Selective and Lightweight Closure Conversion

Mitchell Wand and Paul Steckler*

College of Computer Science
Northeastern University
360 Huntington Avenue, 161CN
Boston, MA 02115, USA
{wand,steck}@ccs.neu.edu

Abstract

We consider the problem of selective and lightweight closure conversion, in which multiple procedure-calling protocols may coexist in the same code. Flow analysis is used to match the protocol expected by each procedure and the protocol used at each of its possible call sites. We formulate the flow analysis as the solution of a set of constraints, and show that any solution to the constraints justifies the resulting transformation. Some of the techniques used are suggested by those of abstract interpretation, but others arise out of alternative approaches.

1 Introduction

Modern compilers perform a variety of program analyses in order to produce good code. The goal of the analyses is to annotate a program with certain propositions about the behavior of that program. One can then apply optimizations to the program that are justified by those propositions.

For imperative languages, the justification of such optimizations or transformations has been investigated for many years and is reasonably well-understood. For higher-order languages such as Scheme, ML, or Haskell, however, it has proven remarkably difficult to specify the semantics of these propositions in a way that justifies the resulting transformations.

In this paper, we study one such transformation, called *closure conversion*. In a higher-order language, a procedure is typically represented by a record consisting of a piece of pure code (a λ-expression with no free variables) and the values of the free variables of the original procedure. This data structure is called a *closure*. In closure conversion, these data structures are built at the source-language level. Procedure creation is replaced by closure creation, and procedure application is replaced by invocation of the first component of the closure on the actual parameter and the closure itself. This is sometimes called *closure-passing style* [5].

For example, we might convert

$$\begin{aligned}&\textbf{let}\, f = \lambda xy.\lambda z.(x-y-z)\\ &\quad g = \lambda h.h3\\ &\textbf{in}\; g(fuv)\end{aligned}$$

to

$$\begin{aligned}&\textbf{let}\, f = \lambda xy.[(\lambda ez.\,\textbf{destr}\; e(\lambda xy.(x-y-z))),[x,y]]\\ &\quad g = \lambda h.\,\textbf{app}\; h\,3\\ &\textbf{in}\; g(fuv)\end{aligned}$$

where **app** is defined by

$$\textbf{app} = \lambda r.(\text{fst } r)(\text{snd } r)$$

Here f takes two arguments and emits a procedure of one argument; in the closure-converted version, the procedure is represented by a record. g takes a procedure represented as a closure record and applies it by taking the piece of code and applying it to the record of free variables and the actual parameter. **destr** destructures the record of free variables and binds the pieces to x and y, thus recreating the desired environment for the procedure body $(x-y-z)$.

Closure conversion is a crucial step in the compilation process for higher-order languages. The code parts of closures are closed terms, so they can be moved to the top level of the program if desired. Furthermore, this conversion is a source-to-source transformation, so the representation of closures can be optimized using all the tools available to the compiler.

In general, a closure-converted program is not equivalent to the original program under β-conversion or under any reasonable λ-theory, so it is a difficult step to justify. In this paper, we prove the correctness of a closure conversion algorithm that is generalized in two ways:

1. Closure conversion is not performed everywhere, so call sites obeying the closure record application protocol coexist with sites which use ordinary λ-application. We call this feature *selective* closure conversion.

   For selective conversion, the analysis must distinguish data flows consisting of procedures converted to closures from those which the translation will leave unchanged. Our example already showed the utility of this feature: the value returned by f, which was a procedure that escaped the scope of its free variables, was represented as a closure, but the other procedures were not.

*Work supported by the National Science Foundation and DARPA under grants CCR-9002253 and CCR-9014603

POPL 94- 1/94, Portland Oregon,USA
© 1994 ACM 0-89791-636-0/94/001..$3.50

In general, one might have many different procedure-calling protocols coexisting in the same program. For example, procedures coming from some library might have a different protocol from those defined in the client program, and the compiler must keep these separate. Another example is an interpreter based on denotational semantics, in which one might have radically different representations for environments, procedures, and continuations; one needs to replace each application by `apply-env`, `apply-proc` or `apply-cont`.

2. In many cases, a closure need not include all the free variables of a procedure, because some of the free variables are available at all the places where the closure might be invoked. We call such variables *dynamic*, because they behave the same as if the language used dynamic scope, and we call the incomplete closures *lightweight closures*. In the extreme, the closure might be able to find all its free variables at the call site, so no closure need be created. Avoiding closure creation plays an important part in creating good code for higher-order languages [4]. Our results give a method for proving the correctness of conversion algorithms which in appropriate cases avoid building closures.

These results are significant in several ways:

- First, we believe the analysis may be turn out to be a useful one for real languages.
- Second, it provides a example of the formal justification of a fairly complex analysis-based program transformation.
- We believe the techniques and approach we have used will be useful for other analyses and transformations.
- Last, the techniques are quite different from those of conventional abstract interpretation. We discuss this further in Section 8.

The outline of the paper is as follows:

1. Section 2 presents the transformation and some examples in more detail. Section 3 presents the input and output languages to be used. Our input language is an untyped λ-calculus with constants, boolean values, and conditionals. Our output language adds records to the input language. Both input and output languages use call-by-value evaluation [12].
2. In order to distinguish distinct occurrences of identical terms and to eliminate explicit substitution, we use a representation of terms using *occurrences* (i.e., parse tree nodes) and environments. We prove that the occurence evaluator simulates the original (Section 4).
3. In Section 5 we give the syntax and semantics of a language of propositions for occurrences. The semantics allows us to judge whether an proposition is true at a particular value or environment.
4. We annotate a program by assigning every phrase in the program a set of input propositions and an output proposition. We give a set of conditions defining when a program's annotations are *locally consistent*. Every program has a minimal locally-consistent annotation, computable by a conventional iterative algorithm. We show the soundness of our annotation scheme by proving that any consistently-annotated program makes all its annotations true: if the inputs to a node satisfy the node's input proposition, then its outputs satisfy its output proposition. This is analogous to the soundness of ordinary Hoare logic. (Section 6).
5. In Section 7 we finally define the closure-conversion transformation from annotated source programs into the target language, and we show the correctness of the transformation. The correctness theorem says that if we have a consistently-annotated source program that evaluates to some value, then the transformed program evaluates to the transform of that value. In particular, if the source program evaluates to some constant, then the transformed program evaluates to the same constant.

We close with a comparison with related work and some directions for further work.

2 Some Examples

In closure conversion, a λ-calculus term $(\lambda x.M)$ is replaced by a closure record

$$r = [(\lambda ex.\,\mathbf{destr}\ e\ (\lambda u_1 \ldots u_n.M)), [u_1, \ldots, u_n]]$$

where $u_1, \ldots, u_n$ are the free variables of $(\lambda x.M)$, and **destr** is a record-destructuring operation (see Section 3). The record consists of a closed λ-term and a record of the free variables. If r is bound to such a closure, then r can be applied to x by writing $\mathbf{app}\,f\,x$, where **app** is defined by

$$\mathbf{app} = \lambda r.(\mathbf{fst}\ r)(\mathbf{snd}\ r)$$

An easy calculation shows that $\mathbf{app}\ r$ is convertible to $\lambda x.M$.

Another version of closure conversion defines **app** to be $\lambda r.(\mathbf{fst}\ r)r$. This is similar to the protocol used in [5]. With this protocol, the code body would become

$$(\lambda r v_1 \ldots v_m x.\,\mathbf{destr}\,(\mathbf{snd}\ r)(\lambda u_1 \ldots u_n.M'))$$

This protocol allows more freedom in converting λ-expressions; for example, a closure with no saved variables could be represented by

$$[(\lambda r v_1 \ldots v_m x.M')]$$

This protocol would have the added benefit that recursive procedures could be represented easily. All the results in this paper work for this protocol as well; the only thing that would need to be reproved is Theorem 3. Having this protocol coexist with the one in Figure 4 is, of course, another application of the analysis we have presented.

2.1 Selective Conversion

Our goal is to use flow analysis to distinguish procedures and call sites for translation to a language in which there are multiple procedure-calling protocols. Initially, let us consider the case where there are exactly two protocols: "ordinary"

λ-calculus application and closure applications, as in our introductory example. We assume that unknown procedures are λ-calculus functions.

Thus each application must use one of the two protocols, depending on whether closure records or ordinary functions are involved. Therefore, the transformation must assure that all the procedures that might be applied at a given call site agree on their application protocol. To do this, we must perform a flow analysis. Thus, in

$$\begin{array}{l}\textbf{let}\, f = \lambda x.\ldots \\ \quad g = \lambda x.\ldots \\ \textbf{in}\,\textbf{let}\ h = \lambda y.\,\textbf{if zero?}\ y\ \textbf{then}\ f\ \textbf{else}\ g \\ \quad \textbf{in}\ \lambda xy.(hy)x\end{array}$$

the λ-expressions for f, g, and h can be converted to closures, and both the applications in the last line of the program can be converted to applications of closures.

On the other hand, in

$$\begin{array}{l}\lambda g.\ \textbf{let}\, f = \lambda x.\ldots \\ \quad \textbf{in let}\ h = \lambda y.\,\textbf{if zero?}\ y\ \textbf{then}\ f\ \textbf{else}\ g \\ \quad\quad \textbf{in}\ \lambda xy.(hy)x\end{array}$$

the value of (hy) might be either f or the unknown function g. Therefore the application of (hy) to x must be an ordinary application, and f cannot be closure-converted.

2.2 Lightweight closures

If some of the free variables in a function are available at the call site, we may be able to omit those variables from the closure. To do so, we must be sure that the bindings for the omitted variables are the same at the call site as at the point of function definition. Consider:

$$\begin{array}{l}\lambda y.\textbf{let}\, f = \lambda x.\lambda z.(x - y - z) \\ \quad\quad g = \lambda h.h3 \\ \quad \textbf{in}\ g(fv)\end{array}$$

One might expect to create a closure record for the $(\lambda z.\ldots)$ procedure, as we did before, containing values for the variables x and y. But the only call site for this procedure (at $(h3)$) is within the scope of y, so we can omit y from the closure and make it an additional argument at the call site:

$$\begin{array}{l}\lambda y.\textbf{let}\, f = \lambda x.[(\lambda eyz.(x - y - z)), [x]] \\ \quad\quad g = \lambda h.\,\textbf{app}\ h\, y\, 3 \\ \quad \textbf{in}\ g(fv)\end{array}$$

We call a variable like y, which the procedure expects to pick up at the call site, a *dynamic* variable. We call a closure like this one, in which the dynamic variables are not included, a *lightweight* closure.

Now things are more complicated: all procedures at a call site must agree not only on whether to use ordinary application or closure application, but in the case of closure application they must also agree on which variables are dynamic, and on the order in which they should be supplied at the call site. So now we have exponentially many possible procedure-calling protocols to be managed.

The flow analysis must assure not only that the dynamic variables of each procedure are present at every call site for that procedure, but also that those variables have the same values they had at the procedure-creation site. A free variable of a procedure may be lexically available at a call site, but with a different binding. This can happen when a procedure escapes from its original bindings but is then passed in to a new invocation of the original lexical scope. For example, consider

$$\begin{array}{l}\textbf{let}\, g = (\lambda xy.\ \textbf{let}\, f = (\lambda z.zx) \\ \quad\quad\quad\quad\quad \textbf{in}\ yf) \\ \textbf{in}\ g\ c_1\ (g\ c_2\ (\lambda v.v))\end{array}$$

Variable x is in scope at the call site (yf). But there are two calls to g, with two invocations of the scope for x. For the left-hand call, x is bound to c_1; for the right-hand call, x is bound to c_2. During the left-hand call to g, y is bound to $(\lambda z.zx)$, which has escaped from the other invocation of x's scope. Therefore, the flow analysis must assure that x is not made a dynamic variable.

3 The Language

Our input language Λ_{in} is an untyped λ-calculus with constants, boolean values, and conditionals. Our output language Λ_{out} adds records to the input language. We use $[M_1', \ldots, M_n']$ to build a record with n components $(n \geq 0)$. If the value of M is a record with n components, then the term **destr** MN applies N to the n components. For example, we define

$$\begin{array}{l}\textbf{fst} = \lambda r.\,\textbf{destr}\ r\ (\lambda xy.x) \\ \textbf{snd} = \lambda r.\,\textbf{destr}\ r\ (\lambda xy.y)\end{array}$$

Note that the rule for **destr** includes the possibility that $n = 0$. It would be straightforward to extend the results of this paper to include records in the input language.

Both input and output languages use call-by-value evaluation. Following [12], we define a *value* to be a term that is a constant, variable, abstraction, or a record of values. We define a relation $\underset{t}{\Longrightarrow}$ between terms and values. The rules defining $\underset{t}{\Longrightarrow}$ are given in Figure 1. In addition to these rules, there are the usual rules for application of constants such as **succ**, **pred**, **zero?**, etc.

4 The Occurrence Evaluator

Our flow analysis needs to keep track of what procedures in the source program are applied where. Unfortunately, the call-by-value evaluator sketched above uses substitution, which throws away information about the source of a procedure: it builds terms that no longer contain information about how the term was obtained from the source program. To keep track of the source of each term, write an *occurrence evaluator* that represents a term as a substitution instance of a subterm of the original program.

To do this, we will need to refer to particular *occurrences* of terms, allowing identical subterms with differing annotations to be distinguished. To do this, we imagine that we are traversing the parse tree of a single program. An *occurrence* is then an address in the tree, given as a finite string over the alphabet

$$\{\textit{rator}, \textit{rand}, \textit{bv}, \textit{body}, \textit{test}, \textit{then}, \textit{else}\}$$

We can use these terms to navigate in the parse tree in the usual way: the empty string represents the root of the tree,

$$v \underset{t}{\Longrightarrow} v$$

$$c \underset{t}{\Longrightarrow} c$$

$$\lambda x.M \underset{t}{\Longrightarrow} \lambda x.M$$

$$\frac{M \underset{t}{\Longrightarrow} \lambda x.Q \quad N \underset{t}{\Longrightarrow} N' \quad Q[N'/x] \underset{t}{\Longrightarrow} V}{MN \underset{t}{\Longrightarrow} V}$$

$$\frac{M \underset{t}{\Longrightarrow} \textbf{true}}{\textbf{if } M \textbf{ then } N \textbf{ else } P \underset{t}{\Longrightarrow} N}$$

$$\frac{M \underset{t}{\Longrightarrow} \textbf{false}}{\textbf{if } M \textbf{ then } N \textbf{ else } P \underset{t}{\Longrightarrow} P}$$

$$\frac{\forall i, 1 \le i \le n, M_i \underset{t}{\Longrightarrow} V_i}{[M_1, \ldots, M_n] \underset{t}{\Longrightarrow} [V_1, \ldots, V_n]}$$

$$\frac{M \underset{t}{\Longrightarrow} [V_1, \ldots, V_n] \quad N \underset{t}{\Longrightarrow} \lambda x_1 \ldots x_n.Q \quad Q[V_1/x_1, \ldots, V_n/x_n] \underset{t}{\Longrightarrow} W}{\textbf{destr } M \; N \underset{t}{\Longrightarrow} W}$$

Figure 1: Rules for the term evaluator

i.body.rand represents the operand of the body of node i, if i is a node that denotes an abstraction whose body is an application, etc. We write $[\![i]\!]$ for the term represented by node i, and $App(i)$, $Abs(i)$ for the predicates that test whether node i is an application, abstraction, etc. We will similarly provide addresses for constants, and for terms that appear in the initial environment.

We will write $M\{\rho\}$ for the term obtained by performing a substitution ρ on term M.

We next define the data structures manipulated in the evaluator.

Definition 1 The following definitions are mutually referential.

1. An *occurrence closure* (i, ψ) is a pair consisting of an occurrence index and an occurrence environment.
2. An *occurrence environment* ψ is a finite map from variables to occurrence closures, where for each $x \in Dom(\psi)$, $\psi(x) = (i, \psi')$ implies $[\![i]\!]$ is a value in Λ_{tn}.

The meaning of occurrence closures is given by the map $\mathcal{U}[\![-]\!]$ defined recursively on this definition by:

$$\mathcal{U}[\![(i, \psi)]\!] = [\![i]\!]\{\mathcal{U}[\![-]\!] \circ \psi\}$$

This definition is well-founded since occurrence closures have finite depth by Definition 1; termination occurs when ψ is empty. We call $\mathcal{U}[\![-]\!]$ the *unwinding* function.

The occurrence evaluator uses occurrence closures to simulate substitutions. We define a relation $\underset{oc}{\Longrightarrow}$ between occurrence closures that represent terms and occurrence closures that represent values. The rules for the occurrence evaluator are given in Figure 2.

The correctness theorem for the occurrence evaluator says that the occurrence evaluator simulates the operation of the term evaluator on the unwound occurrence closures:

Theorem 1 (Simulation) *If*

$$(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$$

then

$$\mathcal{U}[\![(i, \psi)]\!] \underset{t}{\Longrightarrow} \mathcal{U}[\![(j, \psi')]\!]$$

Proof: By induction on the definition of $\underset{oc}{\Longrightarrow}$. ∎

5 The Language of Annotations

We want to annotate each occurrence with a proposition that describes its possible values. For our analysis, this description includes the following information:

1. A *flow*, which describes the set of procedures to which the given occurrence might evaluate,
2. A *protocol tag*, which describes the protocol used by all the procedures to which the occurrence might evaluate, and
3. An *invariance set*, which describes the set of variables whose values are left invariant by evaluation of the occurrence.

An invariance set is a set of variables; flows will be defined below. A protocol tag is an element of $\{id\} \cup \{cl_\sigma \mid \sigma \in Var^*\}$. If σ is the sequence $\langle v_1, \ldots, v_n \rangle$, we write $\lceil \sigma \rceil$ for the unordered set $\{v_1, \ldots, v_n\}$.

$$\frac{Var\,(i)}{(i,\psi) \underset{oc}{\Longrightarrow} \psi(\llbracket i \rrbracket)}$$

$$\frac{Const\,(i)}{(i,\psi) \underset{oc}{\Longrightarrow} (i,\emptyset)}$$

$$\frac{Abs\,(i)}{(i,\psi) \underset{oc}{\Longrightarrow} (i,\psi)}$$

$$\frac{\begin{array}{l} Cond\,(i) \\ (i.test,\psi) \underset{oc}{\Longrightarrow} (j,\psi') \\ \llbracket j \rrbracket = \mathbf{true} \\ (i.then,\psi) \underset{oc}{\Longrightarrow} (k,\psi'') \end{array}}{(i,\psi) \underset{oc}{\Longrightarrow} (k,\psi'')}$$

$$\frac{\begin{array}{l} Cond\,(i) \\ (i.test,\psi) \underset{oc}{\Longrightarrow} (j,\psi') \\ \llbracket j \rrbracket = \mathbf{false} \\ (i.else,\psi) \underset{oc}{\Longrightarrow} (k,\psi'') \end{array}}{(i,\psi) \underset{oc}{\Longrightarrow} (k,\psi'')}$$

$$\frac{\begin{array}{l} App\,(i) \\ (i.rator,\psi) \underset{oc}{\Longrightarrow} (j,\psi'), \quad Abs\,(j) \\ (i.rand,\psi) \underset{oc}{\Longrightarrow} (k,\psi'') \\ (j.body,\psi')[\llbracket j.bv \rrbracket \mapsto (k,\psi'')] \underset{oc}{\Longrightarrow} (m,\psi''') \end{array}}{(i,\psi) \underset{oc}{\Longrightarrow} (m,\psi''')}$$

Figure 2: Rules for the occurrence evaluator

Definition 2 The following definitions are mutually referential.

1. A *proposition* $\mathcal{P}$ is a triple consisting of a flow, a protocol tag, and a set of variables.
2. A *flow* ϕ is a finite set of abstract closures.
3. An *abstract closure* (i, A) is a pair consisting of the occurrence index of an abstraction and a proposition environment.
4. An *proposition environment* A is a finite map from variables to propositions.

Our goal is to annotate each occurrence i with an annotation environment A_i and a proposition $\mathcal{P}_i$ such that if this occurrence i is evaluated in some occurrence environment ψ, and the value $\psi(x)$ of each variable x free in $\llbracket i \rrbracket$ satisfies the assumption $A_i(x)$ that A_i makes about it, then the output satisfies the proposition $\mathcal{P}_i$. Thus the pair $(A_i, \mathcal{P}_i)$ is the functional analog to a Hoare-style partial correctness assertion.

To do this, of course, we have to explain what it means for an occurrence closure to satisfy a proposition. Define a *protocol assignment* to be a map from occurrences to protocol tags.

Definition 3 The following definitions are mutually recursive.

1. An occurrence environment ψ satisfies a set of annotation assumptions A under protocol assignment Π, as follows:
$$\psi \overset{\Pi}{\models} A$$
iff $\forall x \in Dom(A)$
   (a) $x \in Dom(\psi)$ and $\psi(x) \overset{\Pi}{\models} A(x)$, and
   (b) If $A(x) = (\phi, \pi, \theta)$, $\psi(x) = (i, \psi')$, and $y \in \theta$, then $y \in Dom(\psi) \cap Dom(\psi')$, and $\psi(y) = \psi'(y)$
2. An occurrence closure (i, ψ) satisfies a proposition (ϕ, π, θ) under a protocol assignment Π, as follows:
$$(i, \psi) \overset{\Pi}{\models} (\phi, \pi, \theta)$$
iff
   (a) $\llbracket i \rrbracket$ is a constant, variable, **true**, or **false**, or
   (b) $\Pi(i) = \pi$ and there exists A such that $(i, A) \in \phi$ and $\psi \overset{\Pi}{\models} A$.

For $\psi \overset{\Pi}{\models} A$, the first condition says that for any variable x, the occurrence closure $\psi(x)$ satisfies the associated proposition $A(x)$. The second condition says that if we look at the occurrence closure (i, ψ') that we get by looking up a variable x in ψ, then ψ and ψ' agree on the variables in the invariance set θ.

For $(i, \psi) \overset{\Pi}{\models} (\phi, \pi, \theta)$, the first condition tells us that the propositions only deal with the λ-terms that may arise as values. The second condition says that i must be a appear in an abstract closure listed in the flow ϕ, and furthermore that its accompanying environment must satisfy the proposition environment associated with i in the flow. This is roughly equivalent to saying that $\mathcal{U}[\![(i,\psi)]\!]$ is a substitution instance of $[\![i]\!]$ where the substitution satisfies A.

We can order the set of propositions by setting $(\phi,\pi,\theta) \leq (\phi',\pi',\theta')$ iff $\phi \subseteq \phi'$, $\pi = \pi'$, and $\theta' \subseteq \theta$. Then we have

Lemma 1

1. If $(i,\psi) \overset{\Pi}{\models} (\phi,\pi,\theta)$ and $(\phi,\pi,\theta) \leq (\phi',\pi',\theta')$, then $(i,\psi) \overset{\Pi}{\models} (\phi',\pi',\theta')$.
2. If for all x, $A(x) \leq A'(x)$, then $\psi \overset{\Pi}{\models} A$ implies $\psi \overset{\Pi}{\models} A'$.

Proof: Immediate. ∎

The analysis could be generalized to track scalar values or other flow properties; see [11]. In this case, the set of tags would be replaced by a partial order, and the condition $\Pi(i) = \pi$ in Definition 3.2(b) would be replaced by $\Pi(i) \sqsubseteq \pi$.

6 Consistent Annotations

An *annotation map* is a map Γ that associates with each occurrence i an proposition environment A_i and a proposition $\mathcal{P}_i = (\phi_i,\pi_i,\theta_i)$.

An annotation map is the functional analog of a Floyd-Hoare-style annotation of a flowchart or imperative program. As in the imperative case, the plan is to find some local conditions such that if the local conditions are satisfied at every node, then the annotations will all be true.

Our local conditions are shown in Figure 3. If an annotation map Γ satisfies these conditions at every node, we say Γ is *locally consistent.* Observe that those portions of the conditions not dealing with protocol tags and invariance sets are the same as those for ordinary closure analysis in the style of [13], transposed to our notation. The conditions on protocol tags and invariance sets are a bit more involved; the motivation behind all the conditions will be discussed further in the proof of Theorem 2.

Furthermore, we say that an annotation map Γ is *monovariant* iff for every flow ϕ considered in the annotations of Γ, if $(j,B) \in \phi$, then $B = A_j$ (recall that A_j is the proposition environment associated with occurrence j by Γ). More formally, we can say

Definition 4 (Monovariance)

1. *An annotation map Γ is monovariant iff for all occurrences i, the annotations $(A_i,\mathcal{P}_i)$ are Γ-monovariant.*
2. *A flow ϕ is Γ-monovariant iff for all $(j,B) \in \phi$, $B = A_j$ and B is Γ-monovariant.*
3. *A proposition (ϕ,π,θ) is Γ-monovariant iff ϕ is Γ-monovariant.*
4. *A proposition environment A is Γ-monovariant iff for all $x \in Dom(A)$, $A(x)$ is Γ-monovariant.*

As before, the empty proposition environment provides the base case for the definition.

In a monovariant annotation, flows can be represented as sets of occurrences, rather than sets of occurrence closures. Hence there are only finitely many monovariant annotations for any given program.

Most monovariant closure analyses put only occurrences, but not environments, in their flows. By including proposition environments in abstract closures, our version allows the semantics of propositions to be defined independently of a global annotation map.

We can use the conditions of Figure 3 as an iteration operator on monovariant annotations: Given a monovariant annotation, we can use the rules of Figure 3 to increase the $\mathcal{P}_i$ by adding new elements to the ϕ_i and shrinking the θ_i. (See [11] for additional discussion.) Notice that this iteration operator is the identity on the protocol tags.

We can obtain a monovariant, locally consistent annotation by initializing the algorithm with a $\leq$-minimal annotation (initializing all the ϕ_i to empty, all the π_i to id, and all the θ_i to the set of all variables in the program), and then applying the iteration operator until a fixed point is reached. Since all the protocol tags are id, the condition

$$\pi_{i.rator} = cl_\sigma \implies \lceil \sigma \rceil \subseteq \theta_{i.rator}$$

cannot fail. (One could equally well initialize all the protocol tags to $cl_{()}$; the iterative algorithm would compute the same flows and invariance sets). Since there are only finitely many monovariant annotations, and the algorithm monotonically improves each annotation according to the $\leq$ order, the algorithm always terminates, and yields a locally-consistent monovariant annotation.

One can then search for other protocol assignments that are consistent; we leave the algorithmics of this search as an open problem.

We can now state the first main theorem:

Theorem 2 (Soundness)
Let Γ be a locally consistent, monovariant annotation map. Let Π be the protocol assignment defined by $\Pi(i) = \pi_i$. Let i be any occurrence index, and let ψ be an occurrence environment such that $\psi \overset{\Pi}{\models} A_i$. If $(i,\psi) \underset{oc}{\Longrightarrow} (j,\psi')$, then

1. $(j,\psi') \overset{\Pi}{\models} (\phi_i,\pi_i,\theta_i)$
2. *$\forall x \in \theta_i$, $x \in Dom(\psi)$, $x \in Dom(\psi')$, and $\psi(x) = \psi'(x)$.*

Proof: (Sketch). The proof is by induction on the structure of the proof that $(i,\psi) \underset{oc}{\Longrightarrow} (j,\psi')$. Since the definition of $\underset{oc}{\Longrightarrow}$ is by cases on the form of $[\![i]\!]$, the proof follows this form as well. Here we sketch the proof to give some intuition behind the consistency rules; pieces of a more detailed proof is given in the appendix.

The various conditions on A_i embody the usual lexical scoping rules.

For an abstraction, we have $(i,\psi) \underset{oc}{\Longrightarrow} (i,\psi)$. We have $\{(i,A_i)\} \subseteq \phi_i$ by the local consistency of Γ, $\psi \overset{\Pi}{\models} A_i$ by assumption, and $\pi_i = \Pi(i)$ by construction of Π, so $(i,\psi) \overset{\Pi}{\models} (\phi_i,\pi_i,\theta_i)$. The second condition holds trivially.

$$Var\,(i) \Longrightarrow A_i([\![i]\!]) = \mathcal{P}_i$$

$$Const\,(i) \Longrightarrow \theta_i = \emptyset$$

$$Abs\,(i) \Longrightarrow \left\{ \begin{array}{l} A_{i.body} = A_i[[\![i.bv]\!] \mapsto \mathcal{P}_{i\;bv}] \\ \{(i, A_i)\} \subseteq \phi_i, \text{ and} \\ \theta_i \subseteq Dom(A_i) \end{array} \right.$$

$$App\,(i) \text{ and } Const\,(i.rator) \Longrightarrow A_{i.rand} = A_i$$

$$App\,(i) \text{ and } \neg\, Const\,(i.rator) \Longrightarrow \left\{ \begin{array}{l} A_{i.rator} = A_{i.rand} = A_i, \\ \theta_i \subseteq \theta_{i.rator}, \\ \pi_{i.rator} = cl_\sigma \Longrightarrow \lceil \sigma \rceil \subseteq \theta_{i.rator}, \text{ and} \\ \forall (j, B) \in \phi_{i.rator} \\ \quad \left\{ \begin{array}{l} Abs\,(j), \\ \mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}, \\ \mathcal{P}_{j.body} \leq \mathcal{P}_i, \\ \theta_{j.bv} \subseteq \theta_{i.rator}, \text{ and} \\ [\![j.bv]\!] \notin \theta_{j.bv} \cup \theta_i \end{array} \right. \end{array} \right.$$

$$Cond\,(i) \Longrightarrow \left\{ \begin{array}{l} A_{i\;test} = A_{i.then} = A_{i.else} = A_i, \\ \mathcal{P}_{i.then} \leq \mathcal{P}_i, \text{ and} \\ \mathcal{P}_{i.else} \leq \mathcal{P}_i, \end{array} \right.$$

Figure 3: Local Consistency Conditions for Annotations

Suppose (i, ψ) is an application, with the following proof for its evaluation:

$$\frac{\begin{array}{l} (i.rator, \psi) \underset{oc}{\Longrightarrow} (j, \psi'),\; Abs\,(j) \\ (i.rand, \psi) \underset{oc}{\Longrightarrow} (k, \psi'') \\ (j.body, \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]) \underset{oc}{\Longrightarrow} (m, \psi''') \end{array}}{(i, \psi) \underset{oc}{\Longrightarrow} (m, \psi''')}$$

The operator evaluates to (j, ψ'), so by the induction hypothesis (j, A_j) must be in $\phi_{i.rator}$. So any value of the operand $i.rand$ is a possible value of $j.bv$, and any value of the procedure body $j.body$ is a possible value of the application i. These observations give rise to the requirements that $\mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}$ and $\mathcal{P}_{j.body} \leq \mathcal{P}_i$.

What values in ψ are left invariant by this calculation? ψ'' is obtained from ψ in two steps: First the operator is evaluated, yielding a ψ' agreeing with ψ on the variables in $\theta_{i.rator}$. We then evaluate $j.body$ in an extension of ψ', yielding ψ'', which agrees with ψ' on $\theta_{j.body} - [\![j.bv]\!]$. Hence we must have $\theta_i \subseteq \theta_{i.rator} \cup \theta_{j.body} - [\![j.bv]\!]$. This explains the conditions $\theta_i \subseteq \theta_{i.rator}$, $\theta_i \subseteq \theta_{j.body}$, and $[\![j.bv]\!] \notin \theta_i$. The condition $[\![j.bv]\!] \notin \theta_{j.bv}$ is necessary so that $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \overset{\Pi}{\models} A_{j\;body}$ to support the induction. A more detailed exposition is given in the appendix.

∎

7 The Closure-Conversion Transformation

Once we have annotated the program, we are ready to perform closure conversion. In closure conversion, each abstraction term $(\lambda x.M)$ that is marked by the tag cl_σ is replaced by a closure record

$$[(\lambda e v_1 \ldots v_m x.\, \mathbf{destr}\; e(\lambda u_1 \ldots u_n.M')), [u_1, \ldots, u_n]]$$

where $\langle v_1 \ldots v_m \rangle = \sigma$, $\{u_1 \ldots u_n\} = FV(\lambda x.M) - \lceil \sigma \rceil$, and M' is the closure-conversion of M.

Each application MN whose operator is marked by the tag cl_σ is replaced by $\mathbf{app}\; M'\; v_1 \ldots v_m\, N'$, where $\langle v_1 \ldots v_m \rangle = \sigma$, $\mathbf{app} = \lambda r.(\mathbf{fst}\; r)(\mathbf{snd}\; r)$ and M' and N' are the converted versions of M and N. It is easy to see that if r is the record above, then $\mathbf{app}\; r\, v_1 \ldots v_m$ converts to $\lambda x.M'$. In this way the dynamic variables are passed at call-time. For $\sigma = \langle\rangle$, this is the same as ordinary closure conversion. Note that the exact details of r are unimportant, so long as $\mathbf{app}\; r\, v_1 \ldots v_m$ converts to $\lambda x.M'$; for example, we have slightly different code for the case where $n = 0$.

Of course, this description is not quite accurate, since the transformation needs to know the annotations, not just the terms. Therefore we formulate the transformation as a map from the set of occurrences to the language of output terms. The rules are given in Figure 4.

We extend the transformation to occurrence closures by recursively applying it in each occurrence environment:

$$\hat{\Phi}(i, \psi) = \Phi(i)\{\hat{\Phi} \circ \psi\}$$

Since occurrence environments are of finite depth, $\hat{\Phi}$ is well-founded.

Now we are finally in a position to formulate the correctness of the transformation. The correctness theorem says that evaluating a transformed program gives the same result as transforming the value of the original program. In particular, if the program evaluates to a constant, then the transformed program evaluates to the same constant.

We can illustrate this theorem by a commutative diagram:

$$Var\,(i) \Longrightarrow \Phi(i) = [\![i]\!]$$

$$Const\,(i) \Longrightarrow \Phi(i) = [\![i]\!]$$

$$Abs\,(i) \wedge \pi_i = id \Longrightarrow \Phi(i) = \lambda x.\Phi(i.body)$$

$$\begin{aligned} Abs\,(i) \wedge \pi_i = cl_\sigma \Longrightarrow \Phi(i) = \; & [(\lambda e\vec{v}x.\mathbf{destr}\ e\ (\lambda\vec{u}.\Phi(i.body))), [\vec{u}]] \\ & \text{where } e \text{ fresh} \\ & \text{and } \vec{v} = \sigma \\ & \text{and } \lceil\vec{u}\rceil = FV([\![i]\!]) - \lceil\sigma\rceil \end{aligned}$$

$$App\,(i) \;\wedge Const\,(i.rator) \Longrightarrow \Phi(i) = \Phi(i.rator)\ \Phi(i.rand)$$

$$App\,(i) \wedge \neg\, Const\,(i.rator) \wedge \pi_{i\ rator} = id \Longrightarrow \Phi(i) = \Phi(i.rator)\ \Phi(i.rand)$$

$$\begin{aligned} App\,(i) \wedge \neg\, Const\,(i.rator) \wedge \pi_{i\ rator} = cl_\sigma \Longrightarrow \; & \Phi(i) = \mathbf{app}\ \Phi(i.rator)\ v_1\ \ldots\ v_n\ \Phi(i.rand) \\ & \text{where } \vec{v} = \sigma \end{aligned}$$

$$Cond\,(i) \Longrightarrow \Phi(i) = \mathbf{if}\ \Phi(i.test)\ \mathbf{then}\ \Phi(i.then)\ \mathbf{else}\ \Phi(i.else)$$

Figure 4: Definition of the transform

$$\begin{tikzcd} (i,\psi) \arrow[r, Rightarrow, "oc"'] \arrow[d, "\hat{\Phi}"] & (j,\psi') \arrow[d, "\hat{\Phi}"] \\ \hat{\Phi}(i,\psi) \arrow[r, Rightarrow, "t"'] & \hat{\Phi}(j,\psi') \end{tikzcd}$$

Theorem 3 (Correctness) *Let* Γ *be a monovariant locally consistent annotation map, and* Π *the protocol assignment defined by* $\forall i, \Pi(i) = \pi_i$. *Let* ψ *be an occurrence environment such that* $\psi \overset{\Pi}{\models} A_i$. *If*

$$(i,\psi) \underset{oc}{\Longrightarrow} (j,\psi')$$

then

$$\hat{\Phi}(i,\psi) \underset{t}{\Longrightarrow} \hat{\Phi}(j,\psi')$$

Proof: (Sketch) The proof is by induction on the size of the derivation that $(i,\psi) \underset{oc}{\Longrightarrow} (j,\psi')$. The soundness theorem is used to guarantee that for each invocation of the induction hypothesis for an occurrence closure (k,ψ''), the needed condition $\psi'' \overset{\Pi}{\models} A_k$ is satisfied. ∎

Corollary 1 Let Γ be a monovariant locally consistent annotation map, and Π the protocol assignment defined by $\forall i$, $\Pi(i) = \pi_i$. Let ψ be an occurrence environment such that $\psi \overset{\Pi}{\models} A_i$. If $(i,\psi) \underset{oc}{\Longrightarrow} (j,\psi')$, where $[\![j]\!]$ is a constant c. Then $\hat{\Phi}(i,\psi) \underset{t}{\Longrightarrow} c$.

8 Comparison with Other Work

The immediate sources for our work are Sestoft [13] and Shivers [15]. Sestoft [13] presents an analysis called *closure analysis* which he uses to detect definition-use chains in a higher-order language. His closure analysis is equivalent to our use of flows. His soundness theorem [13, Lemma 5.1-1] appears to be equivalent to our Theorem 2 specialized to flows. He then uses this to prove the correctness of a transformation that replaces parameter-passing by assignment to global variables.

Shivers [15] presents a number of analyses, including one he calls 0CFA. 0CFA is equivalent to Sestoft's closure analysis. He proves a soundness theorem for this analysis for a rather complex language, including side-effects, but he does not prove the correctness of any transformations based on the analysis. Shivers' work is done in the context of programs in continuation-passing style. Our original development of the flow semantics in Sections 5–6 was motivated by an attempt to generalize and simplify Shivers' proof.

The idea of approximating an occurrence closure by an abstract closure comes from [9]; this is the straightforward extension of the idea of record types from [10] to the occurrence-closure evaluator.

In [16] we gave an alternative proof of a closure-conversion algorithm by semantic methods. However, that was a proof of a different property: it showed that if the converted term produced a constant under certain restrictions, then the original term produced the same constant. Here we show the converse: that if the original term succeeds, so does the converted term.

The paradigm of formulating a flow analysis as the solution of a set of equations is, of course, traditional in the realm of imperative languages. The idea of showing that any solution to the constraints yields a justified transformation was introduced in [17] in the context of binding-time analysis and off-line partial evaluation.

Appel [4] gives a useful catalog of transformations; many of these will provide good challenges for transformation-correctness proofs. Our transformation is superficially similar to Appel's callee-saves transformation, in that both pass extra variables to the procedure. However, in callee-saves, the procedure is responsible to passing those values unchanged to its continuation; we do not use continuation-passing style, and the variables are passed to the procedure for its own use.

Heintze [8] and Aitken *et al.* [2, 3] have modeled program analyses as solving constraint equations. Their work, however, has centered on the algorithmics of solving constraints, in forms more general than those considered here, rather than on the use of the solutions to justify program transformations.

Abadi *et al.* [1] use a similar representation of λ-terms as substitution instances, and prove a result (Lemma 3.5) similar to our Simulation Theorem (Theorem 1). However, there is nothing like our flow analysis or lightweight closures in their system.

It is enlightening to compare our approach to a more conventional abstract-interpretation approach. In abstract interpretation, one normally starts with a denotational semantics. One then defines a non-standard interpretation by adding a cache or similar device to keep track of the different values to which each program variable can be bound. Then one takes an abstraction of the cache interpretation. Because this analysis is more indirect, it is more difficult to relate the results of the abstract interpretation to the original semantics, and therefore to use the results to justify a transformation. Furthermore, because we deal directly with operational semantics, we avoid any domain-theoretic difficulties.

The great advantage of the abstract-interpretation approach is that the algorithm for the analysis comes for free, as a fixed point of the abstract-interpretation equations. Our approach factors the problem differently: we formulate the analysis as a constraint problem. We then show that any solution to the constraints yields a correct transformation. Our technique does not provide an automatic method for solving the constraints, though in many cases, such as the one here, the form of the equations leads to easy solution methods.

9 Future Research

We believe the techniques and approach we have used will be useful for other analyses and transformations. It would be useful to compare this analysis with the lifetime analyses [6, 7] for replacing heap-based closures by stack representations of procedures. As pointed out in [11, 14], closure analysis works for call-by-name evaluation also; extending our analysis to call-by-name languages should be straightforward. Another area for investigation is the integration of these techniques with the verified-compiler technology sketched in [18]. Last, it would be desirable to have a better understanding of the relationship between our approach and that of conventional abstract interpretation.

Acknowledgements

We would like to thank Bob Muller, Flemming Nielson, and John Hughes for useful discussions about closure analysis. Richard Kelsey provided useful insights into escape analysis and possible applications.

A Appendix: Proof of Theorem 2

Proof: Induction on structure of proof that $(i,\psi) \underset{oc}{\Longrightarrow} (j, \psi')$

We show only the cases where $[\![i]\!]$ is in the subset of Λ_{in} corresponding to terms of the pure λ-calculus. The remaining cases are straightforward.

case *Var*(i): Then $(i,\psi) \underset{oc}{\Longrightarrow} (j,\psi')$, where $(j,\psi') = \psi([\![i]\!])$. Since Γ is locally consistent, we have $A_\iota([\![i]\!]) = (\phi_i, \pi_\iota, \theta_\iota)$. Since $\psi \overset{\Pi}{\models} A_\iota$, we know that $\psi([\![i]\!]) \overset{\Pi}{\models} (\phi_\iota, \pi_\iota, \theta_\iota)$, so $(j,\psi') \overset{\Pi}{\models} (\phi_\iota, \pi_\iota, \theta_\iota)$. Also since $\psi \overset{\Pi}{\models} A_\iota$, $x \in \theta_\iota$ implies $x \in Dom(\psi) \cap Dom(\psi')$, and $\psi(x) = \psi'(x)$.

case *Abs*(i):

For abstractions, we have $(i,\psi) \underset{oc}{\Longrightarrow} (i,\psi)$. First we show that $(i,\psi) \overset{\Pi}{\models} (\phi_\iota, \pi_\iota, \theta_\iota)$. By the definition of $\overset{\Pi}{\models}$, this holds if $\Pi(i) = \pi_\iota$ (true by the construction of Π), and if there is some $(j, A) \in \phi_\iota$ such that $(i,\psi) \overset{\Pi}{\models} (j, A)$. Γ is locally consistent, so $\{(i, A_\iota)\} \subseteq \phi_\iota$, and by assumption, $\psi \overset{\Pi}{\models} A_\iota$. By the local consistency condition, we have $x \in \theta_\iota$ implies $x \in Dom(A_\iota)$. Since $\psi \overset{\Pi}{\models} A_\iota$, $x \in Dom(A_\iota)$ implies $x \in Dom(\psi)$, so $\forall x \in \theta_\iota$, $x \in Dom(\psi)$.

case *App*(i): Suppose $(i,\psi) \underset{oc}{\Longrightarrow} (m,\psi''')$, so we have:

$$\frac{\begin{array}{l}(i.rator, \psi) \underset{oc}{\Longrightarrow} (j,\psi'),\ Abs(j)\\ (i.rand, \psi) \underset{oc}{\Longrightarrow} (k,\psi'')\\ (j.body, \psi'[[\![j.bv]\!] \mapsto (k,\psi'')]) \underset{oc}{\Longrightarrow} (m,\psi''')\end{array}}{(i,\psi) \underset{oc}{\Longrightarrow} (m,\psi''')}$$

Then we deduce the desired result by calculating as follows:

(1) $\psi \overset{\Pi}{\models} A_{\iota.rator}, A_{\iota\ rand}$
  $[A_\iota = A_{\iota\ rator} = A_{\iota\ rand}]$

(2) $(i.rator, \psi) \underset{oc}{\Longrightarrow} (j,\psi')$
  [proof that $(i,\psi) \underset{oc}{\Longrightarrow} (m,\psi''')$]

(3) $(j,\psi') \overset{\Pi}{\models} (\phi_{\iota.rator}, \pi_{\iota.rator}, \theta_{\iota.rator})$
  [IH re soundness at *i.rator*]

(4) $y \in \theta_{\iota.rator} \Longrightarrow$
  $y \in Dom(\psi) \cap Dom(\psi')$,
  $\psi(y) = \psi'(y)$
  [IH re invariant at *i.rator*]

(5) $(j, A_j) \in \phi_{\iota.rator}$
  [defs of $\overset{\Pi}{\models}$, monovariance, (3)]

(6) $(j,\psi') \overset{\Pi}{\models} (j, A_j)$
  [by (5), def of $\overset{\Pi}{\models}$]

(7) $\psi' \overset{\Pi}{\models} A_j$
  [by (6), def of $\overset{\Pi}{\models}$]

(8) $(i.rand, \psi) \underset{oc}{\Longrightarrow} (k,\psi'')$
  [proof that $(i,\psi) \underset{oc}{\Longrightarrow} (m,\psi''')$]

(9) $(k, \psi'') \overset{\Pi}{\models} (\phi_{i.rand}, \pi_{i.rand}, \theta_{i.rand})$
[IH re soundness at *i.rand*]

(10) $y \in \theta_{i.rand} \Longrightarrow$
$y \in Dom(\psi) \cap Dom(\psi'')$,
$\psi(y) = \psi''(y)$
[IH re invariant at *i.rand*]

(11) $\phi_{i.rand} \subseteq \phi_{j.bv}$
[$\mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}$]

(12) $\phi_{j.body} \subseteq \phi_i$
[$\mathcal{P}_{j.body} \leq \mathcal{P}_i$]

(13) $\pi_{i.rand} = \pi_{j.bv}$
[$\mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}$]

(14) $\pi_{j.body} = \pi_i$
[$\mathcal{P}_{j.body} \leq \mathcal{P}_i$]

(15) $\theta_i \subseteq \theta_{i.rator}$
[invariance set constraints]

(16) $\theta_{j.bv} \subseteq \theta_{i.rand}$
[$\mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}$]

(17) $\theta_i \subseteq \theta_{j.body}$
[$\mathcal{P}_{j.body} \leq \mathcal{P}_i$]

(18) $\theta_{j.bv} \subseteq \theta_{i.rator}$
[invariance set constraints]

(19) $[\![j.bv]\!] \notin \theta_{j.bv}$
[invariance set constraints]

(20) $[\![j.bv]\!] \notin \theta_i$
[invariance set constraints]

(21) $y \in (\theta_{i.rator} \cap \theta_{i.rand}) \Longrightarrow$
$y \in Dom(\psi') \cap Dom(\psi'')$,
$\psi'(y) = \psi''(y)$
[by (4), (10)]

(22) $y \in \theta_{j.bv} \Longrightarrow$
$y \in Dom(\psi') \cap Dom(\psi'')$,
$\psi'(y) = \psi''(y)$
[by (18), (16)]

(23) $y \in \theta_{j.bv} \Longrightarrow$
$y \in Dom(\psi'[[\![j.bv]\!] \mapsto (k, \psi'')]) \cap Dom(\psi'')$,
$\psi'[[\![j.bv]\!] \mapsto (k, \psi'')](y) = \psi''(y)$
[by (19)]

(24) $(k, \psi'') \overset{\Pi}{\models} (\phi_{j.bv}, \pi_{j.bv}, \theta_{j.bv})$
[by (9), (11), (13)]

(25) $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \overset{\Pi}{\models} A_j[[\![j.bv]\!] \mapsto \mathcal{P}_{j.bv}]$
[by (7), (23), (24)]

(26) $A_{j.body} = A_j[[\![j.bv]\!] \mapsto \mathcal{P}_{j.bv}]$
[def of consistent annotation]

(27) $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \overset{\Pi}{\models} A_{j.body}$
[(25), (26)]

(28) $(m, \psi''') \overset{\Pi}{\models} (\phi_{j.body}, \pi_{j.body}, \theta_{j.body})$
[IH re soundness at *j.body*]

(29) $y \in \theta_{j.body} \Longrightarrow$
$y \in Dom(\psi'[[\![j.bv]\!] \mapsto (k, \psi'')]) \cap Dom(\psi''')$,
$\psi'[[\![j.bv]\!] \mapsto (k, \psi'')](y) = \psi'''(y)$
[IH re invariant at *j.body*]

(30) $(m, \psi''') \overset{\Pi}{\models} (\phi_i, \pi_i, \theta_i)$
[(12), (14), and (28)]

(31) $y \in (\theta_{j.body} - [\![j.bv]\!]) \Longrightarrow$
$y \in Dom(\psi') \cap Dom(\psi''')$,
$\psi'(y) = \psi'''(y)$
[by (29)]

(32) $y \in (\theta_{i.rator} \cap (\theta_{j.body} - [\![j.bv]\!])) \Longrightarrow$
$y \in Dom(\psi) \cap Dom(\psi''')$,
$\psi(y) = \psi'''(y)$
[by (4), (31)]

(33) $y \in \theta_i \Longrightarrow$
$y \in Dom(\psi) \cap Dom(\psi''')$,
$\psi(y) = \psi'''(y)$
[by (17), (15), (20), (32)]

References

[1] Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Levy. Explicit Substitutions. *Journal of Functional Programming*, 1(4):375–417, October 1991.

[2] Alexander Aiken and Brian R. Murphy. Static Type Inference in a Dynamically Typed Language. In *Conf. Rec. 18th ACM Symposium on Principles of Programming Languages*, pages 279–290, 1990.

[3] Alexander Aiken and Edward L. Wimmers. Solving Systems of Set Constraints (Extended Abstract). In *Proc. 7th IEEE Symposium on Logic in Computer Science*, pages 329–340, 1992.

[4] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, Cambridge, 1992.

[5] Andrew W. Appel and Trevor Jim. Continuation-Passing, Closure-Passing Style. In *Conf. Rec. 16th ACM Symposium on Principles of Programming Languages*, pages 293–302, 1989.

[6] Alain Deutsch. On Determining Lifetime and Aliasing of Dynamically Allocated Data in Higher-order Functional Specifications. In *Conf. Rec. 17th ACM Sympo-*

sium on Principles of Programming Languages, pages 157–168, 1990.

[7] Benjamin Goldberg and Young Gil Park. Higher Order Escape Analysis: Optimizing Stack Allocation in Functional Program Implementations. In *Proc. 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 152–160, Berlin, Heidelberg, and New York, 1990. Springer-Verlag.

[8] Nevin Heintze. *Set Based Program Analysis.* PhD thesis, Carnegie-Mellon University, October 1992.

[9] Neil D. Jones. Flow Analysis of Lambda Expressions. In *International Colloquium on Automata, Languages, and Programming*, 1981.

[10] Neil D. Jones and Steven S. Muchnick. A Flexible Approach to Interprocedural Data Flow Analysis and Progrms with Recursive Data Structures. In *Conf. Rec. 9th ACM Symposium on Principles of Programming Languages*, pages 66–74, 1982.

[11] Jens Palsberg and Michael I. Schwartzbach. Safety Analysis vs. Type Inference. *Information and Computation*, to appear.

[12] Gordon D. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[13] Peter Sestoft. Replacing Function Parameters by Global Variables. Master's thesis, DIKU, University of Copenhagen, October 1988.

[14] Peter Sestoft. *Analysis and Efficient Implementation of Functional Programs.* PhD thesis, DIKU, University of Copenhagen, October 1991.

[15] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie-Mellon University, May 1991.

[16] Mitchell Wand. Correctness of Procedure Representations in Higher-Order Assembly Language. In S. Brookes, editor, *Proceedings Mathematical Foundations of Programming Semantics '91*, volume 598 of *Lecture Notes in Computer Science*, pages 294–311. Springer-Verlag, Berlin, Heidelberg, and New York, 1992.

[17] Mitchell Wand. Specifying the Correctness of Binding-Time Analysis. In *Conf. Rec. 20th ACM Symposium on Principles of Programming Languages*, pages 137–143, 1993. To appear in *Journal of Functional Programming.*

[18] Mitchell Wand and Dino P. Oliva. Proving the Correctness of Storage Representations. In *Proc. 1992 ACM Symposium on Lisp and Functional Programming*, pages 151–160, 1992.