate-program1 (caddr e)))]
               [(isPrimop? e)
                (list (car e)
                      (bt-make-var)
                      (cadr e)
                      (annotate-program1 (caddr e))
                      (annotate-program1 (caddr (cdr e))))]
               [(isLet? e)
                (list 'let
                      (cadr e)
                      (list 'forall (make-forall-symbol)
                            (annotate-program1 (caddr e)))
                      (annotate-program1 (cadddr e)))]
               [else
                (cons (car e)
                      (cons (bt-make-var)
                            (map annotate-program1 (cdr e))))]))
        (error 'annotate-program "Error in program."))))

(define delift
  (lambda (e)
    (cond
     [(isLift? e)
      (let ((from (get-lift-from e))
            (to (get-lift-to e)))
        (if (eq? from to)
            (delift (get-lift-exp e))
            (list 'lift
                  from
                  to
                  (delift (get-lift-exp e)))))]
     [(and (isVar? e) (not (list? (get-var-number e))))
      (list 'var (get-annotation e) (get-var e))]
     [(isVar? e)
      (list 'quanapp
            (list 'var
                  (get-annotation e)
                  (get-var e))
            (get-var-number e))]
     [(or (isBoolean? e)(isInteger? e)(isVar? e))
      e]
     [(or (isLambda? e)(isFix? e))
      (list (car e)
            (cadr e)
            (caddr e)
            (delift (caddr (cdr e))))]
     [(isPrimop? e)
      (list (car e)
            (cadr e)
            (caddr e)
            (delift (get-primop-arg1 e))
            (delift (get-primop-arg2 e)))]
     [(isQuanapp? e)
      (list (car e)
            (delift (get-quanapp-exp e))
            (get-quanapp-args e))]
     [else
      (cons (car e)
            (cons (cadr e)
                  (map delift (cddr e))))])))

(define (make-static-subst l)
  (if (null? l)
      identity
      (compose-subst (make-subst (car l) type-static)
                     (make-static-subst (cdr l)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Occurs in

(define (occur-inside? var type-exp)   ; On standard or binding time types
  (cond [(or (std-type-constant? type-exp) (bt-type-constant? type-exp))
         #f]
        [(or (std-type-var? type-exp) (bt-type-var? type-exp))
         (if (eq? var type-exp)
             #t
             #f)]
        [(type-function? type-exp)
         (or (occur-inside? var (get-std-type (type-func-operand type-exp)))
             (occur-inside? var (get-std-type (type-func-return type-exp))))]
        [(type-product? type-exp)
         (or (occur-inside? var (get-std-type (type-fst type-exp)))
             (occur-inside? var (get-std-type (type-snd type-exp))))]
        [(type-forall? type-exp)
         (occur-inside? var (get-std-type (get-forall-result type-exp)))]
        [(type-rec? type-exp)
         (occur-inside? var (get-rec-type type-exp))]
        [else (error 'occur-inside
                     "Unknown type in occur-inside? ~s~%" type-exp)]))

(define (occurs-in? var type-exp)   ; On compound types
  (if (type-forall? type-exp)
      (occurs-in? var (get-forall-result type-exp))
      (let ((st (get-std-type type-exp))
            (bt (get-bt-type type-exp)))
        (or (eq? var bt)
            (cond
             [(or (std-type-var? st) (std-type-constant? st))
              #f]
             [(type-function? st)
              (or (occurs-in? var (type-func-operand st))
                  (occurs-in? var (type-func-return st)))]
             [(type-product? st)
              (or (occurs-in? var (type-fst st))
                  (occurs-in? var (type-snd st)))]
             [(type-rec? st)
              (occurs-in? var (list (get-rec-type st) 'not-a-var))]
             [else (error 'occurs-in
                          "Unknown type in occurs-in? ~s~%" st)])))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Finding free variables
(define (find-freevars-in-exp e applenv)
  (sort-out-consts-and-duplicates
   (find-freevars-in-exp1 e applenv)))

(define (find-freevars-in-exp1 e applenv)
  (cond
   [(isBoolean? e)
    '()]
   [(isInteger? e)
    '()]
   [(isVar? e)
    (if (isApplVar? e applenv)
        (get-vars-in-list (lookup-env (get-var-number e) applenv))
        '())
    ]
   [(isIf? e)
    (cons (get-annotation e)
          (append (find-freevars-in-exp1 (get-conditional e) applenv)
                  (find-freevars-in-exp1 (get-cond-then e) applenv)
                  (find-freevars-in-exp1 (get-cond-else e) applenv)))]
   [(isLambda? e)
    (cons (get-annotation e)
          (find-freevars-in-exp1 (get-lambda-exp e) applenv))]
   [(isApply? e)
    (cons (get-annotation e)
          (append (find-freevars-in-exp1 (get-appl-funct e) applenv)
                  (find-freevars-in-exp1 (get-appl-arg e) applenv)))]
   [(isPrimop? e)
    (cons (get-annotation e)
          (append (find-freevars-in-exp1 (get-primop-arg1 e) applenv)
                  (find-freevars-in-exp1 (get-primop-arg2 e) applenv)))]
   [(isFix? e)
    (cons (get-annotation e)
          (find-freevars-in-exp1 (get-fix-exp e) applenv))]
   [(isLet? e)
    (append (find-freevars-in-exp1 (get-let-exp1 e) applenv)
            (find-freevars-in-exp1 (get-let-exp2 e) applenv))]
   [(isPair? e)
    (cons (get-annotation e)
          (append (find-freevars-in-exp1 (get-pair-fst e) applenv)
                  (find-freevars-in-exp1 (get-pair-snd e) applenv)))]
   [(isFst? e)
    (cons (get-annotation e)
          (find-freevars-in-exp1 (get-fst-exp e) applenv))]
   [(isSnd? e)
    (cons (get-annotation e)
          (find-freevars-in-exp1 (get-snd-exp e) applenv))]
   [(isLift? e)
    (find-freevars-in-exp1 (get-lift-exp e) applenv)
    ]
   [(isNil? e)(list (get-annotation e))]
   [(isNull? e)
    (cons (get-annotation e)
          (find-freevars-in-exp1 (get-null-exp e) applenv))]
   [(isForall? e)
    (find-freevars-in-exp1 (get-forall-exp e) applenv)]))

(define (get-vars-in-list l)
  (if (null? l)
      '()
      (let ((b (car l)))
        (if (bt-type-var? b)
            (cons b (get-vars-in-list (cdr l)))
            (get-vars-in-list (cdr l))))))

(define (find-freevars-in-type t)
  (find-freevars-in-type1 t '()))

(define (find-freevars-in-type0 tl)
  (if (null? tl)
      '()
      (append (find-freevars-in-type2 (cadar tl))
              (find-freevars-in-type0 (cdr tl)))))

(define (find-freevars-in-type1 t not-free)
  (if (type-forall? t)
      (find-freevars-in-type1 (get-forall-result t)
                              (append (get-forall-arg t) not-free))
      (let ((st (car t))
            (bt (cadr t)))
        (cond
         [(type-boolean? st)
          (list bt)]
         [(type-integer? st)
          (list bt)]
         [(type-rec? st)
          (find-freevars-in-type1 (get-rec-type st)
                                  (cons (get-rec-var st)
                                        not-free))]
         [(type-product? st)
          (cons bt (append (find-freevars-in-type1 (type-fst st) not-free)
                           (find-freevars-in-type1 (type-snd st) not-free)))]
         [(type-function? st)
          (cons bt (append (find-freevars-in-type1 (type-func-operand st)
                                                   not-free)
                           (find-freevars-in-type1 (type-func-return st)
                                                        not-free)))]
         [(std-type-var? st)
          (if (member bt not-free)
              '()
              (list bt))]))))

(define (find-freevars-in-type2 st)
  (cond
   [(type-boolean? st)
    '()]
   [(type-integer? st)
    '()]
   [(type-product? st)
    (append (find-freevars-in-type1 (type-fst st))
            (find-freevars-in-type1 (type-snd st)))]
   [(type-function? st)
    (append (find-freevars-in-type1 (type-func-operand st))
            (find-freevars-in-type1 (type-func-return st)))]
   [(std-type-var? st)
    '()]))

(define (find-free-vars-in-constraints s)
  (sort-out-consts-and-duplicates (find-free-vars-in-constraints1 s)))

(define (find-free-vars-in-constraints1 s)
  (if (null? s)
      '()
      (let ((b1 (cadar s))
            (b2 (caddar s)))
        (cons b1 (cons b2 (find-free-vars-in-constraints1 (cdr s)))))))

(define (make-free-static pgm)
  ((make-free-static1 '()) pgm))
(define (make-free-static1 not-free)
  (lambda (l)
    (cond
     [(null? l) '()]
     [(atom? l)
      (if (and (symbol? l)(bt-type-var? l)(not (member l not-free)))
          type-static
          l)]
     [(list? l)
      (if (isforall? l)
          (let ((args (get-forall-args l)))
            (list 'forall
                  args
                  ((make-free-static1 (append (car (get-forall-args l))
                                              not-free))
                   (get-forall-exp l))))
          (map (make-free-static1 not-free) l))])))
```

```
(define make-free-static-subst
  (lambda (l)
    (cond
     [(null? l) '()]
     [(atom? l)
      (if (and (symbol? l)(bt-type-var? l))
          (make-subst l type-static)
          identity)]
     [(list? l)
      (compose-subst* (map make-free-static-subst l))])))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Unification.

(define (unify t1 t2)
  (if (> test 5)(begin (printf "trying to unify:")(newline)
                       (pretty-print t1)
                       (printf "with")(newline)
                       (pretty-print t2)
                       (newline)))
  (compose-subst (unify-std (get-std-type t1) (get-std-type t2))
                 (unify-bt (get-bt-type t1) (get-bt-type t2))))

(define (unify-bt t1 t2)
  (cond
   [(and (bt-type-constant? t1)(bt-type-constant? t2))
    (if (or (and (type-static? t1)(type-static? t2))
            (and (type-dynamic? t1)(type-dynamic? t2)))
        identity
        (unification-error t1 t2))]
   [(and (bt-type-var? t1)(bt-type-var? t2))
    (make-subst t1 t2)]
   [(bt-type-var? t1)
    (make-subst t1 t2)]
   [(bt-type-var? t2)
    (make-subst t2 t1)]
   [else (unification-error t1 t2)]))

(define (unify-std t1 t2)
  (cond
   [(and (std-type-constant? t1)(std-type-constant? t2))
    (if (or (and (type-boolean? t1)(type-boolean? t2))
            (and (type-integer? t1)(type-integer? t2)))
        identity
        (unification-error t1 t2))]
   [(and (std-type-var? t1)(std-type-var? t2))
    (make-subst t1 t2)]
   [(std-type-var? t1)
```

```
    (if (occur-inside? t1 t2)
        (begin (if (> test 4)
                   (begin (newline)
                          (printf "  Recursive type: Variable ~s occurs in ~s"
                                  t1 t2)
                          (newline)(newline)))
               (make-subst t1 (make-rec-type t1 t2)))
        (make-subst t1 t2))]
   [(std-type-var? t2)
    (if (occur-inside? t2 t1)
        (begin (if (> test 4)
                   (begin (newline)
                          (printf "  Recursive type: Variable ~s occurs in ~s"
                                  t2 t1)
                          (newline)(newline)))
               (make-subst t1 (make-rec-type t1 t2)))
        (make-subst t2 t1))]
   [(and (type-function? t1)(type-function? t2))
    (let* ([s1 (unify (type-func-return t1)
                      (type-func-return t2))]
           [s2 (unify (apply-subst s1 (type-func-operand t1))
                      (apply-subst s1 (type-func-operand t2)))])
      (compose-subst s2 s1))]
   [(and (type-product? t1)(type-product? t2))
    (let* ([s1 (unify (type-fst t1)
                      (type-fst t2))]
           [s2 (unify (apply-subst s1 (type-snd t1))
                      (apply-subst s1 (type-snd t2)))])
      (compose-subst s2 s1))]
   [(and (type-rec? t1)(type-rec? t2))
    (let* ((s1 (unify-std (get-rec-var t1) (get-rec-var t2)))
           (s2 (unify-std (apply-subst s1 (get-rec-type t1))
                          (apply-subst s1 (get-rec-type t2)))))
      (compose-subst s2 s1))]
   [(type-rec? t1)
    (unify-std (apply-subst (make-subst (get-rec-var t1) t1) (get-rec-type t1))
               t2)]
   [(type-rec? t2)
    (unify-std (apply-subst (make-subst (get-rec-var t2) t2) (get-rec-type t2))
               t1)]
   [else (unification-error t1 t2)]))

(define (unification-error t1 t2)
  (error 'unify "~s and ~s not unifiable" t1 t2))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Make environment (v -> (type-var bt-type))* from bt-env (v -> bt-type)*
(define (make-env bt-env)
  (if (null? bt-env)
      '()
      (cons (list (caar bt-env)
                  (make-compound-type (std-make-var) (cadar bt-env)))
            (make-env (cdr bt-env)))))
```

C.2 Reduction Procedures

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                          ;;
;; The Perplex System                                                       ;;
;; Partial Evaluation with Polyvariant Lets for the EXtended lambda calculus ;;
;;                                                                          ;;
;; Christian Mossin                                                         ;;
;;                                                                          ;;
;; This File:                                                               ;;
;;   Reduction                                                              ;;
;;                                                                          ;;
;; Created: Fri May 7                                                       ;;
;; Last changed: Thu Jul 15                                                 ;;
;;                                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Reduction of constraints
(define time-red 1)

(define (reduce-constraints constraints type env e applenv s)
  (let* ((text (if (> test -1)
                   (begin (printf "Reducing Constraints")(newline))))
         (s-env (map (lambda (t)
                       (list (car t)
                             (apply-subst s (cadr t))))
                     env))
         (text (if (> test 2) (begin (printf "  S env = ~s" s-env)
                                     (newline))))
         (text (if (> test 0) (begin (printf "  Finding Intv")(newline))))
         (free-vars-in-constraints (find-free-vars-in-constraints constraints))
         (free-vars-in-type (sort-out-consts-and-duplicates
                             (find-freevars-in-type type)))
         (free-vars-in-env
          (sort-out-consts-and-duplicates
           (flatten
            (map (lambda (a)
                   (find-freevars-in-type (cadr a)))
                 s-env))))
         (intv (sort-out-consts-and-duplicates
                (set-minus free-vars-in-constraints
                           (append free-vars-in-type
                                   free-vars-in-env))))
```

```
(text (if (> test 2) (begin (printf "  Intv = ~s" intv)
                            (newline))))
(text (if (> test -1)
          (begin (printf "  C-reduction:")(newline))))
(text (if (= time-red 1)(begin (printf "C-reduction")(newline))))
(s0 (if (= time-red 0)
        (eliminate-cycles constraints intv)
        (time (eliminate-cycles constraints intv))))
(c1 (apply-subst s0 constraints))
(text (if (> test 3)
          (begin (printf "  Constraints after eliminating cycles: ~s"
                         c1)(newline))))
(type (apply-subst s0 type))
(env (apply-subst s0 s-env))
(subst (compose-subst s0 s))
(all-vars (cons type-static
                (cons type-dynamic
                      (find-free-vars-in-constraints constraints))))
(text (if (> test -1)
          (begin (printf "  T-reduction:")(newline))))
(text (if (= time-red 1)(begin (printf "T-reduction")(newline))))
(red1 (if (= time-red 0)
          (Conditioned-reduction constraints intv identity '())
          (time (Conditioned-reduction constraints intv identity '())
                )))
(s1 (car red1))
(constraints1 (remove-duplicates-constraints (cadr red1)))
(text (if (> test 3)
          (begin (printf "T-reduction gave constraints")
                 (newline)
                 (printf "~s" constraints1)(newline))))

(all-vars1 (sort-out-duplicates (apply-subst s1 all-vars)))
(above-env (compute-above-env all-vars1 all-vars1 constraints1))
(below-env (compute-below-env all-vars1 all-vars1 constraints1))
(text (if (> test -1)
          (begin (printf "  G-reduction:")(newline))))
(free-vars-in-exp (apply-subst
                   s1
                   (apply-subst subst
                                (find-freevars-in-exp e applenv))))
(text (if (> test 3)
          (begin (printf "FreeVars(e) = ~s" free-vars-in-exp)
                 (newline))))
(type1 (apply-subst s1 type))
(free-vars-in-constraints
 (find-free-vars-in-constraints constraints1))
(free-vars-in-type (sort-out-consts-and-duplicates
                    (find-freevars-in-type type1)))
(free-vars-in-env
 (sort-out-consts-and-duplicates
  (flatten
   (map (lambda (a)
          (apply-subst s1 (find-freevars-in-type (cadr a))))
        env))))
(obv (sort-out-consts-and-duplicates
      (append free-vars-in-type
              free-vars-in-env)))
(text (if (> test 5)
          (begin (printf "obv = ~s" obv)(newline))))
(text (if (= time-red 1)(begin (printf "G-reduction")(newline))))
(red2 (if (= time-red 0)
          (G-reduction all-vars1 all-vars1 all-vars1
                       above-env below-env
                       constraints1 obv identity
                       free-vars-in-exp)
          (time (G-reduction all-vars1 all-vars1 all-vars1
                             above-env below-env
                             constraints1 obv identity
                             free-vars-in-exp))))
(s2 (car red2))
(constraints2 (remove-duplicates-constraints (cadr red2)))
(above-env2 (sort-out-env-duplicates (caddr red2)))
(below-env2 (sort-out-env-duplicates (cadddr red2)))

(text (if (> test 3)
          (begin (printf "G-reduction gave constraints")(newline)
                 (printf "~s" constraints2)(newline))))
(s12 (compose-subst s2 s1))
(type-closure (apply-subst s2 type1))
(all-vars2 (sort-out-duplicates (apply-subst s2
                                              all-vars1)))
(text (if (> test -1)
          (begin (printf "  S-reduction:")(newline))))
(text (if (= time-red 1)(begin (printf "S-reduction")(newline))))
(red3 (if (= time-red 0)
          (S-reduction constraints2 identity
                       above-env2 below-env2
                       0 (length constraints2)
                       type-closure all-vars2
                       (sort-out-duplicates
                        (apply-subst s2 free-vars-in-exp))
                       (apply-subst s2 free-vars-in-env))
          (time (S-reduction constraints2 identity
                             above-env2 below-env2
                             0 (length constraints2)
                             type-closure all-vars2
                             (sort-out-duplicates
                              (apply-subst s2 free-vars-in-exp))
                             (apply-subst s2 free-vars-in-env)))))
```

```
         (s3 (car red3))
         (constraints3 (reduce-equal (cadr red3)))
         (text (if (> test 3)
                   (begin (printf "S-reduction gave constraints")(newline)
                          (printf "~s" constraints3)(newline))))

         (text (if (> test -1)
                   (begin (printf "  Finding least solution:")(newline))))
         (s4 (find-least-solution constraints3))
         (constraints4 (apply-subst s4 constraints3))

         (substitution (compose-subst s4 (compose-subst s3 s12)))
         )
    (list (compose-subst substitution subst)
          (remove-duplicates-constraints constraints4))))

(define (G-reduction x-vars y-vars all-vars above-env below-env
                     constraints obv subst free-vars-in-exp)
  (if (null? y-vars)
      (list subst constraints above-env below-env)
      (if (null? x-vars)
          (G-reduction all-vars (cdr y-vars)
                       all-vars above-env below-env constraints
                       obv subst free-vars-in-exp)
          (let ((x (car x-vars))
                (y (car y-vars)))
            (if (and (not (eq? x y))
                     (bt-type-var? x)
                     (g-subsumes x y below-env above-env constraints
                                 obv free-vars-in-exp)
                     )
                (let ((s (make-subst x y)))
                  (if (> test 0)
                      (begin
                        (printf
                         "    lead to subtitution ~s->~s"
                         x y)
                        (newline)))
                  (G-reduction (apply-subst s (cdr x-vars))
                               (apply-subst s y-vars)
                               (apply-subst s all-vars)
                               (map
                                (lambda (x)
                                  (cons (car x) (apply-subst s (cdr x))))
                                above-env)
                               (map
                                (lambda (x)
                                  (cons (car x) (apply-subst s (cdr x))))
                                below-env)
                               (apply-subst s constraints)
```

```
                              (apply-subst s obv)
                              (compose-subst s subst)
                              (apply-subst s free-vars-in-exp)))
                (G-reduction (cdr x-vars) y-vars all-vars
                             above-env below-env constraints
                             obv subst free-vars-in-exp))))))

(define (G-subsumes x y below-env above-env constraints obv free-vars-in-exp)
  (let* ((cwx (conditioned-with x y constraints above-env below-env))
         (cwy (conditioned-with y x constraints above-env below-env))
         (yes (G-subsumes1 cwx cwy cwx obv free-vars-in-exp)))
    (if (and (> test 4) yes)
        (begin
          (printf "Reducing: cwx = ~s, cwy = ~s" cwx cwy)
          (newline)))
    yes))
(define (G-subsumes1 cwx cwy all-cwx obv free-vars-in-exp)
  (if (null? cwy)
      #t
      (if (null? cwx)
          (G-subsumes1 all-cwx (cdr cwy) all-cwx obv free-vars-in-exp)
          (let* ((x (caar cwx))
                 (y (caar cwy))
                 (above-x (cadar cwx))
                 (above-y (cadar cwy))
                 (below-x (caddar cwx))
                 (below-y (caddar cwy)))
            (and (bt-type-var? x)
                 (not (member x obv))
                 (or (member y below-x)
                     (not (member x free-vars-in-exp)))
                 (subset? (set-minus above-x (list x)) above-y)
                 (subset? (set-minus below-x (list x)) below-y)
                 (G-subsumes1 (cdr cwx) cwy all-cwx obv free-vars-in-exp))))))

(define (S-reduction constraints subst above-env below-env
                     n len type-closure all-vars free-vars-in-exp obv)
  (if (or (null? constraints) (> n len)) ; Finished
      (list subst constraints)
      (let* ((c (car constraints))
             (x (cadr c))
             (y (caddr c))
             (rest-constraints (cdr constraints)))
        (cond
         [(and (not (eq? x y))
               (bt-type-var? x)
               (S-subsumes-a x y above-env
                             rest-constraints type-closure
                             obv free-vars-in-exp))
          (let ((s (make-subst x y)))
            (if (> test 0)
                (begin
                  (printf
                   "    lead to subtitution ~s->~s"
                   x y)
                  (newline)))
            (S-reduction (apply-subst s rest-constraints)
                         (compose-subst s subst)
                         (map
                          (lambda (x)
                            (cons (car x) (apply-subst s (cdr x))))
                          above-env)
                         (map
                          (lambda (x)
                            (cons (car x) (apply-subst s (cdr x))))
                          below-env)
                         0 (- len 1) (apply-subst s type-closure)
                         all-vars (apply-subst s free-vars-in-exp)
                         (apply-subst s obv)))]
         [(and (not (eq? x y))
               (bt-type-var? y)
               (S-subsumes-b x y below-env
                             rest-constraints type-closure
                             obv free-vars-in-exp))
          (let ((s (make-subst y x)))
            (if (> test 0)
                (begin
                  (printf
                   "    lead to subtitution ~s->~s"
                   y x)
                  (newline)))
            (S-reduction (apply-subst s rest-constraints)
                         (compose-subst s subst)
                         (map
                          (lambda (x)
                            (cons (car x) (apply-subst s (cdr x))))
                          above-env)
                         (map
                          (lambda (x)
                            (cons (car x) (apply-subst s (cdr x))))
                          below-env)
                         0 (- len 1) (apply-subst s type-closure)
                         all-vars (apply-subst s free-vars-in-exp)
                         (apply-subst s obv)))]
         [else (S-reduction (append rest-constraints (list c))
                            subst above-env below-env
                            (+ n 1) len type-closure
                            all-vars free-vars-in-exp obv)]))))

(define (S-subsumes-a x y above-env constraints type-closure
                      obv free-vars-in-exp)
  (let* ((cwx (conditioned-with-a x y constraints above-env))
         (cwy (conditioned-with-a y x constraints above-env))
         (yes (S-subsumes-a1 cwx cwy cwx type-closure obv free-vars-in-exp)))
    (if (and (> test 4) yes)
        (begin
          (printf "Reducing down: cwx = ~s, cwy = ~s" cwx cwy)
          (newline)))
    yes))
(define (S-subsumes-a1 cwx cwy all-cwx type-closure obv free-vars-in-exp)
  (if (null? cwy)
      #t
      (if (null? cwx)
          (S-subsumes-a1 all-cwx (cdr cwy) all-cwx
                         type-closure obv free-vars-in-exp)
          (let* ((x (caar cwx))
                 (y (caar cwy))
                 (above-x (cadar cwx))
                 (above-y (cadar cwy)))
            (and (bt-type-var? x)
                 (not (member x free-vars-in-exp))
                 (not (member x obv))
                 (occurs-in? x type-closure)
                 (or (not (bt-type-var? y))
                     (not (occurs-in? y type-closure)))
                 (subset? (set-minus above-x (list x)) above-y)
                 (member y above-x)
                 (subset? (expand-polarity x type-closure) '(down))
                 (S-subsumes-a1 (cdr cwx) cwy all-cwx
                                type-closure obv free-vars-in-exp))))))

(define (S-subsumes-b x y below-env constraints type-closure
                      obv free-vars-in-exp)
  (let* ((cwx (conditioned-with-b x y constraints below-env))
         (cwy (conditioned-with-b y x constraints below-env))
         (yes (S-subsumes-b1 cwx cwy cwx type-closure obv free-vars-in-exp)))
    (if (and (> test 4) yes)
        (begin
          (printf "Reducing up: cwx = ~s, cwy = ~s" cwx cwy)
          (newline)))
    yes))
(define (S-subsumes-b1 cwx cwy all-cwx type-closure obv free-vars-in-exp)
  (if (null? cwy)
      #t
      (if (null? cwx)
          (S-subsumes-b1 all-cwx (cdr cwy) all-cwx
                         type-closure obv free-vars-in-exp)
          (let* ((x (caar cwx))
                 (y (caar cwy))
                 (below-x (cadar cwx))
                 (below-y (cadar cwy)))
            (and (bt-type-var? y)
                 (not (member y obv))
                 (occurs-in? y type-closure)
                 (or (not (bt-type-var? x))
                     (not (occurs-in? x type-closure)))
                 (subset? (set-minus below-y (list y)) below-x)
                 (member x below-y)
                 (subset? (expand-polarity y type-closure) '(up))
                 (S-subsumes-b1 (cdr cwx) cwy all-cwx
                              type-closure obv free-vars-in-exp))))))

(define (Conditioned-reduction constraints intv subst reduced-constraints)
  (if (null? constraints)
      (list subst reduced-constraints)
      (let* ((c (car constraints))
             (x (cadr c))
             (y (caddr c))
             (conditioned (and (= (length c) 4)
                               (not (type-boolean? (cadddr c)))
                               (not (type-integer? (cadddr c)))))
             (rest-constraints (cdr constraints)))
        (if (and conditioned
                 (not (eq? x y))
                 (or (type-product? (cadddr c))
                     (type-function? (cadddr c))
                     (type-rec? (cadddr c))))
            (let ((s (cond [(and (bt-type-var? x)
                                 (bt-type-var? y)
                                 (member x intv))
                            (if (> test 0)
                                (begin
                                  (printf
                                   "    lead to substitution ~s->~s" x y)
                                  (newline)))
                            (make-subst x y)]
                           [(and (bt-type-var? x)
                                 (bt-type-var? y)
                                 (member y intv))
                            (if (> test 0)
                                (begin
                                  (printf
                                   "    lead to substitution ~s->~s" y x)
                                  (newline)))
                            (make-subst y x)]
                           [(bt-type-var? x)
                            (if (> test 0)
                                (begin
                                  (printf
                                   "    lead to substitution ~s->~s" x y)
                                  (newline)))
                            (make-subst x y)]
                           [else
                            (if (> test 0)
                                (begin
                                  (printf
                                   "    lead to substitution ~s->~s" y x)
                                  (newline)))
                            (make-subst y x)])))
              (Conditioned-reduction (apply-subst s rest-constraints)
                                     intv (compose-subst s subst)
                                     (apply-subst s reduced-constraints)))
            (Conditioned-reduction rest-constraints intv subst
                                   (cons c reduced-constraints))))))

(define (find-least-solution constraints)
  (make-dynamic constraints identity 0 (length constraints)))
(define (make-dynamic constraints subst n len)
  (if (or (> n len) (null? constraints))
      subst
      (let* ((c (car constraints))
             (x (cadr c))
             (y (caddr c))
             (rest-constraints (cdr constraints)))
        (cond
         [(and (type-dynamic? x)(bt-type-var? y))
          (let ((s (make-subst y x)))
            (if (> test 0)
                (begin
                  (printf "    lead to substitution:  ~s->~s" y x)
                  (newline)))
            (make-dynamic (apply-subst s rest-constraints)
                          (compose-subst s subst)
                          0 (- len 1)))]
         [(and (bt-type-var? x)
               (type-dynamic? y)
               (= (length c) 4)
               (or (type-product? (cadddr c))
                   (type-function? (cadddr c))
                   (type-rec? (cadddr c))))
          (let ((s (make-subst x y)))
            (if (> test 0)
                (begin
                  (printf "    lead to substitution:  ~s->~s"
                          (cadddr c) y x)
                  (newline)))
            (make-dynamic (apply-subst s rest-constraints)
                          (compose-subst s subst)
                          0 (- len 1)))]
         [else
```

```
          (make-dynamic (append rest-constraints (list c))
                        subst (+ n 1) len)]))))

(define (make-static-constraints all-vars)
  (if (null? all-vars)
      '()
      (cons (list 'leq type-static (car all-vars))
            (make-static-constraints (cdr all-vars)))))
(define (make-dynamic-constraints all-vars)
  (if (null? all-vars)
      '()
      (cons (list 'leq (car all-vars) type-dynamic)
            (make-dynamic-constraints (cdr all-vars)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; C-reduction
(define (eliminate-cycles c intv)
  (let ((graph (make-graph c)))
    (compose-subst*
     (map
      (lambda (c)
        (let ((po (pickone c intv '())))
          (make-subst-for-cycle (car po) (cdr po))))
      (ec1 1 graph graph)))))

(define (ec1 tagno g graph)
  (if (null? g)
      '()
      (let* ((f (car g))
             (v (car f))
             (edges (cadr f))
             (tag (caddr f))
             (cyc (cadr (ec3 v (list v) tagno edges graph))))
        (append cyc (ec1 (+ tagno 1) (cdr g) graph)))))

(define (ec2 ver vl tagno v graph)
  (let* ((vertice (lookup-vertice v graph))
         (edges (car vertice))
         (tag (cadr vertice)))
    (if (eq? v ver) ; A cycle
        (begin
          (if (> test 0) (begin (printf "    Found cycle ~s" (member v vl))
                                (newline)))
          (list graph (list (cdr (member v vl)))))
        (if (= tag 0) ; Not visited
            (if (null? edges)
                (list graph '())
                (let ((new-graph (update-graph graph v tagno)))
                  (ec3 ver vl tagno edges new-graph)))
            ; else (on another cycle)
            (list graph '()))))))

(define (ec3 ver vl tagno edges graph)
  (if (null? edges)
      (list graph '())
      (let* ((v (car edges))
             (res (ec2 ver (append vl (list v)) tagno v graph))
             (new-graph (car res))
             (cyc (cadr res))
             (res1 (ec3 ver vl tagno (cdr edges) new-graph))
             (new-graph1 (car res1))
             (cyc1 (cadr res1)))
        (list
         new-graph1
         (append cyc cyc1)))))

(define (lookup-vertice v graph)
  (if (null? graph)
      '()
      (if (eq? v (caar graph))
          (cdar graph)
          (lookup-vertice v (cdr graph)))))

(define (update-graph graph v tagno)
  (if (null? graph)
      '()
      (if (eq? v (caar graph))
          (cons (list v (cadar graph) tagno)
                (cdr graph))
          (cons (car graph) (update-graph (cdr graph) v tagno)))))

(define (remove-1 g)
  (map (lambda (x)
         (let* ((v (car x))
                (edges (cadr x))
                (tag (caddr x)))
           (if (= tag -1)
               (list v edges 0)
               x)))
       g))

; If there is an element in c which is not in Intv, it is important to pick
; this one.
(define (pickone c intv acc)
  (if (null? c)
      acc
      (let ((e (car c)))
        (if (not (member e intv))
            (cons e (append c acc))
            (pickone (cdr c) intv (cons e acc))))))
```

```
;; Making substitutions from the found cycles
(define (make-subst-for-cycle x l)
  (if (null? l)
      identity
      (let ((x1 (car l)))
        (compose-subst (make-subst x1 x)
                       (make-subst-for-cycle x (cdr l))))))

(define (make-graph c)
  (let ((vars (find-free-vars-in-constraints c)))
    (map (lambda (v)
           (list v
                 (flatten
                  (map
                   (lambda (con)
                     (if (and (eq? (cadr con) v) (bt-type-var? (caddr con)))
                         (list (caddr con))
                         '()))
                   c))
                 0))
         vars)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Misc.
(define (conditioned-with a b constraints above-env below-env)
  (reverse (conditioned-with1 (list (list a
                                          (lookup-env a above-env)
                                          (lookup-env a below-env)))
                              b constraints above-env below-env)))
(define (conditioned-with1 a b constraints above-env below-env)
  (let ((a1 (conditioned-with2 a a b constraints above-env below-env)))
    (if (eq? a1 a)
        a1
        (conditioned-with1 a1 b constraints above-env below-env))))
(define (conditioned-with2 a as b constraints above-env below-env)
  (if (null? a)
      as
      (conditioned-with2 (cdr a)
                         (conditioned-with3 (car a) as b constraints
                                            above-env below-env)
                         b constraints above-env below-env)))
(define (conditioned-with3 a as b constraints above-env below-env)
  (if (null? constraints)
      as
      (let* ((c (car constraints))
             (x (cadr c))
             (y (caddr c)))
        (if (= (length c) 4)
            (cond
             [(and (eq? x a) (not (eq? y b)) (not (member-car y as)))
              (conditioned-with3
               a
               (cons (list (list y
                                 (sort-out-duplicates (lookup-env y above-env))
                                 (sort-out-duplicates (lookup-env y below-env))
                                 ))
                     as)
               b (cdr constraints) above-env below-env)]
             [(and (eq? y a) (not (eq? x b)) (not (member-car x as)))
              (conditioned-with3
               a
               (cons (list (list x
                                 (sort-out-duplicates (lookup-env y above-env))
                                 (sort-out-duplicates (lookup-env y below-env))
                                 ))
                     as)
               b (cdr constraints) above-env below-env)]
             [else (conditioned-with3 a as b (cdr constraints)
                                      above-env below-env)])
            (conditioned-with3 a as b (cdr constraints)
                               above-env below-env)))))

(define (conditioned-with-a a b constraints above-env)
  (reverse (conditioned-with-a1 (list (list a
                                            (lookup-env a above-env)))
                                b constraints above-env)))
(define (conditioned-with-a1 a b constraints above-env)
  (let ((a1 (conditioned-with-a2 a a b constraints above-env)))
    (if (eq? a1 a)
        a1
        (conditioned-with-a1 a1 b constraints above-env))))
(define (conditioned-with-a2 a as b constraints above-env)
  (if (null? a)
      as
      (conditioned-with-a2 (cdr a)
                           (conditioned-with-a3 (car a) as b constraints
                                                above-env)
                           b constraints above-env)))
(define (conditioned-with-a3 a as b constraints above-env)
  (if (null? constraints)
      as
      (let* ((c (car constraints))
             (x (cadr c))
             (y (caddr c)))
        (if (= (length c) 4)
            (cond
             [(and (eq? x a) (not (eq? y b)) (not (member-car y as)))
              (conditioned-with-a3
               a
               (cons (list (list y
                                 (sort-out-duplicates (lookup-env y above-env))
                                 ))
                     as)
               b (cdr constraints) above-env)]
             [(and (eq? y a) (not (eq? x b)) (not (member-car x as)))
              (conditioned-with-a3
               a
               (cons (list (list x
                                 (sort-out-duplicates (lookup-env y above-env))
                                 ))
                     as)
               b (cdr constraints) above-env)]
             [else (conditioned-with-a3 a as b (cdr constraints)
                                        above-env)])
            (conditioned-with-a3 a as b (cdr constraints)
                                 above-env)))))

(define (conditioned-with-b a b constraints below-env)
  (reverse (conditioned-with-b1 (list (list a
                                            (lookup-env a below-env)))
                                b constraints below-env)))
(define (conditioned-with-b1 a b constraints below-env)
  (let ((a1 (conditioned-with-b2 a a b constraints below-env)))
    (if (eq? a1 a)
        a1
        (conditioned-with-b1 a1 b constraints below-env))))
(define (conditioned-with-b2 a as b constraints below-env)
  (if (null? a)
      as
      (conditioned-with-b2 (cdr a)
                           (conditioned-with-b3 (car a) as b constraints
                                                below-env)
                           b constraints below-env)))
(define (conditioned-with-b3 a as b constraints below-env)
  (if (null? constraints)
      as
      (let* ((c (car constraints))
             (x (cadr c))
             (y (caddr c)))
        (if (= (length c) 4)
            (cond
             [(and (eq? x a) (not (eq? y b)) (not (member-car y as)))
              (conditioned-with-b3
               a
               (cons (list (list y
                                 (sort-out-duplicates (lookup-env y below-env))
                                 ))
                     as)
               b (cdr constraints) below-env)]
             [(and (eq? y a) (not (eq? x b)) (not (member-car x as)))
              (conditioned-with-b3
               a
               (cons (list (list x
                                 (sort-out-duplicates (lookup-env y below-env))
                                 ))
                     as)
               b (cdr constraints) below-env)]
             [else (conditioned-with-b3 a as b (cdr constraints)
                                        below-env)])
            (conditioned-with-b3 a as b (cdr constraints)
                                 below-env)))))

(define (compute-above-env vars all-vars constraints)
  (compute-above-env1 vars all-vars constraints '()))
(define (compute-above-env1 vars all-vars constraints env)
  (if (null? vars)
      env
      (let ((x (car vars)))
        (compute-above-env1 (cdr vars) all-vars constraints
                            (cons (list x (above x constraints all-vars))
                                  env)))))
(define (compute-below-env vars all-vars constraints)
  (compute-below-env1 vars all-vars constraints '()))
(define (compute-below-env1 vars all-vars constraints env)
  (if (null? vars)
      env
      (let ((x (car vars)))
        (compute-below-env1 (cdr vars) all-vars constraints
                            (cons (list x (below x constraints all-vars))
                                  env)))))
(define (nowhere-conditioned-G x y s)
  (if (or (null? s) (type-static? x) (type-dynamic? x))
      #t
      (let* ((c (car s))
             (a (cadr c))
             (b (caddr c))
             (conditioned (and (= (length c) 4)
                               (not (type-boolean? (cadddr c)))
                               (not (type-integer? (cadddr c))))))
        (if (and (or (and (eq? x a) (not (eq? y b)))
                     (and (eq? x b) (not (eq? y a))))
                 conditioned)
            #f
            (nowhere-conditioned-G x y (cdr s))))))

(define (reduce-equal constraints)
  (if (null? constraints)
      '()
      (let* ((c (car constraints))
             (x (cadr c))
             (y (caddr c)))
        (if (eq? x y)
            (reduce-equal (cdr constraints))
            (cons c (reduce-equal (cdr constraints)))))))

(define (remove-duplicates-constraints c)
  (remove-duplicates-constraints1 c '()))
(define (remove-duplicates-constraints1 c r)
  (if (null? c)
      r
      (let* ((con (car c))
             (x (cadr con))
             (y (caddr con)))
        (if (or (eq? x y)
                (and (or (eq? x type-static)
                         (eq? y type-dynamic))
                     (= (length con) 3)))
            (remove-duplicates-constraints1 (cdr c) r)
            (letrec ((uin (lambda (r res)
                            (if (null? r)
                                (cons con res)
                                (let* ((con1 (car r))
                                       (x1 (cadr con1)) ; r = (leq y1 y2 ?)::rs
                                       (y1 (caddr con1))
                                       (rs (cdr r)))
                                  (if (or (not (eq? x x1))
                                          (not (eq? y y1)))
                                      (uin rs (cons con1 res))
                                      (if (= (length con) 4)
                                          (cons con (append rs res))
                                          (cons con1 (append rs res)))))))))
              (remove-duplicates-constraints1 (cdr c) (uin r '()))))))))
(define (update-if-necessary x x1 x2 r res)
  (if (null? r)
      (cons x res)
      (let* ((y (car r))
             (y1 (cadr y))     ; r = (leq y1 y2 ?)::rs
             (y2 (caddr y))
             (rs (cdr r)))
      (if (or (not (eq? x1 y1))
              (not (eq? x2 y2)))
          (update-if-necessary x x1 x2 rs (cons y res))
          (if (= (length x) 4)
              (cons x (append rs res))
              (cons y (append rs res)))))))

(define (above x c all-vars)
  (if (type-static? x)
      all-vars
      (above1 (list x) c)))
(define (above1 a c)
  (let ((a1 (above2 a a c)))
    (if (eq? a1 a)
        a1
        (above1 a1 c))))
(define (above2 a as c)
  (if (null? a)
      as
      (above2 (cdr a)
              (above3 (car a) as c)
              c)))
(define (above3 a as c)
  (if (null? c)
      as
      (let ((y (caddar c)))
        (if (and (eq? a (cadar c)) (not (member y as)))
            (above3 a (cons y as) (cdr c))
            (above3 a as (cdr c))))))

(define (below x c all-vars)
  (if (type-dynamic? x)
      all-vars
      (below1 (list x type-static) c)))
(define (below1 a c)
  (let ((a1 (below2 a a c)))
    (if (eq? a1 a)
        a1
        (below1 a1 c))))
(define (below2 a as c)
  (if (null? a)
      as
      (below2 (cdr a) (union (below3 (car a) (list (car a)) c) as) c)))
(define (below3 a as c)
  (if (null? c)
      as
      (if (eq? a (caddar c))
          (below3 a (cons (cadar c) as) (cdr c))
          (below3 a as (cdr c)))))

(define (nowhere-conditioned x s)
  (if (or (null? s) (type-static? x) (type-dynamic? x))
      #t
      (let* ((c (car s))
             (a (cadr c))
             (b (caddr c))
             (conditioned (and (= (length c) 4)
                               (not (type-boolean? (cadddr c)))
                               (not (type-integer? (cadddr c))))))
        (if (and (or (eq? x a) (eq? x b))
                 conditioned)
            #f
            (nowhere-conditioned x (cdr s))))))

(define (expand-polarity alpha t)
  (let ((st (get-std-type t))
        (bt (get-bt-type t)))
    (if (atom? st)
        (if (eq? alpha bt)
            '(up)
            '())
        (if (eq? alpha bt)
            '(noway)
            (if (type-function? st)
                (union (contract-polarity alpha (type-func-operand st))
                       (expand-polarity alpha (type-func-return st)))
                (if (occurs-in? alpha t)
                    '(noway)
                    '())))))))

(define (contract-polarity alpha t)
  (let ((st (get-std-type t))
        (bt (get-bt-type t)))
    (if (std-type-constant? st)
        (if (eq? alpha bt)
            '(down)
            '())
        (if (occurs-in? alpha t)
            '(noway)
            '())))))
```

C.3 Auxiliary Procedures

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                            ;;
;; The Perplex System                                                         ;;
;; Partial Evaluation with Polyvariant Lets for the EXtended lambda calculus ;;
;;                                                                            ;;
;; Christian Mossin                                                           ;;
;;                                                                            ;;
;; This File:                                                                 ;;
;;   Misc. function for BTA                                                   ;;
;;                                                                            ;;
;; Created: Mon Feb 8                                                         ;;
;; Last changed: Thu Jul 15                                                   ;;
;;                                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Expressions
(define (isBoolean? e)
  (eq? (car e) 'boolean))

(define (isInteger? e)
  (eq? (car e) 'integer))

(define (isVar? e)
  (eq? (car e) 'var))

(define (isApplVar? e applenv)
  (and (eq? (car e) 'var) (member (get-var-number e) (map car applenv))))

(define (isIf? e)
  (eq? (car e) 'if))

(define (isLambda? e)
  (eq? (car e) 'lambda))

(define (isApply? e)
  (eq? (car e) 'apply))

(define (isPrimop? e)
  (eq? (car e) 'primop))

(define (isFix? e)
  (eq? (car e) 'fix))

(define (isLet? e)
  (eq? (car e) 'let))

(define (isPair? e)
  (eq? (car e) 'pair))

(define (isFst? e)
  (eq? (car e) 'fst))

(define (isSnd? e)
  (eq? (car e) 'snd))

(define (isLift? e)
  (eq? (car e) 'lift))

(define (isNil? e)
  (eq? (car e) 'nil))

(define (isNull? e)
  (eq? (car e) 'null?))
```

```
(define (isForall? e)
  (eq? (car e) 'forall))

(define (isQuanapp? e)
  (eq? (car e) 'quanapp))

(define get-annotation cadr)

(define get-boolean caddr)

(define get-integer caddr)

(define get-var caddr)

(define (get-var-number e) (cadr (cddr e)))

(define get-conditional caddr)

(define (get-cond-then e) (cadr (cddr e)))

(define (get-cond-else e) (caddr (cddr e)))

(define get-lambda-var caddr)

(define (get-lambda-exp e) (cadr (cddr e)))

(define get-appl-funct caddr)

(define (get-appl-arg e) (cadr (cddr e)))

(define get-primop-op caddr)

(define (get-primop-arg1 e) (cadr (cddr e)))

(define (get-primop-arg2 e) (caddr (cddr e)))

(define get-fix-var caddr)

(define (get-fix-exp e) (cadr (cddr e)))

(define get-let-var cadr)

(define (get-let-exp1 e) (cadr (cdr e)))

(define (get-let-exp2 e) (caddr (cdr e)))

(define get-pair-fst caddr)

(define (get-pair-snd e) (cadr (cddr e)))
```

```
(define get-fst-exp caddr)

(define get-snd-exp caddr)

(define get-null-exp caddr)

(define get-lift-from cadr)

(define (get-lift-to e) (cadr (cdr e)))

(define (get-lift-exp e) (cadr (cddr e)))

(define get-forall-exp caddr)

(define get-forall-id cadr)

(define get-forall-args cadr)

(define get-quanapp-exp cadr)

(define get-quanapp-args caddr)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Environments
(define (lookup-env x env)
  (if (null? env)
      (begin
        (printf "Error in lookup-env Unknown variable: ~s" x)(newline)
        (error 'lookup-env "Unknown variable: ~s" x))
      (if (eq? (caar env) x)
          (cadar env)
          (lookup-env x (cdr env)))))
(define (update-env x type env)
  (cons (list x type) env))

(define empty-appl-env '())

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Selectors and predicates for type variables.
(define type-static
  'static)
(define type-dynamic
  'dynamic)
(define type-integer
  'integer)
(define type-boolean
  'boolean)

(define (get-std-type t) (car t))
(define (get-bt-type t) (cadr t))
```

```
;; Make a normal type variable.
(define (bt-make-var)
  (bt-make-symbol))
(define (std-make-var)
  (std-make-symbol))

(define (make-compound-type std bt)
  (list std bt))
(define (make-fun-type arg res bt)
  (make-compound-type (list 'fun arg res) bt))
(define (make-pair-type fst snd bt)
  (make-compound-type (list 'pair fst snd) bt))
(define (make-rec-type x y)
  (let ((v (std-make-var)))
    (list 'mu v (apply-subst (make-subst x v) y))))
(define (make-forall-type vars constraints type)
  (list 'forall vars constraints type))

;; Type Expression predicates.
(define (type-static? type)
  (eq? type 'static))

(define (type-dynamic? type)
  (eq? type 'dynamic))

(define (type-var? type)
  (or (bt-type-var? type)
      (std-type-var? type)
      (var-type-var? type)
      (forall-type-var? type)))

(define (std-type-var? type)
  (and (symbol? type)
       (let ((type-l (string->list (symbol->string type))))
         (and (> (length type-l) 2)
              (eq? (car type-l) #\s)
              (eq? (cadr type-l) #\t)
              (not (eq? (caddr type-l) #\a))))))

(define (bt-type-var? type)
  (and (symbol? type)
       (let ((type-l (string->list (symbol->string type))))
         (and (> (length type-l) 2)
              (eq? (car type-l) #\b)
              (eq? (cadr type-l) #\t)))))

(define (var-type-var? type)
  (and (symbol? type)
       (let ((type-l (string->list (symbol->string type))))
         (and (> (length type-l) 2)
              (eq? (car type-l) #\v)
              (eq? (cadr type-l) #\a)
              (eq? (caddr type-l) #\r)))))

(define (forall-type-var? type)
  (and (symbol? type)
       (let ((type-l (string->list (symbol->string type))))
         (and (> (length type-l) 2)
              (eq? (car type-l) #\f)
              (eq? (cadr type-l) #\o)
              (eq? (caddr type-l) #\r)
              (eq? (caddr (cdr type-l)) #\a)
              (eq? (caddr (cddr type-l)) #\l)
              (eq? (caddr (cdddr type-l)) #\l)))))

(define (type-boolean? type)
  (eq? type 'boolean))

(define (type-integer? type)
  (eq? type 'integer))

(define (std-type-var? type)
  (and (symbol? type)
       (let ((type-l (string->list (symbol->string type))))
         (and (> (length type-l) 2)
              (eq? (car type-l) #\s)
              (eq? (cadr type-l) #\t)
              (not (eq? (caddr type-l) #\a))))))

(define (std-type-constant? type)
  (or (eq? type 'integer)
      (eq? type 'boolean)))

(define (bt-type-constant? type)
  (or (eq? type 'static)
      (eq? type 'dynamic)))

(define (type-function? type)
  (and (list? type)
       (eq? (car type) 'fun)))

(define (type-forall? type)
  (and (list? type)
       (eq? (car type) 'forall)))

(define (type-product? type)
  (and (list? type)
       (eq? (car type) 'pair)))
```

```
(define (type-rec? type)
  (and (list? type)
       (eq? (car type) 'mu)))

(define type-func-return caddr)

(define type-func-operand cadr)

(define type-fst cadr)

(define type-snd caddr)

(define get-forall-arg cadr)

(define get-forall-constraints caddr)

(define get-forall-result cadddr)

(define get-rec-var cadr)

(define get-rec-type caddr)

(define (make-subst-to-new l)
  (if (null? l)
      identity
      (compose-subst
       (make-subst (car l) (bt-make-symbol))
       (make-subst-to-new (cdr l)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Substitutions
(define identity (lambda (x) x))

(define (make-subst var exp)
  (if (eq? var exp)
      identity
      (lambda (x) (if (eq? x var) exp x))))

(define (compose-subst-old f g)
  (lambda (x)((apply-subst1 f) (g x))))

(define (compose-subst f g)
  (lambda (x)
    (let ((l (g x)))
      (if (atom? l)
          (f l)
          (map (apply-subst1 f) l)))))

(define (compose-subst* s*)
  (if (null? s*)
      identity
      (compose-subst (car s*)
                     (compose-subst* (cdr s*)))))

;(define (apply-subst s l)
;  (printf "Applying:")(newline)
;  (time ((apply-subst1 s) l)))
(define (apply-subst s l)
  ((apply-subst1 s) l))
(define (apply-subst1 s)
  (lambda (l)
    (if (atom? l)
        (s l)
        (map (apply-subst1 s) l))))

;Applying a substitution represented as a list of pairs (an environment)
(define (apply-env s l)
  (if (null? s)
      l
      (apply-env (cdr s) ((apply-env1 (car s)) l))))

(define (apply-env1 s)
  (lambda (l)
    (cond
     [(null? l) '()]
     [(atom? l)
      (if (eq? l (car s))
          (cadr s)
          l)]
     [(list? l)
      (map (apply-env1 s) l)])))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Constraint set
(define empty-constraint-set '())

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; A new gen-sym
(define bt-symbol-nr 0)
(define (bt-make-symbol)
  (set! bt-symbol-nr (1+ bt-symbol-nr))
  (string->symbol (format "bt~a" bt-symbol-nr)))

(define std-symbol-nr 0)
(define (std-make-symbol)
  (set! std-symbol-nr (1+ std-symbol-nr))
  (string->symbol (format "st~a" std-symbol-nr)))

(define forall-symbol-nr 0)
(define (make-forall-symbol)
  (set! forall-symbol-nr (1+ forall-symbol-nr))
  (string->symbol (format "forall~a" forall-symbol-nr)))

(define var-symbol-nr 0)
(define (make-var-symbol)
  (set! var-symbol-nr (1+ var-symbol-nr))
  (string->symbol (format "var~a" var-symbol-nr)))

(define (reset-symbols)
  (set! bt-symbol-nr 0)
  (set! std-symbol-nr 0)
  (set! forall-symbol-nr 0)
  (set! var-symbol-nr 0)
  '())

(define (symbol<? s1 s2)
  (string<? (symbol->string s1)
            (symbol->string s2)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; misc.
(define (reverse l)
  (reverse1 l '()))
(define (reverse1 l acc)
  (if (null? l)
      acc
      (reverse1 (cdr l) (cons (car l) acc))))

(define (append-list l)
  (if (null? l)
      '()
      (append (car l) (append-list (cdr l)))))

(define (remove-duplicates l)
  (if (null? l)
      l
      (let ((x (car l))
            (lr (cdr l)))
        (if (member x lr)
            (remove-duplicates lr)
            (cons x (remove-duplicates lr))))))

(define (flatten l) ; Turns a list of lists into a list of elements
  (if (null? l)
      '()
      (append (car l) (flatten (cdr l)))))

(define (sort-out-consts-and-duplicates l)
  (sort-out-consts-and-duplicates1 l '()))
```

```
(define (sort-out-consts-and-duplicates1 l a)
  (if (null? l)
      a
      (let ((t (car l))
            (lr (cdr l)))
        (if (or (type-static? t)
                (type-dynamic? t)
                (member t lr))
            (sort-out-consts-and-duplicates1 lr a)
            (sort-out-consts-and-duplicates1 lr (cons t a))))))
(define (sort-out-duplicates l)
  (sort-out-duplicates1 l '()))
(define (sort-out-duplicates1 l a)
  (if (null? l)
      a
      (let ((t (car l))
            (lr (cdr l)))
        (if (member t lr)
            (sort-out-duplicates1 lr a)
            (sort-out-duplicates1 lr (cons t a))))))
(define (sort-out-env-duplicates l)
  (sort-out-env-duplicates1 l '()))
(define (sort-out-env-duplicates1 l a)
  (if (null? l)
      a
      (let ((t (car l))
            (lr (cdr l)))
        (if (member-car (car t) lr)
            (sort-out-env-duplicates1 lr a)
            (sort-out-env-duplicates1 lr (cons t a))))))
(define (member-car x xs)
  (and (not (null? xs))
       (or (eq? x (caar xs))
           (member-car x (cdr xs)))))

(define (set-minus l1 l2)
  (if (null? l1)
      '()
      (let ((x (car l1)))
        (if (member x l2)
          (set-minus (cdr l1) l2)
          (cons x (set-minus (cdr l1) l2))))))

(define (union s1 s2)
  (union1 s1 s2 '()))
(define (union1 s1 s2 un)
  (if (null? s1)
      (append un s2)
      (let ((x (car s1)))
        (if (member x s2)
            (union1 (cdr s1) s2 un)
            (union1 (cdr s1) s2 (cons x un))))))

(define (subset? s1 s2)
  (or (null? s1)
      (and (member (car s1) s2)
           (subset? (cdr s1) s2))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Printing programs in LaTeX-able form
(define (pretty-print-latex e)
  (format
   (string-append
    "\\begin{tabbing}\\small~c"
    (pp-latex e 0) "~c"
    "\\end{tabbing}~c")
   #\newline #\newline #\newline))

(define (pp-latex e l)
  (cond
   [(isBoolean? e)
    (string-append (symbol->string (get-boolean e))
                   "$^{" (pp-latex-bt (get-annotation e)) "}$")]
   [(isInteger? e)
    (string-append (number->string (get-integer e))
                   "$^{" (pp-latex-bt (get-annotation e)) "}$")]
   [(isVar? e)
    (string-append (symbol->string (get-var e))
                   "$^{" (pp-latex-bt (get-annotation e)) "}$")]
   [(isIf? e)
    (format
     (string-append (make-indent l) "{\\tt if}$^{ "
                    (pp-latex-bt (get-annotation e))
                    "}$ "
                    (pp-latex (get-conditional e) (+ l 1)) "\\\\~c"
                    (indent (+ l 1))
                    "{\\tt then } "
                    (pp-latex (get-cond-then e) (+ l 1))
                    "\\\\~c"
                    (indent (+ l 1))
                    "{\\tt else } "
                    (pp-latex (get-cond-else e) (+ l 1)))
     #\newline #\newline)]
   [(isLambda? e)
    (format
     (string-append "$\\lambda^{" (pp-latex-bt (get-annotation e)) "}$"
                    (symbol->string (get-var e)) ".~c"
                    (pp-latex (get-lambda-exp e) l))
     #\newline)]
   [(isApply? e)
```

```
 (format
  (string-append "(" (make-indent l) (pp-latex (get-appl-funct e) (+ l 1))
                 "\\\\~c"
                 (indent (+ l 1))
                 "\\appl$^{" (pp-latex-bt (get-annotation e))"}$"
                 (pp-latex (get-appl-arg e) (+ l 1)) ")")
  #\newline)]
[(isPrimop? e)
 (format
  (string-append "(" (make-indent l)(pp-latex (get-primop-arg1 e) (+ l 1))
                 "\\\\~c"
                 (indent (+ l 1))
                 "$" (symbol->string (get-primop-op e))
                 "^{" (pp-latex-bt (get-annotation e))"}$"
                 (pp-latex (get-primop-arg2 e) (+ l 1)) ")")
  #\newline)]
[(isFix? e)
 (format
  (string-append "{\\tt f}" (make-indent l) "{\\tt ix}$^{"
                 (pp-latex-bt (get-annotation e)) "}$"
                 (symbol->string (get-fix-var e)) "."
                 "\\\\~c"
                 (indent (+ l 1))
                 (pp-latex (get-fix-exp e) (+ l 1)))
  #\newline)]
[(isLet? e)
 (format
  (string-append "{\\tt let} " (make-indent l) " "
                 (symbol->string (get-let-var e)) " = "
                 (pp-latex (get-let-exp1 e) l) "\\\\~c"
                 (indent l)
                 "{\\tt in }" (pp-latex (get-let-exp2 e) l))
  #\newline)]
[(isPair? e)
 (format
  (string-append "{\\tt pair}$^{" (pp-latex-bt (get-annotation e)) "}$"
                 (make-indent l)
                 "(" (pp-latex (get-pair-fst e) (+ l 1))
                 "\\\\~c"
                 (indent (+ l 1))
                 "," (pp-latex (get-pair-snd e) (+ l 1)) ")")
  #\newline)]
[(isFst? e)
 (string-append "{\\tt fst}$^{" (pp-latex-bt (get-annotation e)) "}$("
                (pp-latex (get-fst-exp e) l) ")")]
[(isSnd? e)
 (string-append "{\\tt snd}$^{" (pp-latex-bt (get-annotation e)) "}$("
                (pp-latex (get-snd-exp e) l) ")")]
[(isLift? e)
 (string-append "[$" (pp-latex-bt (get-lift-from e)) "\\coercesto"
                   (pp-latex-bt (get-lift-to e)) "$] "
                   (pp-latex (get-lift-exp e) l))]
   [(isNil? e)
    "{\\tt nil}"]
   [(isNull? e)
    (string-append "{\\tt null?}$^{" (pp-latex-bt (get-annotation e)) "}$("
                   (pp-latex (get-null-exp e) l) ")")]
   [(and (isForall? e) pp-with-constraints)
    (let ((ll (get-forall-args e)))
      (format
       (string-append "{\\large $\\Lambda$}" (make-indent l)
                      "$\\left(\\mynomath{"
                      "\\begin{tabular}{l}" (pp-latex-bts (car ll))
                      "\\end{tabular}}\\right)$"
                      "\\mynomath{"
                      "\\raisebox{1ex}{$\\left(\\mynomath{"
                      "\\begin{tabular}{l}"
                      (pp-latex-constraints (cadr ll))
                      "\\end{tabular}}\\right)$}}.\\\\~c"
                      (indent (+ l 1))
                      (pp-latex (get-forall-exp e) (+ l 1)))
       #\newline))]
   [(and (isForall? e) (not pp-with-constraints))
    (let ((ll (get-forall-args e)))
      (format
       (string-append "{\\large $\\Lambda$}$\\left(\\mynomath{"
                      "\\begin{tabular}{l}" (pp-latex-bts (car ll))
                      "\\end{tabular}}\\right)$"
                      ".\\\\~c"
                      (indent (+ l 1))
                      (pp-latex (get-forall-exp e) (+ l 1)))
       #\newline))]
   [(isQuanapp? e)
    (string-append "(" (pp-latex (get-quanapp-exp e) l)
                   "\\quanapp$\\left(\\mynomath{\\begin{tabular}{l}"
                   (pp-latex-bts (get-quanapp-args e))
                   "\\end{tabular}}\\right)$)")]
   ))

(define (indent l)
  (if (> l 10)
      (string-append "\\hspace{2mm}" (indent (- l 1)))
      (indent1 l)))

(define (indent1 l)
  (if (= l 0)
      ""
      (string-append "\\> " (indent1 (- l 1)))))
(define (make-indent l)
  (if (> l 9)
      ""
      "\\="))

(define (pp-latex-bts bs)
  (pp-latex-bts1 bs 0))
(define (pp-latex-bts1 bs n)
  (if (null? bs)
      ""
      (if (= n 8)
          (format
           (string-append "\\\\~c$" (pp-latex-bt (car bs)) "$"
                          (pp-latex-bts1 (cdr bs) 1))
           #\newline)
          (string-append "$" (pp-latex-bt (car bs)) "$"
                         (pp-latex-bts1 (cdr bs) (+ n 1))))))

(define (pp-latex-bt b)
  (cond
   [(type-static? b)
    "\\static"]
   [(type-dynamic? b)
    "\\dynamic"]
   [else   ; A type-variable
    (let ((s (symbol->string b)))
      (string-append "\\beta_{" (substring s 2 (string-length s)) "}"))]))

(define (print-constraints-latex s)
  (if (string? s)
      ""
      (string-append "Cannot solve: " (pp-latex-constraints s 0))))

(define (pp-latex-constraints s)
  (pp-latex-constraints1 s 0))
(define (pp-latex-constraints1 s n)
  (if (null? s)
      ""
      (if (= n 3)
          (format
           (string-append "\\\\~c{\\footnotesize($" (pp-latex-bt (cadar s))
                          "$$\\leq$$" (pp-latex-bt (caddar s)) "$"
                          (if (= (length (car s)) 4)
                              (string-append ","
                                             (pp-latex-st-short
                                              (caddr (cdar s)) 0))
                              "")
                          ")} " (pp-latex-constraints1 (cdr s) 1))
           #\newline)
          (string-append "{\\footnotesize($" (pp-latex-bt (cadar s))
                         "$$\\leq$$" (pp-latex-bt (caddar s)) "$"
                         (if (= (length (car s)) 4)
                            (string-append ","
                                           (pp-latex-st-short
                                            (caddr (cdar s)) 0))
                            "")
                        ")} " (pp-latex-constraints1 (cdr s) (+ n 1))))))

(define (print-type-latex t)
  (format
   (string-append
    "\\noindent Type:~c\\begin{tabbing}\\small~c"
    (pp-latex-type t 0)
    "~c"
    "\\end{tabbing}~c")
   #\newline #\newline #\newline #\newline))

(define (pp-latex-type t l)
  (string-append "(" (pp-latex-st (car t) l)
                 ",$" (pp-latex-bt (cadr t)) "$)"))

(define (pp-latex-types t)
  (if (null? t)
      ""
      (string-append "(" (pp-latex-st (caar t) 0) "$\\mapsto$"
                     (pp-latex-st (cadar t) 0) ")"
                     (format "\\\\~c" #\newline)
                     (pp-latex-types (cdr t)))))

(define (pp-latex-st st l)
  (cond
   [(type-boolean? st)
    "{\\tt bool}"]
   [(type-integer? st)
    "{\\tt int}"]
   [(type-rec? st)
    (format
     (string-append "$\\mu$" (pp-latex-st (get-rec-var st) l)
                    "." (pp-latex-st (get-rec-type st) l)))]
   [(type-product? st)
    (format
     (string-append "(" (pp-latex-type (type-fst st) l)
                    "$\\times$"
                    (pp-latex-type (type-snd st) l) ")"))]
   [(type-function? st)
    (format
     (string-append "(" (pp-latex-type (type-func-operand st) l)
                    "$\\rightarrow$"
                    (pp-latex-type (type-func-return st) l) ")"))]
   [(std-type-var? st)
    (let ((s (symbol->string st)))
      (string-append "$\\tau_{" (substring s 2 (string-length s)) "}$"))]))
```

```
(define (pp-latex-st-short st l)
  (cond
   [(type-boolean? st)
    "{\\tt bool}"]
   [(type-integer? st)
    "{\\tt int}"]
   [(type-product? st)
    "({\\tiny $\\sqcup$}$\\times${\\tiny $\\sqcup$},{\\tiny $\\sqcup$})"]
   [(type-function? st)
    "({\\tiny $\\sqcup$}$\\rightarrow${\\tiny $\\sqcup$},{\\tiny $\\sqcup$})"]
   [(std-type-var? st)
    (let ((s (symbol->string st)))
      (string-append "$\\tau_{" (substring s 2 (string-length s)) "}$"))]
   [(type-rec? st)
    (string-append "$\\mu$" (symbol->string (get-rec-var st)))]))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; File handling

(define (writef obj file)
  (begin
    (if (file-exists? file)(delete-file file))
    (let ((port (open-output-file (eval file))))
      (begin
        (display obj port)
        (newline port)
        (close-output-port port)))))

(define (writefpp obj file)
  (begin
    (if (file-exists? file)(delete-file file))
    (let ((port (open-output-file (eval file))))
      (begin
        (pretty-print obj port)
        (newline port)
        (close-output-port port)))))

(define (remove-file file)
  (system (string-append "/bin/rm " (eval file))))

(define (file->item file)
  (let ((port@ (open-input-file (eval file))))
    (let ((res (read port@)))
      (close-input-port port@)
      res)))
```

C.4 Specializer

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                            ;;
;; The Perplex System                                                         ;;
;; Partial Evaluation with Polyvariant Lets for the EXtended lambda calculus ;;
;;                                                                            ;;
;; March 1993                                                                 ;;
;; Christian Mossin                                                           ;;
;;                                                                            ;;
;; This File:                                                                 ;;
;;   Specializer                                                              ;;
;;                                                                            ;;
;; Created: Mon Feb 8                                                         ;;
;; Last changed: Thu Jul 15                                                   ;;
;;                                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (specialize e env)
  (let ((res-exp (specialize1 e env '())))
    res-exp))

(define (specialize1 e env bt-env)
  (if (and (> test 0)
           (isApply? e)
           (or (isVar? (get-appl-funct e))
               (isFix? (get-appl-funct e))))
      (begin (printf "Applying ~s"
                     (get-fix-var (get-appl-funct e)))
             (newline)))
  (if (and (> test 3)
           (isQuanapp? e))
      (begin (printf "Let-bound variable ~s" (get-quanapp-exp e))(newline)
             (let ((quanargs
                    (get-quanapp-args e))
                   (form-args
                     (car (get-forall-args
                           (lookup-env (get-var (get-quanapp-exp e)) env)))))
               (printf "Formal args: ~s" form-args)
               (newline)
               (printf "Actual args: ~s" quanargs)
               (newline)
               (printf "Which are:   ~s"
                       (map (lambda (b) (if-var-then-lookup b bt-env))
                            quanargs))
               (newline))))
  (cond
   [(isBoolean? e)
    (list 'boolean (get-boolean e))]
```

```
[(isInteger? e)
 (list 'integer (get-integer e))]
[(isVar? e)
 (lookup-env (get-var e) env)]
[(isIf? e)
 (let ((ann (if-var-then-lookup (get-annotation e) bt-env))
       (cond-s (specialize1 (get-conditional e) env bt-env)))
   (if (> test 1) (begin (printf "if~s ~s" ann cond-s)(newline)))
   (if (type-static? ann)
       (if (eq? (get-int-boolean cond-s) 'true)
           (specialize1 (get-cond-then e) env bt-env)
           (specialize1 (get-cond-else e) env bt-env))
       (if (type-dynamic? ann)
           (list 'if       ; else dynamic
                 cond-s
                 (specialize1 (get-cond-then e) env bt-env)
                 (specialize1 (get-cond-else e) env bt-env))
           (error 'if "Annotation ~s not bound" ann))
       ))]
[(isLambda? e)
 (let ((ann (if-var-then-lookup (get-annotation e) bt-env)))
   (if (type-static? ann)
       (list 'closure e env bt-env)
       (if (type-dynamic? ann)
           (let* ((var (get-lambda-var e))
                  (fresh-var (make-fresh var)))
             (list 'lambda    ; else dynamic
                   fresh-var
                   (specialize1 (get-lambda-exp e)
                                (update-env var (list 'var fresh-var) env)
                                bt-env)))
           (error 'lambda "Annotation ~s not bound" ann))
       ))]
[(isApply? e)
 (let ((ann (if-var-then-lookup (get-annotation e) bt-env))
       (funct-s (specialize1 (get-appl-funct e) env bt-env))
       (arg-s (specialize1 (get-appl-arg e) env bt-env)))
   (if (type-static? ann)
       (let* ((new-env (caddr funct-s))
              (lam (cadr funct-s))
              (new-e (get-lambda-exp lam))
              (new-v (get-lambda-var lam))
              (new-bt-env (cadddr funct-s)))
         (specialize1 new-e (update-env new-v arg-s new-env) new-bt-env))
       (if (type-dynamic? ann)
           (list 'apply     ; else dynamic
                 funct-s
                 arg-s)
           (error 'apply "Annotation ~s not bound" ann))
       ))]
```

```
[(isPrimop? e)
 (let* ((ann (if-var-then-lookup (get-annotation e) bt-env))
        (v1 (specialize1 (get-primop-arg1 e) env bt-env))
        (v2 (specialize1 (get-primop-arg2 e) env bt-env))
        (op (get-primop-op e))
        (text (if (> test 2)
                  (begin (printf "result of ~s~s ~s ~s" op ann v1 v2)
                         (newline)))
              ))
   (if (type-static? ann)
       (let ((res ((eval (lookup-env op (primop-env-spec))) v1 v2)))
         (if (> test 2) (begin (printf " is ~s" res)(newline)))
         res)
       (if (type-dynamic? ann)
           (list 'primop op v1 v2) ; else dynamic
           (error 'primop "Annotation ~s not bound" ann))
       ))]
[(isFix? e)
 (let ((ann (if-var-then-lookup (get-annotation e) bt-env)))
   (if (type-static? ann)
       (let ((var (get-fix-var e))
             (exp (get-fix-exp e)))
         (specialize1 (substitute-for-var var e exp) env bt-env))
       (if (type-dynamic? ann)
           (let* ((var (get-fix-var e))
                  (fresh-var (make-fresh var)))
             (list 'fix ; else dynamic
                   fresh-var
                   (specialize1 (get-fix-exp e)
                                (update-env var (list 'var fresh-var) env)
                                bt-env)))
           (error 'fix "Annotation ~s not bound" ann))
       ))]
[(isLet? e)
 (let* ((var (get-let-var e))
        (e1 (get-let-exp1 e))
        (e2 (get-let-exp2 e))
        (new-env (update-env var e1 env)))
   (specialize1 e2 new-env bt-env))]
[(isPair? e)
 (let ((ann (if-var-then-lookup (get-annotation e) bt-env))
       (v1 (specialize1 (get-pair-fst e) env bt-env))
       (v2 (specialize1 (get-pair-snd e) env bt-env)))
   (if (type-static? ann)
       (list 'pair v1 v2)
       (if (type-dynamic? ann)
           (list 'pair v1 v2) ; else dynamic
           (error 'primop "Annotation ~s not bound" ann))
       ))]
[(isFst? e)
```

```
    (let ((ann (if-var-then-lookup (get-annotation e) bt-env))
          (v (specialize1 (get-fst-exp e) env bt-env)))
      (if (type-static? ann)
          (cadr v)
          (if (type-dynamic? ann)
              (list 'fst v) ; else dynamic
              (error 'fst "Annotation ~s not bound" ann))
          ))]
   [(isSnd? e)
    (let ((ann (if-var-then-lookup (get-annotation e) bt-env))
          (v (specialize1 (get-snd-exp e) env bt-env)))
      (if (type-static? ann)
          (caddr v)
          (if (type-dynamic? ann)
              (list 'snd v) ; else dynamic
              (error 'snd "Annotation ~s not bound" ann))
          ))]
   [(isLift? e)
    (let ((from (if-var-then-lookup (get-lift-from e) bt-env))
          (to (if-var-then-lookup (get-lift-to e) bt-env))
          (v (specialize1 (get-lift-exp e) env bt-env))
          )
      v)]
   [(isNil? e)
    '(nil)]
   [(isNull? e)
    (let ((ann (if-var-then-lookup (get-annotation e) bt-env))
          (v (specialize1 (get-null-exp e) env bt-env)))
      (if (> test 2) (begin (printf "Null?~s ~s" ann v)(newline)))
      (if (type-static? ann)
          (list 'boolean (if (eq? (car v) 'nil) 'true 'false))
          (list 'null? v) ; else dynamic
          ))]
   [(isQuanapp? e)
    (let* ((exp (get-quanapp-exp e))
           (act-args (get-quanapp-args e))
           (forall (lookup-env (get-var exp) env))
           (e1 (get-forall-exp forall))
           (for-args (car (get-forall-args forall)))
           (new-bt-env (update-env-list for-args act-args bt-env))
           )
      (specialize1 e1 env new-bt-env))]
   ))

(define (if-var-then-lookup b bt-env)
  (if (and (symbol? b) (bt-type-var? b))
      (if-var-then-lookup (bt-lookup-env b bt-env) bt-env)
      b))

(define (bt-lookup-env b bt-env)
  (if (null? bt-env)
      (begin
        (printf "ERROR!!!!!")(newline)
        (error 'bt-lookup-env "Either is ~s not there or only to itself" b))
      (if (and (eq? b (caar bt-env)) (not (eq? (caar bt-env) (cadar bt-env))))
          (cadar bt-env)
          (bt-lookup-env b (cdr bt-env)))))

(define (update-env-list for-args act-args bt-env)
  (if (or (null? for-args) (null? for-args))
      (if (and (null? for-args) (null? for-args))
          bt-env
          (error 'update-env-list
                 "Formal and actual parameter lists not of equal length"))
      (begin
        (update-env-list (cdr for-args)
                         (cdr act-args)
                         (update-env (car for-args)
                                     (if-var-then-lookup (car act-args) bt-env)
                                     bt-env)))))

(define (substitute-for-var v e exp)
  (cond
   [(null? exp) '()]
   [(atom? exp)
    exp]
   [(and (isVar? exp) (eq? (get-var exp) v))
    e]
   [(and (isFix? exp) (eq? (get-fix-var exp) v))
    exp]
   [(list? exp)
    (map (lambda (sub-exp) (substitute-for-var v e sub-exp)) exp)]))

(define var-nr 0)
(define (make-fresh v)
  (set! var-nr (1+ var-nr))
  (string->symbol (format "~s~a" v var-nr)))


(define (primop-env-spec) '((+ (lambda (x y)
                                  (if (and (isInteger? x) (isInteger? y))
                                      (list
                                       'integer
                                       (+ (get-int-integer x)
                                          (get-int-integer y))))))
                             (- (lambda (x y)
                                  (if (and (isInteger? x) (isInteger? y))
                                      (list
                                       'integer
                                       (- (get-int-integer x)
             (get-int-integer y))))))
(* (lambda (x y)
     (if (and (isInteger? x) (isInteger? y))
         (list
          'integer
          (* (get-int-integer x)
             (get-int-integer y))))))
(/ (lambda (x y)
     (if (and (isInteger? x) (isInteger? y))
         (list
          'integer
          (/ (get-int-integer x)
             (get-int-integer y))))))
(and (lambda (x y)
     (if (and (isBoolean? x) (isBoolean? y))
         (list
          'boolean
          (if (and (eq? (get-int-boolean x) 'true)
                   (eq? (get-int-boolean y) 'true))
              'true
              'false)))))
(or (lambda (x y)
     (if (and (isBoolean? x) (isBoolean? y))
         (list
          'boolean
          (if (or (eq? (get-int-boolean x) 'true)
                  (eq? (get-int-boolean y) 'true))
              'true
              'false)))))
(> (lambda (x y)
     (if (and (isInteger? x) (isInteger? y))
         (list
          'boolean
          (if (> (get-int-integer x)
                 (get-int-integer y))
              'true
              'false)))))
(< (lambda (x y)
     (if (and (isInteger? x) (isInteger? y))
         (list
          'boolean
          (if (< (get-int-integer x)
                 (get-int-integer y))
              'true
              'false)))))
(= (lambda (x y)
     (if (and (isInteger? x) (isInteger? y))
         (list
          'boolean
          (if (= (get-int-integer x)
                                                  (get-int-integer y))
                                               'true
                                               'false)))))))
```

C.5 Interpreter

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                          ;;
;; The Perplex System                                                       ;;
;; Partial Evaluation with Polyvariant Lets for the EXtended lambda calculus ;;
;;                                                                          ;;
;; Christian Mossin                                                         ;;
;;                                                                          ;;
;; This File:                                                               ;;
;;   Interpreter                                                            ;;
;;                                                                          ;;
;; Created: Wed Apr 8                                                       ;;
;; Last changed: Thu Jul 15                                                 ;;
;;                                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (interpret e env)
  (cond
   [(isBoolean? e)
    e]
   [(isInteger? e)
    e]
   [(isVar? e)
    (lookup-env (get-int-var e) env)]
   [(isIf? e)
    (let ((cond-s (interpret (get-int-conditional e) env)))
      (if (get-int-boolean cond-s)
          (interpret (get-int-cond-then e) env)
          (interpret (get-int-cond-else e) env))
      )]
   [(isLambda? e)
    (list e env)]
   [(isApply? e)
    (let* ((funct-s (interpret (get-int-appl-funct e) env))
           (arg-s (interpret (get-int-appl-arg e) env))
           (new-env (cadr funct-s))
           (lam (car funct-s))
           (new-e (get-int-lambda-exp lam))
           (new-v (get-int-lambda-var lam)))
      (interpret new-e (update-env new-v arg-s new-env)))]
   [(isPrimop? e)
    (let* ((v1 (interpret (get-int-primop-arg1 e) env))
           (v2 (interpret (get-int-primop-arg2 e) env))
           (op (get-int-primop-op e))
           )
      (let ((res ((eval (lookup-env op primop-env-int)) v1 v2)))
        res))]
   [(isFix? e)
    (let ((var (get-int-fix-var e))
          (exp (get-int-fix-exp e)))
      (interpret (substitute-int-for-var var e exp) env))]
   [(isLet? e)
    (let* ((var (get-int-let-var e))
           (e1 (interpret (get-int-let-exp1 e) env))
           (e2 (get-int-let-exp2 e))
           (new-env (update-env var e1 env)))
      (interpret e2 new-env))]
   [(isPair? e)
    (let ((v1 (interpret (get-int-pair-fst e) env))
          (v2 (interpret (get-int-pair-snd e) env)))
      (list 'pair v1 v2))]
   [(isFst? e)
    (let ((v (interpret (get-int-fst-exp e) env)))
      (cadr v))]
   [(isSnd? e)
    (let ((v (interpret (get-int-snd-exp e) env)))
      (caddr v))]
   [(isNil? e)
    '(nil)]
   [(isNull? e)
    (let ((v (interpret (get-int-null-exp e) env)))
      (list 'boolean (eq? (car v) 'nil)))]))

(define (substitute-int-for-var v e exp)
  (cond
   [(null? exp) '()]
   [(atom? exp)
    exp]
   [(and (isFix? exp) (eq? (get-int-fix-var exp) v))
    exp]
   [(and (isVar? exp) (eq? (get-int-var exp) v))
    e]
   [(list? exp)
    (map (lambda (sub-exp) (substitute-int-for-var v e sub-exp)) exp)]))

(define primop-env-int '((+ (lambda (x y)
                              (if (and (isInteger? x) (isInteger? y))
                                  (list
                                   'integer
                                   (+ (get-int-integer x)(get-int-integer y)))
                                  (error
                                   'primop
```

```
                "Type-error: + applied to non integer"))
           ))
      (- (lambda (x y)
           (if (and (isInteger? x) (isInteger? y))
               (list
                'integer
                (- (get-int-integer x)(get-int-integer y)))
               (error
                'primop
                "Type-error: - applied to non integer"))
           ))
      (* (lambda (x y)
           (if (and (isInteger? x) (isInteger? y))
               (list
                'integer
                (* (get-int-integer x)(get-int-integer y)))
               (error
                'primop
                "Type-error: * applied to non integer"))
           ))
      (/ (lambda (x y)
           (if (and (isInteger? x) (isInteger? y))
               (list
                'integer
                (/ (get-int-integer x)(get-int-integer y)))
               (error
                'primop
                "Type-error: / applied to non integer"))
           ))
      (and (lambda (x y)
             (if (and (isBoolean? x) (isBoolean? y))
                 (list
                  'boolean
                  (and (get-int-boolean x)
                       (get-int-boolean y)))
                 (error
                  'primop
                  "Type-error: and applied to non integer"))
             ))
      (or (lambda (x y)
            (if (and (isBoolean? x) (isBoolean? y))
                (list
                 'boolean
                 (+ (get-int-boolean x)(get-int-boolean y)))
                (error
                 'primop
                 "Type-error: or applied to non integer"))
            ))
      (> (lambda (x y)
           (if (and (isInteger? x) (isInteger? y))
                                   (list
                                    'boolean
                                    (> (get-int-integer x)(get-int-integer y)))
                                   (error
                                    'primop
                                    "Type-error: > applied to non integer"))
                               ))
                         (< (lambda (x y)
                               (if (and (isInteger? x) (isInteger? y))
                                   (list
                                    'boolean
                                    (< (get-int-integer x)(get-int-integer y)))
                                   (error
                                    'primop
                                    "Type-error: < applied to non integer"))
                               ))
                         (= (lambda (x y)
                               (if (and (isInteger? x) (isInteger? y))
                                   (list
                                    'boolean
                                    (= (get-int-integer x)(get-int-integer y)))
                                   (error
                                    'primop
                                    "Type-error: = applied to non integer"))
                               ))))

(define get-int-boolean cadr)

(define get-int-integer cadr)

(define get-int-var cadr)

(define (get-int-var-number e) (cadr (cdr e)))

(define get-int-conditional cadr)

(define (get-int-cond-then e) (cadr (cdr e)))

(define (get-int-cond-else e) (caddr (cdr e)))

(define get-int-lambda-var cadr)

(define (get-int-lambda-exp e) (cadr (cdr e)))

(define get-int-appl-funct cadr)

(define (get-int-appl-arg e) (cadr (cdr e)))

(define get-int-primop-op cadr)
```

```
(define (get-int-primop-arg1 e) (cadr (cdr e)))

(define (get-int-primop-arg2 e) (caddr (cdr e)))

(define get-int-fix-var cadr)

(define (get-int-fix-exp e) (cadr (cdr e)))

(define get-int-let-var cadr)

(define (get-int-let-exp1 e) (cadr (cdr e)))

(define (get-int-let-exp2 e) (caddr (cdr e)))

(define get-int-pair-fst cadr)

(define (get-int-pair-snd e) (cadr (cdr e)))

(define get-int-fst-exp cadr)

(define get-int-snd-exp cadr)

(define get-int-null-exp cadr)
```

C.6 Post-reduction

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                            ;;
;; The Perplex System                                                         ;;
;; Partial Evaluation with Polyvariant Lets for the EXtended lambda calculus ;;
;;                                                                            ;;
;; Christian Mossin                                                           ;;
;;                                                                            ;;
;; This File:                                                                 ;;
;;   Post reducer                                                             ;;
;;                                                                            ;;
;; Created: Wed Apr 13                                                        ;;
;; Last changed: Thu Jul 15                                                   ;;
;;                                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define post-reduce
  (lambda (e)
    (cond
     [(atom? e) e]
     [(isFst? e)
      (let* ((e1 (get-int-fst-exp e))
             (e1-new (post-reduce e1)))
        (if (isPair? e1-new)
            (get-int-pair-fst e1-new)
            (list 'fst e1-new)))]
     [(isSnd? e)
      (let* ((e1 (get-int-snd-exp e))
             (e1-new (post-reduce e1)))
        (if (isPair? e1-new)
            (get-int-pair-snd e1-new)
            (list 'snd e1-new)))]
     [else
      (cons (car e) (map post-reduce (cdr e)))])))
```

Appendix D

Examples

This appendix shows the examples used for testing the speed of the binding time analysis in chapter 11. The original and residual programs are given in concrete syntax, which should be understandable.

D.1 List of Factorial

We present a program for computing lists of factorials $(n!, (n-1)!, \ldots, 1!)$.

D.1.1 Original Program

```
(let facs
    (lambda x
      (let map (fix m
                    (lambda f
                      (lambda l
                        (if (null? (var l))
                            (nil)
                            (pair (apply (var f) (fst (var l)))
                                  (apply (apply (var m) (var f))
                                         (snd (var l)))
                                  )))))
        (let fac (fix f
                      (lambda n
                        (if (primop = (var n) (integer 0))
                            (integer 1)
                            (primop *
                                    (var n)
                                    (apply (var f)
                                           (primop -
                                                   (var n)
                                                   (integer 1)))))))
          (let mklist
              (fix ml
                   (lambda x
                     (if (primop = (var x) (integer 0))
                         (nil)
                         (pair (var x)
                               (apply (var ml)
                                      (primop - (var x)
                                                (integer 1)))))))
            (apply (apply (var map) (var fac))
                   (apply (var mklist) (var x)))))))
  (pair (apply (var facs) (integer 5))
        (apply (var facs) (var d))))
```

D.1.2 Annotated Program

The number of parameters to the functions in this program was reduced by the reduction presented in chapter 10 as follows:

	Without reduction	With reduction	*FreeVars*(κ)–*FreeVars*(A)
map	11	9	9
fac	7	3	3
mklist	7	4	4
facs	21	5	4

The number of variables in *FreeVars*(κ)–*FreeVars*(A) is shown to give a lower limit on the number of binding time parameters (though this limit is not always achievable due to the way the variables are used as annotations).

```
let facs = $\Lambda\left( \beta_{220}\beta_{168}\beta_{161}\beta_{169}\beta_{164} \right).$
  $\lambda^{\beta_{169}}$x. let map = $\Lambda\left( \begin{array}{l} \beta_{24}\beta_{55}\beta_{34}\beta_{51}\beta_{56}\beta_{35}\beta_{29}\beta_{45} \\ \beta_{58} \end{array} \right).$
          fix$^{\beta_{58}}$m.
           $\lambda^{\beta_{58}}$f. $\lambda^{\beta_{56}}$l. if$^{\beta_{51}}$ null?$^{\beta_{51}}$(l$^{\beta_{51}}$)
                        then  nil
                        else  pair$^{\beta_{55}}$([$\beta_{35}\rightsquigarrow\beta_{24}$] (f$^{\beta_{45}}$
                                         @$^{\beta_{45}}$[$\beta_{34}\rightsquigarrow\beta_{29}$] fst$^{\beta_{51}}$(l$^{\beta_{51}}$))
                                   ,((m$^{\beta_{58}}$
                                      @$^{\beta_{58}}$f$^{\beta_{45}}$)
                                     @$^{\beta_{56}}$snd$^{\beta_{51}}$(l$^{\beta_{51}}$)))
  in let fac = $\Lambda\left( \beta_{64}\beta_{92}\beta_{101} \right).$
       fix$^{\beta_{101}}$f.
        $\lambda^{\beta_{101}}$n. if$^{\beta_{92}}$ (n$^{\beta_{92}}$
                    =$^{\beta_{92}}$[S$\rightsquigarrow\beta_{92}$] 0$^{\mathsf{S}}$)
                then  [S$\rightsquigarrow\beta_{64}$] 1$^{\mathsf{S}}$
                else  ([$\beta_{92}\rightsquigarrow\beta_{64}$] n$^{\beta_{92}}$
                      *$^{\beta_{64}}$(f$^{\beta_{101}}$
                          @$^{\beta_{101}}$(n$^{\beta_{92}}$
                              −$^{\beta_{92}}$[S$\rightsquigarrow\beta_{92}$] 1$^{\mathsf{S}}$)))
  in let mklist = $\Lambda\left( \beta_{123}\beta_{142}\beta_{135}\beta_{144} \right).$
       fix$^{\beta_{144}}$ml.
        $\lambda^{\beta_{144}}$x. if$^{\beta_{135}}$ (x$^{\beta_{135}}$
                    =$^{\beta_{135}}$[S$\rightsquigarrow\beta_{135}$] 0$^{\mathsf{S}}$)
```

$$
\begin{array}{l}
\qquad\qquad\qquad\texttt{then}\ \ \texttt{nil} \\
\qquad\qquad\qquad\texttt{else}\ \ \texttt{pair}^{\beta_{142}}([\beta_{135}\rightsquigarrow\beta_{123}]\ \mathrm{x}^{\beta_{135}} \\
\qquad\qquad\qquad\qquad\qquad ,(\mathrm{ml}^{\beta_{144}} \\
\qquad\qquad\qquad\qquad\qquad\ @^{\beta_{144}}(\mathrm{x}^{\beta_{135}} \\
\qquad\qquad\qquad\qquad\qquad\qquad -^{\beta_{135}}[\mathsf{S}\rightsquigarrow\beta_{135}]\ 1^{\mathsf{S}}))) \\
\quad \texttt{in}\ (((\mathrm{map}^{\mathsf{S}}\diamond\begin{pmatrix}\beta_{220}\beta_{168}\beta_{164}\beta_{164}\mathsf{S}\beta_{164}\beta_{164}\mathsf{S} \\ \mathsf{S}\end{pmatrix}) \\
\qquad @^{\mathsf{S}}(\mathrm{fac}^{\mathsf{S}}\diamond\big(\ \beta_{164}\beta_{164}\mathsf{S}\ \big))) \\
\qquad @^{\mathsf{S}}((\mathrm{mklist}^{\mathsf{S}}\diamond\big(\ \beta_{164}\beta_{164}\beta_{161}\mathsf{S}\ \big)) \\
\qquad\quad @^{\mathsf{S}}\mathrm{x}^{\beta_{161}})) \\
\texttt{in pair}^{\mathsf{D}}(((\mathrm{facs}^{\mathsf{S}}\diamond\big(\ \mathsf{DDSSS}\ \big)) \\
\qquad\quad @^{\mathsf{S}}5^{\mathsf{S}}) \\
\qquad ,((\mathrm{facs}^{\mathsf{S}}\diamond\big(\ \mathsf{DDDSD}\ \big)) \\
\qquad\quad @^{\mathsf{S}}\mathrm{d}^{\mathsf{D}})) \\
\end{array}
$$

Specialization of the list of factorials program will fail to terminate.

D.2 MP-interpreter

D.2.1 Original Program

```
(let generalize1 (lambda e (if (boolean true)
                             (var e)
                             (if (primop = (var d) (integer 1))
                                 (lambda x (nil))
                                 (lambda x (nil)))))
(let length (fix len
                 (lambda l
                   (if (null? (var l))
                       (integer 0)
                       (primop + (integer 1)
                               (apply (var len) (snd (var l)))))))
(let append (fix app
                 (lambda l1
                   (lambda l2
                     (if (null? (var l1))
                         (var l2)
                         (pair (fst (var l1))
                               (apply (apply (var app)
                                             (snd (var l1)))
                                      (var l2)))))))
(let myTrue (integer 1)
(let myFalse (integer 0)
(let isTrue? (lambda v (primop = (var v) (var myTrue)))
(let P->V1 (lambda P (fst (snd (snd (var P)))))
(let P->V2 (lambda P (fst (snd (snd (snd (var P))))))
(let P->B (lambda P (fst (snd (snd (snd (snd (var P)))))))
(let emptyBlock? (lambda B (null? (var B)))
(let headBlock (lambda B (fst (snd (snd (var B)))))
(let tailBlock (lambda B (fst (snd (snd (snd (var B))))))
(let C-Assignment->V (lambda C (fst (snd (var C))))
(let C-Assignment->E (lambda C (fst (snd (snd (var C)))))
(let C-Conditional->E (lambda C (fst (snd (snd (var C)))))
(let C-Conditional->B1 (lambda C (fst (snd (snd (snd (var C))))))
(let C-Conditional->B2 (lambda C (fst (snd (snd (snd (snd (var C)))))))
(let C-While->E (lambda C (fst (snd (snd (var C)))))
(let C-While->B (lambda C (fst (snd (snd (snd (var C))))))
(let isAssignment? (lambda C (primop = (fst (var C)) (integer 1)))
(let isConditional? (lambda C (primop = (fst (var C)) (integer 2)))
(let isWhile? (lambda C (primop = (fst (var C)) (integer 3)))
(let isConstant? (lambda E (primop = (fst (var E)) (integer 1)))
(let isVariable? (lambda E (primop = (fst (var E)) (integer 2)))
(let isPlus? (lambda E (primop = (fst (var E)) (integer 3)))
(let isMinus? (lambda E (primop = (fst (var E)) (integer 4)))
(let isGt? (lambda E (primop = (fst (var E)) (integer 5)))
(let isEq? (lambda E (primop = (fst (var E)) (integer 6)))
(let E->C (lambda E (fst (snd (var E))))
(let E->V (lambda E (fst (snd (var E))))
(let E->E1 (lambda E (fst (snd (snd (var E)))))
(let E->E2 (lambda E (fst (snd (snd (snd (var E))))))
(let init-environment (lambda v1 (lambda v2 (apply (apply (var append)
                                                          (var v1))
                                                   (var v2))))
(let lookup-env1 (lambda v
                   (fix le
                    (lambda n
                      (lambda env
                        (if (primop = (var v) (fst (var env)))
                            (var n)
                            (apply (apply (var le)
                                          (primop + (var n)
                                                    (integer 1)))
                                   (snd (var env))))))))
(let lookup-env (lambda v
                  (lambda env
                    (apply (apply (apply (var lookup-env1) (var v))
                                  (integer 0))
                           (var env))))
(let init-store1 (fix is1
                  (lambda input-v1
                    (lambda length-V1
                      (if (primop = (var length-V1) (integer 0))
                          (nil)
                          (pair (fst (var input-v1))
                                (apply (apply (var is1)
                                              (snd (var input-v1)))
```

```
                                       (primop - (var length-v1)
                                               (integer 1))))))))
(let init-store2 (fix is2
                  (lambda length-V2
                    (if (primop = (var length-V2) (integer 0))
                        (nil)
                        (pair (integer 0)
                              (apply (var is2)
                                     (primop - (var length-v2)
                                               (integer 1)))))))
(let init-store (lambda length-V1
                  (lambda input-V1
                    (lambda length-V2
                      (apply (apply (var append)
                                    (apply (apply (var init-store1)
                                                  (var input-v1))
                                           (var length-V1)))
                             (apply (var init-store2)
                                    (var length-V2))))))
(let update-store (lambda value
                    (fix us
                     (lambda loc
                       (lambda store
                         (if (primop = (var loc) (integer 0))
                             (pair (var value)
                                   (snd (var store)))
                             (pair (fst (var store))
                                   (apply (apply (var us)
                                                 (primop - (var loc)
                                                           (integer 1)))
                                          (snd (var store)))))))))
(let lookup-store (fix ls (lambda loc
                            (lambda store
                              (if (primop = (var loc) (integer 0))
                                  (fst (var store))
                                  (apply (apply (var ls)
                                                (primop - (var loc)
                                                        (integer 1)))
                                         (snd (var store)))))))
(let evalExpression
    (fix ee
         (lambda E
           (lambda env
             (lambda store
               (if
                (apply (var isConstant?) (var E))
                (apply (var E->C) (var E))
                (if
                 (apply (var isVariable?) (var E))
                 (apply (apply (var lookup-store)
                               (apply (apply (var lookup-env)
                                             (apply (var E->V) (var E)))
                                      (var env)))
                        (var store))
                 (let E1 (apply
                          (apply
                           (apply
                            (var ee)
                            (apply (var E->E1) (var E)))
                           (var env))
                          (var store))
                   (let E2 (apply
                            (apply
                             (apply
                              (var ee)
                              (apply (var E->E2) (var E)))
                           (var env))
                          (var store))
                     (if
                      (apply (var isPlus?) (var E))
                      (primop + (var E1) (var E2))
                      (if
                       (apply (var isMinus?) (var E))
                       (primop - (var E1) (var E2))
                       (if
                        (apply (var isGt?) (var E))
                        (if (primop > (var E1) (var E2))
                            (var myTrue)
                            (var myFalse))
                        (if
                         (apply (var isEq?) (var E))
                         (if (primop = (var E1) (var E2))
                             (var myTrue)
                             (var myFalse))
                         (integer 0)))))))))))))
(let evalBCC
    (fix BCC
         (lambda choice     ; Block Commands or Command
           (let eval-Block
            (lambda B
              (lambda dummy ; To make types fit
                (lambda env
                  (lambda store
                    (if (apply (var emptyBlock?) (var B))
                        (nil)
                        (apply
                         (apply
                          (apply
                           (apply
                            (apply (var BCC) (integer 1))
                            (apply (var headBlock) (var B)))
                           (apply (var tailBlock) (var B)))
                          (var env))
                         (var store)))))))
           (let eval-Commands
             (lambda C
               (lambda B
                 (lambda env
                   (lambda store
                     (if (apply (var emptyBlock?) (var B))
                         (apply
                          (apply
                           (apply
                            (apply
                             (apply (var BCC) (integer 2))
                             (var C))
                            (nil))
                           (var env))
                          (var store))
                         (apply
                          (apply
                           (apply
                            (apply
                             (apply (var BCC) (integer 1))
                             (apply (var headBlock) (var B)))
                            (apply (var tailBlock) (var B)))
                           (var env))
                          (apply
                           (apply
                            (apply
                             (apply
                              (apply (var BCC) (integer 2))
                              (var C))
                             (nil))
                            (var env))
                           (var store)))))))))
           (let eval-Command
             (lambda C
               (lambda dummy
                 (lambda env
                   (lambda store
                     (if (apply (var isAssignment?) (var C))
                         (apply
                          (apply
                           (apply
                            (var update-store)
                            (apply         ; value
                             (apply
                              (apply
                               (var evalExpression)
```

```
                         (apply (var C-Assignment->E) (var C)))
                        (var env))
                       (var store)))
                     (apply   ; location
                      (apply
                       (var lookup-env)
                       (apply (var C-Assignment->V)
                              (var C)))
                      (var env)))
     (var store))
    (if (apply (var isConditional?) (var C))
(if (apply (var isTrue?)
   (apply
    (apply
     (apply
      (var evalExpression)
      (apply (var C-Conditional->E)
     (var C)))
     (var env))
    (var store)))
    (apply
     (apply
      (apply
       (apply
(apply (var BCC) (integer 0))
(apply (var C-Conditional->B1)
       (var C)))
       (nil))
      (var env))
     (var store))
    (apply
     (apply
      (apply
       (apply
(apply (var BCC) (integer 0))
(apply (var C-Conditional->B2)
       (var C)))
       (nil))
      (var env))
     (var store)))
(if (apply (var isWhile?) (var C))
    (apply
     (apply
      (var generalize1)
      (fix evalWhile
   (lambda store1
     (if (apply
  (var isTrue?)
  (apply
   (apply
          (apply
           (var evalExpression)
           (apply (var C-While->E)
          (var C)))
          (var env))
         (var store1)))
       (apply
        (var evalWhile)
        (apply
         (apply
          (apply
           (apply
            (apply (var BCC)
           (integer 0))
            (apply (var C-While->B)
           (var C)))
           (nil))
          (var env))
         (var store1)))
       (var store1)))))
           (var store))
          (nil))))))))
      (if
       (primop = (var choice) (integer 0)) ; Block
       (var eval-Block)
       (if
        (primop = (var choice) (integer 1)) ; Block
        (var eval-Commands)
        (var eval-Command))))))))
(let run (lambda P
           (lambda val
      (let V1 (apply (var P->V1) (var P))
        (let V2 (apply (var P->V2) (var P))
          (let env (apply (apply (var init-environment)
         (apply (var P->V1) (var P)))
          (var V2))
            (let store (apply (apply (apply (var init-store)
            (apply (var length)
           (var V1)))
             (var val))
      (apply (var length) (var V2)))
      (apply (apply (apply (apply (apply (var evalBCC)
         (integer 0))
          (apply (var P->B) (var P)))
           (nil))
            (var env))
             (var store))))))))
  (apply (apply (var run) (var P))
         (var value))))))))))))))))))))))))))))))))))))))))))))))
```

D.2.2 Annotated Program

The number of parameters to the functions in this program was reduced by the reduction presented in chapter 10 as follows. As in appendix D.1.2, the number of variables in $\mathit{FreeVars}(\kappa)$–$\mathit{FreeVars}(A)$ is shown to give a lower limit on the number of binding time parameters :

	Without reduction	With reduction	$FreeVars(\kappa)$–$FreeVars(A)$
generalize1	8	1	1
length	7	4	4
append	9	6	6
istrue?	4	3	3
p->v1	9	9	9
p->v2	11	11	11
p->b	13	13	13
emptyblock?	5	5	5
headblock	9	9	9
tailblock	11	11	11
c-assignment->v	7	7	7
c-assignment->e	9	9	9
c-conditional->e	9	9	9
c-conditional->b1	11	11	11
c-conditional->b2	13	13	13
c-while->e	9	9	9
c-while->b	11	11	11
isassignment?	6	5	5
isconditional?	6	5	5
iswhile?	6	5	5
isconstant?	6	5	5
isvariable?	6	5	5
isplus?	6	5	5
isminus?	6	5	5
isgt?	6	5	5
iseq?	6	5	5
e->c	7	7	7
e->v	7	7	7
e->e1	9	9	9
e->e2	11	11	11
init-environment	13	6	6
lookup-env1	12	8	8
lookup-env	17	6	6
init-store1	11	7	7
init-store2	7	4	4
init-store	27	11	9
update-store	11	7	7
lookup-store	10	6	6
e1	2	0	0
e2	2	0	0
evalexpression	89	12	9
eval-block	18	10	10
eval-commands	16	9	9
eval-command	307	13	8
evalbcc	323	12	8
v1	3	0	0
v2	2	0	0
env	15	0	0
store	41	5	2
run	386	16	11

The annotated program is

`let` generalize1 $= \Lambda\Big(\ \beta_{18} \ \Big).$

```
    $\lambda^{\beta_{18}}$e. if$^{\mathsf{S}}$ true$^{\mathsf{S}}$
          then  e$^{\mathsf{D}}$
          else  if$^{\mathsf{D}}$ (d$^{\mathsf{D}}$
                      $=^{\mathsf{D}}[\mathsf{S}\rightsquigarrow\mathsf{D}]$ 1$^{\mathsf{S}}$)
                then  $\lambda^{\mathsf{D}}$x. nil
                else  $\lambda^{\mathsf{D}}$x. nil
in let length = $\Lambda\left(\ \beta_{19}\beta_{977}\beta_{976}\beta_{36}\ \right)$.
       fix$^{\beta_{36}}$len.
        $\lambda^{\beta_{36}}$l. if$^{\beta_{976}}$ null?$^{\beta_{976}}$(l$^{\beta_{976}}$)
             then  $[\mathsf{S}\rightsquigarrow\beta_{19}]$ 0$^{\mathsf{S}}$
             else  ($[\mathsf{S}\rightsquigarrow\beta_{19}]$ 1$^{\mathsf{S}}$
                    $+^{\beta_{19}}$(len$^{\beta_{36}}$
                          $@^{\beta_{36}}$snd$^{\beta_{976}}$(l$^{\beta_{976}}$)))
in let append = $\Lambda\left(\ \beta_{44}\beta_{59}\beta_{61}\beta_{984}\beta_{982}\beta_{62}\ \right)$.
       fix$^{\beta_{62}}$app.
        $\lambda^{\beta_{62}}$l1. $\lambda^{\beta_{61}}$l2. if$^{\beta_{982}}$ null?$^{\beta_{982}}$(l1$^{\beta_{982}}$)
                     then  l2$^{\beta_{59}}$
                     else  pair$^{\beta_{59}}$($[\beta_{984}\rightsquigarrow\beta_{44}]$ fst$^{\beta_{982}}$(l1$^{\beta_{982}}$)
                                ,((app$^{\beta_{62}}$
                                  $@^{\beta_{62}}$snd$^{\beta_{982}}$(l1$^{\beta_{982}}$))
                                 $@^{\beta_{61}}$l2$^{\beta_{59}}$))
in let mytrue = $\Lambda$().
       1$^{\mathsf{S}}$
in let myfalse = $\Lambda$().
       0$^{\mathsf{S}}$
in let istrue? = $\Lambda\left(\ \beta_{69}\beta_{992}\beta_{70}\ \right)$.
       $\lambda^{\beta_{70}}$v. $[\beta_{992}\rightsquigarrow\beta_{69}]$ (v$^{\beta_{992}}$
                       $=^{\beta_{992}}[\mathsf{S}\rightsquigarrow\beta_{992}]$ (mytrue$^{\mathsf{S}}{}_{\diamond}$()))
in let p-¿v1 = $\Lambda\left(\begin{array}{l}\beta_{71}\beta_{999}\beta_{998}\beta_{997}\beta_{996}\beta_{995}\beta_{994}\beta_{993}\\ \beta_{77}\end{array}\right)$.
       $\lambda^{\beta_{77}}$p. $[\beta_{998}\rightsquigarrow\beta_{71}]$ fst$^{\beta_{997}}$(snd$^{\beta_{995}}$(snd$^{\beta_{993}}$(p$^{\beta_{993}}$)))
in let p-¿v2 = $\Lambda\left(\begin{array}{l}\beta_{78}\beta_{1008}\beta_{1007}\beta_{1006}\beta_{1005}\beta_{1004}\beta_{1003}\beta_{1002}\\ \beta_{1001}\beta_{1000}\beta_{85}\end{array}\right)$.
       $\lambda^{\beta_{85}}$p. $[\beta_{1007}\rightsquigarrow\beta_{78}]$ fst$^{\beta_{1006}}$(snd$^{\beta_{1004}}$(snd$^{\beta_{1002}}$(snd$^{\beta_{1000}}$(p$^{\beta_{1000}}$))))
in let p-¿b = $\Lambda\left(\begin{array}{l}\beta_{86}\beta_{1019}\beta_{1018}\beta_{1017}\beta_{1016}\beta_{1015}\beta_{1014}\beta_{1013}\\ \beta_{1012}\beta_{1011}\beta_{1010}\beta_{1009}\beta_{94}\end{array}\right)$.
       $\lambda^{\beta_{94}}$p. $[\beta_{1018}\rightsquigarrow\beta_{86}]$ fst$^{\beta_{1017}}$(snd$^{\beta_{1015}}$(snd$^{\beta_{1013}}$(snd$^{\beta_{1011}}$(snd$^{\beta_{1009}}$(p$^{\beta_{1009}}$)))))
in let emptyblock? = $\Lambda\left(\ \beta_{97}\beta_{1022}\beta_{1021}\beta_{1020}\beta_{98}\ \right)$.
       $\lambda^{\beta_{98}}$b. $[\beta_{1020}\rightsquigarrow\beta_{97}]$ null?$^{\beta_{1020}}$(b$^{\beta_{1020}}$)
in let headblock = $\Lambda\left(\begin{array}{l}\beta_{99}\beta_{1029}\beta_{1028}\beta_{1027}\beta_{1026}\beta_{1025}\beta_{1024}\beta_{1023}\\ \beta_{105}\end{array}\right)$.
       $\lambda^{\beta_{105}}$b. $[\beta_{1028}\rightsquigarrow\beta_{99}]$ fst$^{\beta_{1027}}$(snd$^{\beta_{1025}}$(snd$^{\beta_{1023}}$(b$^{\beta_{1023}}$)))
in let tailblock = $\Lambda\left(\begin{array}{l}\beta_{106}\beta_{1038}\beta_{1037}\beta_{1036}\beta_{1035}\beta_{1034}\beta_{1033}\beta_{1032}\\ \beta_{1031}\beta_{1030}\beta_{113}\end{array}\right)$.
       $\lambda^{\beta_{113}}$b. $[\beta_{1037}\rightsquigarrow\beta_{106}]$ fst$^{\beta_{1036}}$(snd$^{\beta_{1034}}$(snd$^{\beta_{1032}}$(snd$^{\beta_{1030}}$(b$^{\beta_{1030}}$))))
```

`in let` c-assignment-¿v $= \Lambda\left(\beta_{114}\beta_{1043}\beta_{1042}\beta_{1041}\beta_{1040}\beta_{1039}\beta_{119} \right).$

$\lambda^{\beta_{119}}\mathrm{c}.\ [\beta_{1042}\rightsquigarrow\beta_{114}]\ \mathtt{fst}^{\beta_{1041}}(\mathtt{snd}^{\beta_{1039}}(\mathrm{c}^{\beta_{1039}}))$

`in let` c-assignment-¿e $= \Lambda\left(\begin{array}{l} \beta_{120}\beta_{1050}\beta_{1049}\beta_{1048}\beta_{1047}\beta_{1046}\beta_{1045}\beta_{1044} \\ \beta_{126} \end{array} \right).$

$\lambda^{\beta_{126}}\mathrm{c}.\ [\beta_{1049}\rightsquigarrow\beta_{120}]\ \mathtt{fst}^{\beta_{1048}}(\mathtt{snd}^{\beta_{1046}}(\mathtt{snd}^{\beta_{1044}}(\mathrm{c}^{\beta_{1044}})))$

`in let` c-conditional-¿e $= \Lambda\left(\begin{array}{l} \beta_{127}\beta_{1057}\beta_{1056}\beta_{1055}\beta_{1054}\beta_{1053}\beta_{1052}\beta_{1051} \\ \beta_{133} \end{array} \right).$

$\lambda^{\beta_{133}}\mathrm{c}.\ [\beta_{1056}\rightsquigarrow\beta_{127}]\ \mathtt{fst}^{\beta_{1055}}(\mathtt{snd}^{\beta_{1053}}(\mathtt{snd}^{\beta_{1051}}(\mathrm{c}^{\beta_{1051}})))$

`in let` c-conditional-¿b1 $= \Lambda\left(\begin{array}{l} \beta_{134}\beta_{1066}\beta_{1065}\beta_{1064}\beta_{1063}\beta_{1062}\beta_{1061}\beta_{1060} \\ \beta_{1059}\beta_{1058}\beta_{141} \end{array} \right).$

$\lambda^{\beta_{141}}\mathrm{c}.\ [\beta_{1065}\rightsquigarrow\beta_{134}]\ \mathtt{fst}^{\beta_{1064}}(\mathtt{snd}^{\beta_{1062}}(\mathtt{snd}^{\beta_{1060}}(\mathtt{snd}^{\beta_{1058}}(\mathrm{c}^{\beta_{1058}}))))$

`in let` c-conditional-¿b2 $= \Lambda\left(\begin{array}{l} \beta_{142}\beta_{1077}\beta_{1076}\beta_{1075}\beta_{1074}\beta_{1073}\beta_{1072}\beta_{1071} \\ \beta_{1070}\beta_{1069}\beta_{1068}\beta_{1067}\beta_{150} \end{array} \right).$

$\lambda^{\beta_{150}}\mathrm{c}.\ [\beta_{1076}\rightsquigarrow\beta_{142}]\ \mathtt{fst}^{\beta_{1075}}(\mathtt{snd}^{\beta_{1073}}(\mathtt{snd}^{\beta_{1071}}(\mathtt{snd}^{\beta_{1069}}(\mathtt{snd}^{\beta_{1067}}(\mathrm{c}^{\beta_{1067}})))))$

`in let` c-while-¿e $= \Lambda\left(\begin{array}{l} \beta_{151}\beta_{1084}\beta_{1083}\beta_{1082}\beta_{1081}\beta_{1080}\beta_{1079}\beta_{1078} \\ \beta_{157} \end{array} \right).$

$\lambda^{\beta_{157}}\mathrm{c}.\ [\beta_{1083}\rightsquigarrow\beta_{151}]\ \mathtt{fst}^{\beta_{1082}}(\mathtt{snd}^{\beta_{1080}}(\mathtt{snd}^{\beta_{1078}}(\mathrm{c}^{\beta_{1078}})))$

`in let` c-while-¿b $= \Lambda\left(\begin{array}{l} \beta_{158}\beta_{1093}\beta_{1092}\beta_{1091}\beta_{1090}\beta_{1089}\beta_{1088}\beta_{1087} \\ \beta_{1086}\beta_{1085}\beta_{165} \end{array} \right).$

$\lambda^{\beta_{165}}\mathrm{c}.\ [\beta_{1092}\rightsquigarrow\beta_{158}]\ \mathtt{fst}^{\beta_{1091}}(\mathtt{snd}^{\beta_{1089}}(\mathtt{snd}^{\beta_{1087}}(\mathtt{snd}^{\beta_{1085}}(\mathrm{c}^{\beta_{1085}}))))$

`in let` isassignment? $= \Lambda\left(\beta_{172}\beta_{1096}\beta_{1095}\beta_{1094}\beta_{173} \right).$

$\lambda^{\beta_{173}}\mathrm{c}.\ [\beta_{1095}\rightsquigarrow\beta_{172}]\ (\mathtt{fst}^{\beta_{1094}}(\mathrm{c}^{\beta_{1094}})$

$=^{\beta_{1095}}[\mathsf{S}\rightsquigarrow\beta_{1095}]\ 1^{\mathsf{S}})$

`in let` isconditional? $= \Lambda\left(\beta_{180}\beta_{1099}\beta_{1098}\beta_{1097}\beta_{181} \right).$

$\lambda^{\beta_{181}}\mathrm{c}.\ [\beta_{1098}\rightsquigarrow\beta_{180}]\ (\mathtt{fst}^{\beta_{1097}}(\mathrm{c}^{\beta_{1097}})$

$=^{\beta_{1098}}[\mathsf{S}\rightsquigarrow\beta_{1098}]\ 2^{\mathsf{S}})$

`in let` iswhile? $= \Lambda\left(\beta_{188}\beta_{1102}\beta_{1101}\beta_{1100}\beta_{189} \right).$

$\lambda^{\beta_{189}}\mathrm{c}.\ [\beta_{1101}\rightsquigarrow\beta_{188}]\ (\mathtt{fst}^{\beta_{1100}}(\mathrm{c}^{\beta_{1100}})$

$=^{\beta_{1101}}[\mathsf{S}\rightsquigarrow\beta_{1101}]\ 3^{\mathsf{S}})$

`in let` isconstant? $= \Lambda\left(\beta_{196}\beta_{1105}\beta_{1104}\beta_{1103}\beta_{197} \right).$

$\lambda^{\beta_{197}}\mathrm{e}.\ [\beta_{1104}\rightsquigarrow\beta_{196}]\ (\mathtt{fst}^{\beta_{1103}}(\mathrm{e}^{\beta_{1103}})$

$=^{\beta_{1104}}[\mathsf{S}\rightsquigarrow\beta_{1104}]\ 1^{\mathsf{S}})$

`in let` isvariable? $= \Lambda\left(\beta_{204}\beta_{1108}\beta_{1107}\beta_{1106}\beta_{205} \right).$

$\lambda^{\beta_{205}}\mathrm{e}.\ [\beta_{1107}\rightsquigarrow\beta_{204}]\ (\mathtt{fst}^{\beta_{1106}}(\mathrm{e}^{\beta_{1106}})$

$=^{\beta_{1107}}[\mathsf{S}\rightsquigarrow\beta_{1107}]\ 2^{\mathsf{S}})$

`in let` isplus? $= \Lambda\left(\beta_{212}\beta_{1111}\beta_{1110}\beta_{1109}\beta_{213} \right).$

$\lambda^{\beta_{213}}\mathrm{e}.\ [\beta_{1110}\rightsquigarrow\beta_{212}]\ (\mathtt{fst}^{\beta_{1109}}(\mathrm{e}^{\beta_{1109}})$

$=^{\beta_{1110}}[\mathsf{S}\rightsquigarrow\beta_{1110}]\ 3^{\mathsf{S}})$

`in let` isminus? $= \Lambda\left(\beta_{220}\beta_{1114}\beta_{1113}\beta_{1112}\beta_{221} \right).$

$\lambda^{\beta_{221}}\mathrm{e}.\ [\beta_{1113}\rightsquigarrow\beta_{220}]\ (\mathtt{fst}^{\beta_{1112}}(\mathrm{e}^{\beta_{1112}})$

$=^{\beta_{1113}}[\mathsf{S}\rightsquigarrow\beta_{1113}]\ 4^{\mathsf{S}})$

`in let` isgt? $= \Lambda\left(\beta_{228}\beta_{1117}\beta_{1116}\beta_{1115}\beta_{229} \right).$

$\lambda^{\beta_{229}}\mathrm{e}.\ [\beta_{1116}\rightsquigarrow\beta_{228}]\ (\mathtt{fst}^{\beta_{1115}}(\mathrm{e}^{\beta_{1115}})$

$=^{\beta_{1116}}[\mathsf{S}\rightsquigarrow\beta_{1116}]\ 5^{\mathsf{S}})$

```
in let iseq? = $\Lambda\Big(\ \beta_{236}\beta_{1120}\beta_{1119}\beta_{1118}\beta_{237}\ \Big).$
        $\lambda^{\beta_{237}}$e. $[\beta_{1119}\rightsquigarrow\beta_{236}]$ (fst$^{\beta_{1118}}$(e$^{\beta_{1118}}$)
                                   $=^{\beta_{1119}}[\mathsf{S}\rightsquigarrow\beta_{1119}]\ 6^{\mathsf{S}}$)
in let e-¿c = $\Lambda\Big(\ \beta_{238}\beta_{1125}\beta_{1124}\beta_{1123}\beta_{1122}\beta_{1121}\beta_{243}\ \Big).$
        $\lambda^{\beta_{243}}$e. $[\beta_{1124}\rightsquigarrow\beta_{238}]$ fst$^{\beta_{1123}}$(snd$^{\beta_{1121}}$(e$^{\beta_{1121}}$))
in let e-¿v = $\Lambda\Big(\ \beta_{244}\beta_{1130}\beta_{1129}\beta_{1128}\beta_{1127}\beta_{1126}\beta_{249}\ \Big).$
        $\lambda^{\beta_{249}}$e. $[\beta_{1129}\rightsquigarrow\beta_{244}]$ fst$^{\beta_{1128}}$(snd$^{\beta_{1126}}$(e$^{\beta_{1126}}$))
in let e-¿e1 = $\Lambda\begin{pmatrix} \beta_{250}\beta_{1137}\beta_{1136}\beta_{1135}\beta_{1134}\beta_{1133}\beta_{1132}\beta_{1131} \\ \beta_{256} \end{pmatrix}.$
        $\lambda^{\beta_{256}}$e. $[\beta_{1136}\rightsquigarrow\beta_{250}]$ fst$^{\beta_{1135}}$(snd$^{\beta_{1133}}$(snd$^{\beta_{1131}}$(e$^{\beta_{1131}}$)))
in let e-¿e2 = $\Lambda\begin{pmatrix} \beta_{257}\beta_{1146}\beta_{1145}\beta_{1144}\beta_{1143}\beta_{1142}\beta_{1141}\beta_{1140} \\ \beta_{1139}\beta_{1138}\beta_{264} \end{pmatrix}.$
        $\lambda^{\beta_{264}}$e. $[\beta_{1145}\rightsquigarrow\beta_{257}]$ fst$^{\beta_{1144}}$(snd$^{\beta_{1142}}$(snd$^{\beta_{1140}}$(snd$^{\beta_{1138}}$(e$^{\beta_{1138}}$))))
in let init-environment = $\Lambda\Big(\ \beta_{1154}\beta_{272}\beta_{274}\beta_{1151}\beta_{1147}\beta_{275}\ \Big).$
        $\lambda^{\beta_{275}}$v1. $\lambda^{\beta_{274}}$v2. (((append$^{\beta_{267}}\diamond\Big(\ \beta_{1154}\beta_{272}\mathsf{S}\beta_{1151}\beta_{1147}\mathsf{S}\ \Big)$)
                                   $@^{\mathsf{S}}$v1$^{\beta_{1147}}$)
                                  $@^{\mathsf{S}}$v2$^{\beta_{272}}$)
in let lookup-env1 = $\Lambda\Big(\ \beta_{277}\beta_{1160}\beta_{1159}\beta_{303}\beta_{1158}\beta_{304}\beta_{281}\beta_{306}\ \Big).$
        $\lambda^{\beta_{306}}$v. fix$^{\beta_{304}}$le.
                $\lambda^{\beta_{304}}$n. $\lambda^{\beta_{303}}$env. if$^{\beta_{281}}$ (v$^{\beta_{281}}$
                                             $=^{\beta_{281}}[\beta_{1160}\rightsquigarrow\beta_{281}]$ fst$^{\beta_{1159}}$(env$^{\beta_{1159}}$))
                                     then  $[\beta_{1158}\rightsquigarrow\beta_{277}]$ n$^{\beta_{1158}}$
                                     else  ((le$^{\beta_{304}}$
                                             $@^{\beta_{304}}$(n$^{\beta_{1158}}$
                                                  $+^{\beta_{1158}}[\mathsf{S}\rightsquigarrow\beta_{1158}]\ 1^{\mathsf{S}}$))
                                            $@^{\beta_{303}}$snd$^{\beta_{1159}}$(env$^{\beta_{1159}}$))
in let lookup-env = $\Lambda\Big(\ \beta_{307}\beta_{1174}\beta_{1167}\beta_{318}\beta_{311}\beta_{319}\ \Big).$
        $\lambda^{\beta_{319}}$v. $\lambda^{\beta_{318}}$env. $[\beta_{311}\rightsquigarrow\beta_{307}]$ ((((lookup-env1$^{\beta_{309}}\diamond\Big(\ \beta_{311}\beta_{1174}\beta_{1167}\mathsf{SSS}\beta_{311}\mathsf{S}\ \Big)$)
                                             $@^{\mathsf{S}}$v$^{\beta_{311}}$)
                                            $@^{\mathsf{S}}0^{\mathsf{S}}$)
                                           $@^{\mathsf{S}}$env$^{\beta_{1167}}$)
in let init-store1 = $\Lambda\Big(\ \beta_{327}\beta_{345}\beta_{1180}\beta_{347}\beta_{1183}\beta_{1179}\beta_{348}\ \Big).$
        fix$^{\beta_{348}}$is1.
         $\lambda^{\beta_{348}}$input-v1. $\lambda^{\beta_{347}}$length-v1. if$^{\beta_{1180}}$ (length-v1$^{\beta_{1180}}$
                                                  $=^{\beta_{1180}}[\mathsf{S}\rightsquigarrow\beta_{1180}]\ 0^{\mathsf{S}}$)
                                          then  nil
                                          else  pair$^{\beta_{345}}$([$\beta_{1183}\rightsquigarrow\beta_{327}$] fst$^{\beta_{1179}}$(input-v1$^{\beta_{1179}}$)
                                                        ,((is1$^{\beta_{348}}$
                                                          $@^{\beta_{348}}$snd$^{\beta_{1179}}$(input-v1$^{\beta_{1179}}$))
                                                         $@^{\beta_{347}}$(length-v1$^{\beta_{1180}}$
                                                                $-^{\beta_{1180}}[\mathsf{S}\rightsquigarrow\beta_{1180}]\ 1^{\mathsf{S}}$)))
in let init-store2 = $\Lambda\Big(\ \beta_{357}\beta_{367}\beta_{1189}\beta_{369}\ \Big).$
        fix$^{\beta_{369}}$is2.
         $\lambda^{\beta_{369}}$length-v2. if$^{\beta_{1189}}$ (length-v2$^{\beta_{1189}}$
                                   $=^{\beta_{1189}}[\mathsf{S}\rightsquigarrow\beta_{1189}]\ 0^{\mathsf{S}})$
                           then  nil
                           else  pair$^{\beta_{367}}([\mathsf{S}\rightsquigarrow\beta_{357}]\ 0^{\mathsf{S}}$
                                      $,(\text{is2}^{\beta_{369}}$
                                        $@^{\beta_{369}}(\text{length-v2}^{\beta_{1189}}$
                                              $-^{\beta_{1189}}[\mathsf{S}\rightsquigarrow\beta_{1189}]\ 1^{\mathsf{S}})))$
in let init-store $= \Lambda\left(\begin{array}{l}\beta_{1215}\beta_{1216}\beta_{388}\beta_{391}\beta_{1204}\beta_{1194}\beta_{392}\beta_{381}\\ \beta_{393}\beta_{1208}\beta_{1210}\end{array}\right).$
      $\lambda^{\beta_{393}}$length-v1. $\lambda^{\beta_{392}}$input-v1. $\lambda^{\beta_{391}}$length-v2.
      $(((\text{append}^{\beta_{373}}\diamond(\ \beta_{1215}\beta_{1216}\mathsf{S}\beta_{1208}\beta_{1210}\mathsf{S}\ ))$
        $@^{\mathsf{S}}(((\text{init-store1}^{\beta_{376}}\diamond(\ \beta_{1208}\beta_{1210}\beta_{381}\mathsf{S}\beta_{1204}\beta_{1194}\mathsf{S}\ ))$
            $@^{\mathsf{S}}\text{input-v1}^{\beta_{1194}})$
           $@^{\mathsf{S}}\text{length-v1}^{\beta_{381}}))$
        $@^{\mathsf{S}}((\text{init-store2}^{\beta_{386}}\diamond(\ \beta_{1215}\beta_{1216}\beta_{388}\mathsf{S}\ ))$
           $@^{\mathsf{S}}\text{length-v2}^{\beta_{388}}))$
in let update-store $= \Lambda(\ \beta_{1221}\beta_{1220}\beta_{428}\beta_{1219}\beta_{429}\beta_{1218}\beta_{431}\ ).$
      $\lambda^{\beta_{431}}$value. fix$^{\beta_{429}}$us.
               $\lambda^{\beta_{429}}$loc. $\lambda^{\beta_{428}}$store. if$^{\beta_{1219}}$ $(\text{loc}^{\beta_{1219}}$
                                            $=^{\beta_{1219}}[\mathsf{S}\rightsquigarrow\beta_{1219}]\ 0^{\mathsf{S}})$
                                  then  pair$^{\beta_{1220}}([\beta_{1218}\rightsquigarrow\beta_{1221}]\ \text{value}^{\beta_{1218}}$
                                              $,$snd$^{\beta_{1220}}(\text{store}^{\beta_{1220}}))$
                                  else  pair$^{\beta_{1220}}($fst$^{\beta_{1220}}(\text{store}^{\beta_{1220}})$
                                              $,((\text{us}^{\beta_{429}}$
                                                $@^{\beta_{429}}(\text{loc}^{\beta_{1219}}$
                                                     $-^{\beta_{1219}}[\mathsf{S}\rightsquigarrow\beta_{1219}]\ 1^{\mathsf{S}}))$
                                               $@^{\beta_{428}}$snd$^{\beta_{1220}}(\text{store}^{\beta_{1220}})))$
in let lookup-store $= \Lambda(\ \beta_{432}\beta_{1231}\beta_{1230}\beta_{457}\beta_{1229}\beta_{458}\ ).$
      fix$^{\beta_{458}}$ls.
       $\lambda^{\beta_{458}}$loc. $\lambda^{\beta_{457}}$store. if$^{\beta_{1229}}$ $(\text{loc}^{\beta_{1229}}$
                                   $=^{\beta_{1229}}[\mathsf{S}\rightsquigarrow\beta_{1229}]\ 0^{\mathsf{S}})$
                         then  $[\beta_{1231}\rightsquigarrow\beta_{432}]$ fst$^{\beta_{1230}}(\text{store}^{\beta_{1230}})$
                         else  $((\text{ls}^{\beta_{458}}$
                                 $@^{\beta_{458}}(\text{loc}^{\beta_{1229}}$
                                      $-^{\beta_{1229}}[\mathsf{S}\rightsquigarrow\beta_{1229}]\ 1^{\mathsf{S}}))$
                                $@^{\beta_{457}}$snd$^{\beta_{1230}}(\text{store}^{\beta_{1230}}))$
in let evalexpression $= \Lambda\left(\begin{array}{l}\beta_{460}\beta_{1264}\beta_{1239}\beta_{593}\beta_{1270}\beta_{1238}\beta_{594}\beta_{1242}\\ \beta_{595}\beta_{1283}\beta_{481}\beta_{484}\end{array}\right).$
      fix$^{\beta_{595}}$ee.
       $\lambda^{\beta_{595}}$e. $\lambda^{\beta_{594}}$env. $\lambda^{\beta_{593}}$store.
       if$^{\beta_{1242}}$ $((\text{isconstant?}^{\beta_{462}}\diamond(\ \beta_{1242}\mathsf{S}\beta_{1242}\mathsf{SS}\ ))$
             $@^{\mathsf{S}}\text{e}^{\mathsf{S}})$
       then  $[\mathsf{S}\rightsquigarrow\beta_{460}]\ ((\text{e-¿c}^{\beta_{468}}\diamond(\ \mathsf{SSSS}\beta_{1242}\mathsf{SS}\ ))$
                     $@^{\mathsf{S}}\text{e}^{\mathsf{S}})$
       else  if$^{\beta_{1242}}$ $((\text{isvariable?}^{\beta_{474}}\diamond(\ \beta_{1242}\mathsf{S}\beta_{1242}\mathsf{SS}\ ))$
                 $@^{\mathsf{S}}\mathrm{e}^{\mathsf{S}})$
            then  $[\beta_{1283}\rightsquigarrow\beta_{460}]$ $(((\text{lookup-store}^{\beta_{480}}\diamond\left(\ \beta_{1283}\beta_{1264}\beta_{1239}\mathsf{S}\beta_{481}\mathsf{S}\ \right))$
                                    $@^{\mathsf{S}}[\beta_{484}\rightsquigarrow\beta_{481}]$ $(((\text{lookup-env}^{\beta_{483}}\diamond\left(\ \beta_{484}\beta_{1270}\beta_{1238}\mathsf{S}\beta_{484}\mathsf{S}\ \right))$
                                                   $@^{\mathsf{S}}[\mathsf{S}\rightsquigarrow\beta_{484}]$ $((\text{e-¿v}^{\beta_{486}}\diamond\left(\ \mathsf{SSSS}\beta_{1242}\mathsf{SS}\ \right))$
                                                              $@^{\mathsf{S}}\mathrm{e}^{\mathsf{S}}))$
                                                   $@^{\mathsf{S}}\mathrm{env}^{\beta_{1238}}))$
                                    $@^{\mathsf{S}}\mathrm{store}^{\beta_{1239}})$
            else  let $\mathrm{e1} = \Lambda()$.
                    $(((\mathrm{ee}^{\beta_{595}}$
                        $@^{\beta_{595}}((\text{e-¿e1}^{\beta_{502}}\diamond\begin{pmatrix}\mathsf{SSSSSS}\beta_{1242}\mathsf{S}\\ \mathsf{S}\end{pmatrix})$
                              $@^{\mathsf{S}}\mathrm{e}^{\mathsf{S}}))$
                      $@^{\beta_{594}}\mathrm{env}^{\beta_{1238}})$
                     $@^{\beta_{593}}\mathrm{store}^{\beta_{1239}})$
            in let $\mathrm{e2} = \Lambda()$.
                  $(((\mathrm{ee}^{\beta_{595}}$
                      $@^{\beta_{595}}((\text{e-¿e2}^{\beta_{516}}\diamond\begin{pmatrix}\mathsf{SSSSSSSS}\\ \beta_{1242}\mathsf{SS}\end{pmatrix})$
                            $@^{\mathsf{S}}\mathrm{e}^{\mathsf{S}}))$
                    $@^{\beta_{594}}\mathrm{env}^{\beta_{1238}})$
                   $@^{\beta_{593}}\mathrm{store}^{\beta_{1239}})$
            in if$^{\beta_{1242}}$ $((\text{isplus?}^{\beta_{527}}\diamond\left(\ \beta_{1242}\mathsf{S}\beta_{1242}\mathsf{SS}\ \right))$
                     $@^{\mathsf{S}}\mathrm{e}^{\mathsf{S}})$
               then  $((\mathrm{e1}^{\beta_{460}}\diamond())$
                      $+^{\beta_{460}}(\mathrm{e2}^{\beta_{460}}\diamond()))$
               else  if$^{\beta_{1242}}$ $((\text{isminus?}^{\beta_{539}}\diamond\left(\ \beta_{1242}\mathsf{S}\beta_{1242}\mathsf{SS}\ \right))$
                          $@^{\mathsf{S}}\mathrm{e}^{\mathsf{S}})$
                     then  $((\mathrm{e1}^{\beta_{460}}\diamond())$
                            $-^{\beta_{460}}(\mathrm{e2}^{\beta_{460}}\diamond()))$
                     else  if$^{\beta_{1242}}$ $((\text{isgt?}^{\beta_{551}}\diamond\left(\ \beta_{1242}\mathsf{S}\beta_{1242}\mathsf{SS}\ \right))$
                                $@^{\mathsf{S}}\mathrm{e}^{\mathsf{S}})$
                           then  if$^{\beta_{460}}$ $((\mathrm{e1}^{\beta_{460}}\diamond())$
                                        $>^{\beta_{460}}(\mathrm{e2}^{\beta_{460}}\diamond()))$
                                 then  $[\mathsf{S}\rightsquigarrow\beta_{460}]$ $(\mathrm{mytrue}^{\mathsf{S}}\diamond())$
                                 else  $[\mathsf{S}\rightsquigarrow\beta_{460}]$ $(\mathrm{myfalse}^{\mathsf{S}}\diamond())$
                           else  if$^{\beta_{1242}}$ $((\text{iseq?}^{\beta_{569}}\diamond\left(\ \beta_{1242}\mathsf{S}\beta_{1242}\mathsf{SS}\ \right))$
                                      $@^{\mathsf{S}}\mathrm{e}^{\mathsf{S}})$
                                 then  if$^{\beta_{460}}$ $((\mathrm{e1}^{\beta_{460}}\diamond())$
                                              $=^{\beta_{460}}(\mathrm{e2}^{\beta_{460}}\diamond()))$
                                       then  $[\mathsf{S}\rightsquigarrow\beta_{460}]$ $(\mathrm{mytrue}^{\mathsf{S}}\diamond())$
                                       else  $[\mathsf{S}\rightsquigarrow\beta_{460}]$ $(\mathrm{myfalse}^{\mathsf{S}}\diamond())$
                                 else  $[\mathsf{S}\rightsquigarrow\beta_{460}]$ $0^{\mathsf{S}}$
in let $\mathrm{evalbcc} = \Lambda\begin{pmatrix}\beta_{1655}\beta_{1451}\beta_{1429}\beta_{1654}\beta_{1652}\beta_{1648}\beta_{1336}\beta_{901}\\ \beta_{1646}\beta_{1643}\beta_{1644}\beta_{1645}\end{pmatrix}$.
```

$$
\begin{array}{l}
\mathtt{fix}^{\beta_{901}}\text{bcc}.\\
\quad\lambda^{\beta_{901}}\text{choice. }\mathtt{let}\text{ eval-block} = \Lambda\left(\begin{array}{l}\beta_{1340}\beta_{631}\beta_{1339}\beta_{632}\beta_{1338}\beta_{633}\beta_{1344}\beta_{1343}\\ \beta_{1337}\beta_{634}\end{array}\right).\\
\qquad\qquad\lambda^{\beta_{634}}\text{b. }\lambda^{\beta_{633}}\text{dummy. }\lambda^{\beta_{632}}\text{env. }\lambda^{\beta_{631}}\text{store.}\\
\qquad\qquad\mathtt{if}^{\beta_{1337}}\ ((\text{emptyblock?}^{\beta_{600}}\diamond\left(\ \beta_{1337}\beta_{1344}\beta_{1343}\beta_{1337}\mathsf{S}\ \right))\\
\qquad\qquad\qquad @^{\mathsf{S}}\text{b}^{\beta_{1337}})\\
\qquad\qquad\mathtt{then}\ \ \mathtt{nil}\\
\qquad\qquad\mathtt{else}\ \ (((((\text{bcc}^{\beta_{901}}\\
\qquad\qquad\qquad\quad @^{\beta_{901}}[\mathsf{S}\rightsquigarrow\beta_{1336}]\ 1^{\mathsf{S}})\\
\qquad\qquad\qquad\quad @^{\beta_{1648}}((\text{headblock}^{\beta_{612}}\diamond\left(\begin{array}{l}\mathsf{SSSSS}\beta_{1344}\beta_{1343}\beta_{1337}\\ \mathsf{S}\end{array}\right))\\
\qquad\qquad\qquad\qquad @^{\mathsf{S}}\text{b}^{\beta_{1337}}))\\
\qquad\qquad\qquad\quad @^{\beta_{1652}}((\text{tailblock}^{\beta_{619}}\diamond\left(\begin{array}{l}\mathsf{SSSSSSS}\beta_{1344}\\ \beta_{1343}\beta_{1337}\mathsf{S}\end{array}\right))\\
\qquad\qquad\qquad\qquad @^{\mathsf{S}}\text{b}^{\beta_{1337}}))\\
\qquad\qquad\qquad @^{\beta_{1654}}[\beta_{1339}\rightsquigarrow\beta_{1429}]\ \text{env}^{\beta_{1339}})\\
\qquad\qquad\qquad @^{\beta_{1655}}[\beta_{1340}\rightsquigarrow\mathsf{D}]\ \text{store}^{\beta_{1340}})\\
\quad\mathtt{in}\ \mathtt{let}\text{ eval-commands} = \Lambda\left(\begin{array}{l}\beta_{697}\beta_{1378}\beta_{698}\beta_{1383}\beta_{1382}\beta_{1377}\beta_{699}\beta_{1376}\\ \beta_{700}\end{array}\right).\\
\qquad\lambda^{\beta_{700}}\text{c. }\lambda^{\beta_{699}}\text{b. }\lambda^{\beta_{698}}\text{env. }\lambda^{\beta_{697}}\text{store.}\\
\qquad\mathtt{if}^{\beta_{1377}}\ ((\text{emptyblock?}^{\beta_{637}}\diamond\left(\ \beta_{1377}\beta_{1383}\beta_{1382}\beta_{1377}\mathsf{S}\ \right))\\
\qquad\qquad @^{\mathsf{S}}\text{b}^{\beta_{1377}})\\
\qquad\mathtt{then}\ \ (((((\text{bcc}^{\beta_{901}}\\
\qquad\qquad\quad @^{\beta_{901}}[\mathsf{S}\rightsquigarrow\beta_{1336}]\ 2^{\mathsf{S}})\\
\qquad\qquad\quad @^{\beta_{1648}}[\beta_{1376}\rightsquigarrow\mathsf{S}]\ \text{c}^{\beta_{1376}})\\
\qquad\qquad @^{\beta_{1652}}\mathtt{nil})\\
\qquad\qquad @^{\beta_{1654}}[\beta_{1378}\rightsquigarrow\beta_{1429}]\ \text{env}^{\beta_{1378}})\\
\qquad\qquad @^{\beta_{1655}}\text{store}^{\mathsf{D}})\\
\qquad\mathtt{else}\ \ (((((\text{bcc}^{\beta_{901}}\\
\qquad\qquad\quad @^{\beta_{901}}[\mathsf{S}\rightsquigarrow\beta_{1336}]\ 1^{\mathsf{S}})\\
\qquad\qquad\quad @^{\beta_{1648}}((\text{headblock}^{\beta_{664}}\diamond\left(\begin{array}{l}\mathsf{SSSSS}\beta_{1383}\beta_{1382}\beta_{1377}\\ \mathsf{S}\end{array}\right))\\
\qquad\qquad\qquad @^{\mathsf{S}}\text{b}^{\beta_{1377}}))\\
\qquad\qquad\quad @^{\beta_{1652}}((\text{tailblock}^{\beta_{671}}\diamond\left(\begin{array}{l}\mathsf{SSSSSSS}\beta_{1383}\\ \beta_{1382}\beta_{1377}\mathsf{S}\end{array}\right))\\
\qquad\qquad\qquad @^{\mathsf{S}}\text{b}^{\beta_{1377}}))\\
\qquad\qquad @^{\beta_{1654}}[\beta_{1378}\rightsquigarrow\beta_{1429}]\ \text{env}^{\beta_{1378}})\\
\qquad\qquad @^{\beta_{1655}}(((((\text{bcc}^{\beta_{901}}\\
\qquad\qquad\qquad\quad @^{\beta_{901}}[\mathsf{S}\rightsquigarrow\beta_{1336}]\ 2^{\mathsf{S}})\\
\qquad\qquad\qquad\quad @^{\beta_{1648}}[\beta_{1376}\rightsquigarrow\mathsf{S}]\ \text{c}^{\beta_{1376}})\\
\qquad\qquad\qquad @^{\beta_{1652}}\mathtt{nil})\\
\qquad\qquad\qquad @^{\beta_{1654}}[\beta_{1378}\rightsquigarrow\beta_{1429}]\ \text{env}^{\beta_{1378}})\\
\qquad\qquad\qquad @^{\beta_{1655}}\text{store}^{\mathsf{D}}))\\
\quad\mathtt{in}\ \mathtt{let}\text{ eval-command} = \Lambda\left(\begin{array}{l}\beta_{878}\beta_{879}\beta_{1428}\beta_{880}\beta_{1434}\beta_{1433}\beta_{1427}\beta_{881}\\ \beta_{727}\beta_{730}\beta_{1444}\beta_{1497}\beta_{1574}\end{array}\right).
\end{array}
$$

```
$\lambda^{\beta_{881}}$c. $\lambda^{\beta_{880}}$dummy. $\lambda^{\beta_{879}}$env. $\lambda^{\beta_{878}}$store.
if$^{\beta_{1433}}$ ((isassignment?$^{\beta_{703}}\diamond\big(\ \beta_{1433}\beta_{1434}\beta_{1433}\beta_{1427}\mathsf{S}\ \big)$)
          @$^{\mathsf{S}}$c$^{\beta_{1427}}$)
then  ((((update-store$^{\beta_{709}}\diamond\big(\ \mathsf{DDS}\beta_{727}\mathsf{SDS}\ \big)$)
          @$^{\mathsf{S}}$((((evalexpression$^{\beta_{712}}\diamond\begin{pmatrix}\mathsf{DDDS}\beta_{1451}\beta_{1429}\mathsf{SS}\\ \mathsf{SD}\beta_{1444}\beta_{1444}\end{pmatrix}$)
                @$^{\mathsf{S}}$((c-assignment-¿e$^{\beta_{715}}\diamond\begin{pmatrix}\mathsf{SSSSS}\beta_{1434}\beta_{1433}\beta_{1427}\\ \mathsf{S}\end{pmatrix}$)
                @$^{\mathsf{S}}$c$^{\beta_{1427}}$))
               @$^{\mathsf{S}}$env$^{\beta_{1429}}$)
              @$^{\mathsf{S}}$store$^{\mathsf{D}}$))
          @$^{\mathsf{S}}[\beta_{730}\rightsquigarrow\beta_{727}]$ (((lookup-env$^{\beta_{729}}\diamond\big(\ \beta_{730}\beta_{1451}\beta_{1429}\mathsf{S}\beta_{730}\mathsf{S}\ \big)$)
                        @$^{\mathsf{S}}[\mathsf{S}\rightsquigarrow\beta_{730}]$ ((c-assignment-¿v$^{\beta_{732}}\diamond\big(\ \mathsf{SSS}\beta_{1434}\beta_{1433}\beta_{1427}\mathsf{S}\ \big)$)
                              @$^{\mathsf{S}}$c$^{\beta_{1427}}$))
                        @$^{\mathsf{S}}$env$^{\beta_{1429}}$))
         @$^{\mathsf{S}}$store$^{\mathsf{D}}$)
else  if$^{\beta_{1433}}$ ((isconditional?$^{\beta_{746}}\diamond\big(\ \beta_{1433}\beta_{1434}\beta_{1433}\beta_{1427}\mathsf{S}\ \big)$)
            @$^{\mathsf{S}}$c$^{\beta_{1427}}$)
      then  if$^{\mathsf{D}}$ ((istrue?$^{\beta_{752}}\diamond\big(\ \mathsf{DDS}\ \big)$)
                  @$^{\mathsf{S}}$((((evalexpression$^{\beta_{755}}\diamond\begin{pmatrix}\mathsf{DDDS}\beta_{1451}\beta_{1429}\mathsf{SS}\\ \mathsf{SD}\beta_{1497}\beta_{1497}\end{pmatrix}$)
                        @$^{\mathsf{S}}$((c-conditional-¿e$^{\beta_{758}}\diamond\begin{pmatrix}\mathsf{SSSSS}\beta_{1434}\beta_{1433}\beta_{1427}\\ \mathsf{S}\end{pmatrix}$)
                        @$^{\mathsf{S}}$c$^{\beta_{1427}}$))
                       @$^{\mathsf{S}}$env$^{\beta_{1429}}$)
                      @$^{\mathsf{S}}$store$^{\mathsf{D}}$))
            then  (((((bcc$^{\beta_{901}}$
                        @$^{\beta_{901}}[\mathsf{S}\rightsquigarrow\beta_{1336}]$ 0$^{\mathsf{S}}$)
                        @$^{\beta_{1648}}$((c-conditional-¿b1$^{\beta_{777}}\diamond\begin{pmatrix}\mathsf{SSSSSSS}\beta_{1434}\\ \beta_{1433}\beta_{1427}\mathsf{S}\end{pmatrix}$)
                        @$^{\mathsf{S}}$c$^{\beta_{1427}}$))
                       @$^{\beta_{1652}}$nil)
                      @$^{\beta_{1654}}$env$^{\beta_{1429}}$)
                     @$^{\beta_{1655}}$store$^{\mathsf{D}}$)
            else  (((((bcc$^{\beta_{901}}$
                        @$^{\beta_{901}}[\mathsf{S}\rightsquigarrow\beta_{1336}]$ 0$^{\mathsf{S}}$)
                        @$^{\beta_{1648}}$((c-conditional-¿b2$^{\beta_{797}}\diamond\begin{pmatrix}\mathsf{SSSSSSSS}\\ \mathsf{S}\beta_{1434}\beta_{1433}\beta_{1427}\mathsf{S}\end{pmatrix}$)
                        @$^{\mathsf{S}}$c$^{\beta_{1427}}$))
                       @$^{\beta_{1652}}$nil)
                      @$^{\beta_{1654}}$env$^{\beta_{1429}}$)
                     @$^{\beta_{1655}}$store$^{\mathsf{D}}$)
      else  if$^{\beta_{1433}}$ ((iswhile?$^{\beta_{813}}\diamond\big(\ \beta_{1433}\beta_{1434}\beta_{1433}\beta_{1427}\mathsf{S}\ \big)$)
                                   $@^{\mathsf{S}}\mathrm{c}^{\beta_{1427}})$
                              then  $(((\mathrm{generalize1}^{\beta_{819}\diamond}\left(\ \mathsf{S}\ \right))$
                                   $@^{\mathsf{S}}\mathtt{fix}^{\mathsf{D}}\mathrm{evalwhile}.$
                                       $\lambda^{\mathsf{D}}\mathrm{store1}.$ if$^{\mathsf{D}}$ $((\mathrm{istrue?}^{\beta_{823}\diamond}\left(\ \mathsf{DDS}\ \right))$
                                             $@^{\mathsf{S}}((((\mathrm{evalexpression}^{\beta_{826}\diamond}\left(\begin{array}{l}\mathsf{DDDS}\beta_{1451}\beta_{1429}\mathsf{SS}\\ \mathsf{SD}\beta_{1574}\beta_{1574}\end{array}\right))$
                                             $@^{\mathsf{S}}((\mathrm{c\text{-}while\text{-}¿e}^{\beta_{829}\diamond}\left(\begin{array}{l}\mathsf{SSSSS}\beta_{1434}\beta_{1433}\beta_{1427}\\ \mathsf{S}\end{array}\right))$
                                             $@^{\mathsf{S}}\mathrm{c}^{\beta_{1427}}))$
                                             $@^{\mathsf{S}}\mathrm{env}^{\beta_{1429}})$
                                             $@^{\mathsf{S}}\mathrm{store1}^{\mathsf{D}}))$
                                             then  $(\mathrm{evalwhile}^{\mathsf{D}}$
                                             $@^{\mathsf{D}}(((((\mathrm{bcc}^{\beta_{901}}$
                                             $@^{\beta_{901}}[\mathsf{S}\rightsquigarrow\beta_{1336}]\ 0^{\mathsf{S}})$
                                             $@^{\beta_{1648}}((\mathrm{c\text{-}while\text{-}¿b}^{\beta_{851}\diamond}\left(\begin{array}{l}\mathsf{SSSSSSS}\beta_{1434}\\ \beta_{1433}\beta_{1427}\mathsf{S}\end{array}\right))$
                                             $@^{\mathsf{S}}\mathrm{c}^{\beta_{1427}}))$
                                             $@^{\beta_{1652}}\mathtt{nil})$
                                             $@^{\beta_{1654}}\mathrm{env}^{\beta_{1429}})$
                                             $@^{\beta_{1655}}\mathrm{store1}^{\mathsf{D}}))$
                                             else  $\mathrm{store1}^{\mathsf{D}})$
                                   $@^{\mathsf{D}}\mathrm{store}^{\mathsf{D}})$
                              else  nil
      in if$^{\beta_{1336}}$ $(\mathrm{choice}^{\beta_{1336}}$
                 $=^{\beta_{1336}}[\mathsf{S}\rightsquigarrow\beta_{1336}]\ 0^{\mathsf{S}})$
         then  $(\mathrm{eval\text{-}block}^{\beta_{1648}\diamond}\left(\begin{array}{l}\mathsf{D}\beta_{1655}\beta_{1429}\beta_{1654}\mathsf{S}\beta_{1652}\mathsf{SS}\\ \mathsf{S}\beta_{1648}\end{array}\right))$
         else  if$^{\beta_{1336}}$ $(\mathrm{choice}^{\beta_{1336}}$
                       $=^{\beta_{1336}}[\mathsf{S}\rightsquigarrow\beta_{1336}]\ 1^{\mathsf{S}})$
               then  $(\mathrm{eval\text{-}commands}^{\beta_{1648}\diamond}\left(\begin{array}{l}\beta_{1655}\beta_{1429}\beta_{1654}\mathsf{SSS}\beta_{1652}\mathsf{S}\\ \beta_{1648}\end{array}\right))$
               else  $(\mathrm{eval\text{-}command}^{\beta_{1648}\diamond}\left(\begin{array}{l}\beta_{1655}\beta_{1654}\mathsf{S}\beta_{1652}\mathsf{SSS}\beta_{1648}\\ \beta_{1646}\beta_{1646}\beta_{1645}\beta_{1644}\beta_{1643}\end{array}\right))$
in let $\mathrm{run} = \Lambda\left(\begin{array}{l}\beta_{1704}\beta_{1657}\beta_{960}\beta_{1665}\beta_{1664}\beta_{1663}\beta_{1662}\beta_{1661}\\ \beta_{1660}\beta_{1656}\beta_{961}\beta_{1724}\beta_{1722}\beta_{1723}\beta_{1755}\beta_{1725}\end{array}\right).$
      $\lambda^{\beta_{961}}\mathrm{p}.\ \lambda^{\beta_{960}}\mathrm{val}.$ let $\mathrm{v1} = \Lambda().$
                              $((\mathrm{p\text{-}¿v1}^{\beta_{904}\diamond}\left(\begin{array}{l}\beta_{1664}\beta_{1665}\beta_{1664}\beta_{1663}\beta_{1662}\beta_{1661}\beta_{1660}\beta_{1656}\\ \mathsf{S}\end{array}\right))$
                              $@^{\mathsf{S}}\mathrm{p}^{\beta_{1656}})$
      in let $\mathrm{v2} = \Lambda().$
            $((\mathrm{p\text{-}¿v2}^{\beta_{908}\diamond}\left(\begin{array}{l}\mathsf{SSS}\beta_{1665}\beta_{1664}\beta_{1663}\beta_{1662}\beta_{1661}\\ \beta_{1660}\beta_{1656}\mathsf{S}\end{array}\right))$
            $@^{\mathsf{S}}\mathrm{p}^{\beta_{1656}})$
      in let $\mathrm{env} = \Lambda().$
```

$$
\begin{array}{l}
\qquad\qquad (((\text{init-environment}^{\beta_{912}}\diamond\begin{pmatrix} \mathsf{SSSS}\beta_{1664}\mathsf{S} \end{pmatrix})) \\
\qquad\qquad\quad @^{\mathsf{S}}((\text{p-¿v1}^{\beta_{915}}\diamond\begin{pmatrix} \beta_{1664}\beta_{1665}\beta_{1664}\beta_{1663}\beta_{1662}\beta_{1661}\beta_{1660}\beta_{1656} \\ \mathsf{S} \end{pmatrix}) \\
\qquad\qquad\qquad @^{\mathsf{S}}\text{p}^{\beta_{1656}})) \\
\qquad\qquad\quad @^{\mathsf{S}}(\text{v2}^{\mathsf{S}}\diamond()))) \\
\quad \texttt{in let}\ \text{store} = \Lambda\begin{pmatrix} \beta_{1708}\beta_{1721}\beta_{934}\beta_{1699}\beta_{924} \end{pmatrix}. \\
\qquad\qquad ((((\text{init-store}^{\beta_{923}}\diamond\begin{pmatrix} \beta_{1708}\beta_{1721}\beta_{934}\mathsf{S}\beta_{1704}\beta_{1657}\mathsf{S}\beta_{924} \\ \mathsf{S}\beta_{1699}\beta_{924} \end{pmatrix}) \\
\qquad\qquad\quad @^{\mathsf{S}}[\beta_{1664}\leadsto\beta_{924}]\ ((\text{length}^{\beta_{926}}\diamond\begin{pmatrix} \beta_{1664}\mathsf{S}\beta_{1664}\mathsf{S} \end{pmatrix}) \\
\qquad\qquad\qquad\qquad\qquad @^{\mathsf{S}}(\text{v1}^{\beta_{1664}}\diamond())))) \\
\qquad\qquad\quad @^{\mathsf{S}}\text{val}^{\beta_{1657}}) \\
\qquad\qquad\quad @^{\mathsf{S}}[\mathsf{S}\leadsto\beta_{934}]\ ((\text{length}^{\beta_{936}}\diamond\begin{pmatrix} \mathsf{SSSS} \end{pmatrix}) \\
\qquad\qquad\qquad\qquad @^{\mathsf{S}}(\text{v2}^{\mathsf{S}}\diamond())))) \\
\quad \texttt{in}\ ((((((\text{evalbcc}^{\beta_{941}}\diamond\begin{pmatrix} \mathsf{SSSSSSSS} \\ \beta_{1725}\beta_{1724}\beta_{1723}\beta_{1722} \end{pmatrix}) \\
\qquad\qquad @^{\mathsf{S}}0^{\mathsf{S}}) \\
\qquad\qquad @^{\mathsf{S}}((\text{p-¿b}^{\beta_{946}}\diamond\begin{pmatrix} \mathsf{SSSSS}\beta_{1665}\beta_{1664}\beta_{1663} \\ \beta_{1662}\beta_{1661}\beta_{1660}\beta_{1656}\mathsf{S} \end{pmatrix}) \\
\qquad\qquad\qquad @^{\mathsf{S}}\text{p}^{\beta_{1656}})) \\
\qquad\qquad @^{\mathsf{S}}\texttt{nil}) \\
\qquad\qquad @^{\mathsf{S}}(\text{env}^{\mathsf{S}}\diamond())) \\
\qquad\qquad @^{\mathsf{S}}(\text{store}^{\mathsf{D}}\diamond\begin{pmatrix} \mathsf{DDS}\beta_{1755}\beta_{1664} \end{pmatrix}))) \\
\texttt{in}\ (((\text{run}^{\beta_{962}}\diamond\begin{pmatrix} \mathsf{DDSSSSSS} \\ \mathsf{SSSSSSDS} \end{pmatrix}) \\
\quad @^{\mathsf{S}}\text{p}^{\mathsf{S}}) \\
\quad @^{\mathsf{S}}\text{value}^{\mathsf{D}})
\end{array}
$$

D.2.3 Compiling with MP

We have succesfully compiled programs using the MP-interpreter. The residual/compiled programs are not very readable since the specializer is not memoizing (results in code-duplication) and since it does not do *variable splitting* (the store — being a partially static structure — is constantly being constructed and destructed. For this reason, we have chosen not to show any residual programs.