Polyvariant Analysis of the Untyped Lambda Calculus

Jens Palsberg
palsberg@daimi.aau.dk

Michael I. Schwartzbach
mis@daimi.aau.dk

Department of Computer Science, Aarhus University
Ny Munkegade, DK-8000 Arhus C, Denmark

February 1992

Abstract

We present a polyvariant closure, safety, and binding time analysis for the untyped lambda calculus. The innovation is to analyze each abstraction afresh at all syntactic application points. This is achieved by a semantics-preserving program transformation followed by a novel monovariant analysis, expressed using type constraints. The constraints are solved in cubic time by a single fixed-point computation.

Safety analysis is aimed at determining if a term will cause an error during evaluation. We have recently proved that the monovariant safety analysis accepts strictly more terms than simple type inference. This paper demonstrates that the polyvariant transformation makes even more terms acceptable, even some without higher-order polymorphic types. Furthermore, polyvariant binding time analysis can improve the partial evaluators that base a polyvariant specialization on only monovariant binding time analysis.

1 Introduction

Static analysis of untyped higher order functional programs can be based on either a type analysis [15] or a closure analysis [24]. Both provide basic behaviors information that can be used in further analyses. Their outputs differ as follows.

Type analysis: Transforms an untyped program into a typed program. If this succeeds, then it is guaranteed that constants will not be misused, for example that succ will not be applied to true.

Closure analysis: For every application point, computes a superset of the possible lambda abstractions (closures) that may be applied at that point.

Their treatment of abstractions are fundamentally different, as follows. Type analysis assumes a "most general" context for an abstraction before doing the analysis. This implies that once type checked, an abstraction needs never be reconsidered, even if other parts of the program are modified. Closure analysis considers only the finitely many contexts in the given program. This implies that if the program is modified, then the closure analysis must be redone.

Most traditions approaches to these analyses are similar in *merging* information about contexts. Following Bondorf and Danvy [5], we call such analysis *momovariant*. Monovariant type analysis merges information about all conceivable contexts, whereas monovariant closure analysis merges information about all contexts at hand.

In contrast to the monovariant closure analysis, *polyvariant* analysis [5] will treat an abstraction differently in the finitely many different syntactic contexts. This means that it will analyze each abstraction afresh at all syntactic application points. As exemplified later, a polyvariant closure analysis yields more precise results than a monovariant one.

One use of polyvariant analysis is to obtain better safety checking. Safety analysis is an essential part of type analysis, as follows.

Safety analysis: Decides if constants will be misused.

An approximative analysis says "unsafe" too often; all analyses must of

course be approximative since the problem is inherently undecidable. We have recently proved that the monovariant safety analysis accepts strictly more terms than simple type inference. As exemplified later, the polyvariant transformation makes even more terms acceptable, even some without higher-order polymorphic types.

The imprecision of monovariant analysis is also significant in partial evaluators that perform *polyvariant specialization* [6]. Polyvariant specialization lets every application be either unfolded or lead to the generation of a residual call to a specialized abstraction [12, 13]. If such polyvariant specialization is based on a monovariant binding time analysis rather than a polyvariant one, however, then poorer specialized abstractions are obtained. This is because the monovariant analysis merges the context information of an application point with that of other points. We believe that polyvariant specialization should be bred on polyvariant analysis. The purpose of binding time analysis can be summarized as follows.

Binding time analysis: Decides if the value of an expression is known at compile time.

The prospects of polyvariant analysis are summarized in figure 1.

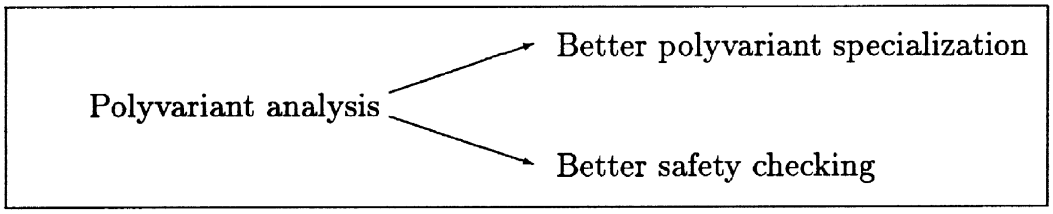


Figure 1: Prospects of polyvariant analysis.

This paper presents a polyvariant closure, safety, and binding time analysis for the untyped lambda calculus. The analysis is applicable to both strict and lazy higher-order functional languages, including ML [16], SCHEME [1], and MIRANDA [25]. The basic component is a monovariant closure analysis; the safety and binding time analyses are developed on top of it. The closure and binding time analyses appears to be slight improvements of those used in the partial evaluator SIMILIX [4]; the safety analysis is new. The analysis is specified using uniform type constraints. Note that we deliberately use the word "type" to denote any kind of behavioral information about a piece of

program text. We use "type variables" to denote unknown such information. The type constraints are notionally derived using a finite *trace graph* which again is obtained directly from the program text. They are then solved in cubic time by a single fixed-point computation over a finite lattice.

We obtain polyvariant analyses by preceding the monovariant analyses with one or more semantics-preserving *program transformations*. In effect we copy all abstractions as many times as might *potentially* be needed by a polyvariant specializer. Note that the explicit transformation is merely a conceptualization; this paper presents an algorithm that does it in a lazy, implicit fashion.

The notion of polyvariant binding time analysis stems from the partial evaluator Schism of Consel [7]. Schism treats only a first-order language, however. The previous approaches to binding time analysis for a higher-order language are all monovariant. The analyses of Nielson and Nielson [19, 18], and Mogensen [17] require programs to be typable with simple recursive types, simple types, or ML-polymorphic types, respectively. The analyses of Bondorf [4], Consel [8] and Gomard and Jones [11, 10], do not impose any typing restrictions. Only the analysis of Nielson and Nielson and that of Jones and Gomard are defined on languages with nested fixed-points. Our analysis imposes no restrictions and is polyvariant.

The closure, safety, and binding time analyses are performed simultaneously, yielding fast execution time. Our safety analysis need not be coupled with the binding time analysis, however. When it is, however, it will guarantee safety for those operators whose arguments are known at compile-time. For comparison, Consel [8] showed how to simultaneously perform monovariant closure and binding time analyses, but not safety analysis, for an untyped higher-order language. The Similix system of Bondorf and Danvy [4, 5] performs a number of analyses one by one. The analysis of Gomard [10] performs both partial type inference and monovariant binding time analysis.

Figure 2 gives an overview of our analysis framework.

In the following section we briefly introduce a small example language L on which to present our analysis. It is a lambda calculus with constants and a letrec construct. In section 3 we extend it into a language XL to allow polyvariant analysis and, potentially, polyvariant specialization. The extra construct needed is a restricted form of product constants and field selection.

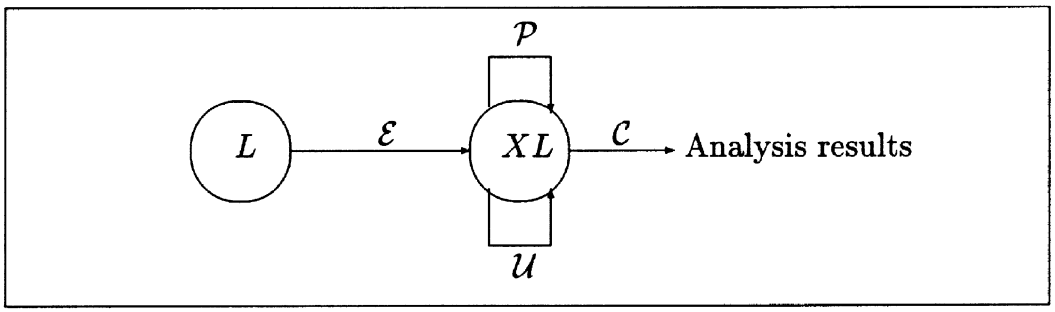


Figure 2: Our analysis framework.

This enables us to define the necessary program transformations. There is a natural embedding $\mathcal{E}$ of L into XL. In section 4 we define a monovariant closure analysis $\mathcal{C}$ of the extended language. In section 5 we extend it to a safety analysis and a binding time analysis. The analyses can then be made polyvariant by first applying the program transformation $\mathcal{P}$ described in section 6. In section 7 we demonstrate that also the standard treatment of the polymorphic let can be explained through a program transformation $\mathcal{U}$. We note that $\mathcal{U} \circ \mathcal{U} = \mathcal{U}$ and that $\mathcal{U} \circ \mathcal{P} = \mathcal{P} \circ \mathcal{U}$. Finally, in section 8 we give an example of how the closure and safety analyses work.

The treatment of tuples and lists is not given here; it is straightforward to incorporate.

2 The Language

$$
\begin{array}{rcl}
\mathrm{E} & ::= & x \mid \lambda x.\mathrm{E} \mid \mathrm{E}_1\ \mathrm{E}_2 \mid \\
 & & \textsf{letrec } x_j \textsf{ is } \mathrm{E}_j \textsf{ in } \mathrm{E} \mid \textsf{if } \mathrm{E}_1\ \mathrm{E}_2\ \mathrm{E}_3 \mid \\
 & & \textsf{true} \mid \textsf{false} \mid \textsf{0} \mid \textsf{succ } \mathrm{E}
\end{array}
$$

Figure 3: Syntax of the example language, L.

Our example language is a lambda calculus with boolean and integer constants, and a letrec construct, see figure 3. The semantics can be either strict or lazy, our analyses will apply in both cases. For example, the language can

be seen as abstract syntax for a subset of SCHEME. Note that the partial evaluator SIMILIX [4, 5] accepts SCHEME programs written in a subset of our example language (except for syntactic differences).

3 The Extended Language

$$
\begin{array}{rl}
\mathrm{E} ::= & x \mid [\alpha_1\colon \lambda x_1.\mathrm{E}_1, \ldots, \alpha_n\colon \lambda x_n.\mathrm{E}_n] \mid \mathrm{E}_1\ \alpha\ \mathrm{E}_2 \mid \\
& \textsf{letrec } x_j \textsf{ is } \mathrm{E}_j \textsf{ in } \mathrm{E} \mid \textsf{if } \mathrm{E}_1\ \mathrm{E}_2\ \mathrm{E}_3 \mid \\
& \textsf{true} \mid \textsf{false} \mid \textsf{0} \mid \textsf{succ } \mathrm{E}
\end{array}
$$

Figure 4: Syntax of the extended language, XL.

To be able to perform the program transformations that are needed for polyvariant analysis and specializations we now extend our example language, see figure 4. The two new constructs are a "lambda-tuple" and a "select-apply". The latter both selects a lambda in a tuple and applies an argument to it. This of course introduces the possibility of a field-not-present error at runtime. This is not significant, however, because there is a semantics-preserving embedding $\mathcal{E}$ of the language in the previous section into the extended language. Let α_0 be a fixed label. We then define $\mathcal{E}$ in the obvious way.

- $\mathcal{E}(x) = x$
- $\mathcal{E}(\textsf{if } E_1E_2E_3) = \textsf{if } \mathcal{E}(E_1)\ \mathcal{E}(E_2)\ \mathcal{E}(E_3)$
- $\mathcal{E}(\textsf{true}) = \textsf{true}$
- $\mathcal{E}(\textsf{false}) = \textsf{false}$
- $\mathcal{E}(\textsf{0}) = \textsf{0}$
- $\mathcal{E}(\textsf{succ } E) = \textsf{succ } \mathcal{E}(E)$
- $\mathcal{E}(\textsf{letrec } x_j \textsf{ is } E_j \textsf{ in } E) = \textsf{letrec } x_j \textsf{ is } \mathcal{E}(E_j) \textsf{ in } \mathcal{E}(E)$
- $\mathcal{E}(E_1E_2) = \mathcal{E}(E_1)\ \alpha_0\ \mathcal{E}(E_2)$
- $\mathcal{E}(\lambda x.E) = [\alpha_0 : \lambda x.\mathcal{E}(E)]$

We claim the following.

Soundness of Embedding Result: The embedding $\mathcal{E}$ is semantics-preserving.
Proof sketch: By induction in the length of executions. □

4 Closure Analysis

We now present a monovariant closure analysis of the extended language. In this context closures will be sets of lambda abstractions. We initially ensure that all bound variables are distinct; hence, an abstraction $\lambda x.E$ can be uniquely denoted by the token λx. We denote by LAMBDA the finite set of all lambda tokens in the current program.

For each (sub)expression E of the program in question we introduce a type variable $[\![E]\!]$ denoting its (unknown) closure information—a subset of LAMBDA. The analysis will proceed in two phases. First we derive the necessmy constraints on these variables; then we compute the minimal sulution by a globs least fixed-point derivation. The idea of generating constraints on type variables from the program syntax is also exploited in [26, 23, 20].

To facilitate the presentation of the constraints, we introduce the concept of a *trace graph*. It has a node for each lambda abstraction, denoted by the corresponding lambda token, and one for the entire expression, denoted MAIN. The edges will reflect possible applications.

We use the following terminology.

Local node: Consider the parse tree for an expression. We shall call a parse tree node *local*, if it can be reached from the root without passing through a lambda abstraction.

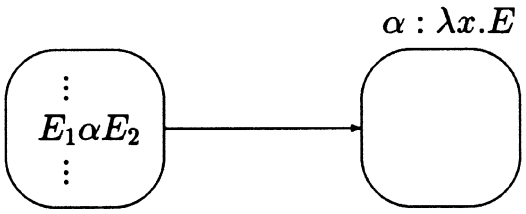
With every trace graph node we associate a set of *local constraints*. They are obtained in the following manner.

Local constraints:

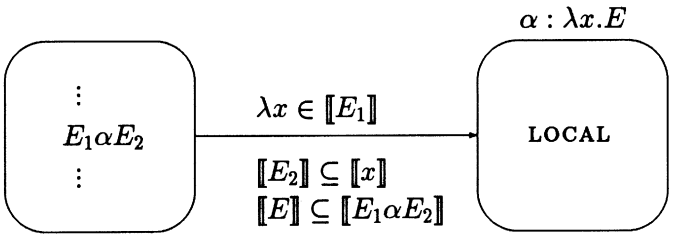
- For every local $\mathsf{if}\ E_1E_2E_3$ we have the constraint $[\![\mathsf{if}\ E_1E_2E_3]\!] \supseteq [\![E_2]\!] \cup [\![E_3]\!]$.
- For every local $[\alpha_i : \lambda x_i.E_i]$ we have the constraint $[\![[\alpha_i : \lambda x_i.E_i]\!] \supseteq \{\lambda x_i\}$.
- For every local $\mathsf{letrec}\ x_j\ \mathsf{is}\ E_j\ \mathsf{in}\ E$ we have the constraints $[\![x_j]\!] = [\![E_j]\!]$.

For example, the first constraint can be read as "the closures produced by $\mathsf{if}\ E_1E_2E_3$ must include both those produced by E_2 and those produced by E_3." We write inclusion, rather than equality, to be able to express the constraints in a uniform manner; however, we obtain the same result, since we compute the minimal solution.

The outgoing trace graph edges arise from lock applications. For every $E_1\alpha E_2$ we have an edge to the trace graph node for any lambda abstraction $\lambda x.E$ that has an α-label. That is, we have the following picture.



With each trace graph edge we associate a *condition* and two *connecting constraints*. The condition is simply $\lambda x \in [\![E_1]\!]$; it states that this edge is only relevant if the closure of the indicated trace graph node is a possible result of E_1. The connecting constraints reflect the relationship between formal and actual arguments and resets. They are $[\![E_2]\!] \subseteq [\![x]\!]$ and $[\![E_1\alpha E_2]\!] \supseteq [\![E]\!]$. Thus, a typical part of the trace graph will now look as follows.

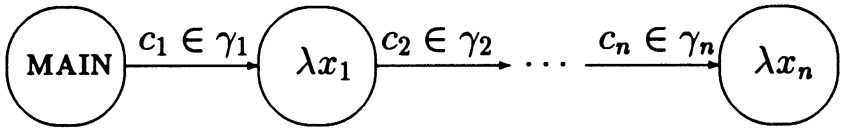


The condition is written above the edge, and the connecting constraints below. A trace graph node corresponds to a lambda abstraction; its body determines both the outgoing trace graph edges (as indicated in the left-hand node) and the lock constraints (as indicated in the right-hand node).

From the trace graph we derive a finite set of *global constraints*. Each of these is a *conditional inclusion* of the form

$$c_1 \in \gamma_1, \ldots, c_n \in \gamma_n \Rightarrow Q$$

where the c_i's are closures and the γ_i's are type variables. The Q is a closure set inclusion that looks like one of $c \subseteq \gamma$, $\gamma \subseteq c$, or $\gamma_1 \subseteq \gamma_2$. Each path in the trace graph from the MAIN node give rise to global constraints as follows. Suppose the path is



The corresponding global constraints are

$$c_1 \in \gamma_1, \ldots, c_n \in \gamma_n \Rightarrow \text{LOCAL} \cup \text{CONNECT}$$

where LOCAL are the local constraints of the final node and CONNECT are the connecting constraints of the final edge. The trace graph may have infinitely many paths, but since there are only finitely many closures and variables, there will only be finitely many different global constraints—for simple combinatorial reasons. In the worst case the size of the global constraint set is exponential in the size of the trace graph, which itself is linear in the size of the program.

We now claim the following.

Soundness of Analysis Result: The global constraints are always satisfiable, and any solution will provide sound closure information about the program.
Proof: The global constraints are satisfiable, since (as yet) no inclusion is of the form $\gamma \subseteq c$. Hence, a (maximal) solution is obtained by assigning the set LAMBDA to each variable. This corresponds to trivial closure information. The minimal solution reflects the best information that we can obt n from the constraints.

In [21] we prove soundness of the closure analysis with respect to both a strict and a lazy operational semantics. □

In appendices A and B we show how to compute the unique minimal solution of a set of globs constraints. The problem is reduced to computing a least fixed-point of a monotonic function in an appropriate lattice. With an incremental algorithm, this fixed-point can be computed in time $O(n^3)$, where n is the size of the lambda term.

5 Safety and Binding Time Analysis

We now show how to extend the closure analysis to also perform safety and binding time analysis. Only new local constraints are needed.

To specify safety constraints, we introduce a set of basic types $\mathcal{B} = \{\mathsf{Bool}, \mathsf{Int}\}$. The idea is that a type variable now denotes a subset of LAMBDA $\cup\, \mathcal{B}$. The type of free variables of the entire expression can be specified by *initial* constraints of the form $[\![x]\!] = \{\mathsf{Bool}\}$, etc. This corresponds to the initial static environment in the analysis of Gomard and Jones [11]. We now add the following local constraints.

Additional local constraints:

- For every local $\mathsf{if}\ E_1E_2E_3$ we add $[\![E_1]\!] = \{\mathsf{Bool}\}$.
- For every local $\mathsf{succ}\ E$ we add $[\![E]\!] = \{\mathsf{Int}\}$ and $[\![\mathsf{succ}\ E]\!] = \{\mathsf{Int}\}$.
- For every local constants we add $[\![\mathsf{true}]\!] = [\![\mathsf{false}]\!] = \{\mathsf{Bool}\}$ and $[\![0]\!] = \{\mathsf{Int}\}$.
- For every local $E_1\alpha E_2$ we add $[\![E_1]\!] \subseteq \textsc{lambda}$.

Soundness of Analysis Result: If the constraints are satisfiable, then no execution will lee to the misuse of constants.
Proof: In [21] we give a proof of this with respect to both a strict and a lazy operations semantics. □

Note that satisfiability is no longer guaranteed, since we now have inclusions of the form $\gamma \subseteq c$. Thus, conflicting constraints can be phrased; in particular, the maximal assignment will no longer necessarily be a solution.

Since safety analysis shares an important ambitions with type inference, namely the avoidance of run-time errors. However, they are bred on rather different perspectives. We are able to compare the two approaches in a formal qualitative sense. The basis for comparison will be simple type inference [26]; the extension to type inference in ML is closely paralled by the $\mathcal{U}$-transformation presented in section 7.

Comparison with Type Inference Result: The set of safe lambda terms typable in the simply typed lambda calculus is a strict subset of those accepted by the safety analysis. This is still true if we allow recursive types, as in the $\lambda\mu$-calculus [2].
Proo: This is also proved in [21]. □

In particular, safety analysis accepts two special families of safe terms: those without constants, and those in normal form.

To specify the binding time analysis, we introduce yet another basic "type" called Code. It plays the rôle of "unknown" or "dynamic" in other approaches. It can only be introduced in the initial constraints, and it will then be propagated automatically by the connecting constraints and some additional local constraints. We now summarize the definition of local con-

straints for all three analyses.

Summary of loyal constraints:

- For every local $\mathsf{if} E_1 E_2 E_3$ we have $[\![E_1]\!] \subseteq \{\mathsf{Bool}, \mathsf{Code}\}$ and $[\![\mathsf{If}\ E_1 E_2 E_3]\!] \supseteq [\![E_2]\!] \cup [\![E_3]\!]$.
- For every local $\mathsf{succ}\ E$ we have $[\![E]\!] = \{\mathsf{Int}, \mathsf{Code}\}$, $\mathsf{Int} \in [\![E]\!] \Rightarrow \{\mathsf{Int}\} \subseteq [\![\mathsf{succ}\ E]\!]$ and $\mathsf{Code} \in [\![E]\!] \Rightarrow \{\mathsf{Code}\} \subseteq [\![\mathsf{succ}\ E]\!]$.
- For every local constants we have $[\![\mathsf{true}]\!] = [\![\mathsf{false}]\!] = \{\mathsf{Bool}\}$ and $[\![0]\!] = \{\mathsf{Int}\}$.
- For every local $E_1 \alpha E_2$ we add $[\![E_1]\!] \subseteq \textsc{lambda} \cup \{\mathsf{Code}\}$ and $\mathsf{Code} \in [\![E_1]\!] \Rightarrow \{\mathsf{Code}\} \subseteq [\![E_1 \alpha E_2]\!]$.
- For every local $[\alpha_i : \lambda x_i.E_i]$ we have $[\![[\alpha_i : \lambda x_i.E_i]]\!] \supseteq \{\lambda x_i\}$.
- For every local $\mathsf{letrec}\ x_j\ \mathsf{is}\ E_j\ \mathsf{in}\ E$ we have $[\![x_j]\!] = [\![E_j]\!]$.

If the constraints are satisfiable, then any expression whose type variable does not contain Code can be executed at compile time. A formal statement of soundness must refer to a concrete code generation scheme. Note that if we summarize the closure and binding time constraints alone, then the constraints *will* be satisfiable.

6 The Polyvariant Tansformation

The closure analysis is monovariant; it analyzes each abstraction once. It can be made *polyvariant* by preceding it by a program transformation $\mathcal{P}$ on the extended language. The key idea is to give a different analysis of each lambda abstraction for every syntactic application in the program. To be able to do this, $\mathcal{P}$ generates a copy of each abstraction for every syntactic application. It is here we need the "lambda-tuple" and "select-apply": to collect the different copies and distinguish between them.

The transformation $\mathcal{P}$ is a map from programs to programs. We first assign unique labels $\beta_1, \ldots, \beta_k$ to the syntactic applications in the program, i.e., every parse tree node which is an application is labeled by a β_j. Next, we

can define $\mathcal{P}$ inductively in the structure of the syntax as follows.

- $\mathcal{P}(x) = x$
- $\mathcal{P}(\mathsf{if}\ E_1E_2E_3) = \mathsf{if}\ \mathcal{P}(E_1)\mathcal{P}(E_2)\mathcal{P}(E_3)$
- $\mathcal{P}(\mathsf{true}) = \mathsf{true}$
- $\mathcal{P}(\mathsf{false}) = \mathsf{false}$
- $\mathcal{P}(\mathsf{0}) = \mathsf{0}$
- $\mathcal{P}(\mathsf{succ}\ E) = \mathsf{succ}\ \mathcal{P}(E)$
- $\mathcal{P}(\mathsf{letrec}\ x_j\ \mathsf{is}\ E_j\ \mathsf{in}\ E) = \mathsf{letrec}\ x_j\ \mathsf{is}\ \mathcal{P}(E_j)\ \mathsf{in}\ \mathcal{P}(E)$
- $\mathcal{P}(E_1\alpha E_2) = \mathcal{P}(E_2)\alpha\beta_j\mathcal{P}(E_2)$, where β_j is the label of this syntactic application, and $\alpha\beta_j$ is a concatenated label.
- $\mathcal{P}([\alpha_j : \lambda x_i.E_j]) = [\alpha_i\beta_j : \lambda x_i.P(E_i)]$. Here we generate k copies of each lambda abstraction. Each label is concatenated with $\beta_1, \ldots, \beta_k$.

We now claim the following.

Soundness of Transformation Result: $\mathcal{P}$ is semantics-preserving.
Proof sketch: By induction in the length of executions. □

The effect of a monovariant analysis of the transformed program is what we define to be a polyvariant analysis of the origins program. The size of $\mathcal{P}(E)$ can become exponential in the size of E. Each type variable $[\![E]\!]$ now exists in multiple versions $[\![E]\!]_1, \ldots, [\![E]\!]_k$. An analysis of the original program could be obtained by for each expression E to compute the union $[\![E]\!]_1 \cup \cdots \cup [\![E]\!]_k$ and remove the information that distinguishes the different copies of closures. However, as we shall see, in the various anayses based on closure analysis the more detailed information can be used directly.

Note that the $\mathcal{P}$ transformation can be applied repeatedly to gain ever more precise information in a subsequent analysis. However, one cannot obtain arbitrarily precise information; it appears that the limit of $\mathcal{P}$ is decidable.

7 The Polymorphic Tansformation

Another transformation that can give more precise closure information is inspired by the let-polymorphism of e.g. ML. The idea is to give a different analysis of each *named* lambda abstraction for every occurrence of its name. Note how this criterion differs from that used in the polyvariant transformation.

- $\mathcal{U}(x) = x$
- $\mathcal{U}(\mathsf{if}\ E_1\ E_2\ E_3) = \mathsf{if}\ \mathcal{U}(E_1)\ \mathcal{U}(E_2)\ \mathcal{U}(E_3)$
- $\mathcal{U}(\mathsf{true}) = \mathsf{true}$
- $\mathcal{U}(\mathsf{false}) = \mathsf{false}$
- $\mathcal{U}(0) = 0$
- $\mathcal{U}(\mathsf{succ}\ E) = \mathsf{succ}\ \mathcal{U}(E)$
- $\mathcal{U}(E_1 \alpha E_2) = \mathcal{U}(E_1)\ \alpha\ \mathcal{U}(E_2)$
- $\mathcal{U}([\alpha_i : \lambda x_i.E_i]) = [\alpha_i : \lambda x_i.\mathcal{U}(E_i)]$.
- $\mathcal{U}(\mathsf{letrec}\ x_j\ \mathsf{is}\ E_j\ \mathsf{in}\ E) = \mathcal{U}(E)[x_i \leftarrow (\mathsf{letrec}\ x_j\ \mathsf{is}\ \mathcal{U}(E_j)\ \mathsf{in}\ x_i)]$. Here free occurrences of the x_i's are beta-substituted by their definitions.

The transformation $\mathcal{U}$ is a map from programs to programs, defined inductively in the structure of the syntax. It basically provides a single unfolding of all letrec-definitions.

We now claim the following.

Soundness of Transformation Result: $\mathcal{U}$ is semantics-preserving.
Proof sketch: By induction in the length of executions. □

The effect of $\mathcal{U}(E)$ is similar to the key idea in ML-style type inference, which conceptually performs a syntactic expansion of all let-definitions. Note that the transformed program may become exponentially larger; this relates to ML type inference being DEXPTIME complete [14].

Properties of Transformation Result: $\mathcal{U}$ is idempotent, i.e., $\mathcal{U} \circ \mathcal{U} =$

$\mathcal{U}$; thus, the the polymorphic transformation cannot be iterated. $\mathcal{U}$ and $\mathcal{P}$ commute, i.e., $\mathcal{P} \circ \mathcal{U} = \mathcal{U} \circ \mathcal{P}$; thus, the polyvariant and the polymorphic transformations are independent.
Proof sketch: By induction in the size of lambda terms. □

8 Examples

We now exemplify the $\mathcal{E}$ and $\mathcal{P}$ transformations and the closure and safety analyses.

Consider the following expression:

$$(\lambda f.\mathsf{if}\ E\ \mathsf{then}\ f\ \mathsf{true}\ \mathsf{else}\ \mathsf{succ}\ (f\ \mathsf{0}))(\mathsf{if}\ E'\ \mathsf{then}\ \lambda x.x\ \mathsf{else}\ \lambda y.\mathsf{0})$$

Note that it has neither a higher-order polymorphic type [9, 22] nor an intersection type [3]. (We assume that E and E' are harmless.) Nevertheless, we will demonstrate that our polyvariant safety analysis correctly guarantees that constants will not be misused when executing the expression.

Choosing α_0 as our fixed label and then applying the $\mathcal{E}$ embedding yields the following expression in the extended language:

$$\alpha_0\ (\mathsf{if}\ \bar{E}'\ \mathsf{then}\ [\alpha_0\ :\ \lambda x.x]\ \mathsf{else}\ [\alpha_0\ :\ \lambda y.\mathsf{x}])$$

If we safety analyze this expression directly, then we will get the answer "unsafe". To see this, let us first perform the closure analysis, ignoring safety and binding time. The result will be that both $\lambda x.x$ and $\lambda y.\mathsf{0}$ can be applied at the points $f\alpha_0\ \mathsf{true}$ and $f\alpha_0\ \mathsf{0}$. When introducing the safety constraints, some of them will say that the type variable for x can contain both Bool and Int. This is because $\lambda x.x$ can be applied to both true and 0. Other constraints will then say that the result of applying $\lambda x.x$ can be both Bool and Int. The analysis conclude that the application $f\alpha_0\mathsf{0}$ can yield both a Bool and an Int, so trying to apply succ is an error. Appendix C shows the trace graph and the global constraints.

Let us now make the polyvariant transformation $\mathcal{P}$ before doing the safety analysis. There are three application points, so let us choose a label for each of them and call the results of concatenating each of them with α_0 for α, β,

and γ. Applying $\mathcal{P}$ yields:

$$\left[\begin{array}{l} \alpha : \lambda f_1.\mathsf{if}\ \bar{E}\ \mathsf{then}\ f_1\ \alpha\ \mathsf{true}\ \mathsf{else}\ \mathsf{succ}\ (f_1\ \beta\ \mathsf{0}) \\ \beta : \lambda f_2.\mathsf{if}\ \bar{E}\ \mathsf{then}\ f_2\ \alpha\ \mathsf{true}\ \mathsf{else}\ \mathsf{succ}\ (f_2\ \beta\ \mathsf{0}) \\ \gamma : \lambda f_3.\mathsf{if}\ \bar{E}\ \mathsf{then}\ f_3\ \alpha\ \mathsf{true}\ \mathsf{else}\ \mathsf{succ}\ (f_3\ \beta\ \mathsf{0}) \end{array}\right]$$

$$\gamma\left(\mathsf{if}\ \bar{E}'\ \mathsf{then}\left[\begin{array}{ll} \alpha : & \lambda x_1.x_1 \\ \beta : & \lambda x_2.x_2 \\ \gamma : & \lambda x_3.x_3 \end{array}\right]\ \mathsf{else}\ \left[\begin{array}{ll} \alpha : & \lambda y_1.\mathsf{0} \\ \beta : & \lambda y_2.\mathsf{0} \\ \gamma : & \lambda y_3.\mathsf{0} \end{array}\right]\right)$$

Applying the closure anaysis to this expression will tell that $\lambda x_1.x_1$ and $\lambda y_1.\mathsf{0}$ can be applied at the point $f_3\ \alpha\ \mathsf{true}$, and that $\lambda x_2.x_2$ and $\lambda y_2.\mathsf{0}$ can be applied at the point $f_3\ \alpha\ \mathsf{0}$. The increaced number of abstractions involved makes the difference in the following. When introducing the safety constraints, some of them will say that the type variable for x_1 can contain Bool. This is because $\lambda x_1.x_1$ can be applied only to true. Other constraints will then say that the result of applying $\lambda x_1.x_1$ can only be Bool. Similar considerations in the remaining three cases make the analysis correctly conclude that the expression is "safe". Appendix D shows the trace graph, the global constraints, and the minimal solutions

9 Conclusion

Our polyvariant analysis can improve partial evaluators based on polyvariant specialization, and it also improves standard polymorphic type inference. The closure and binding time analyses yield information for *all* lambda terms, and both them and the safety analysis can be improved by repeated application of the polyvariant transformation.

Acknowledgement: The authors thank Flemming Nielson and Torben Amtoft for helpful comments on a draft of the paper. Bernard Steffen suggested how to improve the complexity of the fixed-point algorithm. This work has been supported in part by the Danish Research Council under the DART Project (5.21.08.03).

A Solving Conditional Inequalities

This appendix shows how to solve a finite system of conditions inequalities in quadratic time. Conditional inequalities generalize the conditional inclusions used in the analysis.

Definition 1: A *CI-system* consists of

- a complete lattice $\mathcal{D}$.
- a finite set $\{\gamma_i\}$ of *variables*.
- a finite set of *conditions inequalities* of the form

$$C_1, C_2, \ldots, C_k \Rightarrow Q$$

  Each C_i is a *condition* of the form $d \leq \gamma_j$, where $d \in \mathcal{D}$, and Q is an *inequality* of the form $d \leq \gamma_i, \gamma_i \leq d$, or $\gamma_i \leq \gamma_j$.

A *solution* L of the system assigns to each variable γ_i an element $L(\gamma_i) \in \mathcal{D}$ such that all the conditions inequalities are satisfied. □

Note that the lattice $\mathcal{D}$ need *not* be finite. In our application, $\mathcal{D}$ is either the lattice of subsets of LAMBDA (the closure analysis), LAMBDA $\cup\, \mathcal{B}$ (the safety analysis), or LAMBDA $\cup\, \mathcal{B} \cup \{\mathsf{Code}\}$ (the binding time analysis).

Lemma 2: Solutions are closed under greatest lower bound $\sqcap$. Hence, if a CI-system has solutions, then it has a unique minimal one.
Proof: Consider any conditional inequality of the form $C_1, C_2, \ldots, C_k \Rightarrow Q$, and let $\{L_i\}$ be all solutions. We shall show that $\sqcap_i L_i$ is a solution. If a condition $d \leq \sqcap_i L_i(\gamma_i)$ is true, then so is all of $d \leq L_i(\gamma_j)$. Hence, if all the conditions of Q are true in $\sqcap_i L_i$, then they are true in each L_i; furthermore, since they are solutions, Q is also true in each L_i. Since, in general, $A_k \leq B_k$ implies $\sqcap_k A_k \leq \sqcap_k B_k$, it follows that $\sqcap_i L_i$ is a solution. Hence, if thee are any solutions, then $\sqcap_i L_i$ is the unique smallest one. □

Definition 3: Let $\mathcal{C}$ be a CI-system with n distinct variables. An *assignment* is an element of $\mathcal{D}^n \cup \{error\}$ ordered as a lattices see figure 5.

If different from *error*, then it assigns an element of $\mathcal{D}$ to each variable. If V is an assignment, then $\tilde{\mathcal{C}}(V)$ is a new assignment, defined as follows. If

$V = error$, then $\tilde{\mathcal{C}}(V) = error$. An inequality is *enabled* if all of its conditions are true under V. If for any enabled inequality of the form $\gamma_i \leq d$ we do *not* have $V(\gamma_i) \leq d$, then $\tilde{\mathcal{C}}(V) = error$ otherwise, $\tilde{\mathcal{C}}(V)$ is the smallest pointwise extension of V such that

- for every enabled inequality of the form $d \leq \gamma_j$ we have $d \leq \tilde{\mathcal{C}}(V)(\gamma_j)$.
- for every enabled inequality of the form $\gamma_i \leq \gamma_j$ we have $V(\gamma_i) \leq \tilde{\mathcal{C}}(V)(\gamma_j)$.

Clearly, $\tilde{\mathcal{C}}$ is monotonic in the above lattice. □

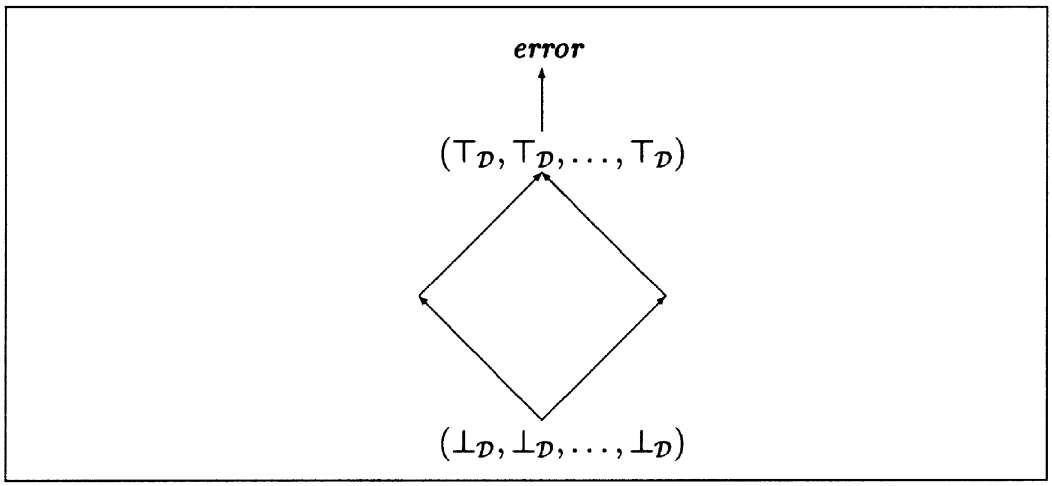


Figure 5: The lattice of assignments.

Lemma 4: An assignment $L \neq error$ is a solution of a CL-system $\mathcal{C}$ *iff* $\tilde{\mathcal{C}}(L)$. If $L = \mathcal{C}$ has no solutions, then *error* is the smallest fixed-point of $\tilde{\mathcal{C}}$.

Proof: If L is a solution of $\mathcal{C}$, then clearly $\tilde{\mathcal{C}}$ will not equal *error* and cannot extend L; hence, L is a fixed-point. Conversely, if L is a fixed-point of $\tilde{\mathcal{C}}$, then all the enabled inequalities must hold. If there are no solutions, then there can be no fixed-point below *error*. Since *error* is by definition a fixed-point, the result follows. □

This means that to find the smallest solution, or to decide that none exists, we need only compute the lest fixed-point of $\tilde{\mathcal{C}}$.

Lemma 5: For any CI-system $\mathcal{C}$, the least fixed-point of $\tilde{\mathcal{C}}$ is equal to

$$\lim_{k \to \infty} \tilde{\mathcal{C}}^k(\bot, \bot, \ldots, \bot)$$

Proof: This is a standard result about monotonic functions on complete lattices. □

B An Incremental Fixed-Point Algorithm

A naïve algorithm would accept a trace graph, derive the global constraints, and compute the minims solution by fixed-point iteration. However, the trace graph for a lambda term of size n may yield $O(2^n)$ different global constraints. Thus, we would get an exponential algorithm.

We now present an algorithm, which computes the minimal solution in time $O(n^3)$. The key idea is to incrementally compute the minimal solution to a larger and larger set of *unconditional* constraints. A loyal constraint is only added when it can be reached from the MAIN node through a path whose conditions hold in the current minims solution. If some local constraint is never added, then it need not hold in the *global* minimal solution. In this manner, each node and edge is at most visited once.

The algorithm is presented by means of two data structures: the Graph and the Solver. The former gives access to the trace graph, and the latter maintains the current minimal solution.

The state of the Graph is the trace graph, in which some nodes have been visited. Initially, all nodes are unvisited. The operations are as follows.

data structure Graph:
  main-node:
    returns the identity of the MAIN node
  outgoing-edges(n):
    returns the edges from the node n and mark it visited
  local-constraints(n):
    returns the local constraints for the node n
  connecting-constraints(e):
    returns the connecting constraints for the edge e
  seen-before(n):
    decides if n has been visited
  destination(e):
    returns the destination node of the edge e
end Graph

The state of the Solver is a set of unconditional constraints and their minims solution. It also maintains a set of trace graph edges called *front edges*.

Initially the set of constrtints is empty, and there are no front edges.

```
data structure Solver:
   add-constraints(c):
      add the constraints c and update the current minimal solution
   add-front-edges(e):
      add the edges e to the set of front edges
   get-front-edge:
      return a front edge whose conditions holds in the
      current minimal solution
   more-front-edges:
      decides if there are more front edges whose conditions hold
end Solver
```

Now, the minimal solution is computed as follows.

```
Solver.add-constraints(Graph.local-constraints(Graph.main-node))
Solver.add-front-edges(Graph.outgoing-edges(Graph.main-node))
while Solver.more-front-edges do
   e := Solver.get-front-edge
   n := Graph.destination(e)
   Solver.add-constraints(Graph.connecting-constraints(e))
   if not Graph.seen-before(n) then
      Solver.add-constraints(Graph.local-constraints(n))
      Solver.add-front-edge(Graph.outgoing-edges(n))
   end
end
```

This algorithm can easily by modifed to implicitly handle polyvariance. The only change is that seen-before will be more restrictive, and that connecting-constraints and local-constraints must rename variables appropriately.

We now sketch implementations of the data structures that yield a time complexity of $O(n^3)$ for the basic analysis. The Graph has $O(n)$ nodes and $O(n^2)$ edges. Thus, $O(n^3)$ time is sufficient to allow a straightforward implementation of the operations.

The interesting aspects relate to the Solver. It is implemented as a dag, where we have a map from constraint variables to nodes. Each node has an associated size $O(n)$ bitvector which represents the set of tokens that all its corresponding variables are assigned in the current minimal solution. The node for a variable can be found in $O(1)$ time. Edges reflect inclusions between variables. The conditions of front edges depend on a single variable and a token; thus, we attach each front edge to the corresponding position in the bitvector in a dag node. Initially, there is a distinct node for each of the $O(n)$ variables, the empty bitvector in each node, no front edges, and no dag edges.

The operation add-constraints does the following for each constraint. If it is of the form $\gamma_1 \subseteq \gamma_2$, then the corresponding dag edge is added. If this forms a cycle, then all the nodes on that cycle are merged. This implies that their bitvectors are unioned together, their lists of front edges are appended, and the map from variables to nodes is updated correspondingly. If the constraint is of the form $c \subseteq \gamma$, then c is unioned to the bitvector in the node for γ. In any case, the dag may now be inconsistent. We must reestablish the inclusion relationships. Along each dag edge, we maintain $O(n)$ imagined "bit-wires", connecting the individual bits in pairs. There can at most be $O(n^3)$ bit-wires in total. When a bit is newly set, we traverse its $O(n)$ outgoing bit-wires, and set the bits they lead to. In this way, set bits are correctly propagated.

We manage front edges as follows. Those with unsatisfied conditions are distributed onto individual positions in bitvectors. Those with satisfied conditions are maintained in a list, on which more-front-edges and get-front-edge operate. When constraints are added, some bits may become set. When this happens for a particular position in a bitvector in some node, then the corresponding list of front edges is appended to the list of front edges with satisfied conditions. The operation add-front-edge first checks if the corresponding bit is set. If so, the edge is appended to the list of front edges with satisfied conditions. If not, the edge is appended to the list of front edges for that bit.

To see that we achive $O(n^3)$ time for the total of Solver operations, we need an amortized argument. A total of $O(n^2)$ constraints are added. However, merging dag nodes can be done at most $O(n)$ times. Each merger involves $O(n)$ nodes, and the union of their bitsets is computed in time $O(n^2)$; the total is time $O(n^3)$. New dag edges are inserted $O(n^2)$ times, once for each

$\gamma_1 \subseteq \gamma_2$ constraint. Each time the union of two bitvectors is computed in time $O(n)$; the total is time $O(n^3)$. Constant sets are included $O(n^2)$ times, once for each $c \subseteq \gamma$ constraint. Each time the union of two bitvectors is computed in time $O(n)$; the total is time $O(n^3)$. Each bit-wire is traversed at most once; the total time is $O(n^3)$. Each front-edge is moved at most twice; the total time is $O(n^2)$. All in all, the total time is $O(n^3)$.

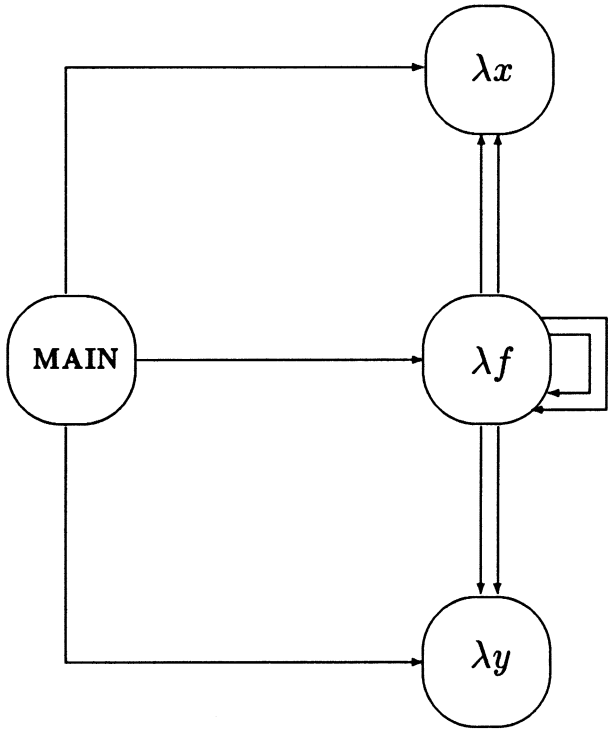
C The Monovariant Example

This appendix shows the trace graph and the global constraints derived from an expression considered in section 8. We only use the closure and safety constraints.

Some abbreviations:

$$
\begin{aligned}
&\textsc{lambda} = \{\lambda f, \lambda x, \lambda y\} \\
&w_0 = \text{the entire term from section 8} = w_1 \alpha_0 w_3 \\
&w_1 = [\alpha_0 : \lambda f.w_2] \\
&w_2 = \mathsf{If}\ \bar{E}\ \mathsf{then}\ f\alpha_0\ \mathsf{true}\ \mathsf{else}\ (f\alpha_0\ \mathsf{0}) \\
&w_3 = \mathsf{If}\ \bar{E}'\ \mathsf{then}\ [\alpha_0 : \lambda x.x]\ \mathsf{else}\ [\alpha_0 : \lambda y.\mathsf{0}]
\end{aligned}
$$

The trace graph has the following structure:



The global constraints are presented below. We indicate in the leftmost column where the inclusions stem from.

$$
\begin{array}{rl}
\text{local (MAIN)} & \left[\begin{array}{l}
[\![w_1]\!] \subseteq \text{LAMBDA} \\
[\![w_2]\!] \supseteq \{\lambda f\} \\
[\![w_3]\!] \supseteq [\![[\alpha_0 : \lambda x.x]]\!] \cup [\![[\alpha_0 : \lambda y.0]]\!] \\
[\![\bar{E}']\!] = \{\mathsf{Bool}\} \\
[\![[\alpha_0 : \lambda x.x]]\!] \supseteq \{\lambda x\} \\
[\![[\alpha_0 : \lambda y.0]]\!] \supseteq \{\lambda y\}
\end{array}\right. \\
\text{connecting (MAIN to } \lambda f) & \left[\begin{array}{l}
\lambda f \in [\![w_1]\!] \Rightarrow \left\{\begin{array}{l}
[\![w_3]\!] \subseteq [\![f]\!] \\
[\![w_0]\!] \supseteq [\![w_2]\!]
\end{array}\right.
\end{array}\right. \\
\text{connecting (MAIN to } \lambda x) & \left[\begin{array}{l}
\lambda x \in [\![w_1]\!] \Rightarrow \left\{\begin{array}{l}
[\![w_3]\!] \subseteq [\![x]\!] \\
[\![w_0]\!] \supseteq [\![x]\!]
\end{array}\right.
\end{array}\right. \\
\text{connecting (MAIN to } \lambda y) & \left[\begin{array}{l}
\lambda y \in [\![w_1]\!] \Rightarrow \left[\begin{array}{l}
[\![w_3]\!] \subseteq [\![y]\!] \\
[\![w_0]\!] \supseteq [\![0]\!]
\end{array}\right.
\end{array}\right. \\
\text{local } (\lambda f) & \left[\begin{array}{l}
\lambda f \in [\![w_1]\!] \Rightarrow \left\{\begin{array}{l}
[\![w_2]\!] \supseteq [\![f\ \alpha_0\ \mathsf{true}]\!] \cup [\![\mathsf{succ}(f\ \alpha_0\ 0)]\!] \\
[\![\bar{E}]\!] = \{\mathsf{Bool}\} \\
[\![\mathsf{true}]\!] = \{\mathsf{Bool}\} \\
[\![0]\!]\{\mathsf{Int}\} \\
[\![f\ \alpha_0\ 0]\!] = \{\mathsf{Int}\} \\
[\![\mathsf{succ}(f\ \alpha_0\ 0]\!] = \{\mathsf{Int}\} \\
[\![f]\!] \subseteq_{LAMBDA}
\end{array}\right.
\end{array}\right. \\
\text{connecting } (\lambda f \text{ to } \lambda f) & \left[\begin{array}{l}
\lambda f \in [\![w_1]\!] \wedge \lambda f \in [\![f]\!] \Rightarrow \cdots \\
\lambda f \in [\![w_1]\!] \wedge \lambda f \in [\![f]\!] \Rightarrow \cdots
\end{array}\right. \\
\text{connecting } (\lambda f \text{ to } \lambda x) & \left[\lambda f \in [\![w_1]\!] \wedge \lambda x \in [\![f]\!] \Rightarrow \left\{\begin{array}{l}
[\![\mathsf{true}]\!] \subseteq [\![x]\!] \\
[\![f\ \alpha_0\ \mathsf{true}]\!] \supseteq [\![x]\!]
\end{array}\right.\right. \\
\text{connecting } (\lambda f \text{ to } \lambda y) & \left[\lambda f \in [\![w_1]\!] \wedge \lambda y \in [\![f]\!] \Rightarrow \left\{\begin{array}{l}
[\![\mathsf{true}]\!] \subseteq [\![y]\!] \\
[\![f\ \alpha_0\ \mathsf{true}]\!] \supseteq [\![0]\!]
\end{array}\right.\right. \\
\text{connecting } (\lambda f \text{ to } \lambda x) & \left[\lambda f \in [\![w_1]\!] \wedge \lambda x \in [\![f]\!] \Rightarrow \left\{\begin{array}{l}
[\![0]\!] \subseteq [\![x]\!] \\
[\![\mathsf{succ}(f\ \alpha_0\ 0)]\!] \supseteq [\![x]\!]
\end{array}\right.\right. \\
\text{connecting } (\lambda f \text{ to } \lambda y) & \left[\lambda f \in [\![w_1]\!] \wedge \lambda y \in [\![f]\!] \Rightarrow \left\{\begin{array}{l}
[\![0]\!] \subseteq [\![y]\!] \\
[\![\mathsf{succ}(f\ \alpha_0\ 0]\!] \supseteq [\![0]\!]
\end{array}\right.\right.
\end{array}
$$

Note that $\lambda f \in \{\lambda f\} \subseteq [\![w_1]\!]$. Using this information to enable conditions, we further note that $\lambda x \in [\![[\alpha_0 : \lambda x.x]]\!] \subseteq [\![w_3]\!] \subseteq [\![f]\!]$. Finally, we conclude that the constraints are *not* satisfiable, since this would require:

$$\{\mathsf{Int}\} = [\![\mathsf{succ}(f\ \alpha_0\ 0)]\!] \supseteq [\![x]\!] \supseteq [\![\mathsf{true}]\!] = \{\mathsf{Bool}\}$$

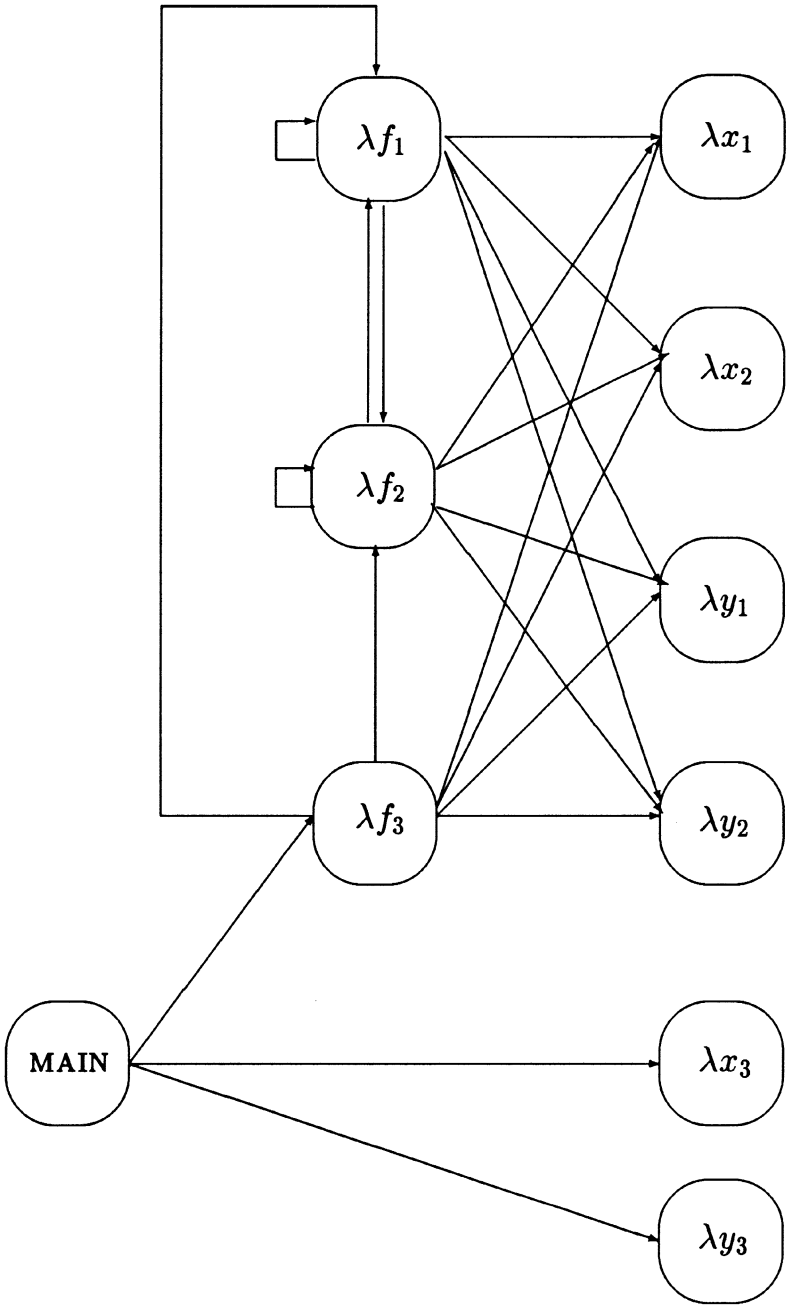
D The Polyvariant Example

This appendix shows the trace graph, global constraints, and minimal solution derived from an expression considered in section 8. The expression is obtained by applying the polyvariant transformation $\mathcal{P}$ to the expression considered in appendix B. As in appendix B, we only use the closure and safety constraints.

Some abbreviations:

$$
\begin{aligned}
\textsc{lambda} &= \{\lambda f_1, \lambda f_2, \lambda f_3, \lambda x_1, \lambda x_2, \lambda x_3, \lambda y_1, \lambda y_2, \lambda y_3\} \\
w_0 &= \text{the entire term from section 8} = w_1 \gamma w_3 \\
w_1 &= \left[\begin{array}{ll} \alpha: & \lambda f_1.\ \mathsf{If}\ \bar{E}\ \mathsf{then}\ f_1\ \alpha\ \mathsf{true}\ \mathsf{else}\ \mathsf{succ}\ (f_1\ \beta\ \mathsf{0}) \\ \beta: & \lambda f_2.\ \mathsf{If}\ \bar{E}\ \mathsf{then}\ f_2\ \alpha\ \mathsf{true}\ \mathsf{else}\ \mathsf{succ}\ (f_2\ \beta\ \mathsf{0}) \\ \gamma: & \lambda f_3.w_2 \end{array}\right] \\
w_2 &= \mathsf{if}\ \bar{E}\ \mathsf{then}\ f_3\ \alpha\ \mathsf{true}\ \mathsf{else}\ \mathsf{succ}\ (f_3\ \beta\ \mathsf{0}) \\
w_3 &= \mathsf{if}\ \bar{E}'\ \mathsf{then}\ w_4\ \mathsf{else}\ w_5 \\
w_4 &= \left[\begin{array}{ll} \alpha: & \lambda x_1.x_1 \\ \beta: & \lambda x_2.x_2 \\ \gamma: & \lambda x_3.x_3 \end{array}\right] \\
w_5 &= \left[\begin{array}{ll} \alpha: & \lambda y_1.\mathsf{0} \\ \beta: & \lambda y_2.\mathsf{0} \\ \gamma: & \lambda y_3.\mathsf{0} \end{array}\right]
\end{aligned}
$$

The trace graph has the following structure:


λf_1
λf_2
λf_3
λx_1
λx_2
λy_1
λy_2
MAIN
λx_3
λy_3

The global constraints are presented below. We indicate in the leftmost column where the inclusions stem from.

$$
\begin{array}{rl}
\text{local (MAIN)} & \left[\begin{array}{l}
[\![w_1]\!] \subseteq \text{LAMBDA} \\
[\![w_2]\!] \supseteq \{\lambda f_1, \lambda f_2, \lambda f_3\} \\
[\![w_3]\!] \supseteq [\![w_4]\!] \cup [\![w_5]\!] \\
[\![\bar{E}']\!] = \{\mathsf{Bool}\} \\
[\![w_4]\!] \supseteq \{\lambda x_1, \lambda x_2, \lambda x_3\} \\
[\![w_5]\!] \supseteq \{\lambda y_1, \lambda y_2, \lambda y_3\}
\end{array}\right. \\
\text{connecting (MAIN to } \lambda f_3) & \left[\begin{array}{l}
\lambda f_3 \in [\![w_1]\!] \Rightarrow \left\{\begin{array}{l} [\![w_3]\!] \subseteq [\![f_3]\!] \\ [\![w_0]\!] \supseteq [\![w_2]\!] \end{array}\right.
\end{array}\right. \\
\text{connecting (MAIN to } \lambda x_3) & \left[\begin{array}{l}
\lambda x_3 \in [\![w_1]\!] \Rightarrow \left\{\begin{array}{l} [\![w_3]\!] \subseteq [\![x_3]\!] \\ [\![w_0]\!] \supseteq [\![x_3]\!] \end{array}\right.
\end{array}\right. \\
\text{connecting (MAIN to } \lambda y_3) & \left[\begin{array}{l}
\lambda y_3 \in [\![w_1]\!] \Rightarrow \left\{\begin{array}{l} [\![w_3]\!] \subseteq [\![y_3]\!] \\ [\![w_0]\!] \supseteq [\![\mathsf{0}]\!] \end{array}\right.
\end{array}\right. \\
\text{local } (\lambda f) & \left[\begin{array}{l}
\lambda f_3 \in [\![w_1]\!] \Rightarrow \left\{\begin{array}{l}
[\![w_2]\!] \subseteq [\![f_3\ \alpha\ \mathsf{true}]\!] \cup [\![\mathsf{succ}(f_3\ \beta\ \mathsf{0})]\!] \\
[\![\bar{E}]\!] = \{\mathsf{Bool}\} \\
[\![\mathsf{true}]\!] = \{\mathsf{Bool}\} \\
[\![\mathsf{0}]\!] = \{\mathsf{Int}\} \\
[\![f_3\ \beta\ \mathsf{0}]\!] = \{\mathsf{Int}\} \\
[\![\mathsf{succ}(f_3\ \beta\ \mathsf{0}]\!] = \{\mathsf{Int}\} \\
[\![f_3]\!] \subseteq \text{LAMBDA}
\end{array}\right.
\end{array}\right. \\
\text{connecting } (\lambda f_3 \text{ to } \lambda f_1) & \left[\begin{array}{l} \lambda f_3 \in [\![w_1]\!] \wedge \lambda f_1 \in [\![f_3]\!] \Rightarrow \cdots \end{array}\right. \\
\text{connecting } (\lambda f_3 \text{ to } \lambda f_2) & \left[\begin{array}{l} \lambda f_3 \in [\![w_1]\!] \wedge \lambda f_2 \in [\![f_3]\!] \Rightarrow \cdots \end{array}\right. \\
\text{connecting } (\lambda f_3 \text{ to } \lambda x_1) & \left[\begin{array}{l}
\lambda f_3 \in [\![w_1]\!] \wedge \lambda x_1 \in [\![f_3]\!] \Rightarrow \left\{\begin{array}{l} [\![\mathsf{true}]\!] \subseteq [\![x_1]\!] \\ [\![f_3\ \alpha\ \mathsf{true}]\!] \supseteq [\![x_1]\!] \end{array}\right.
\end{array}\right. \\
\text{connecting } (\lambda f_3 \text{ to } \lambda y_1) & \left[\begin{array}{l}
\lambda f_3 \in [\![w_1]\!] \wedge \lambda y_1 \in [\![f_3]\!] \Rightarrow \left\{\begin{array}{l} [\![\mathsf{true}]\!] \subseteq [\![y_1]\!] \\ [\![f_3\ \alpha\ \mathsf{true}]\!] \supseteq [\![\mathsf{0}]\!] \end{array}\right.
\end{array}\right. \\
\text{connecting } (\lambda f_3 \text{ to } \lambda x_2) & \left[\begin{array}{l}
\lambda f_3 \in [\![w_1]\!] \wedge \lambda x_2 \in [\![f_3]\!] \Rightarrow \left\{\begin{array}{l} [\![\mathsf{0}]\!] \subseteq [\![x_2]\!] \\ [\![\mathsf{succ}(f_3\ \beta\ \mathsf{0})]\!] \supseteq [\![x_2]\!] \end{array}\right.
\end{array}\right. \\
\text{connecting } (\lambda f_3 \text{ to } \lambda y_2) & \left[\begin{array}{l}
\lambda f_3 \in [\![w_1]\!] \wedge \lambda y_2 \in [\![f_3]\!] \Rightarrow \left\{\begin{array}{l} [\![\mathsf{0}]\!] \subseteq [\![y_2]\!] \\ [\![\mathsf{succ}(f_3\ \beta\ \mathsf{0}]\!] \supseteq [\![\mathsf{0}]\!] \end{array}\right.
\end{array}\right.
\end{array}
$$

Excerpts from the minimal solution:

$$
\begin{array}{l}
[\![w_0]\!] = [\![w_2]\!] = [\![f_3\ \alpha\ \mathsf{true}]\!] = \{\mathsf{Bool}, \mathsf{Int}\} \\
[\![w_1]\!] = \{\lambda f_1, \lambda f_2, \lambda f_3\} \\
[\![w_3]\!] = [\![f_3]\!] = \{\lambda x_1, \lambda x_2, \lambda x_3, \lambda y_1, \lambda y_2, \lambda y_3\} \\
[\![w_4]\!] = \{\lambda x_1, \lambda x_2, \lambda x_3\} \\
[\![w_5]\!] = \{\lambda y_1, \lambda y_2, \lambda y_3\} \\
[\![x_1]\!] = [\![y_1]\!] = \{\mathsf{Bool}\} \\
[\![x_2]\!] = [\![y_2]\!] = [\![f_3\ \beta\ \mathsf{0}]\!] = [\![\mathsf{succ}(f_3\ \beta\ \mathsf{true})]\!] = \{\mathsf{Int}\}
\end{array}
$$

References

[1] Harald Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] Barendregt and Hemerik. Types in lambda calculi and programing languages. In *Proc. ESOP'90, European Symposium on Programming.* Springer-Verlag (*LNCS* 432), 1990.

[3] H. P. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48:931–940, 1983.

[4] Anders Bondorf. Automatic autoprojection of higher order recursive equations. In *Proc. ESOP'90, European Symposium on Programming.* Springer-Verlag (*LNCS* 432), 1990.

[5] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[6] M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484,1984.

[7] Charles Consel. New insights into partial evaluation: the SCHISM experiment. In *Proc. ESOP'88, European Symposium on Programming.* Springer-Verlag (*LNCS* 300), 1988.

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

[9] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'order superieur.* PhD thesis, University of Paris VII, 1972.

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

[11] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69,1991.

[12] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Proc. Rewriting Techniques and Applications*, pages 225–282. Springer-Verlag (*LNCS* 202), 1985. .

[13] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: a self-applicable partial evaluator for experiements in compiler generation. *Journal of LISP and Symbolic Computation*, 2:9–50, 1989.

[14] H. G. Mairson. Decidability of ML typing is complete for deterministic exponential time. In *Seventeenth Sympsium on Principles of Programming Languages*, pages 382–401. ACM Press, January 1990.

[15] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 1978.

[16] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[17] Torben Æ. Mogensen. Binding time analysis for polymorphically typed higher order languages. In *Proc. TAPSOFT'89*. Springer-Verlag (*LNCS* 352), March 1989.

[18] Flemming Nielson and Hanne Riis Nielson. Two-level functional languages. Draft book. To be published by Cambridge University Press, 1991.

[19] Hanne R. Nielson and Flemming Nielson. Automatic binding time analysis for a typed λ-calculus. *Science of Computer Programming*, 10:139–176,1988.

[20] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proc. OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, 1991.

[21] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. Computer Science Department, Aarhus University. Submitted fur publication, 1992.

[22] John C. Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation.* Springer-Verlag (*LNCS* 19), 1974.

[23] Michael I. Schwartzbach. Type inference with inequalities. In *Proc. TAPSOFT'91.* Springer-Verlag (*LNCS* 493), 1991.

[24] Peter Sestoft. Replacing function parameters by global variables. In *Proc. Conference on Functional Programming languages and Computes Architecture*, pages 39–53, 1989.

[25] David A. Turner. Miranda: A non-strict functions language with polymorphic types. In *Proc. Conference on Funtional Programming Languages and Computer Architecture*, pages 1–16. Springer-Verlag(*LNCS* 201), 1985.

[26] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamentae Informaticae*, X:115–122, 1987.