Appeared in ESOP'94, LNCS 788, p. 287–301
Slightly Revised Version

Polymorphic Binding-Time Analysis

Fritz Henglein & Christian Mossin*

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

Abstract. Binding time analysis is an important part of off-line partial evaluation, annotating expressions as being safely evaluable from known data or possibly depending on unknown data. Most binding-time analyses have been *monovariant*, allowing only one binding-time description for each function. The idea of *polyvariance* is to allow multiple binding time descriptions of each function, by duplicating the function [6, 2] or by associating a set of binding time descriptions to each function [3].
Instead we present an inference based binding time analysis polymorphic in binding time values. This polymorphism captures a very powerful notion of polyvariance limited only by the (standard) types of the language. Polymorphism gives a much simpler definition than the known polyvariant schemes allowing us to reason formally about the system and prove it correct.
This paper is based on work in [14].

1 Introduction

In *binding-time analysis* input is divided into *static* (denoted S) and *dynamic* (denoted D). The job of binding-time analysis is to find out which operations and program phrases can be executed with only static inputs available. Note that the analysis has no knowledge of the actual *values* of the static inputs, only *that* they will be available before the dynamic inputs.

The result of a binding-time analysis can be used to guide the actions of both on-line and off-line partial evaluators. In either case operations classified as static are guaranteed to be executable by the partial evaluator once the static inputs are available. In an on-line partial evaluator the remaining operations, classified as dynamic, require (specialization time) checking to determine whether an operation can be executed or must be deferred to run-time. In an off-line partial evaluator the dynamic operations are automatically deferred to run-time. Thus

* Supported by the Danish Technical Research Council

an off-line partial evaluator is generally more efficient than its on-line counterpart, at the expense of deferring more operations to run-time.[2]

In many existing partial evaluators (*e.g.* Similix [1]) every user-defined function is assigned exactly one binding time description. This description has to be a safe approximation to all calls to the function, so if the function is called with actual parameters of different binding times (*e.g.* with (S,D) and (D,S)), the description of the function can only be a "widened" binding time, in this example (D,D). A *polyvariant* binding-time analysis seeks to remedy this problem by keeping function calls made in different binding-time contexts seperate.

We avoid widening by generalizing type-based monovariant binding-time analysis to a form of *polymorphism* in binding times by adding explicit *abstraction* over binding-time parameters to our binding-time descriptions. Function definitions are *parameterized* over binding times instead of committing these to be S or D. This allows keeping the binding times of the calls to a function separate by instantiating its binding-time parameters to different binding times at the call sites.

2 New Results

We present a polyvariant binding-time analysis for an ML-like language that extends a type-based monovariant binding-time analysis by explicit polymorphism over binding-time parameters. We show the following:

- The type system has the *principal typing* property. This guarantees that every program has a (parameterized) binding-time annotation that subsumes all others for the same program; i.e., it can be used in any context where the program could occur. This admits modular ("local") binding-time analysis of a (function) definition, independent of any of its applications.
- Explicit binding-time application can be interpreted by a specializer as ordinary parameter passing.
- The result of binding-time analysis is an annotation of the *original source* program, not a transformed program. This allows integration into a programming environment and, specifically, support for *static binding-time debugging* [12]. Binding-time debugging is helpful in improving the *binding-time separation* in a program (by rewriting it) [9, 13].
- The binding-time analysis is proved to be *correct* with respect to a canonical semantics (corresponding to a canonical specializer) for binding-time annotated programs.
- The binding-time analysis supports polymorphism in the binding-times independent of the polymorphism of the type discipline of the language. Specifically, it allows binding-time parameterized types both in let-expressions (nonrecursive definitions) and in fix-expressions (recursive definitions). Since

[2] While binding-time analysis seems to have been applied exclusively in off-line partial evaluators this shows that binding-time analysis can be employed with advantage by on-line partial evaluators since it eliminates some checking actions.

the polymorphism in binding times is restricted by the structure of the underlying types this still gives a decidable type inference system.
- The binding-time analysis is *partially complete*. We show that unfolding a definition — whether recursive or not — cannot improve the results of the analysis.
- The analysis is straightforwardly extended to handle structured data. In particular, there is no interference of polyvariance and partially static structures.

We have not systematically analyzed the size of explicitly annotated programs in relation to the original (explicitly typed) source programs. Experimental results indicate, however, that the principal annotation of an individual function definition rarely has more than 10 binding-time parameters after reduction of binding-time constraint sets [14].

3 Related Work

Most of the previous work on polyvariant binding-time analysis is based on abstract interpretation [10, 11, 8]. These binding-time analyses are polyvariant as they provide detailed and almost exact binding-time information for functions and even higher-order functions. Since they are too detailed and "extensional" in nature — i.e., they carry out the analysis without a record of *how* a binding-time value arises — there is no obvious way of using their results in a partial evaluator.

Rytz and Gengler take a pragmatic approach to polyvariant binding-time analysis: monovariant binding-time analysis is trapped when a function's binding-time value would need to be "widened" . Instead they duplicate the function's definition and reiterate both control flow and binding time analysis [6] until no more widening steps occur. Being an extension of Similix' binding time analysis [1] the analysis handles a higher order functional language, but reiterating the analysis is very expensive and no proofs of correctness are given.

Bulyonkov shows how the copying can be accomplished systematically for a first-order language by rewriting the program to contain variables corresponding to binding time values and using a polyvariant (memoizing) *specializer* [2]. This approach makes the technique independent of a particular partial evaluator, and avoids reiterating the analyses.

Consel combines closure and binding time analysis to avoid reiteration [3]. Also, actual copying is avoided by keeping a *set* of different binding-time descriptions with function definitions. The binding-time analysis is extended to such sets of descriptions. The degree of polyvariance is bounded by a function describing how the set of different binding time descriptions is indexed. It appears that in a typed language, this function can be made precise enough to capture the same degree of polyvariance as we obtain. In the partial evaluator Schism [4] a much simpler version of the analysis is implemented. No estimate of the cost of achieving the high degree of polyvariance is given. In its full generality the analysis is still likely to be very expensive, though.

Type-based binding-time analysis for typed languages originated with the Nielsons' development of the two-level λ-calculus for simply-typed λ-expressions [16] and Gomard's inference system for untyped λ-expressions [7]. Their analyses are monovariant, and until recently there was no type-based polyvariant analysis.

Contemporary with our work Consel and Jouvelot have developed a type- and effect-based binding-time analysis similar in spirit to ours [5]. They annotate standard types with binding-time *effects*. These effect-annotated types correspond more or less directly to our binding-time annotated types, though with nested binding-time polymorphism. The resulting binding-time analysis can be seen to be analogous to a "sticky" strictness analysis restricted to the simply typed λ-calculus. The programmer must, however, specify explicitly standard types as well as binding times for all formal parameters. With explicit assumptions for all variables the binding times of expressions are computed automatically. Correctness is formulated relative to an operational semantics geared towards reduction to head-normal form that doubles as a standard and specialization semantics.

4 Language

We will use a monomorphic call-by-name language *Exp* as our source language, which is simple enough to avoid important aspects being blurred by syntactical details, but powerful enough to show the generality of our method. The language syntax (including static semantics) is defined by the type rules of Fig. 1.

$$\text{(const)} \quad A \vdash \mathrm{c}:CT(\mathrm{c})$$

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

$$\text{(if)} \quad \frac{A \vdash \mathrm{e}_1:\mathrm{Bool} \quad A \vdash \mathrm{e}_2:t \quad A \vdash \mathrm{e}_3:t}{A \vdash \mathsf{if}\ \mathrm{e}_1\ \mathsf{then}\ \mathrm{e}_2\ \mathsf{else}\ \mathrm{e}_3 : t}$$

$$\text{(abstr)} \quad \frac{A \bigcup \{\mathrm{x}:t\} \vdash \mathrm{e}:t_1}{A \vdash \lambda \mathrm{x}:t.\mathrm{e} : t \to t_1}$$

$$\text{(appl)} \quad \frac{A \vdash \mathrm{e}_1:t_2 \to t_1 \quad A \vdash \mathrm{e}_2:t_2}{A \vdash \mathrm{e}_1 @ \mathrm{e}_2 : t_1}$$

$$\text{(let)} \quad \frac{A \vdash \mathrm{e}_2:t_2 \quad A \bigcup \{\mathrm{x}:t_2\} \vdash \mathrm{e}_1:t_1}{A \vdash \mathsf{let}\ \mathrm{x}:t_2 = \mathrm{e}_2\ \mathsf{in}\ \mathrm{e}_1:t_1}$$

$$\text{(fix)} \quad \frac{A \bigcup \{\mathrm{x}:t\} \vdash \mathrm{e}:t}{A \vdash \mathsf{fix}\ \mathrm{x}:t.\mathrm{e}:t}$$

Fig. 1. Type System

Here c denotes constants (including integers, booleans and operators on these) and CT is a function mapping constants to their type. Furthermore we have variables, conditional, abstraction, application (denoted @), let-expressions and a fixed point operator.

The denotational semantics of the language is given in Fig. 2. We call this semantics the *standard semantics* in contrast to the semantics of the specializer for annotated *Exp*-programs, to be defined in Section 6.

In the semantics we assume a function CV mapping constant names to their meaning. Function $\lfloor\cdot\rfloor$ maps elements of a domain D into the corresponding lifted domain $D_\perp$ and μ is the fixed point operator.

Semantic Domains:

$V_{bool} = \{T, F\}_\perp$
$V_{int} = \mathbb{N}_\perp$
$V_{t_1 \to t_2} = V_{t_1} \to V_{t_2}$

$\mathcal{E}[\![\mathrm{c}]\!]\rho = \lfloor CV(\mathrm{c})\rfloor$
$\mathcal{E}[\![\mathrm{x}]\!]\rho = \rho(\mathrm{x})$
$\mathcal{E}[\![\mathsf{if}\ \mathrm{e}\ \mathsf{then}\ \mathrm{e}'\mathsf{else}\ \mathrm{e}'']\!]\rho = \mathcal{E}[\![\mathrm{e}]\!]\rho \to \mathcal{E}[\![\mathrm{e}']\!]\rho \parallel \mathcal{E}[\![\mathrm{e}'']\!]\rho$
$\mathcal{E}[\![\lambda \mathrm{x}{:}t.\mathrm{e}]\!]\rho = \lambda v \in V_t.\mathcal{E}[\![\mathrm{e}]\!]\rho[\mathrm{x} \mapsto v]$
$\mathcal{E}[\![\mathrm{e}@\mathrm{e}']\!]\rho = (\mathcal{E}[\![\mathrm{e}]\!]\rho)(\mathcal{E}[\![\mathrm{e}']\!]\rho)$
$\mathcal{E}[\![\mathsf{let}\ \mathrm{x}{:}t = \mathrm{e}'\ \mathsf{in}\ \mathrm{e}]\!]\rho = \mathcal{E}[\![\mathrm{e}]\!]\rho[\mathrm{x} \mapsto \mathcal{E}[\![\mathrm{e}']\!]\rho]$
$\mathcal{E}[\![\mathsf{fix}\ \mathrm{x}.\mathrm{e}]\!]\rho = \mu(\lambda v.\mathcal{E}[\![\mathrm{e}]\!]\rho[\mathrm{x} \mapsto v])$

Fig. 2. Denotational Semantics

5 Binding-Time Analysis

The aim of binding-time analysis is to *annotate* a source program with binding-time information about its individual parts. They are used to control the actions of a program specializer (see Section 6), but they can also be used during program design to convey static binding-time properties to a programmer. Binding-time analysis is the *automatic translation* of a source program to a language enriched with such annotations. In this section we describe the annotation-enriched language *2Exp* and its syntactic properties. Section 6 contains its semantics. The algorithmic aspects will be covered elsewhere (see also [14]).

In the type inference framework, the annotation-enriched language is usually modeled by a *two-level* syntax [16, 7] having two versions of each language construct (with a non-underlined version corresponding to static/unfold and underlined corresponding to dynamic/residualize). We will follow this scheme, but to achieve polymorphism in unfold/residualize, we replace the underline notation by a superscript *annotation* b (binding-time value), which can also be a bound binding-time variable. Further we add *coercions* (*lift*s) and explicit binding-time abstraction and application, all of which are also considered annotations. This gives us the following syntax for annotated expressions *2Exp*:

$$\begin{aligned}\mathrm{e} ::= {} & \mathrm{c} \mid \mathrm{x} \mid \mathsf{if}^b\ \mathrm{e}\ \mathsf{then}\ \mathrm{e}\ \mathsf{else}\ \mathrm{e} \mid \lambda^b \mathrm{x}.\mathrm{e} \mid \mathrm{e}@^b\mathrm{e} \mid \mathsf{let}\ \mathrm{x} = \mathrm{e}\ \mathsf{in}\ \mathrm{e} \mid \mathsf{fix}\ \mathrm{x}.\mathrm{e} \\ & \Lambda\beta.\mathrm{e} \mid \mathrm{e}\{b\} \mid [\kappa \rightsquigarrow \kappa]\mathrm{e}\end{aligned}$$

Note that there are no annotations on let and fix. Operationally this indicates that they can *always* be unfolded — this is semantically safe, since unfolding a let or fix will not depend on dynamic (unavailable) data. It is, however, clear, that unfolding is not always desirable since it might lead to duplication or nonterminating specialization. In our problem set-up controlling unfolding is not part of the binding-time analysis proper, but is left to later analyses (possibly exploiting binding-time information). $\Lambda\beta$.e is abstraction over binding-time variables, $\{\cdot\}$ denotes application of such abtractions to binding-time values and $[\kappa \rightsquigarrow \kappa']$ is a coercion from κ to κ'. The *binding times* b, *compound types* κ and type schemes σ are defined by:

$$\begin{aligned} b &::= \mathsf{S} \mid \mathsf{D} \mid \beta \\ \kappa &::= \kappa \rightarrow^b \kappa \mid \mathrm{Int}^b \mid \mathrm{Bool}^b \\ \sigma &::= \kappa \mid \forall\beta.\sigma \mid\ b \leq b' \Rightarrow \sigma \end{aligned}$$

Compound types are standard types with binding-time superscript annotations on every type and type constructor. *E.g.* $\mathrm{Int}^{\mathsf{D}} \rightarrow^{\mathsf{S}} \mathrm{Bool}^{\mathsf{D}}$ is the type of a statically applicable (the value "S" is the binding-time of "$\rightarrow$") predicate on dynamic integers giving dynamic results. We write t^b to match compound types κ.

The type scheme $\forall\beta.\sigma$ is the type of $\Lambda\beta$.e expressing the parameterization with binding-time values. A *constraint* is a pair of two binding time values written $b_1 \leq b_2$. Constraints $\mathsf{S} \leq \mathsf{S}$, $\mathsf{S} \leq \mathsf{D}$ and $\mathsf{D} \leq \mathsf{D}$ are said to *hold*; $\mathsf{D} \leq \mathsf{S}$ does not hold. Then e:$b \leq b' \Rightarrow \sigma$ is read: if $b \leq b'$ holds then e has type σ. A binding-time judgement has a type environment A and a set of constraints C as assumptions. Thus the $b \leq b' \Rightarrow \sigma$ construct allows us to discharge an assumption and make it explicit in the type.

Our type system for inferring polymorphic binding-time information is given in Fig. 3. We assume a function *2CT* mapping constant names to their type (*e.g.* $\mathit{2CT}(5) = \mathrm{Int}^{\mathsf{S}}$ and $\mathit{2CT}(\mathit{not}) = \forall\beta.\mathrm{Bool}^{\beta} \rightarrow^{\beta} \mathrm{Bool}^{\beta}$).

Variables bound by let or fix can have types polymorphic in *binding-time variables* even though the standard type system is monomorphic. Thus the polymorphism in the analysis does not depend on the availability of polymorphism in the underlying standard type system (though it can, of course, accommodate such). Despite the adoption of a polymorphic binding-time typing rule for recursive definitions the binding-time type system is decidable as binding-time annotations are anchored to the standard types.

We illustrate polymorphism by an annotated Ackermann function:

$$\begin{aligned}&\mathsf{let}\ \mathrm{ack} = \mathsf{fix}\ \mathrm{a}.\Lambda\beta_i\beta_j\beta_{res}. \\ &\qquad \lambda\mathrm{i}.\lambda\mathrm{j}.\mathsf{if}^{\beta_i}\ \mathrm{i}=^{\beta_i}0\ \mathsf{then}\ [\mathrm{Int}^{\beta_j} \rightsquigarrow \mathrm{Int}^{\beta_{res}}](\mathrm{j}+^{\beta_j}1) \\ &\qquad\qquad \mathsf{else}\ \mathsf{if}^{\beta_j}\ \mathrm{j}=^{\beta_j}0\ \mathsf{then}\ [\mathrm{Int}^{\beta_i} \rightsquigarrow \mathrm{Int}^{\beta_{res}}](\mathrm{a}\{\beta_i\mathsf{S}\beta_i\}@^{\mathsf{S}}(\mathrm{i}\text{-}^{\beta_i}1)@^{\mathsf{S}}\ 1) \\ &\qquad\qquad\qquad \mathsf{else}\ \mathrm{a}\{\beta_i\beta_{res}\beta_{res}\} \\ &\qquad\qquad\qquad\qquad @^{\mathsf{S}}(\mathrm{i}\text{-}^{\beta_i}1)@^{\mathsf{S}}(\mathrm{a}\{\beta_i\beta_j\beta_{res}\}@^{\mathsf{S}}\mathrm{i}@^{\mathsf{S}}(\mathrm{j}\text{-}^{\beta_j}1)) \\ &\mathsf{in}\ \mathrm{ack}\{\mathsf{SSS}\}@^{\mathsf{S}}2@^{\mathsf{S}}3+\mathrm{ack}\{\mathsf{SDD}\}@^{\mathsf{S}}2@^{\mathsf{S}}\mathrm{d}+\mathrm{ack}\{\mathsf{DSD}\}@^{\mathsf{S}}\mathrm{d}@^{\mathsf{S}}3\end{aligned}$$

$$\text{(const)} \quad A,C\vdash \mathrm{c}:\mathit{2CT}(\mathrm{c})$$

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

$$\text{(if)} \quad \frac{A,C\vdash \mathrm{e}_1:\mathrm{Bool}^{b_1} \quad A,C\vdash \mathrm{e}_2:t^{b_2} \quad A,C\vdash \mathrm{e}_3:t^{b_2} \quad C\vdash b_1\le b_2}{A,C\vdash \mathsf{if}^{b_1}\ \mathrm{e}_1\ \mathsf{then}\ \mathrm{e}_2\ \mathsf{else}\ \mathrm{e}_3 : t^{b_2}}$$

$$\text{(abstr)} \quad \frac{A\bigcup\{\mathrm{x}:t^b\},C\vdash \mathrm{e}:t_1^{b_1} \quad C\vdash b_2\le b \quad C\vdash b_2\le b_1}{A,C\vdash \lambda^{b_2}\mathrm{x}:t^b.\mathrm{e} : t^b\to^{b_2} t_1^{b_1}}$$

$$\text{(appl)} \quad \frac{A,C\vdash \mathrm{e}_1:t_2^{b_2}\to^b t_1^{b_1} \quad A,C\vdash \mathrm{e}_2:t_2^{b_2} \quad C\vdash b\le b_1 \quad C\vdash b\le b_2}{A,C\vdash \mathrm{e}_1@^b\mathrm{e}_2 : t_1^{b_1}}$$

$$\text{(let)} \quad \frac{A,C\vdash \mathrm{e}_2:\sigma \quad A\bigcup\{\mathrm{x}:\sigma\},C\vdash \mathrm{e}_1:\kappa}{A,C\vdash \mathsf{let}\ \mathrm{x}:\sigma=\mathrm{e}_2\ \mathsf{in}\ \mathrm{e}_1:\kappa}$$

$$\text{(fix)} \quad \frac{A\bigcup\{\mathrm{x}:\sigma\},C\vdash \mathrm{e}:\sigma}{A,C\vdash \mathsf{fix}\ \mathrm{x}:\sigma.\mathrm{e}:\sigma}$$

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

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

$$\text{(}\Rightarrow\text{–introduction)} \quad \frac{A,C\bigcup\{b\le b_1\}\vdash \mathrm{e}:\sigma}{A,C\vdash \mathrm{e} : b\le b_1\Rightarrow\sigma}$$

$$\text{(}\Rightarrow\text{–elimination)} \quad \frac{A,C\vdash \mathrm{e} : b\le b_1\Rightarrow\sigma \quad C\vdash b\le b_1}{A,C\vdash \mathrm{e}:\sigma}$$

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

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

$$\text{(id)} \quad C\vdash b\le b$$

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

$$\text{(use)} \quad \frac{C\vdash b\le b'}{C\vdash \mathsf{Base}^b\le\mathsf{Base}^{b'}}$$

Fig. 3. Polymorphic BTA

Note how the function is used polymorphically both in external and recursive calls. The binding time description of the Ackermann function will be:

$$\forall\beta_i\beta_j\beta_{res}.\beta_i\le\beta_{res}\Rightarrow\beta_j\le\beta_{res}\Rightarrow \mathrm{Int}^{\beta_i}\to\mathrm{Int}^{\beta_j}\to\mathrm{Int}^{\beta_{res}}$$

Definition 1. Every annotated expression e has an underlying standard expression denoted by $|\mathrm{e}|$, which can be obtained by *erasing* annotations. We call $|\mathrm{e}^{ann}|$ the *erasure* of e^{ann}. Conversely, e^{ann} is called a *(binding-time) completion* of $|\mathrm{e}^{ann}|$. Similarly, $|\cdot|$ can be applied to type schemes and (binding time) environments. For expression $e\in \mathit{Exp}$ we write $A,C\vdash e:\sigma$ if e has a completion $e^{ann}\in \mathit{2Exp}$ such that $A,C\vdash e^{ann}:\sigma$. □

We now define the notion of *generic instance*. Our definition differs from the standard one since we have explicit polymorphism and coercions. We introduce

the notation C[ ] for a *coercion context* containing Λ's, binding-time applications and coercions. Note that for any coercion context C[ ] we have $|\mathrm{C[e]}| = |\mathrm{e}|$.

Definition 2. We say σ' is a *generic instance* of σ and write $\sigma \subseteq \sigma'$ if there exists a coercion context C[ ] such that $\mathrm{x} : \sigma \vdash \mathrm{C[x]} : \sigma'$. □

E.g. we have $(\forall\beta_3.\mathrm{Int}^{\mathsf{S}} \rightarrow^{\beta_3} \mathrm{Int}^{\mathsf{D}}) \subseteq (\forall\beta_1.\forall\beta_2.\beta_1 \leq \beta_2 \Rightarrow \forall\beta_3.\mathrm{Int}^{\beta_1} \rightarrow^{\beta_3} \mathrm{Int}^{\beta_2})$.

Definition 3. Given A,C we call σ_{pt} a *principal (binding-time) type* of $\mathrm{e} \in \mathit{Exp}$ under A,C if

- $A, C \vdash \mathrm{e} : \sigma_{pt}$
- $\forall\sigma : A, C \vdash \mathrm{e} : \sigma \Longrightarrow \sigma_{pt} \subseteq \sigma$

□

Our binding-time analysis has the *principality property* for binding-time types (corresponding to a principal typing property, but on binding time values); that is, *all* expressions have a principal type. The significance of the principal typing property for an expression $\mathrm{e} \in \mathit{Exp}$ is that there is a single derivable binding-time type that can be used in a binding-time analysis for *all* possible contexts in which e might occur. This is the key to the modularity and partial completeness of the analysis with respect to unfolding of let- and fix-expressions. (See Section 8.)

Theorem 4. *(Principal Typing Property)*
Every expression $\mathrm{e} \in \mathit{Exp}$ *has a principal type under* A,C *if it has any (binding-time) type under* A,C *at all.*

Proof. By induction over the standard type derivation tree (otherwise similar in style to [15]). □

For $A \vdash \mathrm{e}: t$ it can be seen that e has a (principal) type for any completion A^{ann} of A if all type assumptions are first-order; that is, not of functional type.

A principal type of an expression is clearly not unique due to, amongst other things, reordering of $\forall$ and $\cdot \leq \cdot \Rightarrow \cdot$. Note also that there are generally several different completions of an expression with the *same* (principal) type. This is due to the fact that principality only considers the *externally* visible binding-time behavior of an expression, but not its internal structure.

6 Specialization

In Fig. 4 we present the denotational semantics for binding-time analyzed programs. This is essentially a specializer consuming the information generated by the polymorphic binding-time analysis from the last section.

The semantic function takes two environments: a standard environment ρ, and a binding-time environment η mapping binding time variables to binding-time values in BtV. By abuse of notation we take $\eta(\mathsf{S}) = \mathsf{S}$ and $\eta(\mathsf{D}) = \mathsf{D}$. The specializer is "natural" in the sense that every clause for static expressions (including the rules for let and fix) is identical to the corresponding rule of the standard semantics and all rules for dynamic expressions simply evaluate

Semantic Domains:

$$
\begin{array}{ll}
2V_{\mathrm{Bool}^{\mathsf{S}}} & = \{T, F\}_\perp \\
2V_{\mathrm{Int}^{\mathsf{S}}} & = \mathbb{N}_\perp \\
BtV & = \{\mathsf{S},\mathsf{D}\} \\
2V_{\kappa_1 \to^{\mathsf{S}} \kappa_2} & = 2V_{\kappa_1} \to 2V_{\kappa_2} \\
2V_{t^{\mathsf{D}}} & = Exp \\
2V_{\forall\beta.\sigma} & = BtV \to (2V_\sigma)_\perp \\
2V_{b_1 \le b_2 \Rightarrow \sigma} & = 2V_\sigma
\end{array}
$$

$$
\begin{array}{l}
\mathcal{T}[\![\mathrm{c}]\!]\rho\eta = \lfloor 2CV(\mathrm{c}) \rfloor \\
\mathcal{T}[\![\mathrm{x}]\!]\rho\eta = \rho(\mathrm{x}) \\
\mathcal{T}[\![\mathsf{if}^b\ \mathrm{e}\ \mathsf{then}\ \mathrm{e}'\mathsf{else}\ \mathrm{e}'']\!]\rho\eta = \mathit{case}\ \eta(b)\ \mathit{of} \\
\qquad\qquad \mathsf{S}\text{: } \mathcal{T}[\![\mathrm{e}]\!]\rho\eta \to \mathcal{T}[\![\mathrm{e}']\!]\rho\eta \parallel \mathcal{T}[\![\mathrm{e}'']\!]\rho\eta \\
\qquad\qquad \mathsf{D}\text{: } \mathit{build\text{-}if}(\mathcal{T}[\![\mathrm{e}]\!]\rho\eta, \mathcal{T}[\![\mathrm{e}']\!]\rho\eta, \mathcal{T}[\![\mathrm{e}'']\!]\rho\eta) \\
\mathcal{T}[\![\lambda^b \mathrm{x}{:}t^{b'}.\mathrm{e}]\!]\rho\eta = \mathit{case}\ \eta(b)\ \mathit{of} \\
\qquad\qquad \mathsf{S}\text{: } \lambda v \in (V_{t^{b'}})_\perp . \mathcal{T}[\![\mathrm{e}]\!]\rho[\mathrm{x} \mapsto v]\eta \\
\qquad\qquad \mathsf{D}\text{: } \mathit{build\text{-}\lambda}(\mathrm{x}, t, \mathcal{T}[\![\mathrm{e}]\!]\rho\eta) \\
\mathcal{T}[\![\mathrm{e}@^b\mathrm{e}']\!]\rho\eta = \mathit{case}\ \eta(b)\ \mathit{of} \\
\qquad\qquad \mathsf{S}\text{: } (\mathcal{T}[\![\mathrm{e}]\!]\rho\eta)(\mathcal{T}[\![\mathrm{e}']\!]\rho\eta) \\
\qquad\qquad \mathsf{D}\text{: } \mathit{build\text{-}@}(\mathcal{T}[\![\mathrm{e}]\!]\rho\eta, \mathcal{T}[\![\mathrm{e}']\!]\rho\eta) \\
\mathcal{T}[\![\mathsf{let}\ \mathrm{x}{:}\sigma = \mathrm{e}'\ \mathsf{in}\ \mathrm{e}]\!]\rho\eta = \mathcal{T}[\![\mathrm{e}]\!]\rho[\mathrm{x} \mapsto \mathcal{T}[\![\mathrm{e}']\!]\rho\eta]\eta \\
\mathcal{T}[\![\mathsf{fix}\ \mathrm{x}.\mathrm{e}]\!]\rho\eta = \mu(\lambda v. \mathcal{T}[\![\mathrm{e}']\!]\rho[\mathrm{x} \mapsto v]\eta) \\
\mathcal{T}[\![[\mathsf{Base}^{b_1} \rightsquigarrow \mathsf{Base}^{b_2}]\mathrm{e}]\!]\rho\eta = \mathit{case}\ (\eta(b_1), \eta(b_2))\ \mathit{of} \\
\qquad\qquad (\mathsf{S},\mathsf{D})\text{: } \mathit{build\text{-}const}(\mathcal{T}[\![\mathrm{e}]\!]\rho\eta) \\
\qquad\qquad (\mathsf{S},\mathsf{S}) \text{ or } (\mathsf{D},\mathsf{D})\text{: } \mathcal{T}[\![\mathrm{e}]\!]\rho\eta \\
\qquad\qquad (\mathsf{D},\mathsf{S})\text{: } \mathit{error} \\
\mathcal{T}[\![\Lambda\beta.\mathrm{e}]\!]\rho\eta = (\lambda b. \mathcal{T}[\![\mathrm{e}]\!]\rho\eta[\beta \mapsto b]) \\
\mathcal{T}[\![\mathrm{e}\{b\}]\!]\rho\eta = (\mathcal{T}[\![\mathrm{e}]\!]\rho\eta)(\eta(b))
\end{array}
$$

Fig. 4. Denotational Semantics for Binding-time Analyzed Expressions

subexpressions and emit code (corresponding to off-line partial evaluation). We assume a function *2CV* mapping 2-level constant names to their meaning.

Note that there are now two kinds of values: real values that you can "do computations with" and residual values that are pure syntax to appear in the residual program. The functions *build-if*, *build-*λ and *build-*@ take arguments from $2V_{t^{\mathsf{D}}}$ and return an element in $2V_{t^{\mathsf{D}}}$ (all arguments and the result of appropriate matching types). Function *build-const* takes an argument from $2V_{bool^{\mathsf{S}}}$ or $2V_{int^{\mathsf{S}}}$ and returns an element in $2V_{bool^{\mathsf{D}}}$, *resp.* $2V_{int^{\mathsf{D}}}$.

7 Correctness

In Theorem 9 we state correctness. The definitions and proof is inspired by Gomard [7]. Correctness includes consistency of the specializer, *i.e.* for all closed

e ∈ *2Exp*: if ⊢e:σ then $\mathcal{T}[\![\mathrm{e}]\!]\rho\eta \in 2V_\sigma$ for all ρ,η. We write $\rho \in V_A$ iff for all x bound by ρ, $\rho(x) \in V_{A(x)}$.

Relation $\mathcal{R}$ defines the relationship that (under suitable conditions) must hold between standard evaluation of an expression e and specialization of the same expression: if e is (first order) static, standard evaluation and specialization should yield the same result; if e is (first order) dynamic, standard evaluation should yield the same result as standard evaluation of the specialized expression (we take $\forall\rho_d : \mathcal{E}[\![\bot]\!]\rho_d = \bot$). Note that specialization might not terminate while standard evaluation does; this is expressed in the definition of $\mathcal{R}$ using $\sqsubseteq$. The relation is extended to a logical relation on higher order and polymorphic types.

Definition 5. Given *CV* and *2CV*, the relation $\mathcal{R}_\sigma$ holds for $(\mathrm{e}, \rho_s, \rho_d, \rho, \eta) \in$ *2Exp* × *2VarEnv* × *VarEnv* × *VarEnv* × *BtVarEnv* iff one of the following holds

1. $\sigma = \mathsf{Base}^{\mathsf{S}}$ and $\mathcal{T}[\![e]\!]\rho_s\eta = \mathcal{E}[\![\,|e|\,]\!]\rho$
2. $\sigma = \mathsf{Base}^{\mathsf{D}}$ or $\sigma = \kappa \to^{\mathsf{D}} \kappa'$ and $\mathcal{E}[\![\mathcal{T}[\![e]\!]\rho_s\eta]\!]\rho_d \sqsubseteq \mathcal{E}[\![\,|e|\,]\!]\rho$
3. $\sigma = \kappa' \to^{\mathsf{S}} \kappa$ and $\forall \mathrm{e}' : \mathcal{R}_{\kappa'}(\mathrm{e}', \rho_s, \rho_d, \rho, \eta)$ implies $\mathcal{R}_\kappa(\mathrm{e}@^{\mathsf{S}}\mathrm{e}', \rho_s, \rho_d, \rho, \eta)$
4. $\sigma = \forall\beta.\sigma'$ and $\forall b \in \mathit{BtVal} : \mathcal{R}_{\sigma'[b/\beta]}(\mathrm{e}\{b\}, \rho_s, \rho_d, \rho, \eta)$.
5. $\sigma = b \le b' \Rightarrow \sigma'$ and $C \vdash b \le b'$ implies $\mathcal{R}_{\sigma'}(\mathrm{e}, \rho_s, \rho_d, \rho, \eta)$.

□

The specialization time environment ρ_s, the run time environment ρ_d and the standard evaluation environment ρ should *agree*: If a variable x is (first order) static, ρ_s and ρ hold the same value; if x is (first order) dynamic, ρ_s maps x to a new variable name which when looked up in ρ_d yields the same value as ρ(x). This extends to higher order values in a natural way:

Definition 6. Given a set of identifiers, *VarSet*, three environments, ρ, ρ_s, ρ_d and a type environment A such that $\rho_s \in V_A$, we say that ρ_s, ρ_d, ρ *agree* on *VarSet* iff $\forall \mathrm{x} \in \mathit{Varset}, \forall \eta \in \mathit{BtVarEnv} : \mathcal{R}_{A(\mathrm{x})}(\mathrm{x}, \rho_s, \rho_d, \rho, \eta)$. □

Definition 7. We say that *2CT*,*CV*,*2CV match* iff $\forall \rho_s, \rho_d, \rho, \eta : \mathcal{R}_{2CT_{(\mathrm{c})}}(\mathrm{c}, \rho_s, \rho_d, \rho, \eta)$. □

We say that a binding-time environment $\eta \in$ *BtVarEnv* is *ground* if it maps all binding-time variables to binding-time values S or D. We allow binding-time environments to be applicable to types and type environments in the obvious way (essentially working as substitutions).

Definition 8. Given a ground environment η and a coercion set C, the relation $\eta \models C$ (η *solves* C) is defined by:

$\eta \models \{(b \le b')\}$ iff $(\eta(b), \eta(b')) \in \{(\mathsf{S},\mathsf{S}), (\mathsf{S},\mathsf{D}), (\mathsf{D},\mathsf{D})\}$
$\eta \models C_1 \bigcup C_2$ iff $\eta \models C_1$ and $\eta \models C_2$ □

Theorem 9. *(Correctness)*
$\forall \mathrm{e}, \rho_s, \rho_d, \rho, A, C, \eta, \sigma$, *2CT*, *CV*, *2CV* :

$$\left.\begin{array}{l} \mathit{2CT,CV,2CV}\ \text{match} \\ \eta \models C \\ \rho_s \in V_{\eta A} \\ \rho_s,\rho_d,\rho\ \text{agree on FreeVars(e)} \\ A,C \vdash \mathrm{e}:\sigma \end{array}\right\} \Longrightarrow \mathcal{R}_{\eta\sigma}(\mathrm{e},\rho_s,\rho_d,\rho,\eta)$$

Proof. Proceeds by induction over the derivation of $A, C \vdash \mathrm{e} : \sigma$. The interesting cases are proved in Lemma 14 to 16 which can be found in appendix A. □

8 Partial Completeness

We prove the subject expansion property for let and fix. This implies that performing binding-time analysis after unfolding let- or fix-expressions will not give any better binding-time types. Making different variants of a (top-level) function for each context in which it is used, corresponds to unfolding let and fix bound expressions. Thus our polymorphic binding time analysis yields as good results *without* changing the program at hand as any polyvariant binding time analysis using this technique. This is a *partial completeness* result for our analysis[3] as it shows that the analysis is *invariant* under the equality induced by *let-* and *fix-*unfolding. With *induced* coercions it is also complete with respect to η-equality (not shown here). As can be expected it is *not* complete with respect to (non-linear) β-reduction and simplification of conditionals. Note, however, that the weaker subject *reduction* property holds also for these cases.

If e is an expression with k occurences of variable x, we refer to the i'th occurence as $\mathrm{x}^{(i)}$ and use $\mathrm{e}[\mathrm{e}'/\mathrm{x}^{(i)}]$ to denote the substitution of e' for the i'th occurence of x.

The following lemma states that any type derivation tree can be cut at any point inserting variables with the same type as the cut branch. The converse holds too (that variables can be replaced by expressions of the same type). This is basically the well-known substitution lemma.

Lemma 10. *Let e be an expression with k occurences of variable x. Then*

$$\begin{array}{l} A,C \vdash \mathrm{e}[\mathrm{e}_i/\mathrm{x}^{(i)}]:\sigma \Longleftrightarrow \\ \quad (\exists \sigma_1 \ldots \sigma_k)\ A, C \vdash \mathrm{e}_i : \sigma_i \wedge A \bigcup \{\mathrm{x}_1 : \sigma_1, \cdots, \mathrm{x}_\mathrm{k} : \sigma_\mathrm{k}\}, C \vdash \mathrm{e}[\mathrm{x}_\mathrm{i}/\mathrm{x}^{(\mathrm{i})}] : \sigma \end{array}$$

Proof. By induction over the type inference tree. □

Theorem 11. *(Subject Expansion for* let*)*

$$A, C \vdash \mathsf{let}\ \mathrm{x} = \mathrm{e}'\ \mathsf{in}\ \mathrm{e} : \sigma \Leftrightarrow A, C \vdash \mathrm{e}[\mathrm{e}'/\mathrm{x}] : \sigma.$$

[3] It can't be both computable and totally complete for obvious reasons.

Proof. ($\Rightarrow$) This is the subject-reduction property for let, which follows from Lemma 10. ($\Leftarrow$) Similar to Theorem 13 (see below). □

The following lemma is used in the proof of subject expansion for fix.

Lemma 12. *If σ_{pt} is a principal type of* fix x.e $\in$ *Exp under A, C then σ_{pt} is also a principal type of* e *under* $A \bigcup \{\mathrm{x} : \sigma_{pt}\}, C$.

Proof. Assume it isn't; i.e., there exists σ with $\sigma \subseteq \sigma_{pt}$ and $\sigma_{pt} \not\subseteq \sigma$ such that $A \bigcup \{\mathrm{x} : \sigma_{pt}\}, C \vdash \mathrm{e} : \sigma$. From the definition of generic instantiation, Definition 2, it follows that $A \bigcup \{\mathrm{x} : \sigma\}, C \vdash \mathrm{e} : \sigma$. This implies $A, C \vdash \mathsf{fix}\ \mathrm{x.e} : \sigma$. Since σ_{pt} is principal for fix x.e we get $\sigma_{pt} \subseteq \sigma$, which is in contradiction to our proof assumption. □

Theorem 13. *(Subject Expansion for* fix*)*

$$A, C \vdash \mathsf{fix}\ x.e : \sigma \Leftrightarrow A, C \vdash e[\mathsf{fix}\ x.e/x] : \sigma.$$

Proof. ($\Rightarrow$) (Subject reduction) Follows from Lemma 10. ($\Leftarrow$) (Subject expansion) Let e contain k occurrences of x. Assume $A, C \vdash \mathrm{e}[\mathsf{fix}\mathrm{x.e}/\mathrm{x}] : \sigma$. By Lemma 10 there exist $\sigma_1, \ldots, \sigma_k$ such that

1. $A, C \vdash \mathsf{fix}\ \mathrm{x.e} : \sigma_i$ for $1 \leq i \leq k$, and
2. $A \bigcup \{\mathrm{x}_1 : \sigma_1, \ldots, \mathrm{x}_k : \sigma_k\}, C \vdash \mathrm{e}[\mathrm{x}_i/\mathrm{x}^{(i)}] : \sigma$.

By Theorem 4 we know that fix x.e has a principal type σ_{pt} under A, C. Thus $\sigma_{pt} \subseteq \sigma_i$ for all $1 \leq i \leq k$. From the definition of $\subseteq$, Definition 2, and point 2 it follows that

$$A \bigcup \{\mathrm{x} : \sigma_{pt}\}, C \vdash \mathrm{e} : \sigma.$$

Since, by Lemma 12, σ_{pt} is a principal type for e under $A \bigcup \{$ x: σ_{pt} $\}$, C we obtain $\sigma_{pt} \subseteq \sigma$. Now we can conclude from $A, C \vdash \mathsf{fix}\ \mathrm{x.e} : \sigma_{pt}$ together with $\sigma_{pt} \subseteq \sigma$ that $A, C \vdash \mathsf{fix}\ \mathrm{x.e} : \sigma$. □

9 Implementation

A working prototype implementation of the system exists (described in [14]) using standard type inference methods. Some work is needed to improve the efficiency of the implementation. An important issue in the implementation is to keep the principal types small (to avoid big program size increases). Reductions capable of doing this are described in [14], but they are neither proved correct nor implemented efficiently.

References

1. A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17 (Selected papers of ESOP '90, the 3rd European Symposium on Programming)(1-3):3–34, Dec. 1991.
2. M. A. Bulyonkov. Extracting polyvariant binding time analysis from polyvariant specializer. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 59–65. ACM Press, 1993.
3. C. Consel. Polyvariant binding-time analysis for applicative languages. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 66–77. ACM Press, 1993.
4. C. Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154. ACM Press, 1993.
5. C. Consel and P. Jouvelot. Separate polyvariant binding-time analysis. Research report, Pacific Software Research Center, Oregon Graduate Institute of Science and Technology, Beaverton, Oregon, USA, 1993.
6. M. Gengler and B. Rytz. A polyvariant binding time analysis. In *1992 ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 21–28. Yale University, 1992.
7. C. K. Gomard. A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics. *Transactions on Programming Languages and Systems*, 14(2):147–172, Apr. 1992.
8. 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.
9. J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Proc. 19th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 258–268. ACM Press, Jan. 1992.
10. J. Launchbury. *Projection Factorisations in Partial Evaluation*. PhD thesis, University of Glasgow, Jan. 1990.
11. T. Mogensen. Binding time analysis for polymorphically typed higher order languages. In J. Diaz and F. Orejas, editors, *Proc. Int. Conf. Theory and Practice of Software Development (TAPSOFT)*, pages 298–312, March 1989. LNCS 352.
12. C. Mossin. Similix binding time debugger manual, system version 4.0. Included in Similix distribution, Sept. 1991.
13. C. Mossin. Partial evaluation of general parsers. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 13–21. ACM Press, 1993.
14. C. Mossin. Polymorphic binding time analysis. Master's thesis, DIKU, University of Copenhagen, Denmark, July 1993.
15. A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proc. 6th Int. Conf. on Programming, LNCS 167*, 1984.
16. H. Nielson and F. Nielson. Automatic binding time analysis for a typed lambda calculus. *Science of Computer Programming*, 10:139–176, 1988.

A Proof of Correctness

We prove Theorem 9. Let $\mathcal{H}(A,C\vdash e{:}\sigma)$ be the induction hypothesis. Though Gomard's correctness theorem has been slightly altered *w.r.t.* to standard types,

coercion sets and binding time variables, the induction cases from his proof [7] carry directly over to our proof. We show the cases for fix, binding-time abstraction and binding time application. The let case is similar to the fix case, constrained binding time types are trivial and the reamining cases correspond to cases by Gomard.

Note that without losing generality any type scheme can be written in the form $\forall\beta_1.\cdots\forall\beta_n.b_1' \leq b_1'' \Rightarrow \cdots b_m' \leq b_m'' \Rightarrow \kappa$. Let *2CT*,*CV* and *2CV* be given throughout the proof.

Lemma 14. $\forall \mathrm{e}, \mathrm{e}' : \mathcal{H}(A\bigcup\{\mathrm{x} : \sigma\}, C\vdash \mathrm{e} : \sigma) \Rightarrow \mathcal{H}(A, C\vdash \mathsf{fix}\ \mathrm{x.e}: \sigma)$

Proof. Let ρ_s,ρ_d,ρ and η be given such that the conditions of Theorem 9 hold. Define $\rho_s' = \rho_s[\mathrm{x} \mapsto val]$ and $\rho' = \rho[\mathrm{x} \mapsto val']$, where *val*, *val*′ are given such that $\mathcal{R}(\mathrm{x},\rho_s',\rho_d,\rho',\eta,\sigma)$. From the induction hypothesis we have

$$\mathcal{R}(\mathrm{e},\rho_s',\rho_d,\rho',\eta,\sigma) \tag{1}$$

The idea is to use (1) to prove the induction step of an induction proof. We have that $\eta\sigma$ has the form $\forall\beta_1.\cdots\forall\beta_n.b_1'\leq b_1''\Rightarrow \cdots b_m'\leq b_m''\Rightarrow\kappa_1 \rightarrow^{\mathsf{S}} \cdots \rightarrow^{\mathsf{S}} \kappa_p$ where κ_p is either $\mathsf{Base}^{\mathsf{S}}$ or one of $\mathsf{Base}^{\mathsf{D}}$ or $\kappa' \rightarrow^{\mathsf{D}} \kappa''$.

We only prove the case where $\kappa_p = \mathsf{Base}^{\mathsf{D}}$. Case $\kappa_p = \kappa' \rightarrow \kappa''$ is identical and $\kappa_p = \mathsf{Base}^{\mathsf{S}}$ similar though simpler.

Let $b_1 \ldots b_n$ and $\mathrm{e}_1 \ldots \mathrm{e}_{p-1}$ be given such that $\mathcal{R}(\mathrm{e}_i,\rho_s,\rho_d,\rho,\eta,\kappa_i)$. Now $\mathcal{R}(\mathsf{fix}\ \mathrm{x.e},\rho_s,\rho_d,\rho,\eta,\eta\sigma)$ may also be written

$$\mathcal{R}((\mathsf{fix}\ \mathrm{x.e})\{b_1\}\cdots\{b_n\}@^{\mathsf{S}}\mathrm{e}_1@^{\mathsf{S}}\cdots@^{\mathsf{S}}\mathrm{e}_{p-1}, \rho_s, \rho_d, \rho, \eta, \kappa_p)$$

This is equivalent to

$$\begin{aligned}\mathcal{E}[\![\mathcal{T}[\![(\mathsf{fix}\ \mathrm{x.e})\{b_1\}\cdots\{b_n\}@^{\mathsf{S}}\mathrm{e}_1@^{\mathsf{S}}\cdots@^{\mathsf{S}}\mathrm{e}_{p-1}]\!]\rho_s\eta]\!]\rho_d \sqsubseteq\\ \mathcal{E}[\![\,|(\mathsf{fix}\ \mathrm{x.e})\{b_1\}\cdots\{b_n\}@^{\mathsf{S}}\mathrm{e}_1@^{\mathsf{S}}\cdots@^{\mathsf{S}}\mathrm{e}_{p-1}|\,]\!]\rho\end{aligned}$$

This can be rewritten to

$$\begin{aligned}\mathcal{E}[\![(\mathcal{T}[\![\mathsf{fix}\ \mathrm{x.e}]\!]\rho_s\eta)b_1\cdots b_n(\mathcal{T}[\![\mathrm{e}_1]\!]\rho_s\eta)\cdots(\mathcal{T}[\![\mathrm{e}_{p-1}]\!]\rho_s\eta)]\!]\rho_d \sqsubseteq\\ (\mathcal{E}[\![\,|\mathsf{fix}\ \mathrm{x.e}|\,]\!]\rho)(\mathcal{E}[\![\,|\mathrm{e}_1|\,]\!]\rho)\cdots(\mathcal{E}[\![\,|\mathrm{e}_{p-1}|\,]\!]\rho)\end{aligned}$$

by the semantics we get

$$\begin{aligned}\mathcal{E}[\![\mu(\lambda v.\mathcal{T}[\![\mathrm{e}]\!]\rho_s[\mathrm{x} \mapsto v]\eta)b_1\cdots b_n(\mathcal{T}[\![\mathrm{e}_1]\!]\rho_s\eta)\cdots(\mathcal{T}[\![\mathrm{e}_{p-1}]\!]\rho_s\eta)]\!]\rho_d \sqsubseteq\\ \mu(\lambda v.\mathcal{E}[\![\,|\mathrm{e}|\,]\!]\rho[\mathrm{x} \mapsto v])(\mathcal{E}[\![\,|\mathrm{e}_1|\,]\!]\rho)\cdots(\mathcal{E}[\![\,|\mathrm{e}_{p-1}|\,]\!]\rho)\end{aligned}$$

We prove this by simultaneous fixed point induction. If we use $\perp_{std}$ to denote $(\lambda v_1.\cdots\lambda v_{p-1}.\perp)$ and $\perp_d$ to denote $(\Lambda b v_1.\cdots\Lambda b v_n.\lambda v_1.\cdots\lambda v_{p-1}.\perp)$ we can write the induction base as

$$\mathcal{E}[\![\perp_d b_1\cdots b_n(\mathcal{T}[\![\mathrm{e}_1]\!]\rho_s\eta)\cdots(\mathcal{T}[\![\mathrm{e}_{p-1}]\!]\rho_s\eta)]\!]\rho_d \sqsubseteq \perp_{std}(\mathcal{E}[\![\,|\mathrm{e}_1|\,]\!]\rho)\cdots(\mathcal{E}[\![\,|\mathrm{e}_{p-1}|\,]\!]\rho)$$

If we use f to denote $(\lambda v.\mathcal{E}[\![\,|\mathrm{e}|\,]\!]\rho[\mathrm{x} \mapsto v])$ and f_s to denote $(\lambda v.\mathcal{T}[\![\mathrm{e}]\!]\rho_s[x \mapsto v]\eta)$ then to prove the induction step, we prove for all m:

$$\mathcal{E}[\![(f_s^m \bot_d) b_1 \cdots b_n (\mathcal{T}[\![\mathrm{e}_1]\!]\rho_s\eta) \cdots (\mathcal{T}[\![\mathrm{e}_{p-1}]\!]\rho_s\eta)]\!]\rho_d \sqsubseteq (f^m \bot_{std})(\mathcal{E}[\![\,|\mathrm{e}_1|\,]\!]\rho) \cdots (\mathcal{E}[\![\,|\mathrm{e}_{p-1}|\,]\!]\rho)$$
$$\Longrightarrow \mathcal{E}[\![(f_s^{m+1} \bot_d) b_1 \cdots b_n (\mathcal{T}[\![\mathrm{e}_1]\!]\rho_s\eta) \cdots (\mathcal{T}[\![\mathrm{e}_{p-1}]\!]\rho_s\eta)]\!]\rho_d \quad (2)$$
$$\sqsubseteq (f^{m+1} \bot_{std})(\mathcal{E}[\![\,|\mathrm{e}_1|\,]\!]\rho) \cdots (\mathcal{E}[\![\,|\mathrm{e}_{p-1}|\,]\!]\rho)$$

This follows from (where we exploit the fact, that x does not appear free in e_i)

$$\begin{aligned}
&\mathcal{E}[\![(f_s^{m+1} \bot_d) b_1 \cdots b_n (\mathcal{T}[\![\mathrm{e}_1]\!]\rho_s\eta) \cdots (\mathcal{T}[\![\mathrm{e}_{p-1}]\!]\rho_s\eta)]\!]\rho_d = \\
&\quad \mathcal{E}[\![(\lambda v.\mathcal{T}[\![\mathrm{e}]\!]\rho_s[\mathrm{x} \mapsto v]\eta)(f_s^m \bot_d) b_1 \cdots b_n (\mathcal{T}[\![\mathrm{e}_1]\!]\rho_s\eta) \cdots (\mathcal{T}[\![\mathrm{e}_{p-1}]\!]\rho_s\eta)]\!]\rho_d = \\
&\quad \mathcal{E}[\![(\mathcal{T}[\![\mathrm{e}]\!]\rho_s[\mathrm{x} \mapsto (f_s^m \bot_d)]\eta) b_1 \cdots b_n (\mathcal{T}[\![\mathrm{e}_1]\!]\rho_s\eta) \cdots (\mathcal{T}[\![\mathrm{e}_{p-1}]\!]\rho_s\eta)]\!]\rho_d = \\
&\quad \mathcal{E}[\![\mathcal{T}[\![\mathrm{e}\{b_1\} \cdots \{b_n\}@^{\mathsf{S}}\mathrm{e}_1@^{\mathsf{S}} \cdots @^{\mathsf{S}}\mathrm{e}_{p-1}]\!]\rho_s[\mathrm{x} \mapsto (f_s^m \bot_d)]\eta]\!]\rho_d \quad (3)
\end{aligned}$$

If we in the definition of $\rho_s{}'$ and ρ let *val* assume the value $f_s^m \bot_d$ and *val'* assume the value $f^m \bot_{std}$, then the (local) induction hypothesis gives us $\mathcal{R}(\mathrm{x},\rho_s{}',\rho_d,\rho',\eta,\sigma)$ and thus by (1) we have $\mathcal{R}(\mathrm{e},\rho_s{}',\rho_d,\rho',\eta,\sigma)$. This allows us to rewrite (3) to

$$\begin{aligned}
&\sqsubseteq \mathcal{E}[\![\,|\mathrm{e}\{b_1\} \cdots \{b_n\}@^{\mathsf{S}}\mathrm{e}_1@^{\mathsf{S}} \cdots @^{\mathsf{S}}\mathrm{e}_{p-1}|\,]\!]\rho[\mathrm{x} \mapsto (f_s^m \bot_{std})] = \\
&\quad (\mathcal{E}[\![\,|\mathrm{e}|\,]\!]\rho[\mathrm{x} \mapsto (f^m \bot_{std})])(\mathcal{E}[\![\,|\mathrm{e}_1|\,]\!]\rho) \cdots (\mathcal{E}[\![\,|\mathrm{e}_{p-1}|\,]\!]\rho) = \\
&\quad (\lambda v.\mathcal{E}[\![\,|\mathrm{e}|\,]\!]\rho[\mathrm{x} \mapsto v])(\mathrm{f^m} \bot_{\mathrm{std}})(\mathcal{E}[\![\,|\mathrm{e}_1|\,]\!]\rho) \cdots (\mathcal{E}[\![\,|\mathrm{e}_{\mathrm{p}-1}|\,]\!]\rho) = \\
&\quad (f^{m+1} \bot_{std})(\mathcal{E}[\![\,|\mathrm{e}_1|\,]\!]\rho) \cdots (\mathcal{E}[\![\,|\mathrm{e}_{p-1}|\,]\!]\rho)
\end{aligned}$$

□

Lemma 15. $\forall \mathrm{e} : \mathcal{H}(A, C \vdash \mathrm{e} : \sigma) \Rightarrow \mathcal{H}(A, C \vdash \Lambda\beta.\mathrm{e} : \forall\beta.\sigma)$

Proof. Let ρ_s,ρ_d,ρ and η be given such that the conditions of Theorem 9 hold. Let $b \in BtVal$ be given and let $\eta' = \eta[\beta \mapsto b]$. Since FreeVars(e) = FreeVars($\Lambda\beta$.e), we have by the induction hypothesis that $\mathcal{R}(\mathrm{e},\rho_s,\rho_d,\rho,\eta',\eta'\sigma)$ holds.

We have that $\eta'\sigma$ has the form $\forall\beta_1.\cdots\forall\beta_n.b_1{}' \leq b_1{}'' \Rightarrow \cdots b_m{}' \leq b_m{}'' \Rightarrow \kappa_1 \rightarrow^{\mathsf{S}} \cdots \rightarrow^{\mathsf{S}} \kappa_{p+1}$ where κ_{p+1} is either $\mathsf{Base}^{\mathsf{S}}$ or one of $\mathsf{Base}^{\mathsf{D}}$ or $\kappa' \rightarrow^{\mathsf{D}} \kappa''$. Let $b_1 \cdots b_n \in BtVal$ and $\mathrm{e}_1 \cdots \mathrm{e}_{p-1}$ be given such that $\mathcal{R}(\mathrm{e}_i,\rho_s,\rho_d,\rho,\eta',\kappa_i)$. Now we have

$$\mathcal{R}(\mathrm{e}\{b_1\} \cdots \{b_n\}@^{\mathsf{S}}\mathrm{e}_1@^{\mathsf{S}} \cdots @^{\mathsf{S}}\mathrm{e}_{p-1}, \rho_s, \rho_d, \rho, \eta', \kappa_{p+1})$$

Assume $\kappa_{p+1} = \mathsf{Base}^{\mathsf{S}}$ ($\kappa_{p+1} = \mathsf{Base}^{\mathsf{D}}$ or $\kappa_{p+1} = \kappa' \rightarrow^{\mathsf{D}} \kappa''$ are similar). Then

$$\mathcal{E}[\![\,|\mathrm{e}\{b_1\} \cdots \{b_n\}@^{\mathsf{S}}\mathrm{e}_1@^{\mathsf{S}} \cdots @^{\mathsf{S}}\mathrm{e}_p|\,]\!]\rho = \mathcal{T}[\![\mathrm{e}\{b_1\} \cdots \{b_n\}@^{\mathsf{S}}\mathrm{e}_1@^{\mathsf{S}} \cdots @^{\mathsf{S}}\mathrm{e}_p]\!]\rho_s\eta'$$

This can be rewritten as:

$$(\mathcal{E}[\![\,|\mathrm{e}|\,]\!]\rho)(\mathcal{E}[\![\,|\mathrm{e}_1|\,]\!]\rho) \cdots (\mathcal{E}[\![\,|\mathrm{e}_p|\,]\!]\rho) = (\mathcal{T}[\![\mathrm{e}]\!]\rho_s\eta')b_1 \cdots b_n(\mathcal{T}[\![\mathrm{e}_1]\!]\rho_s\eta') \cdots (\mathcal{T}[\![\mathrm{e}_p]\!]\rho_s\eta')$$

By alpha-conversion can assume that β does not appear free in e_i:

$$(\mathcal{E}[\![\,|\mathrm{e}|\,]\!]\rho)(\mathcal{E}[\![\,|\mathrm{e}_1|\,]\!]\rho) \cdots (\mathcal{E}[\![\,|\mathrm{e}_{\mathrm{p}}|\,]\!]\rho) = (\mathcal{T}[\![\mathrm{e}]\!]\rho_s\eta')\mathrm{b}_1 \cdots \mathrm{b}_{\mathrm{n}}(\mathcal{T}[\![\mathrm{e}_1]\!]\rho_s\eta) \cdots (\mathcal{T}[\![\mathrm{e}_{\mathrm{p}}]\!]\rho_s\eta)$$

Now by using the binding time application rule and the definition of η

$$\begin{aligned}
&(\mathcal{E}[\![\,|(\Lambda\beta.\mathrm{e})\{b\}|\,]\!]\rho)(\mathcal{E}[\![\,|\mathrm{e}_1|\,]\!]\rho) \cdots (\mathcal{E}[\![\,|\mathrm{e}_p|\,]\!]\rho) = \\
&\quad (\mathcal{T}[\![(\Lambda\beta.\mathrm{e})\{b\}]\!]\rho_s\eta)b_1 \cdots b_n(\mathcal{T}[\![\mathrm{e}_1]\!]\rho_s\eta) \cdots (\mathcal{T}[\![\mathrm{e}_p]\!]\rho_s\eta)
\end{aligned}$$

Apply the binding time application rule and the @-rule to get

$$\mathcal{E}[\![\, |\Lambda\beta.\mathrm{e}\{b\}\{b_1\}\cdots\{b_n\}@^{\mathsf{S}}\mathrm{e}_1@^{\mathsf{S}}\cdots@^{\mathsf{S}}\mathrm{e}_p|\,]\!]\rho =$$
$$\mathcal{T}[\![\Lambda\beta.\mathrm{e}\{b\}\{b_1\}\cdots\{b_n\}@^{\mathsf{S}}\mathrm{e}_1@^{\mathsf{S}}\cdots@^{\mathsf{S}}\mathrm{e}_p]\!]\rho_s\eta$$

This implies $\mathcal{R}(\Lambda\beta.\mathrm{e}\{b\},\rho_s,\rho_d,\rho,\eta,\eta([b/\beta]\sigma))$, and the desired conclusion $\mathcal{R}(\Lambda\beta.\mathrm{e},\rho_s,\rho_d,\rho,\eta,\eta(\forall\beta.\sigma))$ follows from the definition of $\mathcal{R}$. □

Lemma 16. $\forall \mathrm{e}, b : \mathcal{H}(A, C\vdash \mathrm{e} : \forall\beta.\sigma) \Rightarrow \mathcal{H}(A, C\vdash \mathrm{e}\{b\} : [b/\beta]\sigma)$

Proof. Let ρ_s,ρ_d,ρ and η be given such that the conditions of Theorem 9 hold. From FreeVars(e) = FreeVars(e{b}), we have $\mathcal{R}(\mathrm{e},\rho_s,\rho_d,\rho,\eta,\eta\forall\beta.\sigma)$ by the induction hypothesis. The conclusion $\mathcal{R}(\mathrm{e}\{b\},\rho_s,\rho_d,\rho,\eta,\eta([b/\beta]\sigma))$ follows immediately. □

This article was processed using the LaTeX macro package with LLNCS style