CENTRE DE RECHERCHE EN INFORMATIQUE

Separate Polyvariant Binding Time Reconstruction

Ch. Consel, P. Jouvelot, P. Orbaek

Juillet 1994

Document EMP–CRI A/261

Separate Polyvariant Binding Time Reconstruction

Charles Consel*
Charles.Consel@irisa.fr

Pierre Jouvelot†
jouvelot@cri.ensmp.fr

Peter Ørbæk‡
poe@brics.aau.dk

Ecole des Mines, CRI Report A/261
October 1994

Abstract

Binding time analysis aims at determining which identifiers 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 instantiation of its type. By using an effect system, binding time information can be propagated across compilation boundaries.

This paper gives a complete description of this new binding time analysis framework, show how both type and binding time information can be expressed and how they relate to the dynamic semantics, and describes a type and binding time reconstruction algorithm which is proved correct with respect to the static semantics.

Keywords: partial evaluation, type inference
Paper Word Count: 5132 (excluding bibliography)

*IRISA, Rennes, France. This research was partially supported by NSF grant CCR-9224375.
†Centre de Recherche en Informatique, Ecole des Mines de Paris, France.
‡Basic Research in Computer Science, Dept. of Comp. Sci., Aarhus University, Denmark, Centre of the Danish National Research Foundation.

1 Introduction

Analyzing the binding time properties of identifiers in programs aims at determining when the value of these identifiers 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 identifiers 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.

Existing binding time analyses (see Section 2) 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 [20, 21] 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.

The paper is organized as follows. Section 2 surveys the prior work related to binding time analysis. Section 3 presents the syntax and dynamic semantics of the language. Section 4 describes the binding time analysis type and effect system, together with its correctness with respect to the dynamic semantics. Section 5 presents the type and effect reconstruction algorithm, together with a proof of its correctness with respect to the static semantics. We discuss possible extensions to this work in Section 6, and give some concluding remarks in Section 7.

2 Related Work

Binding time analysis is primarily used for partial evaluation [17] 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, 4, 6, 8, 9, 10, 13, 27] and type systems [23, 24, 14, 15]. By using a more symbolic approach, our framework allows binding time analysis to be expressed without the added complexity of the previous systems, in particular two-level lambda-calculi and annotated terms. Also, contrarily to [14], function types are compatible with the usual functionality of expressions since binding time properties are kept in effects; this is particularly visible for dynamic lambdas. Finally, all previous analyses assume that the whole program text is available, although admittedly two-level lambda-calculi could be adapted to modular design.

Effect systems have proven quite an effective framework for the specification and analysis of higher-order functional and imperative program behaviors. Examples are side-effects [21], communication in

parallel programs [18, 25], time complexity [12], or control-flow analysis [30]. They are particularly well-designed for program analysis that operate across module boundaries, i.e. within the realm of separate compilation and optimization. Since effect systems and abstract interpretation can be used simultaneously using separate abstract interpretation [31], our approach could possibly also be merged with existing systems, based on either type systems or abstract interpretation.

Inspired by a previous unpublished version of our work [7], Henglein and Mossin [16] introduce a polyvariant binding time analysis which uses an extended type system with a two-level syntax. Here polyvariance is limited to let-bound identifiers, while our approach can simply see `let` as syntactic sugar (see Section 6). Their correcteness proof uses a denotationally-specified specializer, while our approach, more in tune with the nature of partial evaluation (i.e., compile-time code generation) is operationally-based. Also, they restrict their consistency relation to first-order values, while our proof does not need to introduce such a concept. No mention is made of how type-checking is performed. Finally, our approach allows across-module information to be more intuitively conveyed than theirs.

Link-time optimizations in compilers, e.g. for register allocation or address calculation, have been used in compilers for a long time [32, 28]. Our new approach suggests that binding time analysis could be used in a similar way, e.g. for performing specialized code generation at link time to optimize operating systems [11].

3 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 6 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 [3] 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.

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

Not only does the function type describe the abstraction as a function from `Int` to `Int`, but it also captures the 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 `x` of the formal. Noteworthy, this binding time behavior is determined independently of any given context; it is in this regard that our binding time analysis is *polyvariant*.

Next, consider the following function:

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

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 show in this paper that the binding time behavior of functions is always captured by a simple disjunctive boolean term.

We give below the abstract syntax of our kernel language (I denotes identifiers):

$$
\begin{array}{lcll}
B & ::= & I & \\
 & | & B_0 \vee B_1 & \text{Combination of Effects} \\
 & & & \\
T & ::= & I & \\
 & | & (I : T_1) \xrightarrow{B} T_2 & \text{Function Type} \\
 & & & \\
E & ::= & I & \\
 & | & (\texttt{lambda}\ (I)\ E) & \text{Lambda} \\
 & | & (\texttt{lambda}\ (I : T)\ E_0) & \text{Typed Lambda} \\
 & | & (E_0\ E_1) & \text{Application}
\end{array}
$$

Our language allows both explicitly and implicitly typed function definitions. For the sake of simplicity, it does not include constants (see Section 6). Also, note that, although 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. This combination of types and binding time information within an effect system enables our analysis framework to be well-suited to *separate* analysis, types appearing within module signatures.

The dynamic semantics of our language is expressed by a structural operational semantics [26] on type-erased[1] expressions. The dynamic semantics presented in Figure 1 is composed of a set of rules that inductively defines the evaluation relation $E \rightarrow E'$ on the structure of expressions. It associates to each expression E its reduced expression. In our simple language, computed values are reduced lambda terms:

$$V \in Values$$

Since the partial evaluation process deals with syntactic objects, the dynamic semantics uses a syntactic substitution-based approach. To limit the number of trivial rules, the (Exp_d) rule ensures that $E \rightarrow E$; reductions $E \rightarrow E'$ of interest are those in which $E \not\equiv E'$, where $\equiv$ denotes syntactical equality. Concerning the remaining rules, if the ($\rightarrow \text{E}_d$) rewriting rule that implements the beta-reduction of the lambda-calculus is standard, the ($\rightarrow \text{I}_d$) 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. In other words, partial evaluation reduces expressions to their head normal form, while preserving termination properties.

We note $E[E'/I]$ the syntactic substitution of I by E' in E.

4 Static Semantics

The static semantics specifies the type and binding time information of expressions. We assume that the types of function arguments can either be provided by the programmer (for instance, to appear

[1] The straightforward $TypeErase$ function removes explicit types from expressions.

$$(\mathrm{Exp}_d) \qquad E \to E$$

$$(\to \mathrm{I}_d) \qquad \frac{E \to E'}{(\texttt{lambda}\ (I)\ E) \to (\texttt{lambda}\ (I)\ E')}$$

$$(\to \mathrm{E}_d) \qquad \frac{\begin{array}{rcl} E_0 & \to & (\texttt{lambda}\ (I)\ E') \\ E_1 & \to & E'_1 \\ E'[E'_1/I] & \to & E'' \end{array}}{(E_0\ E_1) \to E''}$$

Figure 1: Dynamic Semantics

within module signatures) or be left unspecified.

4.1 Binding Time and Type Algebras

The binding time algebra is a boolean algebra ordered with $\sqsubseteq$. We note $\vee$ the associative, commutative and idempotent combination operator and use the predefined effect identifiers `stat` for minimum and `dyn` for maximum elements. As expected, `stat` is unitary and `dyn` is absorbent.

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. For function types, latent binding times and return types behave monotonically while argument types use antimonotonicity:

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

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 wherever necessary to avoid capture problems or to test for type ordering. The equality relation $=$ on types is defined, as usual, by double inclusion with $\sqsubseteq$.

4.2 Typing Rules

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

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

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

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

$$(\rightarrow \text{I}_s)\quad \frac{A[I : T_1] \vdash E : T \# B}{A \vdash (\texttt{lambda}\ (I : T_1)\ E) : (I : T_1) \xrightarrow{B} T\ \#\ \texttt{stat}}$$

$$(\text{i}\rightarrow \text{I}_s)\quad \frac{A[I : T_1] \vdash E : T \# B}{A \vdash (\texttt{lambda}\ (I)\ E) : (I : T_1) \xrightarrow{B} T\ \#\ \texttt{stat}}$$

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

Figure 2: Static Semantics

In the $(\rightarrow \text{I}_s)$ and $(\text{i}\rightarrow \text{I}_s)$ rules, the type and binding time of the function body are integrated into the function type as its result type and latent binding time; the argument type may or may not be explicitly provided, in which case it is automatically inferred. This information is used when a function is applied in the $(\rightarrow \text{E}_s)$ 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 $(\rightarrow \text{E}_s)$ rule ensure that no bound expression identifier 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 *text* is available at compile (or program-transformation) time. This captures the usual behavior of partial evaluation. If we wanted to define the binding time of an expression as `stat` when it can actually be *evaluated* (and *not* partially evaluated) at compile time, a more appropriate rule for $(\rightarrow \text{I}_s)$, and similarly for $(\text{i}\rightarrow \text{I}_s)$, would have been:

$$\frac{A[I : T_1] \vdash E : T \# B}{A \vdash (\texttt{lambda}\ (I : T_1)\ E) : (I : T_1) \xrightarrow{B} T\ \#\ B - I}$$

where the $-$ operation is defined as $(x \vee y) - x = y$. [2]

[2] Using the unitary rule, one can see that $x - x = \texttt{stat}$.

4.3 Correctness

A major issue is to relate the static information to the dynamic semantics. The following adequacy theorem, akin to Subject Reduction, states that evaluation on type-erased expressions preserves type and binding time information.

Theorem 1 (Adequacy) *If* $A \vdash E : T \# B$ *and* $TypeErase(E) \rightarrow V$, *then* $A \vdash V : T \# B$.

Proof: See Appendix A. □

If the adequacy theorem ensures the consistency of the static information with respect to evaluation, it does not relate it with our intuition of partial evaluation. The following binding time property expresses that the static semantics indeed defines a binding time analysis for the partial evaluation defined by the dynamic semantics. The idea is that, if the binding time analysis says that the result of a computation can be computed at compile time, i.e. is static, then the partial evaluation is indeed able to evaluate that expression. Here, "E' cannot be further reduced" means that there does not exist $E'' \not\equiv E'$ such that $E' \rightarrow E''$.

Theorem 2 (Binding Time Property) *Suppose* $A \vdash E : T \# B$ *and* $TypeErase(E) \rightarrow E'$ *such that* E' *cannot be further reduced. If* $B = \texttt{stat}$, *then* $E' \in Values$.

Proof: See Appendix B. □

Whereas, in our simple language, the only values are lambda expressions, the extension of this property to a full-fledged language would naturally incorporate constants as well.

5 Type and Binding Time Effect Reconstruction

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

5.1 Simple Types

Following an approach suggested in [30] for control-flow analysis, the type and binding time effect reconstruction process uses a two-staged approach. First, programs are assumed to be explicitly typed via the simply-typed lambda calculus, using a standard type reconstruction algorithm (see for instance [22]). Simple types t are defined below:

$$\begin{array}{rcll} t & ::= & I & \\ & | & t_1 \longrightarrow t_2 & \text{Function Types} \end{array}$$

Then, type and binding time information is reconstructed from expressions decorated with simple types.

The reconstruction algorithm ensures that all latent binding times in function types are unification variables ν. The functions New and $Erase$ map between simple types and types used in $\mathcal{R}$.

$$Erase(I) = I$$
$$Erase((I : T_1) \xrightarrow{\nu} T\ ) = Erase(T_1) \longrightarrow Erase(T)$$

$$New(I) = I$$
$$New(t_1 \longrightarrow t) = (I : \mathrm{New}(t_1)) \xrightarrow{\nu} \mathrm{New}(t)$$

where I and ν are a new identifier and a new unification variable respectively.

5.2 Effect Constraints

The reconstruction algorithm builds up constraints that are inequalities or equalities between effects that may include unification variables. Type inequalities or equalities are destructured into effect inequalities via the following *Constraint* function. The relation r below can either be $\sqsubseteq$ or $=$.

$$Constraint(I, r, J) = \begin{array}{l} \{\} \text{ if } I \equiv J \\ \{(N_1, =, N_2)\}, \text{ otherwise} \end{array}$$

$$Constraint((I : T_1) \xrightarrow{B} T\ , r, (I : T'_1) \xrightarrow{B'} T'\ ) =$$
$$Constraint(T, r, T') \cup Constraint(T'_1, r, T_1) \cup \{(B, r, B')\}$$

where $\cup$ denotes set union, $\{\}$ the empty set and N_1 and N_2 are two distinct fresh constant binding time identifiers. ($\{(N1, =, N_2)\}$ is equivalent to fail.)

The satisfiability of constraints is formalized via the notion of a model, which is a substitution mapping unification variables to effect values (without unification variables). More generally, a substitution m on types and effects is defined as follows:

$$mI = I, \text{ if } I \text{ is not in the domain of } m$$
$$m(I), \text{ otherwise}$$

$$m((I : T_1) \xrightarrow{B} T\ ) = (I : mT_1) \xrightarrow{mB} mT$$
$$m(B_1 \vee B_2) = mB_1 \vee mB_2$$

where alpha-renaming within function types (for identifiers I) is assumed to be performed, if need be, to make the definition well-formed. As a matter of convenience, $m(x)$ is often written mx.

The definition of substitutions is generalized to environments in the usual way. The extension of substitutions m and n of distinct domains is noted $m + n$, and general substitution composition mn. A substitution can be defined on one point $[x \rightarrow y]$ or by extension $[x \rightarrow fx \mid x \in S]$. As a matter of convenience, a substitution application $[x \rightarrow y]z$ is sometimes noted $z[y/x]$.

A substitution m on unification variables models a constraint $\mathcal{C}$, noted $m \models \mathcal{C}$, if and only if, for all triple (B, r, B') in $\mathcal{C}$, $mB\ r\ mB'$. The following lemmas relate models and constraints.

Lemma 1 $m \models (\mathcal{C} \cup \mathcal{C}') \Longrightarrow (m \models \mathcal{C} \wedge m \models \mathcal{C}')$

Lemma 2 $m \models Constraint(T, r, T') \Longrightarrow (mT\ r\ mT')$

Note that models are substitutions defined on unification variables and *not* the (lambda-bound) identifiers that appear in user programs.

5.3 Type Reconstruction Algorithm

The type and binding time reconstruction algorithm $\mathcal{R}$ is presented in Figure 3. The first stage of the algorithm, which implements type reconstruction for the simply-typed lambda-calculus, is assumed to have been performed. Expressions analyzed by $\mathcal{R}$ are thus type-correct and decorated with simple type information, as shown by the simple type annotation t appearing in the implicit lambda case.

In a type environment A that binds identifiers to their type, for an expression E, $\mathcal{R}$ returns its type T, its binding time B and an effect constraint $\mathcal{C}$ that defines the possible values of the binding time unification variables present in T and B. The actual types and binding times of E (there are many of them via the (Conv_s) rule) can be obtained by instantiating T and B with any model of $\mathcal{C}$.

$\mathcal{R}(A,\ E)$ = case E in
I =>
    if $I \in domain(A)$ then
        let new variable ν
            $T = New(Erase(A(I)))$
        in $(T, \nu, Constraint(A(I), \sqsubseteq, T) \cup \{(I, \sqsubseteq, \nu)\})$
    else fail
(lambda $(I : t)$ E) =>
    let $T_1 = New(t)$
        $(T, B, \mathcal{C}) = \mathcal{R}(A[I : T_1],\ E)$
        new variables ν, ν'
        $\mathcal{C}' = \mathcal{C} \cup \{(B, =, \nu)\}$
        $\{\nu_{I_i}\} = FreeVariables(\mathcal{C}')$
        new constant N_I
        new variables $\{\nu'_{I_i}\}$
    in $((I : T_1) \xrightarrow{\nu} T\ , \nu', \mathcal{C}' \cup \mathcal{C}'[N_I/I][\nu'_{I_i}/\nu_{I_i}])$
(lambda $(I : T_1)$ E) =>
    let $t_1 = Erase(T_1)$
        $(T', B', \mathcal{C}') = \mathcal{R}(A,$ (lambda $(I : t_1)$ $E))$
        $(I : T'_1) \xrightarrow{\nu} T\ = T'$
    in $(T', B', \mathcal{C}' \cup Constraint(T_1, =, T'_1))$
$(E_0\ E_1)$ =>
    let $((I : T_1\text{'}) \xrightarrow{\nu} T\ , B_0, \mathcal{C}_0) = \mathcal{R}(A,\ E_0)$
        $(T_1, B_1, \mathcal{C}_1) = \mathcal{R}(A,\ E_1)$
    in $(T, B_0 \vee \nu, \mathcal{C}_1 \cup \mathcal{C}_0[B_1/I] \cup Constraint(T_1, =, T_1\text{'}))$

Figure 3: Type Reconstruction Algorithm

The various cases are rather straightforward and only need a couple of explanations. First, an invariant of the algorithm is that the triple returned by $\mathcal{R}$ is already prepared to deal with a possible use of the (Conv_s) rule by the caller.

Then, in the implicit lambda case, the reason for the added constraint $\mathcal{C}'[N_I/I][\nu'_{I_i}/\nu_{I_i}]$ is formally

visible in the soundness proof. But, intuitively, its purpose is to ensure that the constraint $\mathcal{C}'$ is satisfiable for any possible value of I. Indeed, notice that if the lambda is applied, I is replaced in $\mathcal{C}'$ in the application case by the binding time B_1 of the argument. The satisfiability of the resulting constraint is thus not enough to ensure that the function itself would be typable in any context. The free unification variables $\{\nu_{I_i}\}$ of $\mathcal{C}'$ are gathered with the *FreeVariables* function and replaced by new variables $\{\nu'_{I_i}\}$ which, together with the replacement of I by a fresh constant identifier N_I, ensure that $\mathcal{C}'$ is satisfiable in any application (thus the new constant N_I representing an arbitrary argument binding time).

Finally, in the application case, since expressions are assumed to be already simply-typed correct, the result function type from $\mathcal{R}(A,\ E_0)$ is well-defined; there is no need for unification in the type domain. A similar argument holds for the explicit lambda case.

5.4 Correctness Theorems

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

Theorem 3 (Termination) *$\mathcal{R}(A, E)$ terminates or fails.*

Proof: $\mathcal{R}$ is defined by induction over the domain of finite

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

Theorem 4 (Soundness) *Let $(T, B, \mathcal{C}) = \mathcal{R}(A, E)$ and m be a substitution:*

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

Proof: See Appendix C. □

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

Theorem 5 (Completeness) *If $mA \vdash E : T \# B$, there exist T', B', $\mathcal{C}'$ and a model m' such that:*

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

Proof: See Appendix D. □

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

Theorem 6 (Decidability) *Every constraint constructed by $\mathcal{R}$ on a type-correct expression admits a model.*

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

Practically, the following algorithm can be used to determine a constraint model. First, add new slack unification variables to inequality constraints to replace them by effect equalities: $B \sqsubseteq B'$ is equivalent to $B = B' \vee \nu$ for a new variable ν. Then, non-deterministically choose which unification variables are to be assigned the binding time `dyn`. This eliminates some of the now trivial equations since `dyn` is absorbent. The resulting constraint can then be solved in polynomial time by ACUI unification using the algorithm defined in [19] on an algebra without a maximum element.

Note that, if one is willing to not use explicitly typed lambda expressions for which type equality is mandated, or the `dyn` absorbent element, constraints can be built in such a way that only effect inequalities occur. A polynomial-time reconstruction algorithm in the spirit of [29] can then be designed for such a system (see [30] for a similar example related to control-flow analysis).

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 (exactly) has a given binding time, it would be easy to add a new special form to the language, such as `(ensures` B E`)`, which would check that, among the possible binding times of E, one is equal to B in the binding time algebra.

6 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 [22] could easily be added to it using the following rewriting:

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

This rewriting is purely syntactical 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 I_0 is bound is replicated within the `let` body, thus relaxing the constraints between the different uses of I_0. The type reconstruction algorithm would not necessarily have to be applied on the larger, substituted expression; using caching mechanisms, a more efficient implementation can be designed [19].

Finally note that polymorphism is only useful for type information in our framework. Indeed, already having a polyvariant analysis entails that no new binding information is gathered with this rule.

Data structures

The language defined in Section 3 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. Significantly, no new static rules need to be added, just a proper extension of initial environments; this is a tribute to the generality of our approach.

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 [29]. 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[3].

Recursion

As defined in Section 3, our language is strongly normalizing. One way to add general recursion would be to introduce a set of fixpoint combinator constants $Y_{\mathtt{t}}$ for each type `t` and provide a suitable type for them, such as:

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

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. Besides, extending the syntax of binding time expressions would complicate the type reconstruction process. A better way to deal with this problem would be to introduce a `rec` construct in the language with the following dynamic semantics:

$$(\mathrm{Rec}_d) \qquad \frac{\begin{array}{rcl} E & = & (\mathtt{rec}\ (I\ I_1)\ E_0) \\ E_0[E/I] & \rightarrow & E' \end{array}}{E \ \rightarrow \ (\mathtt{lambda}\ (I_1)\ E')}$$

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

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

Note that adding recursion to the language implies that the adequacy theorem of Section 4 must be restricted to terminating expressions. Ultimately, partial evaluation may also fail to terminate in this case; the approach suggested in [12] can be used to detect expressions that are always terminating.

[3] Note that adding mutation operations would badly interact with the simple-minded polymorphism scheme presented above, see [29] for a better approach.

7 Conclusion

We presented a new, simple and elegant 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 instantiation of its polyvariant type. By providing binding time information in function types, module signatures allow binding time analysis to be separately performed.

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 reconstruction 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, via adapted explicit binding time declarations. 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 [5] 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

The authors thank Olivier Danvy (and his students), Vincent Dornic, and Mitch Wand for their comments on a simpler version 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] R. M. Burstall and B. W. Lampson. A kernel language for abstract data types and modules. Research Report 80-430, Cornell University, 1984.

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

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

[6] C. Consel. Polyvariant binding-time analysis for higher-order, applicative languages. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, 1993.

[7] C. Consel and P. Jouvelot. Polyvariant binding-time analysis for separate compilation using type and effect checking. Research report, Pacific Software Research Center, Oregon Graduate Institute of Science and Technology, Beaverton, Oregon, USA, 1993.

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

[9] C. Consel and S. C. Khoo. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, 1993. Extended version of [8].

[10] C. Consel and S.C. Khoo. On-line & off-line partial evaluation: Semantic specifications and correctness proofs. Research report, Yale University, New Haven, Connecticut, USA, 1993. Extended version. To appear in *Journal of Functional Programming.*

[11] C. Consel, C. Pu, and J. Walpole. Incremental specialization: The key to high performance, modularity and portability in operating systems. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 44–46, 1993. Invited paper.

[12] 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.

[13] 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.

[14] C. K. Gomard. Partial type inference for untyped functional programs. In *ACM Conference on Lisp and Functional Programming*, pages 282–287, 1990.

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

[16] F. Henglein and C. Mossin. Polymorphic binding-time analysis. In *ESOP'94*, pages 448–472. Springer-Verlag, 1994.

[17] 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.

[18] P. Jouvelot and D. K. Gifford. Communication effects for message-based concurrency. Technical Memorandum 148, MIT, Cambridge, USA, 1989.

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

[20] 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.

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

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

[23] 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.

[24] 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.

[25] H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *ACM Symposium on Principles of Programming Languages.* ACM Press, 1994.

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

[27] 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.

[28] A. Srivastava and D. W. Wall. Link-time optimization of address calculation on a 64-bit architecture. In *ACM Conference on Programming Language Design and Implementation*, pages 49–60, 1994.

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

[30] Y-M. Tang. *Systèmes d'Effet et Interprétation Abstraite pour l'Analyse de Flot de Contrôle.* PhD thesis, CRI, Ecole des Mines de Paris, Paris, France, 1994.

[31] Y. M. Tang and P. Jouvelot. Separate abstract interpretation for control-flow analysis. In *International Conference on the Theoretical Aspects of Computer Software*, number 789 in Lecture Notes in Computer Science. Springer Verlag, 1994.

[32] D. W. Wall. Global register allocation at link time. In *ACM SIGPLAN Conference on Compiler Construction*, pages 264–275, 1986.

In the following proofs, we assume that all identifiers have been renamed by alpha-conversion so that each identifier is bound at most once.

A Adequacy

Theorem (Adequacy) *If* $A \vdash E : T \# B$ *and* $TypeErase(E) \rightarrow V$, *then* $A \vdash V : T \# B$.

Proof: The proof is by induction on the length of value derivation.

- $E = I$

Assuming $A \vdash I : T \# B$ and $I \rightarrow I$, we must show that $A \vdash I : T \# B$, which is a tautology.

- $E = (\texttt{lambda}\ (I : T_1)\ E_0)$

Assume that $A \vdash (\texttt{lambda}\ (I : T_1)\ E_0) : T \# B$ and $TypeErase((\texttt{lambda}\ (I)\ E_0)) \rightarrow V$. By the $(\rightarrow \mathrm{I}_s)$ rule:

$$\begin{cases} (I : T_1) \xrightarrow{B_0} T_0 \ \sqsubseteq T \\ \texttt{stat} \sqsubseteq B \\ A[I : T_1] \vdash E_0 : T_0 \ \# \ B_0 \end{cases}$$

By the dynamic semantics, we must have:

$$\begin{cases} TypeErase(E_0) \rightarrow E'_0 \\ V = (\texttt{lambda}\ (I)\ E'_0) \end{cases}$$

Induction then yields: $A[I : T_1] \vdash E'_0 : T_0 \ \# \ B_0$. By the $(\mathrm{i}{\rightarrow}\mathrm{I}_s)$ rule of the static semantics, we get:

$$A \vdash (\texttt{lambda}\ (I)\ E'_0) : (I : T_1) \xrightarrow{B_0} T_0 \ \ \# \ \texttt{stat}$$

and, by the (Conv_s) rule, the required result follows. (The case for implicitly typed lambda expressions is similar.)

- $E = (E_0\ E_1)$

Assume that $A \vdash (E_0\ E_1) : T \# B$ and $TypeErase((E_0\ E_1)) \rightarrow V$, we must show that $A \vdash V : T \# B$. By the first part of the antecedent and the static semantics, we must have:

$$\begin{cases} T'[B_1/I] \sqsubseteq T \\ B_0 \vee B'[B_1/I] \sqsubseteq B \\ A \vdash E_0 : (I : T_1) \xrightarrow{B'} T' \ \ \# \ B_0 \\ A \vdash E_1 : T_1 \ \# \ B_1 \end{cases}$$

By the dynamic semantics, we must have:

$$\begin{cases} TypeErase(E_0) \rightarrow (\texttt{lambda}\ (I)\ E') \\ TypeErase(E_1) \rightarrow E'_1 \\ E'[E'_1/I] \rightarrow V \end{cases}$$

By induction, we get:

$$\begin{cases} A \vdash (\texttt{lambda}\ (I)\ E') : (I : T_1) \xrightarrow{B'} T' \ \# \ B_0 \\ A \vdash E'_1 : T_1 \ \# \ B_1 \end{cases}$$

By the (i→I$_s$) rule of the static semantics, we must have: $A[I : T_1] \vdash E' : T' \ \# \ B'$. Using the substitution lemma (see Lemma 3 below), we get:

$$A_I^{B_1} \vdash E'[E'_1/I] : T'[B_1/I] \ \# \ B'[B_1/I]$$

Since all identifiers are bound at most once, $A_I^{B_1} = A$. Using induction once more, we get:

$$A \vdash V : T'[B_1/I] \ \# \ B'[B_1/I]$$

By the (Conv$_s$) rule,we get the required result: $A \vdash V : T \ \# \ B$. □

Lemma 3 (Substitution) *Assume:*

$$I \notin FreeIdentifiers(T', E', B') \quad (1)$$

$$A \vdash E' : T' \ \# \ B' \quad (2)$$

$$A[I : T'] \vdash E : T \ \# \ B \quad (3)$$

and B' does not include any lambda-bound identifier from E, I is not bound in E and E has no explicit types. Then

$$A_I^{B'} \vdash E[E'/I] : T[B'/I] \ \# \ B[B''/I]$$

where $A_I^{B'}$ is as A, but all occurrences of I in the types of bound values are replaced by B'.

Proof: The proof is by structural induction on E.

- $E = I$

By (3) $A[I : T'] \vdash I : T \ \# \ B$, so by (Conv$_s$) and (Var$_s$) we must have $T' \sqsubseteq T$ and $I \sqsubseteq B$. We need to prove that $T' \sqsubseteq T[B'/I]$. Since $T' \sqsubseteq T$ and $I \notin FreeIdentifiers(T')$, the inequality holds for all substitutions, especially for B'. Also $I[B'/I] \equiv B' \sqsubseteq B[B'/I]$. Now by using (2) and (Conv$_s$) we get:

$$A \vdash I[E'/I] : T[B'/I] \ \# \ B[B'/I].$$

As I no longer occurs in the expressions, the same holds with $A_I^{B'}$ for A.

- $E = J$

To complement the preceding case, assume $J \not\equiv I$. By (3) we have:

$$A[I : T'] \vdash J : T \# B$$

Thus, by (Conv_s) and (Var_s), we must have $A(J) \sqsubseteq T$ and $J \sqsubseteq B$. We need to prove:

$$A_I^{B'} \vdash J : T[B'/I] \# B[B'/I]$$

which is obvious by the definition of $A_I^{B'}$.

- $E = (\texttt{lambda}\ (J)\ E_0)$

From (Conv_s), ($\rightarrow\text{I}_s$) and (3), we must have $(J : T_1) \xrightarrow{B_0} T_0 \sqsubseteq T$ where:

$$A[I : T'][J : T_1] \vdash E_0 : T_0 \# B_0$$

By induction, since $A[I : T'][J : T_1] = A[J : T_1][I : T']$:

$$A_I^{B'}[J : T_1[B'/I]] \vdash E_0[E'/I] : T_0[B'/I] \# B_0[B'/I]$$

By ($\text{i}\rightarrow\text{I}_s$):

$$A_I^{B'} \vdash (\texttt{lambda}\ (J)\ E_0[E'/I]) : (J : T_1[B'/I])^{B_0[B'/I]} T_0[B'/I]\ \ \#\ \texttt{stat}$$

Now, $J \notin \mathit{FreeIdentifiers}(B')$, use the definition of substitution and finally use (Conv_s) to get the required result.

- $E = (E_0\ E_1)$

By (convS), ($\rightarrow\text{I}_s$) and (3) we must have $T_2[B_1/J] \sqsubseteq T$ and $B_0 \vee B_2[B_1/J] \sqsubseteq B$ where:

$$\begin{cases} A[I : T'] \vdash E_0 : (J : T_1) \xrightarrow{B_2} T_2\ \ \# B_0 \\ A[I : T'] \vdash E_1 : T_1 \# B_1 \end{cases}$$

By induction:

$$\begin{cases} A_I^{B'} \vdash E_0[E'/I] : ((J : T_1) \xrightarrow{B_2} T_2\ )[B'/I] \# B_0[B'/I] \\ A_I^{B'} \vdash E_1[E'/I] : T_1[B'/I] \# B_1[B'/I] \end{cases}$$

By definition:

$$\begin{cases} (E_0[E'/I]\ E_1[E'/I]) \equiv (E_0\ E_1)[E'/I] \\ ((J : T_1) \xrightarrow{B_2} T_2\ )[B'/I] \equiv (J : T_1[B'/I])^{B_2[B'/I]} T_2[B'/I] \end{cases}$$

By ($\rightarrow$I$_s$)

$$A_I^{B'} \vdash (E_0\ E_1)[E'/I] : (T_2[B'/I])[(B_1[B'/I])/J] \# B_0[B'/I] \vee (B_2[B'/I])[B_1[B'/I]/J]$$

Since, by our assumptions, B' does not contain any lambda-bound identifier from E (here J), we can interchange the order of substitutions and get:

$$A_I^{B'} \vdash (E_0\ E_1)[E'/I] : (T_2[B_1/J])[B'/I] \# (B_0 \vee B_2[B_1/J])[B'/I]$$

Using (Conv$_s$) and the monotonicity of the substitution $[I \rightarrow B']$ we get the required result. □

B Binding Time Property

Theorem (Binding Time Property) *Suppose* $A \vdash E : T \# B$ *and* $TypeErase(E) \rightarrow E'$ *such that* E' *cannot be further reduced. If* $B = \texttt{stat}$, *then* $E' \in Values$.

Proof: The proof is by structural induction on E. In all cases, we have $A \vdash E' : T \# B$ by the adequacy theorem.

- $E' = I$

Using the adequacy theorem, with the (Var$_s$) and (Conv$_s$) rules, $I \sqsubseteq B$. Therefore, $B \neq \texttt{stat}$, and the result holds vacuously.

- $E' = (\texttt{lambda}\ (I)\ E_0)$

E' clearly is of the appropriate form.

- $E' = (E_0\ E_1)$

Since E' is assumed irreducible, the following transistion of the dynamic semantics must be impossible for all I and E'_0:

$$E_0 \rightarrow (\texttt{lambda}\ (I)\ E'_0)$$

as, otherwise, the ($\rightarrow$E$_d$) rule would be applicable. By the static semantics, the following must hold:

$$\begin{cases} A \vdash E_0 : (I : T_1) \xrightarrow{B_2} T_2 \ \# B_0 \\ A \vdash E_1 : T_1 \# B_1 \\ B_0 \vee B_2[B_1/I] \sqsubseteq B \end{cases}$$

As $B = \texttt{stat}$ by assumption, $B_0 = \texttt{stat}$. By induction, since $(E_0\ E_1)$ is fully reduced, E_0 also is, and $E_0 = (\texttt{lambda}\ (I)\ E'_0)$, which is a contradiction. □

C Soundness

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

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

Proof: The proof is by induction on the structure of expressions.

- $E = I$

Since $m \models Constraint(A(I), \sqsubseteq, T) \cup \{(I, \sqsubseteq, \nu)\}$, using Lemma 1 and the definition of models, we get $m(A(I)) \sqsubseteq mT$ and $mI \sqsubseteq m\nu$.

$(T, \nu, \mathcal{C}) = \mathcal{R}(A, I)$
$\Longrightarrow I \in domain(A)$ (by definition of $\mathcal{R}$)
$\Longrightarrow A \vdash I : A(I) \# I$ (by typing rule (Var_s))
$\Longrightarrow mA \vdash I : m(A(I)) \# mI$
$\Longrightarrow mA \vdash I : mT \# m\nu$ (by typing rule (Conv_s))

- $E = (\texttt{lambda}\ (I : t)\ E_0)$

Since $m \models \mathcal{C}' \cup \mathcal{C}'[N_I/I][\nu'_{I_i}/\nu_{I_i}]$, using Lemma 1, we get $m \models \mathcal{C}'$.

$(T_0, B_0, \mathcal{C}_0) = \mathcal{R}(A[I : T_1],\ E_0)$
$\Longrightarrow m(A[I : T_1]) \vdash E_0 : mT_0 \# mB_0$ (by induction hypothesis)
$\Longrightarrow (mA)[I : mT_1] \vdash E_0 : mT_0 \# mB_0$
$\Longrightarrow mA \vdash (\texttt{lambda}\ (I)\ E_0)\ :\ (I : mT_1) \xrightarrow{mB_0} mT_0\ \#\ \texttt{stat}$ (by typing rule ($\text{i}{\rightarrow}\text{I}_s$))
$\Longrightarrow mA \vdash (\texttt{lambda}\ (I)\ E_0)\ : m((I : T_1) \xrightarrow{\nu} T_0\ ) \#\ \texttt{stat}$ (since $mB_0 = m\nu$)
$\Longrightarrow mA \vdash (\texttt{lambda}\ (I)\ E_0)\ : m((I : T_1) \xrightarrow{\nu} T_0\ ) \# m\nu'$ (by typing rule (Conv_s))

- $E = (\texttt{lambda}\ (I : T_1)\ E_0)$

Since $m \models \mathcal{C}' \cup Constraint(T_1, =, T_1')$, using Lemma 1, we get $m \models \mathcal{C}'$.

Since $(T', B', \mathcal{C}') = \mathcal{R}(A, (\texttt{lambda}\ (I : t_1)\ E_0))$, and $(I : T_1') \xrightarrow{\nu} T_0' = T'$, by induction, we get:

$mA \vdash (\texttt{lambda}\ (I : t_1)\ E_0)\ :\ (I : mT'_1) \xrightarrow{m\nu} mT'_0\ \#\ mB'$ (by typing rule ($\rightarrow\text{I}_s$))
$\Longrightarrow mA \vdash (\texttt{lambda}\ (I : t_1)\ E_0)\ :\ (I : mT_1) \xrightarrow{m\nu} mT'_0\ \#\ mB'$ (since $mT_1' = mT_1$)

Thus, by typing rule ($\text{i}{\rightarrow}\text{I}_s$), we get $mA[I : mT_1] \vdash E_0 : mT_0' \# m\nu$. Using rule ($\rightarrow\text{I}_s$), we get:

$mA \vdash (\texttt{lambda}\ (I : T_1)\ E_0)\ :\ (I : mT_1) \xrightarrow{m\nu} mT_0'\ \#\ \texttt{stat}$ (by typing rule ($\rightarrow\text{I}_s$))
$\Longrightarrow mA \vdash (\texttt{lambda}\ (I : T_1)\ E_0)\ : m((I : T_1') \xrightarrow{\nu} T_0'\ ) \#\ \texttt{stat}$ (since $mT_1' = mT_1$)
$\Longrightarrow mA \vdash (\texttt{lambda}\ (I : T_1)\ E_0)\ : mT' \# mB'$ (by typing rule (Conv_s))

- $E = (E_0\ E_1)$

We know that $m \models \mathcal{C}_1 \cup \mathcal{C}_0[B_1/I] \cup Constraint(T_1, =, T_1')$. Since $m \models \mathcal{C}_0[B_1/I]$, there exists a model m_0 such that $m_0 \models \mathcal{C}_0$. Indeed, knowing that $\mathcal{C}_0[B_1/I]$ is satisfiable means that $\mathcal{C}_0$ is too, since $\mathcal{R}$ specially checks that $\mathcal{C}_0$ is satisfiable for any N_I in lieu of I in the lambda case (see the added constraint $\mathcal{C}'[N_I/I][\nu'_{I_i}/\nu_{I_i}]$). This added constraint is needed since, otherwise, $\mathcal{C}_0[B_1/I]$ could be true, but $\mathcal{C}_0$ not, because of the particular current value of B_1.

For instance, assume that the constraint $\mathcal{C}_0$ says $(I, =, \texttt{stat})$ for the body of E_0; this is not satisfiable and the program should not be accepted. But if E_0 is used in an application where the argument E_1 has a binding time B_1 that is `stat`, $\mathcal{C}_0[\texttt{stat}/I]$ *is* satisfiable ... which is something we do not want. We want all subexpressions of an expression to be type and effect correct, which is the reason for the special $\mathcal{C}'[N_I/I][\nu'_{I_i}/\nu_{I_i}]$ constraint.

Let thus m_0 be the model:

$$m_0 \quad = \quad \begin{array}{l} [N_I \rightarrow I]m[\nu_{I_i} \rightarrow \nu'_{I_i} \mid \nu Ii \in \{\nu_{I_i}\}], \text{ on the set of } \nu_{I_i} \\ m, \text{ otherwise} \end{array}$$

then, $m_0 \models \mathcal{C}_0$. Note that m needs to be used after $[N_I \rightarrow I]m[\nu_{I_i} \rightarrow \nu'_{I_i}]$ to recover the values of the ν'_{I_i} related to N_I in $\mathcal{C}_0$ (and corrupted after the substitution $[N_I \rightarrow I]$). Also, note that I can be prevented (via alpha-renaming) from appearing in the domain and image of m

Then, $m = [x \rightarrow (m_0 x)[m_0 B_1/I] \mid x \in domain(m)]$.

By induction, $m_0 A \vdash E_0 : (I : m_0 T_1{}') \xrightarrow{m_0 \nu} m_0 T \ \# \ m_0 B_0$.

Since m_0 differs from m only on I, which could have been alpha-renamed, m_0 can be interchangeably used in lieu of m on any expression but ν and T (which are the only ones truly dependent on I). Thus $m_0 \models \mathcal{C}_1$ and, by induction, $m_0 A \vdash E_1 : m_0 T_1 \ \# \ m_0 B_1$. Using $(\rightarrow \mathrm{E}_s)$, $m_0 T_1 = m_0 T_1{}'$ and $m_0 A = mA$:
$\Longrightarrow mA \vdash (E_0\ E_1) : (m_0 T)[m_0 B_1/I] \ \# \ m_0 B_0 \vee (m_0 \nu)[m_0 B_1/I]$ (by typing rule $(\rightarrow \mathrm{E}_s)$)
$\Longrightarrow mA \vdash (E_0\ E_1) : (mT) \ \# \ mB_0 \vee m\nu$ (by definition of m)
$\Longrightarrow mA \vdash (E_0\ E_1) : (mT) \ \# \ m(B_0 \vee \nu)$ □

D Completeness

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

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

Proof: The proof is by structural induction on E and on the number of argument types in terms.

- $E = I$

Suppose that $mA \vdash I : T \# B$.
$\Longrightarrow I \in mA$ and $(mA)(I) \sqsubseteq T$ (by typing rules (Var_s) and (Conv_s))
$\Longrightarrow I \in A$
$\Longrightarrow$ there exists $T' = New(Erase(A(I))$ and ν such that:

$$(T', \nu, Constraint(A(I), \sqsubseteq, T') \cup \{(I, \sqsubseteq, \nu)\}) = \mathcal{R}(A, E)$$

Let

$$\begin{cases} B' = \nu \\ C' = Constraint(A(I), \sqsubseteq, T') \cup \{(I, \sqsubseteq, \nu)\} \\ m' = m + [\nu \rightarrow B] \\ m' \models Constraint(T', =, T) \end{cases}$$

Note that m' exists since T' only uses new variables, which are mapped in m' to the appropriate latent effect variables of T.

- $mA = m'A$ since ν and the free variables of T' are new.
- $I \sqsubseteq B$
  $\Longrightarrow I \sqsubseteq m'\nu$
  $\Longrightarrow m'I \sqsubseteq m'\nu$, since m' is only defined on variables.
  $\Longrightarrow m' \models \{(I, \sqsubseteq, \nu)\}$

  Also $m' \models Constraint(T', =, T)$
  $\Longrightarrow m'T' = m'T = T$, since T has no variables.
  $\Longrightarrow (m'A)(I) \sqsubseteq m'T'$, since $(mA)(I) \sqsubseteq T$ and $mA = m'A$.
  $\Longrightarrow m' \models Constraint(A(I), \sqsubseteq, T')$
  Thus, $m' \models C'$.
- $m'B' = m'\nu = B$.
- $m'T' = m'T$, by definition of m'.
  $\Longrightarrow m'T' = T$, since T has no variables in $domain(m')$.

- $E = (\texttt{lambda}\ (I)\ E_0)$

Suppose that $mA \vdash (\texttt{lambda}\ (I)\ E_0) : T \# B$. By definition of $\sqsubseteq$ and the ($\text{i}{\rightarrow}\text{I}_s$) rule, there exist T_1, T_0 and B_0 such that

$$\begin{cases} (I : T_1) \xrightarrow{B_0} T_0 \ = T \\ (mA)[I : T_1] \vdash E_0 : T_0 \ \# \ B_0 \end{cases}$$

Let ν_1' be a new variable such that $(mA)[I : T_1] = (m + [\nu_1' \rightarrow T_1])(A[I : \nu_1'])$. By induction, there exist T'_0, B'_0, C'_0, m'_0 such that:

$$\begin{cases} \mathcal{R}(A[I : \nu_1'], E_0) = (T'_0, B'_0, C'_0) \\ m'_0 \models C'_0 \\ m'_0 T'_0 = T_0 \\ m'_0 B'_0 = B_0 \\ m'_0(A[I : \nu_1']) = (m + [\nu_1' \rightarrow T_1])(A[I : \nu_1']) \end{cases}$$

By definition of $\mathcal{R}$, there exist $T', B', C', C'', \nu, \nu', N_I, \{\nu_{I_i}\}, \{\nu'_{I_i}\}$ such that:

$$\begin{cases} \mathcal{R}(A,E) = (T', B', C') \\ T' = (I : \nu'_1) \xrightarrow{\nu} T'_0 \\ B' = \nu' \\ C'' = C'_0 \cup \{(B'_0, =, \nu)\} \\ C' = C'' \cup C''[N_I/I][\nu'_{I_i}/\nu_{I_i}] \end{cases}$$

Since ν, ν' and ν'_{I_i} are not in the domain of m'_0, define m' as follows:

$$m' = m'_0 + [\nu \rightarrow B_0] + [\nu' \rightarrow B] + [\nu'_{I_i} \rightarrow (m'_0 \nu_{I_i})[N_I/I] \mid \nu'_{I_i} \in \{\nu'_{I_i}\}]$$

- $m' \models C'$, by definition of m'.
- $m'B' = m'\nu' = B$.
- $m'T' = (I : m'\nu'_1) \xrightarrow{m'\nu} m'T'_0$
  $\Longrightarrow m'T' = (I : m'_0\nu'_1) \xrightarrow{B_0} T_0$
  $\Longrightarrow \mathrm{m'}T' = \ (I : T_1) \xrightarrow{B_0} T_0 \ = T.$
- $m'A = m'_0A = mA$.

- $E = (\texttt{lambda}\ (I : T_1)\ E_0)$

Suppose that $mA \vdash (\texttt{lambda}\ (I : T_1)\ E_0) : T \# B$. By definition of $\sqsubseteq$ and the $(\rightarrow \mathrm{I}_s)$ rule, there exist T_1, T_0 and B_0 such that

$$\begin{cases} (I : T_1) \xrightarrow{B_0} T_0 \ = T \\ (mA)[I : T_1] \vdash E_0 : T_0 \ \# \ B_0 \end{cases}$$

Using the $(\mathrm{i}\rightarrow \mathrm{I}_s)$ rule, one gets that $mA \vdash (\texttt{lambda}\ (I)\ E_0) : T \# B$. By induction on the number of explicit argument types, there exist T'', B'', C'', m'' such that:

$$\begin{cases} \mathcal{R}(A, (\texttt{lambda}\ (I : t_1)\ E_0)) = (T'', B'', C'') \\ m'' \models C'' \\ m''T'' = T \\ m''B'' = B \\ m''A = mA \end{cases}$$

Let $T' = T'' = (I : T'_1) \xrightarrow{\nu} T\ ), B' = B'', C' = C'' \cup Constraint(T_1, =, T'_1)$ and $m' = m''$.

- $m'T' = m''T'' = T$.
- $m'B' = m''B'' = B$.
- $m' \models C''$.

  Since $m'((I : T'_1) \xrightarrow{\nu} T\ ) = T$ and $T = (I : T_1) \xrightarrow{B_0} T_0$, then $m'T'_1 = T_1 = m'T_1$. One gets $m' \models Constraint(T_1, =, T'_1)$.

  Thus, $m' \models C'$.
- $m'A = m''A = mA$.

- $E = (E_0\ E_1)$

Suppose that $mA \vdash (E_0\ E_1) : T \# B$. By definition of ($\rightarrow$E$_s$) rule, there exist $T_1, B_1, T_2.B_2$ and B_0 such that

$$\begin{cases} mA \vdash E_0 : (I : T_1) \xrightarrow{B_2} T_2 \ \# B_0 \\ mA \vdash E_1 : T_1 \# B_1 \\ T = T_2[B_1/I] \\ B = B_0 \vee B_2[B_1/I] \end{cases}$$

By induction, there exist T'_1, B'_1, C'_1, m'_1 such that

$$\begin{cases} \mathcal{R}(A, E_1) = (T'_1, B'_1, C'_1) \\ m'_1 \models C'_1 \\ m'_1 T'_1 = T_1 \\ m'_1 B'_1 = B_1 \\ m'_1 A = mA \end{cases}$$

Similarly, by induction, there exist $T'_2, B'_2, C'_0, m'_0, T''_1$ such that

$$\begin{cases} \mathcal{R}(A, E_0) = ((I : T''_1) \xrightarrow{B'_2} T'_2\ , B'_0, C'_0) \\ m'_0 \models C'_0 \\ m'_0 B'_0 = B_0 \\ \\ m'_0 T''_1 = T_1 \\ m'_0 B'_2 = B_2 \\ m'_0 T'_2 = T_2 \\ \\ m'_0 A = mA \end{cases}$$

By definition of $\mathcal{R}$, let T', B', C' be such that:

$$\begin{cases} T' = T'_2 \\ B' = B'_0 \vee B'_2 \\ C' = C'_1 \cup C'_0[B'_1/I] \cup Constraint(T'_1, =, T''_1) \end{cases}$$

The domains of m'_1 and m'_0 are disjoint. So, using an idea similar to the one used in the soundness theorem, let m' be defined as:

$$\begin{array}{rcl} m'x & = & [N_I \rightarrow B_1]m'_0[\nu_{I_i} \rightarrow \nu'_{I_i}]x, \quad \text{if } x \text{ is one of the } \nu_{I_i} \text{ in } C'_0 \\ & & m'_0 x, \text{ for the other variables in the domain of } m_0 \\ & & m'_1 x, \text{ otherwise} \end{array}$$

Note that for a variable other than ν_{I_i} in C'_0 (i.e., one of the ν'_{I_i}), one has $(m'_0 x)[B_1/I] = m'_0 x$ since I has been explicitly substituted away in C'_0. Thus, $m'x = (m'_0 x)[B_1/I]$ on the domain of m'_0.

- $m' \models C'_1$, by definition of m'.

  For any (b, r, b') in C'_0, since $m'_0 \models C'_0$, one has: $(m'b)[B_1/I]\ r\ (m'b')[B_1/I]$. Thus, for any $(b[B'_1/I], r, b'[B'_1/I])$ in $C'_0[B'_1/I]$:
  $m'(b[B'_1/I]) = (m'b)[m'B'_1/I] = (m'b)[B_1/I]$
  $m'(b'[B'_1/I]) = (m'b')[m'B'_1/I] = (m'b')[B_1/I]$
  $\Longrightarrow m'(b[B'_1/I])\ r\ m'(b'[B'_1/I])$

$\Longrightarrow m' \models C'_0[B'_1/I]$

Also:
$m'T'_1 = m'_1T'_1 = T_1$
$m'T''_1 = (m'_0T''_1)[B_1/I] = T_1[B_1/I] = T_1$, by possible alpha-renaming of I
$\Longrightarrow m' \models Constraint(T'_1, =, T''_1)$

Thus, $m' \models C'$.

- $\begin{aligned} m'T' &= m'T'_2 \\ &= (m'_0T'_2)[B_1/I] \\ &= T_2[B_1/I] = T \end{aligned}$

- $\begin{aligned} m'B' &= m'(B'_0 \vee B'_2) \\ &= (m'_0B'_0)[B_1/I] \vee (m'_0B'_2)[B_1/I] \\ &= B_0[B_1/I] \vee B_2[B_1/I] \\ &= B_0 \vee B_2[B_1/I] \\ &= B \end{aligned}$

- $m'A = (m'_1 + m'_0)A = mA$

□