Separate Polyvariant Binding-Time Analysis

Charles Consel
Pierre Jouvelot

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 93-006

March 1993

Separate Polyvariant Binding-Time Analysis

Charles Consel
Pacific Software Research Center
Oregon Graduate Institute
(consel@cse.ogi.edu)

Pierre Jouvelot
Ecole des Mines de Paris
(jouvelot@cri.ensmp.fr)

Abstract

Binding-time analysis aims at determining which variables can be bound to their values at compile time. This binding-time information is of utmost importance when performing partial evaluation or constant folding on programs. Existing binding-time analyses are global in that they require complete program texts and descriptions of which of their inputs are available at compile time. As a consequence, such analyses cannot be used in programming languages that support modules or separate compilation. Libraries have to be analyzed every time they are used in some program. This is particularly limiting when considering programming in-the-large; any modification of an application results in the reprocessing of all the modules.

This paper presents a new static analysis for higher-order typed functional languages that relies on a type and effect system to obtain *polyvariant* and *separate* binding-time information. By allowing function types to be parametrized over the binding times of their arguments, different uses of the same function can have different binding times via a straightforward instanciation of its type. By using an effect system, we are able to propagate binding-time information across compilation boundaries.

We give a complete description of our binding-time analysis framework, show how both type and binding-time information can be expressed and how they relate to the dynamic semantics, and give a type and binding-time checking algorithm which is proved correct with respect to the static semantics.

1 Introduction

Analyzing the binding-time properties of variables in programs aims at determining when the value of these variables can be computed. If it is at compile time, their binding time is said to be *static*; otherwise, they are not known until run time and their binding time is said to be *dynamic*. The binding-time properties of variables in an expression determine whether it can be evaluated at compile time or at run time. Given a program, binding-time analysis is performed by propagating the binding-time properties of its inputs.

Binding-time analysis is primarily used for partial evaluation [12] and optimizing compilation [1] where it allows some evaluations to be performed at compile time, thus improving the overall efficiency. It has been studied within the framework of abstract interpretation [2, 3, 5, 6, 7, 8, 10, 20] and type systems [11, 17, 18]. Effective treatments of higher-order functions [3, 2] and data structures [3] have been described. Also, inference of binding-time annotations has been shown to be a very efficient approach to performing binding-*time* analysis [11].

However, regardless of the analysis setting, existing binding-time analyses are global (*i.e.*, the analysis can only be performed on complete programs). As a consequence, such analyses cannot be used in programming languages that support modules or separate compilation. Libraries have to be analyzed every time they are used in some program. This is particularly limiting when considering programming in-the-large; any modification of an application results in the reprocessing of all the modules.

To alleviate these drawbacks, this paper introduces a new approach for performing *polyvariant* and *separate* binding-time analysis in higher-order typed functional languages: the binding-time behavior of functions is independent of their applications. Our approach to binding-time analysis relies on a new type and effect system [14, 15] that allows function types to be parameterized over the binding times of their arguments, thus permitting different uses of the same function to have different binding times. A type and effect system is a static system that allows the dynamic properties of programs to be estimated by a compiler. Types determine what expressions compute, while effects here determine when expressions can be computed; effects denote binding times.

Unlike previous work, the binding times manipulated by the static system are not only defined by simple constants, such as static or dynamic, but are also symbolically parameterized over binding times. For example, we capture the binding-time behavior of the abstraction:

```
(lambda (x:Int y:Int) (+ x y))
```

by the boolean function:

$$(\lambda\ (x, y).\ x \vee y).$$

This boolean function defines the following binding-time behavior: an application of the abstraction evaluates to a dynamic value if either of its arguments is dynamic. Noteworthy, this binding-time behavior is determined independently of any given context. We show in this paper that the binding-time behavior of functions are always captured by a simple disjunctive boolean term. As a result, calculating the binding-time value for a given application from such a term is straightforward.

The paper is organized as follows. Section 2 presents the syntax and dynamic semantics of the language. Section 3 describes the binding-time analysis type and effect system, together with a proof of its correctness with respect to the dynamic semantics. Section 4 presents the type and effect checking algorithm, together with a proof of its correctness with respect to the static semantics. We discuss possible extensions to this work in Section 5, and give some concluding remarks in Section 6.

2 Language Definition

Our kernel language is the simply typed lambda-calculus, extended to accommodate effect information that describes binding-time properties. We discuss in Section 5 how this simple language can be extended to a full-fledged programming language. Binding times occur within function types, where they represent latent binding times. A *latent binding time* is the binding time of the function body. It communicates the expected behavior of a function from its point of definition to its point of use and is expressed in terms of the binding time of the function argument. Thus, function types, besides the type of the argument and result, include a name for its argument and a latent binding time, which may use this argument name. Note that this is similar to a dependent type system in that we need to be able to refer to function arguments in function types, but differs in that the binding time of the argument, and not its value, is required.

Let us look at a couple of examples. First, examine the type of the increment function.

$$\texttt{(lambda (x:Int) (+ x 1))} \ : \ (\texttt{x} : \texttt{Int}) \xrightarrow{\texttt{x}} \texttt{Int}$$

Not only does the function type describe the abstraction as a function from Int to Int, but also it captures its binding-time behavior as the boolean function $(\lambda x.\ x)$. It expresses the fact that the binding-time value of an application of this increment function solely depends on the binding-time value of its argument, denoted here by the name of the formal.

Next, consider the following function:

$$\begin{array}{l}\texttt{(lambda (x:Int)} \\ \quad \texttt{(lambda (f:} \ \ (\texttt{y} : \texttt{Int}) \xrightarrow{\texttt{y}} \texttt{Int} \ \texttt{)} \\ \qquad \texttt{(f (f x))))} \ : \ (\texttt{x} : \texttt{Int}) \xrightarrow{\texttt{stat}} (\texttt{f} : (\texttt{y} : \texttt{Int}) \xrightarrow{\texttt{y}} \texttt{Int} \ ) \xrightarrow{\texttt{f} \vee \texttt{x}} \texttt{Int}\end{array}$$

The binding-time description of this function can be read as follows: the first abstraction evaluates to a static value, namely a lambda expression (see below). The resulting function evaluates to a static value if both the functional argument f and integer parameter x are static.

We give below the abstract syntax of our kernel language:

B	::=	I	Binding Time Effects
	|	$\texttt{B}_0 \vee \texttt{B}_1$	Combination of Effects
T	::=	I	Types
	|	$(\texttt{I} : \texttt{T}_1) \xrightarrow{\texttt{B}} \texttt{T}_2$	Function Types
E	::=	I	Expressions
	|	(lambda (I : T) E)	Typed Lambdas
	|	$(\texttt{E}_0\ \texttt{E}_1)$	Applications

For the sake of simplicity, the language does not include constants (see Section 5). Also, note that though our syntax allows for multiple basic type identifiers, only one, say basic, would be strictly needed to perform binding-time analysis. The crucial information is the arrow structure of types, together with their latent binding-time information.

The dynamic semantics of our language is expressed by structural operational semantics [19]. The dynamic semantics presented in figure 1 is composed of a set of rules that inductively defines the evaluation relation $\texttt{E} \rightarrow \texttt{E}'$ on the structure of expressions. Since the partial evaluation process deals with syntactic objects, the dynamic semantics uses a syntactic substitution-based approach; it associates to each expression E its reduced expression. The computed values are closed reduced terms:

$$\texttt{v} \ \in \ \mathit{Value} \quad \text{Values}$$

If the $(\rightarrow\mathrm{E}_d)$ rewriting rule that implements the beta-reduction of the lambda-calculus is standard, the $(\rightarrow\mathrm{I}_d)$ rule dealing with lambda terms is more unusual. Its semantics requires reductions *inside* lambda bodies to be performed; this takes into account the fact that partial evaluation is also performed within function definitions.

$$(\rightarrow\mathrm{I}_d) \quad \frac{\texttt{E} \ \rightarrow \ \texttt{E}'}{\texttt{(lambda (I : T) E)} \ \rightarrow \ \texttt{(lambda (I : T) E}'\texttt{)}}$$

$$(\rightarrow\mathrm{E}_d) \quad \frac{\begin{array}{rcl}\texttt{E}_0 & \rightarrow & \texttt{(lambda (I : T) E}'\texttt{)} \\ \texttt{E}_1 & \rightarrow & \texttt{E}'_1 \\ \texttt{E[E}'_1\texttt{/I]} & \rightarrow & \texttt{E}''\end{array}}{(\texttt{E}_0\ \texttt{E}_1) \ \rightarrow \ \texttt{E}''}$$

Figure 1: Dynamic Semantics

We note $\texttt{E[E}'\texttt{/I]}$ the syntactic substitution of I by E′ in E.

3 Static Semantics

The static semantics specifies the type and binding-time information of expressions. We assume that the types of function arguments are provided by the programmer; the return type and binding time of function bodies are left unspecified and are automatically reconstructed by the type checking algorithm. We discuss the rationale behind this restriction in Section 5.

3.1 Binding-Time and Type Algebras

The binding-time algebra is a boolean lattice with the binding-time combination operator $\vee$ as the lattice operator and the predefined identifier stat for bottom and dyn for top.

$$\begin{array}{rcll}\texttt{B}_0 \vee (\texttt{B}_1 \vee \texttt{B}_2) & = & (\texttt{B}_0 \vee \texttt{B}_1) \vee \texttt{B}_2 & \text{Associative} \\ \texttt{B}_0 \vee \texttt{B}_1 & = & \texttt{B}_1 \vee \texttt{B}_0 & \text{Commutative} \\ \texttt{B} \vee \texttt{B} & = & \texttt{B} & \text{Idempotent} \\ & & & \\ \texttt{B} \vee \texttt{stat} & = & \texttt{B} & \text{Unitary} \\ \texttt{B} \vee \texttt{dyn} & = & \texttt{dyn} & \text{Absorbing}\end{array}$$

The standard ordering relation is noted $\sqsubseteq$ and is defined in the usual way:

$$\mathtt{B_0} \sqsubseteq \mathtt{B_1} \iff (\mathtt{B_0} \vee \mathtt{B_1}) = \mathtt{B_1}$$

The ordering relation for elements of the type language is also denoted by $\sqsubseteq$. Types are partially ordered by a subtyping relation that extends the subeffecting relation to types. The $\sqsubseteq$ relation is defined by structural induction. The interesting case deals with function types; monotonicity on latent binding times and return types, and antimonotonicity on argument types:

$$(\mathtt{I}:\mathtt{T_1}) \xrightarrow{\mathtt{B}} \mathtt{T_2} \sqsubseteq (\mathtt{I}:\mathtt{T'_1}) \xrightarrow{\mathtt{B'}} \mathtt{T'_2} \iff \begin{cases} \mathtt{B} \sqsubseteq \mathtt{B'} \\ \mathtt{T'_1} \sqsubseteq \mathtt{T_1} \\ \mathtt{T_2} \sqsubseteq \mathtt{T'_2} \end{cases}$$

Since an argument name may appear within the latent binding time or the return type of function types, function types are scoping constructs. We therefore assume that alpha-renaming within function types is used whenever necessary to avoid capture problems.

3.2 Typing Rules

The type and binding-time static semantics, presented in figure 2, is defined by the relation of evaluation $A \vdash \mathtt{E} : \mathtt{T} \# \mathtt{B}$. Given a type environment A, the static semantics associates to each expression E a type T and its binding-time value B.

$$(\mathrm{Conv}_s)\quad \frac{\begin{array}{c} \mathtt{T} \sqsubseteq \mathtt{T'} \\ \mathtt{B} \sqsubseteq \mathtt{B'} \\ A \vdash \mathtt{E} : \mathtt{T} \# \mathtt{B} \end{array}}{A \vdash \mathtt{E} : \mathtt{T'} \# \mathtt{B'}}$$

$$(\mathrm{Var}_s)\quad \frac{\mathtt{I}:\mathtt{T} \in A}{A \vdash \mathtt{I} : \mathtt{T} \# \mathtt{I}}$$

$$(\rightarrow\mathrm{I}_s)\quad \frac{A[\mathtt{I}:\mathtt{T}] \vdash \mathtt{E} : \mathtt{T'} \# \mathtt{B}}{A \vdash (\mathtt{lambda}\,(\mathtt{I}:\mathtt{T})\,\mathtt{E}) : (\mathtt{I}:\mathtt{T}) \xrightarrow{\mathtt{B}} \mathtt{T'} \ \# \ \mathtt{stat}}$$

$$(\rightarrow\mathrm{E}_s)\quad \frac{\begin{array}{c} A \vdash \mathtt{E_0} : (\mathtt{I}:\mathtt{T_1}) \xrightarrow{\mathtt{B}} \mathtt{T_0} \ \# \ \mathtt{B_0} \\ A \vdash \mathtt{E_1} : \mathtt{T_1} \# \mathtt{B_1} \end{array}}{A \vdash (\mathtt{E_0}\ \mathtt{E_1}) : \mathtt{T_0}[\mathtt{B_1}/\mathtt{I}] \ \# \ \mathtt{B_0} \vee \mathtt{B}[\mathtt{B_1}/\mathtt{I}]}$$

Figure 2: Static Semantics

The (Conv_s) rule allows both types and binding times to be upgraded to higher values in their respective lattice. Note however that it is always safe to bound the binding time of an expression with the disjunction of its free expression variables; the dynamic semantics shows that its evaluation can only depends on them.

In the $(\rightarrow\mathrm{I}_s)$ rule, the type and binding time of the function body are integrated into the function type as its result type and latent binding time. This information is used when a function is applied in the $(\rightarrow\mathrm{E}_s)$ rule. The binding time of the whole expression includes (1) the binding time of the function, since partially evaluating an application requires the function value, and (2) the latent binding time, in which the binding time of the argument is substituted for the argument name in order to account for the binding time of the actual. Note that the substitutions performed in the $(\rightarrow\mathrm{E}_s)$ rule ensure that no bound expression variables remain within types or binding times after their lexical scope is exited.

Since we are interested in applications of binding-time information to program transformations, our static semantics assigns a stat binding time to a lambda expression; the meaning here is that the expression is available at compile time. If we wanted to define the binding time of an expression as stat when it can actually be *evaluated* at compile time, a more appropriate rule would have been:

$$(\rightarrow\mathrm{I}_s)\quad \frac{A[\mathtt{I}:\mathtt{T}] \vdash \mathtt{E} : \mathtt{T'} \# \mathtt{B}}{A \vdash (\mathtt{lambda}\,(\mathtt{I}:\mathtt{T})\,\mathtt{E}) : (\mathtt{I}:\mathtt{T}) \xrightarrow{\mathtt{B}} \mathtt{T'} \ \# \ \mathtt{B} - \mathtt{I}}$$

where the $-$ operation is defined as $(x \vee y) - x = y$[1].

3.3 Correctness Theorem

A major issue is to relate the static information to the dynamic semantics. The precise definitions are given in the Appendix A but the intuition behind the following adequacy theorem is as follows. If a given expression E has a type T and a binding-time value B, then, after applying a substitution noted $[_{\mathtt{B},\mathtt{L}}]$ that maps every identifier in B to a value, the resulting expression can be reduced according to the dynamic semantics to a value $\mathtt{v}^*$ without error. Note that the free variables of E that do not appear in B are *not* substituted, and yet the evaluation process succeeds; this is expected since their values are not required according to the binding-time value B.

Theorem 1 (Adequacy) *If* $A \vdash \mathtt{E} : \mathtt{T} \# \mathtt{B}$, *then* $\mathtt{E}^{\mathtt{T}}[_{\mathtt{B},\mathtt{L}}] \rightarrow \mathtt{v}^*$ *and* $\phi \vdash \mathtt{v}^* : \mathtt{T}[_{\mathtt{L}}] \# \mathtt{B'}$ *such that* $\mathtt{B'} \sqsubseteq \mathtt{B}[_{\mathtt{L}}]$.

where ϕ denotes the empty typing environment. We denote $\mathtt{E}^{\mathtt{T}}$ the expression E of type T in which all subexpressions have been tagged with their appropriate type.

Proof: See Appendix A. □

Note that we do not have to worry here about termination conditions since the language, being an extension of the simply typed lambda-calculus, is strongly normalizing. If a fixpoint operation were introduced in the language (see Section 5), the theorem hypothesis would naturally have to include a restriction stating that E terminates.

4 Type and Binding-Time Effect Checking

In the section we show how type and binding-time information can be checked in expressions.

4.1 Type Unification

The unification algorithm $\mathcal{U}$, displayed in figure 3, checks for the type inequality of $\mathtt{T_1}$ and $\mathtt{T_2}$. It returns the set of effect equalities that must be satisfied for the two types to be considered related by the $\sqsubseteq$ relation. We note ν effect unification variables; they are used to convert effect inequalities into equalities.

Note the use of new *constants* when checking function types. The idea is that the two latent binding-time effects and two return types

[1]Using the unitary rule, one can see that $x - x =$ stat.

```
$\mathcal{U}$( T$_1$, T$_2$ ) =
let fail = $\{(N_1,N_2)\}$ in
case T$_1$ in
I$_1$ =>
        if $T_2 = I_1$ then $\emptyset$ else fail
$T_1 = (I_1 : T_{11}) \xrightarrow{B_1} T_{12}$  =>
        if $T_2 = (I_2 : T_{21}) \xrightarrow{B_2} T_{22}$ then
                let $\mathcal{C}_1$ = $\mathcal{U}(T_{21}, T_{11})$
                let new constant N
                let new $\nu$
                let $\mathcal{C}_2$ = $\mathcal{U}(T_{12}[N/I_1], T_{22}[N/I_2])$
                $\mathcal{C}_1 \cup \mathcal{C}_2 \cup \{(B_1[N/I_1] \vee \nu, B_2[N/I_2])\}$
        else fail
else fail
```
where N_1 and N_2 are two distinct fresh constants.

Figure 3: Unification Algorithm

must be equal whatever the binding-time value of the argument is; using a fresh new constant ensures that this equality will be independent of the argument.

A *model* m of a constraint set $\mathcal{C}$, noted $m \models \mathcal{C}$, is an assignment of the effect variables ν of $\mathcal{C}$ such that, for every equality (B_0, B_1) in $\mathcal{C}$, one has $mB_0 = mB_1$.

The unification algorithm $\mathcal{U}$ is sound if, whenever there exists a model that satisfies the binding-time constraint set, the two types are related by the subtyping relationship.

Theorem 2 (Soundness) $\exists m \models \mathcal{U}(x,y) \Longrightarrow x \sqsubseteq y$

The unification algorithm $\mathcal{U}$ is complete if, whenever two types are subtypes, there exists a model for the constraint set returned by $\mathcal{U}$.

Theorem 3 (Completeness) $x \sqsubseteq y \Longrightarrow \exists\ m \models \mathcal{U}(x,y)$

4.2 Type Checking Algorithm

The type and binding-time checking algorithm $\mathcal{R}$ is presented in figure 4. In a type environment A that binds variables to their type, for an expression E, $\mathcal{R}$ returns its type T, its binding time B and an effect constraint $\mathcal{C}$. The actual types and binding times of an expression (there are many of them via the (Conv$_s$) rule) can be obtained by instanciating T and B with any model of $\mathcal{C}$.

Note the use of new effect variables ν to implement the subeffecting (Conv$_s$) rule. The subtyping rule is taken care of by the unification algorithm, which also uses new effect variables to deal with subeffecting relations.

4.3 Correctness Theorems

This section states and proves that the type and binding-time checking algorithm is correct with respect to the static semantics.

Theorem 4 (Termination) *$\mathcal{R}(A, E)$ terminates.*

```
$\mathcal{R}$( $A$, E ) = case E in
I =>
        if $I : T \in A$ then
              let new $\nu$
              $(T, I \vee \nu, \phi)$
        else fail
(lambda (I : T) E) =>
        let $(T', B', \mathcal{C})$ = $\mathcal{R}$( $A$[I : T], E )
        let new $\nu$
        $((I : T) \xrightarrow{B'} T' , \nu, \mathcal{C})$
(E$_0$ E$_1$) =>
        let $(T_0, B_0, \mathcal{C}_0)$ = $\mathcal{R}$( $A$, E$_0$ )
        let $(T_1, B_1, \mathcal{C}_1)$ = $\mathcal{R}$( $A$, E$_1$ )
        if $(I : T) \xrightarrow{B} T'$ = T$_0$ then
              let $\mathcal{C}$ = $\mathcal{U}$( T$_1$, T )
              let B' = $B_0 \vee B[B_1/I]$
              let new $\nu$
              $(T'[B_1/I], B' \vee \nu, \mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C})$
        else fail
else fail
```

Figure 4: Type Checking Algorithm

Proof: Algorithm $\mathcal{R}$ solely recurses on proper subparts of each expression. □

The following theorem ensures the soundness of the type checking algorithm with respect to the inference rules. It states that any model of the binding-time constraint set returned by $\mathcal{R}$ is appropriate to give ground types and binding times consistent with the static semantics.

Theorem 5 (Soundness) *Let* $(T, B, \mathcal{C}) = \mathcal{R}(A, E)$ *and* m *be a model:*

$$(m \models \mathcal{C}) \Longrightarrow A \vdash E : mT \ \# \ mB$$

Proof: See Appendix B. □

The following theorem ensures the completeness of the type checking algorithm with respect to the inference rules. If the static semantics ensures that an expression has a type and a binding time, then the checking algorithm returns a proper type, binding time and constraint set. Moreover, this constraint set admits a model that relates the computed type and binding time to their assumed ones.

Theorem 6 (Completeness) *If* $A \vdash E : T \ \# \ B$*, there exists* T'*,* B'*,* $\mathcal{C}'$ *and a model* m' *such that:*

$$\begin{cases} (T', B', \mathcal{C}') = \mathcal{R}(A, E) \\ m' \models \mathcal{C}' \\ B = m'B' \\ m'T' \sqsubseteq T \end{cases}$$

Proof: See Appendix C. □

We are left with the issue of determining appropriate models for binding time constraint sets.

Theorem 7 (Decidability) *Every constraint set constructed by $\mathcal{R}$ on a type-correct expression admits a model.*

Proof: This theorem is an immediate consequence of the completeness theorem and the fact that the binding-time value dyn can always be conservatively assigned to every type-correct expression in a program. □

Practically, the following algorithm can be used to determine a constraint set model. First, non-deterministically choose which effect variables are to be assigned the binding time dyn. This eliminates some of the now trivial equations since dyn is absorbent. The resulting constraint set can be then solved using the algorithm defined in [13] on a lattice with top element.

Note that constraint sets may admit multiple models. For example, with the following program:

$$\begin{array}{l}
\texttt{(lambda (f:} \quad (\mathtt{x}:\mathtt{Int}) \xrightarrow{\mathtt{stat}} \mathtt{Int}\ \texttt{)} \\
\quad \texttt{(lambda (g:} \quad (\mathtt{y}:(\mathtt{x}:\mathtt{Int}) \xrightarrow{\mathrm{dyn}} \mathtt{Int}\ ) \xrightarrow{\mathtt{stat}} \mathtt{Int}\ \texttt{)} \\
\quad\quad \texttt{(g (lambda (x:Int) (f x))))))}
\end{array}$$

the reconstruction algorithm introduces a unification variable $\nu_{\mathtt{f}}$ for the function call (f x), while the application to g, which uses subtyping, also introduces via the unification algorithm a unification variable ν. The constraint that must be satisfied for the lambda expression to be compatible with the explicit type of g is thus:

$$\nu_{\mathtt{f}} \vee \nu = \mathrm{dyn}$$

This constraint has multiple solutions, which are not comparable via the subeffecting relationship. There are thus, in general, no minimal models for constraint sets.

Since constraint sets always have at least one solution, every type-correct program is effect-correct. However, if the user were interested in checking that a particular expression has a given binding-time value, it would be easy to add a new special form to the language, such as (ensures B E), which would ensure that, among the possible binding-time values of E, one is lower than B in the binding-time lattice.

5 Extensions

This section addresses some possible language extensions of our framework to deal with polymorphism, constants and data structures, recursion, side effects, and more advanced type reconstruction.

Polymorphism

Since the language defined previously is pure, let polymorphism *à la* Standard ML [16] could easily be added to it using the following rewriting:

$$\texttt{(let (}\mathrm{I_0}\ \mathrm{E_0}\texttt{)}\ \mathrm{E_1}\texttt{)} \quad \rightarrow \quad \mathrm{E_1}[\mathrm{E_0}/\mathrm{I_0}]$$

This rewriting is purely syntactic and does not require any change to either the static or the dynamic semantics.

Polymorphism would thus be introduced by specifying that the expression to which $\mathrm{I_0}$ is bound is replicated within the let body, thus relaxing the constraints between the different uses of $\mathrm{I_0}$. Note however that the type checking algorithm would not necessarily have to be applied on the larger, substituted expression; using caching mechanisms, a more efficient implementation could be designed.

Data structures

The language defined in Section 2 does not support constants and, more generally, data structures. Constants could be straightforwardly added to it by extending the initial dynamic and static environments to constant identifiers.

Arbitrary data structures could be implemented as higher-order functions, taking advantage of polymorphism to deal with the various types for which they are used.

Side Effects

Since type and effect systems have been explicitly designed to deal with imperative constructs in the presence of higher-order functions, one could extend our binding-time analysis to functional languages that admit mutation operations, following the lines of [21]. Reference values can be treated as any other data, since types carry enough information through latent binding times to perform binding-time analysis. Our technique would thus be able to perform, within a unique framework, binding-time analysis of both functional and imperative language features, an interesting prospect[2].

Recursion

As defined in Section 2, our language is strongly normalizing. One way to add general recursion would be to introduce a set of fixpoint combinator constants $\mathtt{Y_t}$ for each type t and provide a suitable type for them, such as:

$$\mathtt{Y_t} \ : \qquad (\mathtt{f}:(\mathtt{x}:\mathtt{t}) \xrightarrow{\mathrm{dyn}} \mathtt{t}\ ) \xrightarrow{\mathtt{stat}} (\mathtt{y}:\mathtt{t}) \xrightarrow{\mathrm{dyn}} \mathtt{t}$$

The subtyping rule would coerce any function type to admit a dynamic latent binding time, thus ensuring that a conservative binding time is deduced for the fixpoint value.

Although correct, this approach is not particularly attractive since, for instance, calls with static arguments to recursive functions would be flagged as dynamic. Beside extending the syntax of binding-time expressions, which would complicate the type checking process, a better way to deal with this problem would be to introduce a rec construct in the language with the following dynamic semantics:

$$(\mathrm{Rec}_d) \quad \frac{\begin{array}{rcl} \mathrm{T} & = & (\mathrm{I_1}:\mathrm{T_1}) \xrightarrow{\mathrm{B_0}} \mathrm{T_0} \\ \mathrm{E} & = & (\mathtt{rec}\ (\mathrm{I}:\mathrm{T}\ \mathrm{I_1})\ \mathrm{E_0}) \end{array}}{\mathrm{E} \ \rightarrow \ (\mathtt{lambda}\ (\mathrm{I_1}:\mathrm{T_1})\ \mathrm{E_0}[\mathrm{E}/\mathrm{I}])}$$

while its static semantics could be defined by the following rule:

$$(\mathrm{Rec}_s) \quad \frac{\begin{array}{rcl} \mathrm{T} & = & (\mathrm{I_1}:\mathrm{T_1}) \xrightarrow{\mathrm{B_0}} \mathrm{T_0} \\ A[\mathrm{I}:\mathrm{T}][\mathrm{I_1}:\mathrm{T_1}] & \vdash & \mathrm{E_0} \ : \ \mathrm{T_0}\ \#\ \mathrm{B_0} \end{array}}{A \ \vdash \ (\mathtt{rec}\ (\mathrm{I}:\mathrm{T}\ \mathrm{I_1})\ \mathrm{E_0}) : \ \mathrm{T}\ \#\ \mathtt{stat}}$$

Note that adding recursion to the language implies that the adequacy theorem of Section 3 must be restricted to terminating expressions. Ultimately, partial evaluation may also fail to terminate in this case; the approach suggested in [9] can be used to detect expressions that are always terminating.

[2]Note that adding mutation operations would badly interact with the simple-minded polymorphism scheme presented above, see [21] for a better approach.

Advanced Reconstruction

The paper presents a static system in which argument types have to be provided by the programmer. Some kind of reconstruction is already performed by our system, since neither the binding time, nor the return type of function bodies have to be explicitly provided.

The main difficulty that prevents us from going beyond this level of type reconstruction, *i.e.* allowing more types to be omitted by the programmer, is that function types include latent binding times which depend on the function argument name. Thus, binding times actually are functions from argument names to binding expressions. Allowing them to be left out in the static semantics would require the type and effect reconstruction algorithm to infer these functions. This seems to require some sort of second-order unification, which is undecidable.

We view our current approach as a first step towards a more flexible system and are currently looking at ways to make this reconstruction problem tractable, for instance by performing partial reconstruction (*e.g.*, non-function types could easily be reconstructed).

6 Conclusion

We presented a new static analysis approach to obtain *polyvariant* and *separate* binding-time information for higher-order typed functional languages. It relies on a type and effect system . By allowing function types to be parameterized over the binding times of their arguments, different uses of the same function can have different binding times via a straightforward instanciation of its type.

We give a complete description of our binding-time analysis framework, show how both type and binding-time information can be expressed, how they relate to the dynamic semantics and give a type and binding-time checking algorithm which is proved correct with respect to the static semantics.

Our approach can also be used to control the program transformation process. Indeed, experience in specializing programs shows that in some cases constants should not be propagated to avoid code explosion or non-termination. These problems are usually circumvented by either providing some analysis or asking the user to supply annotations aimed at controlling the propagation process. In our framework, controlling constant propagation is naturally captured by the type language. This provides the user (or analyses) with a uniform and high-level way of annotating programs.

Our approach could be extended to other applications such as strictness analysis. In fact, as shown in [4] for first-order programs, the strictness behavior of a function is naturally captured by a boolean function, just like the binding-time behavior. Our preliminary studies in applying our approach to the strictness problem are very promising.

Acknowledgments

We thank Vincent Dornic for his careful reading of drafts of this paper.

References

[1] A. D. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] A. Bondorf. Automatic autoprojection of higher order recursive equations. In N. D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 70–87. Springer-Verlag, 1990.

[3] C. Consel. Binding time analysis for higher order untyped functional languages. In *ACM Conference on Lisp and Functional Programming*, pages 264–272, 1990.

[4] C. Consel. Fast strictness analysis via symbolic fixpoint iteration. Research Report 867, Yale University, New Haven, Connecticut, USA, 1991.

[5] C. Consel. Polyvariant binding-time analysis for higher-order, applicative languages. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1993. To appear.

[6] C. Consel and S. C. Khoo. Parameterized partial evaluation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 92–106, 1991.

[7] C. Consel and S. C. Khoo. Parameterized partial evaluation (extended version). Research Report 865, Yale University, New Haven, Connecticut, USA, 1991. To appear in *Transactions on Programming Languages and Systems.* Extended version of [6].

[8] C. Consel and S.C. Khoo. On-line & off-line partial evaluation: Semantic specifications and correctness proofs. Research Report 896, Yale University, New Haven, Connecticut, USA, 1992.

[9] V. Dornic, P. Jouvelot, and D. K. Gifford. Polymorphic time systems for estimating program complexity. *ACM Letters on Programming Languages and Systems*, 1(1), 1992.

[10] M. Gengler and B. Rytz. A polyvariant binding time analysis handling partially known values. In *Workshop on Static Analysis*, volume 81-82 of *Bigre Journal*, pages 322–330. IRISA, Rennes, France, 1992.

[11] F. Henglein. Efficient type inference for higher-order binding-time analysis. In *FPCA'91, 5th International Conference on Functional Programming Languages and Computer Architecture*, pages 448–472, 1991.

[12] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, 1985.

[13] P. Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *ACM Symposium on Principles of Programming Languages*, pages 303–310, 1991.

[14] J. M. Lucassen. *Types and Effects, towards the integration of functional and imperative programming.* PhD thesis, M.I.T (L.C.S. LAB.), Massachusetts, U.S.A, 1987. TR-408.

[15] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *ACM Symposium on Principles of Programming Languages*, pages 47–57, 1988.

[16] R. Milner, M. Tofte, and R. Harper. *The Definition of ML.* MIT Press, 1990.

[17] T. Mogensen. Binding time analysis for polymorphically typed higher order languages. In J. Diaz and F. Orejas, editors, *International Joint Conference on Theory and Practice of Software Development*, volume 352 of *Lecture Notes in Computer Science*, pages 298–312. Springer-Verlag, 1989.

[18] H. R. Nielson and F. Nielson. Automatic binding time analysis for a typed λ-calculus. In *ACM Symposium on Principles of Programming Languages*, pages 98–106, 1988.

[19] G. D. Plotkin. *A Structural Approach To Operational Semantics.* University of Aarhus, Aarhus, Denmark, 1981.

[20] B. Rytz and M. Gengler. A polyvariant binding time analysis. In C. Consel, editor, *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 21–28. Yale University, 1992. Research Report 909.

[21] J-P. Talpin and P. Jouvelot. The type and effect discipline. In *IEEE Symposium on Logic in Computer Science*, 1992.

A Adequacy

The language is extended with a set of dedicated values $\top^{\mathtt{T}}$ for any type T. Intuitively, such values denote computations that go wrong. The domain *Value** is defined in the following way:

Definition 1 $\mathtt{v}^* \in Value^*$ *if and only if* $\mathtt{v}^* \in Value$ *and* $\top^{\mathtt{T}}$ *does not occur in* $\mathtt{v}^*$ *for any type* T.

We assume that, for any type T, values $\mathtt{v}^*$ and v of that type exist.

Definition 2 (Rebinding) *A* rebinding $[_{\mathtt{L}}]$ *is a scope-preserving substitution upon binding-time identifiers within types and binding-time expressions defined by:*

$$\begin{array}{lcl}
\mathtt{I}[_{\mathtt{L}}] & = & \mathtt{L}(\mathtt{I}) \textit{ if } \mathtt{I} \in \textit{domain}(\mathtt{L}) \\
 & & \mathtt{I} \textit{ otherwise} \\
(\mathtt{B}_0 \vee \mathtt{B}_1)[_{\mathtt{L}}] & = & (\mathtt{B}_0[_{\mathtt{L}}] \vee \mathtt{B}_1[_{\mathtt{L}}]) \\
((\mathtt{I}:\mathtt{T}) \xrightarrow{\mathtt{B}} \mathtt{T}\ )[_{\mathtt{L}}] & = & (\mathtt{I}:\mathtt{T}[_{\mathtt{L}}]) \xrightarrow{\mathtt{B}[_{\mathtt{L}'}]} \mathtt{T}[_{\mathtt{L}'}]
\end{array}$$

where $\mathtt{L}' = \mathtt{L}[\mathtt{I} \mapsto \mathtt{B}']$ *for some* $\mathtt{B}'$ *and* $f[x \mapsto y]$ *is the function equal to* f *except on* x *where it yields* y.

Note that the definition of a given rebinding requires the specification of what bound variables are substituted to. Rebindings are used to instantiate a given type in a way that is compatible with the binding-time value of arguments of lambda expressions.

Definition 3 (Grounding) *A* grounding $[_{\mathtt{B},\mathtt{L}}]$ *is a scope-preserving substitution upon identifiers within typed expressions* $\mathtt{E}^{\mathtt{T}}$ *defined by:*

$$\begin{array}{lcl}
\mathtt{I}^{\mathtt{T}}[_{\mathtt{B},\mathtt{L}}] & = & \mathtt{v}^* \textit{ if } \mathtt{I} \sqsubseteq \mathtt{B} \\
 & & \mathtt{v} \textit{ otherwise} \\
(\mathtt{E}_0{}^{\mathtt{T}_0}\ \mathtt{E}_1{}^{\mathtt{T}_1})^{\mathtt{T}'}[_{\mathtt{B},\mathtt{L}}] & = & (\mathtt{E}_0{}^{\mathtt{T}_0}[_{\mathtt{B},\mathtt{L}}]\ \mathtt{E}_1{}^{\mathtt{T}_1}[_{\mathtt{B},\mathtt{L}}]) \\
(\mathtt{lambda}\ (\mathtt{I}:\mathtt{T}_1)\ \mathtt{E}_0{}^{\mathtt{T}_0})^{\mathtt{T}}[_{\mathtt{B},\mathtt{L}}] & = & (\mathtt{lambda}\ (\mathtt{I}:\mathtt{T}_1[_{\mathtt{L}}])\ \mathtt{E}_0{}^{\mathtt{T}_0}[_{\mathtt{B}_0,\mathtt{L}'}])
\end{array}$$

where $\mathtt{L}' {=} \mathtt{L}[\mathtt{I} \mapsto \mathtt{B}']$ *for some* $\mathtt{B}'$, $\mathtt{T} = ((\mathtt{I}:\mathtt{T}_1) \xrightarrow{\mathtt{B}_0} \mathtt{T}_0\ )$, $\phi \vdash \mathtt{v}^* : \mathtt{T}[_{\mathtt{L}}]\ \#\ \mathtt{stat}$ *and similarly for* v.

Groundings are similar to structural substitutions with respect to a binding-time value on expressions, as long as the expressions are not lambda expressions; in this latter case, the grounding is performed with respect to this lambda *latent* binding time since this is what defines the behavior of the lambda body. Since the $(\rightarrow\mathrm{I}_d)$ rule requires evaluation of a lambda expression body, a rebinding is maintained to emulate the possible binding times of the actual argument.

Theorem 8 (Adequacy) *If* $A \vdash \mathtt{E} : \mathtt{T}\ \#\ \mathtt{B}$, *then* $\mathtt{E}^{\mathtt{T}}[_{\mathtt{B},\mathtt{L}}] \rightarrow \mathtt{v}^*$ *and* $\phi \vdash \mathtt{v}^* : \mathtt{T}[_{\mathtt{L}}]\ \#\ \mathtt{B}'$ *such that* $\mathtt{B}' \sqsubseteq \mathtt{B}[_{\mathtt{L}}]$.

The proof of adequacy of the static semantics with respect to the dynamic semantics is performed by induction on the length of value derivation. Note that the language is strongly normalizing. For simplicity, we assume that the conversion rule (Conv_s) on types is not used. We note $\mathtt{E}\ \#\sqsubseteq \mathtt{B}$ whenever $\mathtt{E}\ \#\ \mathtt{B}'$ for some $\mathtt{B}' \sqsubseteq \mathtt{B}$.

- E = I
  Suppose that $A \vdash \mathtt{I} : \mathtt{T}\ \#\ \mathtt{I}$.
  By definition of groundings, $\mathtt{I}^{\mathtt{T}}[_{\mathtt{I},\mathtt{L}}] = \mathtt{v}^*$.
  Since values are closed, $\phi \vdash \mathtt{v}^* : \mathtt{T}[_{\mathtt{L}}]\ \#\ \mathtt{stat}$
  $\leadsto \mathtt{stat} \sqsubseteq \mathtt{I}[_{\mathtt{L}}]$

- E = (lambda (I:T₀) E₀)
  Suppose that $A \vdash (\mathtt{lambda}\ (\mathtt{I}:\mathtt{T}_1)\ \mathtt{E}_0) : \mathtt{T}\ \#\ \mathtt{B}$.
  with $\mathtt{T} = (\mathtt{I}:\mathtt{T}_1) \xrightarrow{\mathtt{B}_0} \mathtt{T}_0$.
  $\leadsto \mathtt{E}^{\mathtt{T}}[_{\mathtt{B},\mathtt{L}}] = (\mathtt{lambda}\ (\mathtt{I}:\mathtt{T}_1[_{\mathtt{L}}])\ \mathtt{E}_0{}^{\mathtt{T}_0}[_{\mathtt{B}_0,\mathtt{L}'}])$
  where $\mathtt{L}' = \mathtt{L}[\mathtt{I} \mapsto \mathtt{B}']$ for some $\mathtt{B}'$.
  By the $(\rightarrow\mathrm{I}_s)$ rule, $A[\mathtt{I}:\mathtt{T}_1] \vdash \mathtt{E}_0 : \mathtt{T}_0\ \#\ \mathtt{B}_0$
  By the induction hypothesis, choosing $\mathtt{L}'$ again as rebinding,
  $\leadsto \mathtt{E}_0{}^{\mathtt{T}_0}[_{\mathtt{B}_0,\mathtt{L}'}] \rightarrow \mathtt{v}_0^*$ and $\phi \vdash \mathtt{v}_0^* : \mathtt{T}_0[_{\mathtt{L}'}]\ \#\sqsubseteq \mathtt{B}_0[_{\mathtt{L}'}]$
  By definition of the $(\rightarrow\mathrm{I}_d)$ evaluation rule,
  $\leadsto \mathtt{E}^{\mathtt{T}}[_{\mathtt{B},\mathtt{L}}] \rightarrow \mathtt{v}^*$ and $\phi \vdash \mathtt{v}^* : \mathtt{T}[_{\mathtt{L}}]\ \#\ \mathtt{stat}[_{\mathtt{L}}]$
  with
  $$\left\{\begin{array}{l} \mathtt{v}^* = (\mathtt{lambda}\ (\mathtt{I}:\mathtt{T}_1[_{\mathtt{L}}])\ \mathtt{v}_0^*) \\ \mathtt{T}[_{\mathtt{L}}] = (\mathtt{I}:\mathtt{T}_1[_{\mathtt{L}}]) \xrightarrow{\mathtt{B}_0[_{\mathtt{L}'}]} \mathtt{T}_0[_{\mathtt{L}'}] \end{array}\right.$$

- E = (E₀ E₁)
  Suppose that $A \vdash (\mathtt{E}_0\ \mathtt{E}_1) : \mathtt{T}\ \#\ \mathtt{B}$ with
  $$\left\{\begin{array}{l} \mathtt{T} = \mathtt{T}'[\mathtt{B}_1/\mathtt{I}] \\ \mathtt{B} = \mathtt{B}_0 \vee \mathtt{B}'[\mathtt{B}_1/\mathtt{I}] \end{array}\right.$$
  $\mathtt{E}^{\mathtt{T}}[_{\mathtt{B},\mathtt{L}}] = (\mathtt{E}_0{}^{\mathtt{T}_0}[_{\mathtt{B},\mathtt{L}}]\ \mathtt{E}_1{}^{\mathtt{T}_1}[_{\mathtt{B},\mathtt{L}}])$
  with $\mathtt{T}_0 = (\mathtt{I}:\mathtt{T}_1) \xrightarrow{\mathtt{B}'} \mathtt{T}'$.
  By definition of the static semantics, $A \vdash \mathtt{E}_0 : \mathtt{T}_0\ \#\ \mathtt{B}_0$
  By the induction hypothesis, choosing L again as rebinding,
  $\leadsto \mathtt{E}_0{}^{\mathtt{T}_0}[_{\mathtt{B}_0,\mathtt{L}}] \rightarrow \mathtt{v}_0^*$ and $\phi \vdash \mathtt{v}_0^* : \mathtt{T}_0[_{\mathtt{L}}]\ \#\sqsubseteq \mathtt{B}_0[_{\mathtt{L}}]$
  Since one can show that $\mathtt{E}^{\mathtt{T}}[_{\mathtt{B},\mathtt{L}}] \rightarrow \mathtt{v}^* \Longrightarrow \mathtt{E}^{\mathtt{T}}[_{\mathtt{B}\vee\mathtt{B}_+,\mathtt{L}}] \rightarrow \mathtt{v}^*$,
  $\leadsto \mathtt{E}_0{}^{\mathtt{T}_0}[_{\mathtt{B},\mathtt{L}}] \rightarrow \mathtt{v}_0^* = (\mathtt{lambda}\ (\mathtt{I}:\mathtt{T}_1[_{\mathtt{L}}])\ \mathtt{E}')$
  By the $(\rightarrow\mathrm{I}_s)$ rule, $[\mathtt{I}:\mathtt{T}_1[_{\mathtt{L}}]] \vdash \mathtt{E}' : \mathtt{T}''\ \#\ \mathtt{B}''$ with
  $$\left\{\begin{array}{l} \mathtt{T}'' = \mathtt{T}'[_{\mathtt{L}'}] \\ \mathtt{B}'' = \mathtt{B}'[_{\mathtt{L}'}] \\ \mathtt{L}' = \mathtt{L}[\mathtt{I} \mapsto \mathtt{B}'''] \end{array}\right.$$
  for any $\mathtt{B}'''$. We now face two cases:

- $I \sqsubseteq B'$
  Similarly as above, since $B_1 \sqsubseteq B$,
  $\rightsquigarrow {E_1}^{T_1}[_{B,L}] \rightarrow v_1^* \text{ and } \phi \vdash v_1^* : T_1[_L] \mathrel{\#} \sqsubseteq B_1[_L]$
  Choosing $B''' = I$, we get $I \sqsubseteq B''$.
  Using the $(\rightarrow E_d)$ rule,
  $\rightsquigarrow E^T[_{B,L}] = E'^{T''}[I \mapsto v_1^*] = E'^{T''}[_{B'',L}]$
  since the free variables of E' includes at most I.
- $I \not\sqsubseteq B'$
  $\rightsquigarrow {E_1}^{T_1}[_{B,L}] \rightarrow v_1$
  Even if $\top$ appears in v_1, we have $\phi \vdash v_1 : T_1[_L]$.
  As in the previous case, using the $(\rightarrow E_d)$ rule,
  $\rightsquigarrow E^T[_{B,L}] = E'^{T''}[I \mapsto v_1] = E'^{T''}[_{B'',L}]$

In both cases, by induction,
$\rightsquigarrow E'^{T''}[_{B'',L''}] \rightarrow v'^* \text{ and } \phi \vdash v'^* : T''[_{L''}] \mathrel{\#} \sqsubseteq B''[_{L''}]$
Choosing L'' such that $L''(I') = I'$ and $B''' = B_1$,
$$\begin{cases} T''[_{L''}] = T'' = T'[_{L'}] = (T'[B_1/I])[_L] = T[_L] \\ B''[_{L''}] = B'' = B'[_{L'}] = (B'[B_1/I])[_L] \sqsubseteq B[_L] \end{cases}$$

□

B Soundness

Theorem 9 (Soundness) *Let* $(T, B, \mathcal{C}) = \mathcal{R}(A, E)$ *and* m *be a model:*

$$(m \models \mathcal{C}) \Longrightarrow A \vdash E : mT \mathrel{\#} mB$$

The proof of the soundness of $\mathcal{R}$ with respect to the static semantics is performed by induction on the structure of expressions.

- E = I
  Let $(T, I \vee \nu, \emptyset) = \mathcal{R}(A, I)$ with $m \models \emptyset$.
  $\rightsquigarrow I : T \in A$ (by definition of $\mathcal{R}$)
  $\rightsquigarrow A \vdash I : T \mathrel{\#} I$ (by typing rule (Var_s))
  $\rightsquigarrow A \vdash I : mT \mathrel{\#} I$
  $\rightsquigarrow A \vdash I : mT \mathrel{\#} I \vee m\nu$ (by typing rule (Conv_s))
  $\rightsquigarrow A \vdash I : mT \mathrel{\#} m(I \vee \nu)$ (because I is not an effect variable)
- E = (lambda (I : T) E')
  Let $((I : T) \xrightarrow{B'} T', \nu, \mathcal{C}) = \mathcal{R}(A, (\text{lambda } (I : T)\ E'))$ with $m \models \mathcal{C}$.

  Where $(T', B', \mathcal{C}) = \mathcal{R}(A[I : T], E')$ with $m \models \mathcal{C}$.
  $\rightsquigarrow A[I : T] \vdash E' : mT' \mathrel{\#} mB'$ (by induction hypothesis)

  $\rightsquigarrow A \vdash (\text{lambda } (I : T)\ E) : (I : mT) \xrightarrow{mB'} mT' \mathrel{\#} \text{stat}$ (by typing rule $(\rightarrow I_s)$)

  $\rightsquigarrow A \vdash (\text{lambda } (I : T)\ E) : m((I : T) \xrightarrow{B'} T') \mathrel{\#} m\nu$ (by typing rule (Conv_s))
- E = (E₀ E₁)
  Let $(T'[B_1/I], B' \vee \nu, \mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}) = \mathcal{R}(A, (E_0\ E_1))$ with $m \models (\mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C})$.

  Where $(T_0, B_0, \mathcal{C}_0) = \mathcal{R}(A, E_0)$
  and $(T_1, B_1, \mathcal{C}_1) = \mathcal{R}(A, E_1)$
  and $T_0 = (I : T) \xrightarrow{B} T'$
  and $B' = B_0 \vee B[B_1/I]$

  $\rightsquigarrow A \vdash E_0 : mT_0 \mathrel{\#} mB_0$ with $m \models \mathcal{C}_0$ and $A \vdash E_1 : mT_1 \mathrel{\#} mB_1$ with $m \models \mathcal{C}_1$ (by induction hypothesis)
  $\rightsquigarrow A \vdash E_0 : m((I : T) \xrightarrow{B} T') \mathrel{\#} mB_0$ with $m \models \mathcal{C}_0$
  $\rightsquigarrow A \vdash E_0 : (I : mT) \xrightarrow{mB} mT' \mathrel{\#} mB_0$ with $m \models \mathcal{C}_0$
  $\rightsquigarrow$ By lemma 2 we know that $mT_1 \sqsubseteq mT$, therefore we have $A \vdash E_1 : mT \mathrel{\#} mB_1$

  $\rightsquigarrow A \vdash (E_0\ E_1) : (mT')[mB_1/I] \mathrel{\#} mB_0 \vee (mB)[mB_1/I]$ (by typing rule $(\rightarrow E_s)$)
  $\rightsquigarrow A \vdash (E_0\ E_1) : (mT')[mB_1/I] \mathrel{\#} mB_0 \vee (mB)[mB_1/I] \vee m\nu$ (by typing rule (Conv_s))
  $\rightsquigarrow A \vdash (E_0\ E_1) : (mT')[mB_1/I] \mathrel{\#} m(B_0 \vee B[B_1/I] \vee \nu)$
  $\rightsquigarrow A \vdash (E_0\ E_1) : (mT')[mB_1/I] \mathrel{\#} m(B' \vee \nu)$ □

C Completeness

Theorem 10 (Completeness) *If* $A \vdash E : T \mathrel{\#} B$, *there exists* T', B', $\mathcal{C}'$ *and a model* m' *such that:*
$$\begin{cases} (T', B', \mathcal{C}') = \mathcal{R}(A, E) \\ m' \models \mathcal{C}' \\ B = m'B' \\ m'T' \sqsubseteq T \end{cases}$$

The proof of the completeness of $\mathcal{R}$ with respect to the static semantics is performed by induction on the structure of expressions.

- E = I
  Suppose that $A \vdash I : T \mathrel{\#} B$.
  $\rightsquigarrow I : T_0 \in A \wedge (T_0 \sqsubseteq T)$ (by typing rules (Var_s) and (Conv_s))
  $\rightsquigarrow I \sqsubseteq B$ (by typing rules (Var_s) and (Conv_s))
  $\rightsquigarrow \exists\ T'$ *such that* $I : T' \in A$ *such that*
  $$\begin{cases} (T', B', \mathcal{C}') = \mathcal{R}(A, I) \\ T' = T_0 \\ B' = I \vee \nu \\ \mathcal{C}' = \emptyset \\ m' = \emptyset[\nu \mapsto B] \end{cases}$$
  $\rightsquigarrow m'T' = mT' = T_0 \sqsubseteq T$ (from definitions)
  $$\begin{aligned} \rightsquigarrow m'B' &= m'(I \vee \nu) \\ &= I \vee m'\nu \\ &= I \vee B && \text{(by substitution)} \\ &= B && \text{(since } I \sqsubseteq B\text{)} \end{aligned}$$
- E = (lambda (I:T₀) E₀)
  Suppose that $A \vdash (\text{lambda } (I : T_0)\ E_0) : T \mathrel{\#} B$.
  $\rightsquigarrow \exists\ T_1, B_1$ *such that*
  $$\begin{cases} (I : T_0) \xrightarrow{B_1} T_1 \sqsubseteq T \\ A \vdash (\text{lambda } (I : T_0)\ E_0) : (I : T_0) \xrightarrow{B_1} T_1 \mathrel{\#} \text{stat} \end{cases}$$

  $\rightsquigarrow A[I : T_0] \vdash E_0 : T_1 \mathrel{\#} B_1$ (by typing rule $(\rightarrow I_s)$)
  $\rightsquigarrow \exists\ T'_0, B'_0, \mathcal{C}'_0, m'_0$ *such that*
  $$\begin{cases} (T'_0, B'_0, \mathcal{C}'_0) = \mathcal{R}(A[I : T'_0], E_0) \\ m'_0 \models \mathcal{C}'_0 \\ m'_0 T'_0 \sqsubseteq T_1 \\ m'_0 B'_0 = B_1 \end{cases} \quad \text{(by induction hypothesis)}$$

$$\begin{cases} (\mathtt{T}', \mathtt{B}', \mathcal{C}') = \mathcal{R}(A, (\mathtt{lambda}(\mathtt{I} : \mathtt{T}_0)\ \mathtt{E}_0)) \\ m' = m'_0[\nu \mapsto \mathtt{B}] \\ \mathtt{T}' = (\mathtt{I} : \mathtt{T}_0) \xrightarrow{\mathtt{B}'_0} \mathtt{T}'_0 \\ \mathtt{B}' = \nu \\ \mathcal{C}' = \mathcal{C}'_0 \end{cases}$$

$$\rightsquigarrow m'_0 \models \mathcal{C}'_0 \Longrightarrow m' \models \mathcal{C}'$$

$$\begin{aligned} \rightsquigarrow m'\mathtt{T}' &= m'((\mathtt{I} : \mathtt{T}_0) \xrightarrow{\mathtt{B}'_0} \mathtt{T}'_0\ ) \\ &= (\mathtt{I} : \mathtt{T}_0) \xrightarrow{m'_0\mathtt{B}'_0} m'_0\mathtt{T}'_0 \\ &\sqsubseteq (\mathtt{I} : \mathtt{T}_0) \xrightarrow{\mathtt{B}_1} \mathtt{T}_1 \\ &\sqsubseteq \mathtt{T} \end{aligned}$$

$$\begin{aligned} \rightsquigarrow m'\mathtt{B}' &= m'\nu \\ &= \mathtt{B} \end{aligned}$$

- $\mathtt{E} = (\mathtt{E}_0\ \mathtt{E}_1)$
  Suppose that $A \vdash (\mathtt{E}_0\ \mathtt{E}_1) : \mathtt{T}\ \#\ \mathtt{B}$
  $\rightsquigarrow \exists\ \mathtt{T}_1, \mathtt{B}'_1, \mathtt{B}_1, \mathtt{T}_0, \mathtt{B}'_0$ *such that*

$$\begin{cases} A \vdash \mathtt{E}_0 :\ (\mathtt{I} : \mathtt{T}_1) \xrightarrow{\mathtt{B}'} \mathtt{T}_0\ \ \#\ \mathtt{B}_0 \\ A \vdash \mathtt{E}_1 : \mathtt{T}_1\ \#\ \mathtt{B}_1 \\ \mathtt{T}_0[\mathtt{B}_1/\mathtt{I}] \sqsubseteq \mathtt{T} \\ \mathtt{B}_0 \vee \mathtt{B}'[\mathtt{B}_1/\mathtt{I}] \sqsubseteq \mathtt{B} \end{cases}$$

$\rightsquigarrow \forall\ i\ \in \{0,1\}, \exists\ \mathtt{T}'_i, \mathtt{B}'_i, \mathcal{C}'_i, m'_i$ *such that*

$$\begin{cases} (\mathtt{T}'_i, \mathtt{B}'_i, \mathcal{C}'_i) = \mathcal{R}(A, \mathtt{E}_i) \\ m'_i \models \mathcal{C}'_i \\ m'_0\mathtt{T}'_0 \sqsubseteq\ (\mathtt{I} : \mathtt{T}_1) \xrightarrow{\mathtt{B}'} \mathtt{T}_0 \\ m'_1\mathtt{T}'_1 \sqsubseteq \mathtt{T}_1 \\ m'_i\mathtt{B}'_i = \mathtt{B}_i \end{cases}$$

$\rightsquigarrow \exists\ \mathtt{T}''_1, \mathtt{B}'', \mathtt{T}''_0$ *such that*

$$\begin{cases} \mathtt{T}'_0 =\ (\mathtt{I} : \mathtt{T}''_1) \xrightarrow{\mathtt{B}''} \mathtt{T}''_0 \\ m'_0\mathtt{B}'' \sqsubseteq \mathtt{B}' \\ m'_0\mathtt{T}''_0 \sqsubseteq \mathtt{T}_0 \\ m'_0\mathtt{T}_1 \sqsubseteq \mathtt{T}''_1 \end{cases}$$

$\rightsquigarrow m'_1\mathtt{T}'_1 \sqsubseteq \mathtt{T}_1 = m'_1\mathtt{T}_1 \sqsubseteq m'_1\mathtt{T}''_1$ (by Theorem 2)
$\rightsquigarrow \exists\ C$ *such that* $\mathcal{U}(\mathtt{T}'_1, \mathtt{T}''_1) = \mathcal{C}$
$\rightsquigarrow \exists\ \mathtt{T}', \mathtt{B}', \mathcal{C}', m'$ *such that*

$$\begin{cases} (\mathtt{T}', \mathtt{B}', \mathcal{C}') = \mathcal{R}(A, (\mathtt{E}_0\ \mathtt{E}_1)) \\ \mathtt{T}' = \mathtt{T}''_0[\mathtt{B}'_i/\mathtt{I}] \\ \mathtt{B}' = \mathtt{B}'_0 \vee \mathtt{B}''[\mathtt{B}'_1/\mathtt{I}] \vee \nu \\ \mathcal{C}' = \mathcal{C}'_0 \cup \mathcal{C}'_1 \cup \mathcal{C} \\ m' = m'_1 m'_0[\nu \mapsto \mathtt{B}] \end{cases}$$

$\rightsquigarrow m' \models \mathcal{C}'$ since $m'_1 m'_0 \models \mathcal{C}'_0 \cup \mathcal{C}'_1 \cup \mathcal{C}$

$$\begin{aligned} \rightsquigarrow m'\mathtt{T}' &= (m'_1 m'_0)(\mathtt{T}''_0[\mathtt{B}'_1/\mathtt{I}]) \\ &= (m'_1 m'_0)\mathtt{T}''_0[(m'_1 m'_0)\mathtt{B}'_1/\mathtt{I}] \\ &= (m'_1 m'_0)\mathtt{T}''_0[\mathtt{B}_1/\mathtt{I}] \\ &\sqsubseteq m'_1\mathtt{T}_0[\mathtt{B}_1/\mathtt{I}] \\ &= \mathtt{T}_0[\mathtt{B}_1/\mathtt{I}] \end{aligned}$$

$$\begin{aligned} \rightsquigarrow m'\mathtt{B}' &= m'(\mathtt{B}'_0 \vee \mathtt{B}''[\mathtt{B}'_1/\mathtt{I}] \vee \nu) \\ &= m'_1 m'_0\mathtt{B}'_0 \vee (m'_1 m'_0\mathtt{B}'')[m'_1 m'_0\mathtt{B}'_1/\mathtt{I}] \vee \mathtt{B} \\ &= \mathtt{B}_0 \vee \mathtt{B}'[\mathtt{B}_1/\mathtt{I}] \vee \mathtt{B} \\ &= \mathtt{B} \end{aligned}$$

□