# Lightweight Closure Conversion

PAUL A. STECKLER
University of Technology, Sydney
and
MITCHELL WAND
Northeastern University

We consider the problem of lightweight closure conversion, in which multiple procedure call protocols may coexist in the same code. A lightweight closure omits bindings for some of the free variables of the procedure that it represents. Flow analysis is used to match the protocol expected by each procedure and the protocol used at its possible call sites. We formulate the flow analysis as a deductive system that generates a labeled transition system and a set of constraints. We show that any solution to the constraints justifies the resulting transformation. Some of the techniques used are similar to those of abstract interpretation, but others appear to be novel.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*procedures, functions, and subroutines*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions*; *pre- and postconditions*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*operational semantics*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Closure conversion, compiler construction, global optimization/flow analysis, program transformations

## 1. OPTIMIZING CODE TRANSFORMATIONS

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

For first-order languages, the justification of such optimizations or transforma-

This work was supported by the National Science Foundation and DARPA under grants CCR-9002253 and CCR-9014603. Preliminary presentations of this work appeared in *Conference Record of the 21st ACM Symposium on Principles of Programming Languages* and in the first-named author's Ph.D. dissertation. Both authors were affiliated with Northeastern University when this research was conducted.
Authors' addresses: P.A. Steckler, School of Computing Sciences, University of Technology, Sydney, P.O. Box 123, Broadway 2007 NSW, Australia; email: steck@socs.uts.edu.au; M. Wand, College of Computer Science, Cullinane Hall, Northeastern University, Boston, MA 02115; email: wand@ccs.neu.edu.

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

Here we study one such transformation, called *closure conversion*. In a lexically scoped higher-order language, a procedure is typically represented by a record consisting of a piece of pure code (a closed $\lambda$-abstraction) and the values of the free variables of the original procedure (an environment). This data structure is called a *closure*. In closure conversion, these data structures are built at the source-language level. During compilation, procedure creation is replaced by closure creation, and during execution, procedure application is replaced by invocation of the code part of the closure on an actual parameter and the environment part of the closure. Alternatively, another copy of the closure can be passed instead of just the environment. This strategy is sometimes called *closure-passing style* [Appel and Jim 1989].

For example, we might convert

$$
\begin{array}{l}
\mathbf{let}\ f = \lambda x.\lambda y.(x - y) \\
\quad\ \ g = \lambda h.h\ 3 \\
\mathbf{in}\ g\ (f\ 9)
\end{array}
$$

to

$$
\begin{array}{l}
\mathbf{let}\ f = [(\lambda e, x.[(\lambda e', y.\,\mathbf{let}\ x = (\#1\ e') \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ x - y), \\
\qquad\qquad\qquad\quad [x]]), \\
\qquad\qquad [\,]] \\
\quad\ \ g = [(\lambda e'', h.\,\mathbf{app}\ h\ 3), [\,]] \\
\mathbf{in}\ \mathbf{app}\ g\ (\mathbf{app}\ f\ 9)
\end{array}
$$

where **app** is a combinator, defined as $\lambda r.(\#1\ r)(\#2\ r)$, which applies the code part of a closure to its environment part. In this and subsequent examples, procedures are curried; the procedure in the code part of the closure for $f$ should be read as $\lambda e.\lambda x.\cdots$, for example. In examples, we use **let** as syntactic sugar for procedure application; we do not create closures for the implicit procedures.

Here $f$ takes an argument and emits a procedure of one argument; in the closure-converted version, both $f$ and the procedure it returns are represented by records. $g$ takes the closure record returned by $f$ and applies it by taking the piece of code and applying it to the record of free variables and the actual parameter. Both $f$ and $g$ are closed, so their closures have empty records of free variables.

Closure conversion is a crucial step in the compilation process for higher-order languages. Since the code parts of closures are closed, they may 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 the last example, the original program and its transform both evaluate to the same result. Closure conversion may produce a transform whose result is not convertible with the result of the original program. Let us closure-convert

$$
\begin{array}{l}
\textbf{let } z = 3 \\
\textbf{in } \textbf{let } f = \lambda x.\lambda y.x + y + z \\
\quad\ \textbf{in } f\ 4
\end{array}
$$

to get

$$
\begin{array}{l}
\textbf{let } z = 3 \\
\textbf{in } \textbf{let } f = [(\lambda e, x.\textbf{let } z = (\#1\ e) \\
\qquad\qquad\qquad\qquad \textbf{in } [(\lambda e', y.\textbf{let } x = (\#1\ e') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad z = (\#2\ e') \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } x + y + z), \\
\qquad\qquad\qquad\qquad\qquad [x, z]]), \\
\qquad\qquad\quad [z]] \\
\quad\ \textbf{in } \textbf{app } f\ 4
\end{array}
$$

In this case, the original program evaluates to an ordinary procedure, while the transform evaluates to a closure record.

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 they would if the language used dynamic scope, and we call the incomplete closures *lightweight closures*. If we took this approach to an extreme, a procedure might be able to find all its free variables at the call site, so no closure need be created, as in lambda-lifting [Augustsson 1984]. We do not go so far, but our techniques could clearly be used in this way.

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 an 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, while some of our techniques are similar to those of abstract interpretation, we believe that others represent a new approach. Control-flow analysis, one part of our analysis, is a typical application of abstract interpretation; our invariance sets, which track variable bindings that do not change across subterm evaluation, do not seem readily expressible in that framework.

## 2. EXAMPLES

The dataflow analysis has two significant responsibilities toward the closure conversion transformation. First, all procedures flowing to a given call site must agree on their application protocol. By application protocol, we mean which variables are designated as dynamic and in what order those variables appear as arguments. Second, the dynamic variables at a call site must be bound to the same values as they have at the definition sites for the procedures that flow to that site. As the following examples should suggest, it is not obvious how to perform the dataflow analysis. Consider the following:

$$
\begin{array}{l}
\textbf{let } x = \cdots \\
\quad\; y = \cdots \\
\textbf{in let } f = (\lambda v.\ \cdots\ x\ \cdots) \\
\qquad\quad\; g = (\lambda z.\ \cdots\ y\ \cdots) \\
\quad \textbf{in } (\textbf{if } (\textbf{zero? } \cdots) \textbf{ then } f \textbf{ else } g)\ c
\end{array}
$$

Variable $x$ is free in the procedure $f$, and $y$ is free in the procedure $g$. At the call site on the bottom line, $x$ and $y$ are in scope and can be supplied as extra arguments.

After closure conversion, we might have

$$
\begin{array}{l}
\textbf{let } x = \cdots \\
\quad\; y = \cdots \\
\textbf{in let } f = [(\lambda e, x, y, v. \cdots\ x\ \cdots), [\,]] \\
\qquad\quad\; g = [(\lambda e, x, y, z. \cdots\ y\ \cdots), [\,]] \\
\quad \textbf{in app } (\textbf{if } (\textbf{zero? } \cdots) \textbf{ then } f \textbf{ else } g)\ x\ y\ c
\end{array}
$$

Other choices are possible, subject to the condition that $f$ and $g$ obey the same application protocol. For instance, we could have reversed the order of the dynamic variables, or we could have left the dynamic variables in the closures. Our dataflow analysis will enforce protocol agreement.

Not all variables in scope at a call site are eligible to be made dynamic. A free variable in a procedure may be in the scope of different binders where the procedure is called and where the procedure is closed. Also, a procedure may escape the binding of a free variable and flow back to a call site where the variable is in scope. Even though the variable is visible at the call site, it may be bound to the wrong value. Consider the following:

$$
\begin{array}{l}
\textbf{let } g = \lambda x.\lambda y.\ \textbf{let } f = (\lambda z.x) \\
\qquad\qquad\qquad\quad\; \textbf{in } yf \\
\textbf{in } g\ c\ (g\ c'\ (\lambda v.v))
\end{array}
$$

The result of evaluating this program is $c'$. Variable $x$ is in scope at $f$'s call site $(yf)$, so a naive transformation might produce

$$
\begin{array}{ll}
\textbf{let } g = [(\lambda e, x.[(\lambda e', y.\,\textbf{let } x = (\#1\ e') & \qquad\qquad \texttt{wrong!} \\
\qquad\qquad\qquad\qquad\quad \textbf{in let } f = [(\lambda e'', x, z.x), [\,]] & \\
\qquad\qquad\qquad\qquad\qquad \textbf{in app } y\ x\ f), & \\
\qquad\qquad\quad [x]]), & \\
\qquad\quad [\,]] & \\
\textbf{in app } (\textbf{app } g\ c)\ (\textbf{app } (\textbf{app } g\ c')\ [(\lambda e'', x, v.v), [\,]]) &
\end{array}
$$

But there are two calls to $g$, both of which invoke the scope for $x$. For the left-hand call, $x$ is bound to $c$; for the right-hand call, $x$ is bound to $c'$. During the left-hand call to $g$, $y$ is bound to a copy of $f$ with its free $x$ bound to $c'$. That is, $f$ has escaped from the other invocation of $x$'s scope. This incorrect transform gives $c$ as a result. Therefore, the dataflow analysis needs to assure that $x$ is not considered a dynamic variable. So instead we want

$$
\begin{array}{l}
\textbf{let } g = [(\lambda e, x.[(\lambda e', y.\textbf{let } x = (\#1\ e') \qquad\qquad \texttt{right!}\\
\qquad\qquad\qquad\qquad \textbf{in let } f = [(\lambda e'', z.\textbf{let } x = (\#1\ e'')\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } x),\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad [x]]\\
\qquad\qquad\qquad\qquad\quad \textbf{in app } y\ f),\\
\qquad\qquad [x]]),\\
\qquad [\,]]\\
\textbf{in app}\,(\textbf{app } g\ c)\ (\textbf{app}\,(\textbf{app } g\ c')\ [(\lambda e'', v.v), [\,]])
\end{array}
$$

Our analysis may be seen as a kind of lifetime analysis, which compares the lifetimes of procedures and the bindings for their free variables. This last example demonstrates that the binding lifetime of a variable is not determined by its scope.

Dataflow patterns may be complex. For example, procedures may flow to call sites around loops:[1]

$$
\begin{array}{l}
\textbf{let } x = c\\
\textbf{in let } g = (\lambda y. \cdots\ x\ \cdots)\\
\quad \textbf{in letrec } f = (\lambda h.\lambda n.\textbf{if}\,(\textbf{zero?}\ n)\ \textbf{then}\ (h\ c')\ \textbf{else}\ (f\ h\ (\textbf{pred}\ n)))\\
\qquad \textbf{in } f\ g\ c''
\end{array}
$$

The variable $x$ is free in the procedure $g$, so ordinarily $x$ would appear in $g$'s closure. Now consider the call site $(h\ c')$. At that site, $h$ is bound to $g$, and $x$ is in scope. Certainly $x$ has the same binding there as at $g$'s definition site, since $x$ never gets rebound. Before that call is made, $f$ may be called many times, depending on the magnitude of $c''$. Therefore, the dataflow analysis should allow us to make $x$ a dynamic variable. Each recursive call to $f$ rebinds $h$ and $n$. The analysis detects that at the call site, $h$ is bound to $g$ and that $x$ has not been rebound since its definition. So what we want is

$$
\begin{array}{l}
\textbf{let } x = c\\
\textbf{in let } g = [(\lambda e, x, y. \cdots\ x\ \cdots), [\,]]\\
\quad \textbf{in letrec } f = [(\lambda e', h.[(\lambda e'', n.\textbf{let } h = (\#1\ e'')\\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in if}\,(\textbf{zero?}\ n)\ \textbf{then}\,(\textbf{app } h\ x\ c')\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\,(\textbf{app}\,(\textbf{app } f\ h)\ (\textbf{pred}\ n))),\\
\qquad\qquad\qquad\qquad\quad [h]]),\\
\qquad\qquad\qquad [\,]]\\
\qquad \textbf{in app}\,(\textbf{app } f\ g)\ c''
\end{array}
$$

and our analysis can give results like this.

Certain programming idioms may make lightweight closure conversion useful in practice. Consider the following procedure:

$$
\begin{array}{l}
\lambda t.\textbf{let } f = (\lambda p.\lambda a. \cdots\ t\ \cdots\ p\ \cdots\ a\ \cdots)\\
\quad \textbf{in let } g = f\ P\\
\qquad\quad\ \ h = f\ P'\\
\qquad \textbf{in } \cdots\ (g\ a')\ \cdots\ (h\ a'')\ \cdots
\end{array}
$$

[1] We can define a procedure in our input language that returns the least fixed point of a functional, so we can consider the **letrec** construct in this example as syntactic sugar. Plotkin [1975] gives such a procedure $Z$ as $\lambda f.(\lambda x.f(\lambda y.(xx)y))\ (\lambda x.f(\lambda y.(xx)y))$. This fixed point operator works with call-by-value evaluation, which we use for our input language.

Suppose the argument $t$ is a tree and $f$ a procedure that performs action $a$ on nodes of $t$ for which predicate $p$ holds. Procedures $g$ and $h$ are versions of $f$ specialized to predicates $P$ and $P'$, respectively. Naive closure conversion would include $t$ in the closures for $f$, $g$, and $h$. Lightweight closure conversion produces

$$
\begin{array}{l}
[(\lambda t.\,\mathbf{let}\ f = [(\lambda e, t, p.[(\lambda e', t, a.\,\mathbf{let}\ p = (\#1\ e') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \cdots\ t\ \cdots\ p\ \cdots\ a\ \cdots), \\
\qquad\qquad\qquad\qquad\qquad [p]]), \\
\qquad\qquad\qquad [\,]] \\
\qquad \mathbf{in\ let}\ g = \mathbf{app}\ f\ t\ P \\
\qquad\qquad\quad h = \mathbf{app}\ f\ t\ P' \\
\qquad\quad \mathbf{in}\ \cdots\ (\mathbf{app}\ g\ t\ a')\ \cdots\ (\mathbf{app}\ h\ t\ a'')\ \cdots), \\
\ [\,]]
\end{array}
$$

The tree $t$ is available at the call sites for $f$, $g$, and $h$ and so is omitted from their closures.

The rest of the article is organized as follows:

—In Section 3, we describe the input language, $\Lambda_{in}$. We also describe a data structure called an *occurrence closure* that represents $\Lambda_{in}$ terms, and we give an evaluator for such data structures.

—In Section 4, we describe the syntax and semantics of the annotations given to programs by our analysis. Program annotations are given as the states of a labeled transition system and data associated with such states.

—In Section 5, we show how program annotations may be derived as the solution to a deductive system generated from an input program text.

—In Section 6, we show how the deductive system annotates an example program.

—In Section 7, we show that any program annotation derivable from the deduction rules is sound.

—In Section 8, we describe the language produced by the closure conversion transformation.

—In Section 9, we present an equational reasoning system that will be used to simplify the proof of the correctness of the closure conversion transformation.

—In Section 10, we give the closure conversion transformation.

—In Section 11, we give a correctness result for the transformation.

—In Section 12, we mention related work.

—Finally, in Section 13, we present conclusions.

## 3. LANGUAGES

The language framework given in this section consists of a source language, some notation for describing occurrences of terms, a grammar for environments, and an evaluator that preserves occurrence source information.

### 3.1 The Source Language

We present the source language on which the transformation will operate. The language $\Lambda_{in}$ is an untyped $\lambda$-calculus with integer constants, boolean constants, conditionals, and primitive operators, given by the grammar

$$\begin{aligned} M \ ::= \ & x \mid c \mid \textbf{true} \mid \textbf{false} \mid \lambda x.M \mid \mathit{PrimOp}\ M \mid M M \mid \\ & \textbf{if}\ M\ \textbf{then}\ M\ \textbf{else}\ M \end{aligned}$$

where

$$\mathit{PrimOp} \ ::= \ \textbf{succ} \mid \textbf{pred} \mid \textbf{zero?}$$

The metavariable $c$ ranges over an infinite set of integer constants. Let $\mathit{Var}$ be the set of variables in $\Lambda_{in}$. Sometimes we will use the word "constant" to refer to an integer constant and sometimes to a boolean constant.

Though records appear in our output language for creating closures, we have deliberately omitted them from the input language. Adding records would be straightforward, but would add some tedious cases to our proofs; so we avoid records until we really need them.

The grammar above describes the abstract structure of $\Lambda_{in}$ terms, rather than their concrete syntax. We assume that application associates to the left, as usual; otherwise, we will use parentheses as necessary for disambiguation.

To emphasize that $\Lambda_{in}$ may be considered a programming language, we will usually refer to $\lambda$-abstractions in $\Lambda_{in}$ as "procedures." Procedures are curried, but we will sometimes write $\lambda\vec{x}.M$ as a notational convenience for $\lambda x_1. \ \ldots \ .\lambda x_n.M$. Similarly, we may write $\lambda x, y.M$ for $\lambda x.\lambda y.M$. We will usually refer to the formal parameter of a procedure as its "binding variable."

We write $M = N$ to indicate that $M$ and $N$ are $\alpha$-congruent — that is, the terms are syntactically identical, except for possibly different names for their bound variables.

Values in $\Lambda_{in}$ are given by the grammar

$$\mathit{Val} \ ::= \ x \mid c \mid \textbf{true} \mid \textbf{false} \mid \lambda x.M$$

where the metavariable $M$ ranges over terms in $\Lambda_{in}$. A *scalar value* is a boolean constant or integer constant.

## 3.2 Occurrences of Terms

For the purpose of showing the soundness of our analysis, an ordinary term evaluator that uses substitution has a significant limitation. Given a term and its value, we wish to compare the static annotations of the term and the value. But a substituting term evaluator discards source information about terms: given a substituted term, we cannot always know which parse tree node produced it. To solve this problem, we build an *occurrence evaluator* that preserves source information about terms.

We identify each source program occurrence by a string called an *occurrence index*. Occurrence indices for source programs are finite strings over the alphabet

$$\Sigma = \{\mathit{rator}, \mathit{rand}, \mathit{bv}, \mathit{body}, \mathit{test}, \mathit{then}, \mathit{else}\}.$$

An occurrence index for a source program term describes the path from the root of the parse tree to the occurrence of the term.[2] The empty string is the index

[2]Other approaches used to specify occurrences include generating a unique label for each occur-

for the root of the parse tree. For a procedure with occurrence index $i$, $i.bv$ is the index of its binding variable, and $i.body$ the index of the procedure body. For an application $i$, the operator has index $i.rator$, and the operand has index $i.rand$. For a conditional with index $i$, $i.test$, $i.then$, and $i.else$ are the indices for the test, then-part, and else-part, respectively. Any occurrence of a term may have subterms, except for an occurrence of a binding variable. We can more precisely specify the set of possible *program occurrences* by the regular expression

$$(rator + rand + body + test + then + else)^* \ (bv + \epsilon).$$

We write $[\![i]\!]_M$ for the subterm of $M$ with index $i$. We usually assume we are working with an occurrence in the input term $M_0$; in this case we write $[\![i]\!]$ instead of $[\![i]\!]_{M_0}$.

We can now define the data structures manipulated by the occurrence evaluator.

*Definition* 3.2.1. We simultaneously define *occurrence environments* and *occurrence closures* by the grammar

$$\begin{array}{ll} \psi ::= [\,] \mid \psi[x \mapsto oc] & \text{(occurrence environments)} \\ oc ::= (i, \psi) & \text{(occurrence closures)} \end{array}$$

where $i$ is an occurrence index.

We may write $\psi[v_1, \ldots, v_n \mapsto oc_1, \ldots, oc_n]$ as shorthand for the occurrence environment $\psi[v_1 \mapsto oc_1] \cdots [v_n \mapsto oc_n]$. We shall often rely on the inductively-defined structure of occurrence environments in subsequent definitions and proofs.

We treat occurrence environments as finite functions, in the usual way:

$$\begin{array}{r} \psi[x \mapsto (i, \psi')](x) = (i, \psi') \\ \psi[y \mapsto (i, \psi')](x) = \psi(x) \\ [\,](x) \text{ is undefined} \end{array}$$

We write $Dom(\psi)$ for the domain of an occurrence environment $\psi$.

In certain cases, we will want occurrence environments to be subject to restrictions:

*Definition* 3.2.2. An occurrence environment $\psi$ is a *scalar environment* iff $\forall x \in Dom(\psi)$, $\psi(x) = (j, \psi')$ implies $[\![j]\!]$ is a scalar value.

In particular, the empty occurrence environment $[\,]$ is a scalar environment.

When evaluating an occurrence closure $(i, \psi)$, the occurrence evaluator may manipulate occurrences that originate in the environment $\psi$, besides occurrences of subterms of the source program $[\![i]\!]$. Also, use of the primitive operators may produce constants found in neither a program nor an environment. Therefore, we need to describe occurrence indices for terms other than source terms.

rence [Sestoft 1991], $\alpha$-converting a program so that each binding occurrence is unique, and using the binding variable as a token [Palsberg and Schwartzbach 1995]. In both cited works, only occurrences of procedures are specified. Our approach allows us to refer to occurrences of arbitrary terms. Also, given the index of a term, we can derive the indices of its subterms. Flanagan and Felleisen [1995] use *A-normal forms* to encode occurrences as variable names by assigning the result of each program subexpression to a fresh binder in a **let**-expression.

Let us define a set of indices for occurrences that originate in environments. Let $K$ be an infinite set of strings, disjoint from $\Sigma^*$. We assume $K$ is a regular set over a finite alphabet. An occurrence environment $\psi$ is an *initial environment* iff $\forall x \in \mathit{Dom}(\psi)$, if $\psi(x) = (\kappa, \psi')$ then $\kappa \in K$, and $\psi'$ is an initial environment. This definition is well founded, since it makes the empty environment initial. An occurrence index that has a prefix $\kappa \in K$ is said to be an *environment occurrence*. We assume that subterms of values in an initial environment are assigned occurrence indices as for program subterms.

Scalar values are their own occurrence indices.[3] That is, for any integer constant or boolean constant $c$, $[\![c]\!] = c$. Let $C$ be the set of all such *constant occurrences*. We assume that $C$, like $K$, is coded as a regular set of strings over some finite alphabet, e.g., the set $\{0, 1\}$. $C$ is disjoint from the sets of program and environment occurrences, but observe that program and environment occurrences may also refer to constants.

Now we can give an expression that describes the entire set of occurrence indices. Let $O$ be the regular set:

$$C + ((K + \epsilon)\ (\mathit{rator} + \mathit{rand} + \mathit{body} + \mathit{test} + \mathit{then} + \mathit{else})^*\ (\mathit{bv} + \epsilon)).$$

For a $\Lambda_{in}$ term $M$, $O_M$ indicates the set of occurrence indices in $M$; for an occurrence environment $\psi$, $O_\psi$ is the set of occurrence indices in $\psi$; and $O_{M,\psi}$ is the set $O_M \cup O_\psi$.

### 3.3 The Occurrence Evaluator

The occurrence evaluator is an abstract machine that uses environment extension to simulate the substitutions that would be used by a call-by-value term evaluator. The evaluation relation $\underset{oc}{\Longrightarrow}$ defines a relation between occurrence closures representing arbitrary terms and occurrence closures representing values. The rules for the occurrence evaluator are given in Figure 1.

We use the predicates

$$\mathit{Const},\ \mathit{Var},\ \mathit{Proc},\ \mathit{PrimApp},\ \mathit{App},\ \mathit{Cond}$$

to test whether an occurrence refers to a constant, variable, procedure, primitive application, application, or conditional. The predicate *Const* is true for an occurrence of a scalar value. As for $[\![-]\!]$, we can think of these predicates as parameterized by a particular term. However, we will be informal in our use of such predicates, omitting mention of the term in which an occurrence appears.

### 3.4 Unwinding Occurrence Closures

An occurrence closure represents a term. The meaning of occurrence closures is given by the map $\mathcal{U}[\![-]\!]$ defined by

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

[3]Here we are confounding pointers and integers in the same way for which pre-ANSI C language hackers were once notorious. Our goal in doing so is to make the occurrence evaluator rules involving constants a little simpler.

$$\frac{Var(i) \quad [\![i]\!] \notin Dom(\psi)}{(i,\psi) \underset{oc}{\Longrightarrow} (i,[\,])} \qquad \frac{Var(i) \quad [\![i]\!] \in Dom(\psi)}{(i,\psi) \underset{oc}{\Longrightarrow} \psi([\![i]\!])}$$

$$\frac{Const(i)}{(i,\psi) \underset{oc}{\Longrightarrow} (i,[\,])} \qquad \frac{Proc(i)}{(i,\psi) \underset{oc}{\Longrightarrow} (i,\psi)}$$

$$\frac{PrimApp(i) \quad [\![i.rator]\!] = \mathbf{succ} \quad (i.rand,\psi) \underset{oc}{\Longrightarrow} (j,\psi') \quad [\![j]\!] = c_n}{(i,\psi) \underset{oc}{\Longrightarrow} (c_{n+1},[\,])}$$

$$\frac{PrimApp(i) \quad [\![i.rator]\!] = \mathbf{pred} \quad (i.rand,\psi) \underset{oc}{\Longrightarrow} (j,\psi') \quad [\![j]\!] = c_n}{(i,\psi) \underset{oc}{\Longrightarrow} (c_{n-1},[\,])}$$

$$\frac{PrimApp(i) \quad [\![i.rator]\!] = \mathbf{zero?} \quad (i.rand,\psi) \underset{oc}{\Longrightarrow} (j,\psi') \quad [\![j]\!] = c_0}{(i,\psi) \underset{oc}{\Longrightarrow} (\mathbf{true},[\,])}$$

$$\frac{PrimApp(i) \quad [\![i.rator]\!] = \mathbf{zero?} \quad (i.rand,\psi) \underset{oc}{\Longrightarrow} (j,\psi') \quad [\![j]\!] = c_n,\ n \neq 0}{(i,\psi) \underset{oc}{\Longrightarrow} (\mathbf{false},[\,])}$$

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

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

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

Fig. 1. Rules for the occurrence evaluator.

where the curly braces indicate substitution. This definition is well founded, since occurrence closures have finite depth by Definition 3.2.1; termination occurs when $\psi$ is empty. We call $\mathcal{U}[\![-]\!]$ the *unwinding* function.

For example, suppose

$$\begin{aligned} [\![i]\!] &= x \\ [\![j]\!] &= (\lambda z.zy) \\ \psi &= [\,][x \mapsto (j, \psi')] \\ \psi' &= [\,][y \mapsto (c, [\,])] \end{aligned}$$

Then

$$\begin{aligned} \mathcal{U}[\![(i,\psi)]\!] &= [\![i]\!]\{\mathcal{U}[\![-]\!] \circ \psi\} \\ &= x\{\mathcal{U}[\![-]\!] \circ [\,][x \mapsto (j,\psi')]\} \\ &= \mathcal{U}[\![(j,\psi')]\!] \\ &= [\![j]\!]\{\mathcal{U}[\![-]\!] \circ \psi'\} \\ &= (\lambda z.zy)\{\mathcal{U}[\![-]\!] \circ \psi'\} \\ &= (\lambda z.z\ \mathcal{U}[\![(\psi'(y))]\!]) \\ &= (\lambda z.z\ \mathcal{U}[\![(c,[\,])]\!]) \\ &= (\lambda z.z\ [\![c]\!]\{\mathcal{U}[\![-]\!] \circ [\,]\}) \\ &= (\lambda z.zc) \end{aligned}$$

## 4. ANNOTATIONS

In preparation for the closure conversion transformation, we construct for any program a particular kind of labeled transition system with some additional baggage. Recall that $O$ is the set of all occurrence indices, and $Var$ is the set of all variables.

*Definition* 4.1. An *abstract execution* consists of

—a set of states $S = S_{env} \cup S_{val}$, where $S_{env}$ is a set of *environment states* and $S_{val}$ is a set of *value states*

—a set of transition labels $T = O \cup Var$,

—a $T$-indexed family of transition relations on $S$,

—a map $\Theta\colon S_{val} \to 2^{Var}$, called the *invariance set map*, and

—a map $\Psi\colon S_{val} \to Var^*$, called the *protocol tag map*

such that

—$S_{env}$ and $S_{val}$ are disjoint

—if $s_1 \xrightarrow{i} s_2$ for some $i \in O$, then $s_1 \in S_{val}$ and $s_2 \in S_{env}$

—if $s_1 \xrightarrow{x} s_2$ for some $x \in Var$, then $s_1 \in S_{env}$ and $s_2 \in S_{val}$

—if $s_1 \xrightarrow{t} s_2$ and $s_1 \xrightarrow{t} s_3$ for some $t \in T$, then $s_2 = s_3$.

We use $\mathcal{A}$ as a metavariable ranging over abstract executions. We use $\mathsf{E}$ (sometimes $\mathsf{F}$ and $\mathsf{G}$) to range over elements of $S_{env}$, and $\mathsf{V}$ to range over elements of

$S_{val}$.[4] For an abstract execution $\mathcal{A}$, we may write $\Theta_{\mathcal{A}}$ for its invariance set map and $\Psi_{\mathcal{A}}$ for its protocol tag map.

Because transitions are deterministic, we may think of states as functions. For an environment state $\mathsf{E}$ in an abstract execution $\mathcal{A}$, define $Dom(\mathsf{E})$ to be

$$\{x \mid \exists \mathsf{V} \text{ such that } \mathsf{E} \xrightarrow{x} \mathsf{V} \text{ in } \mathcal{A}\}$$

Similarly, for a value state $\mathsf{V}$ in $\mathcal{A}$, define $Dom(\mathsf{V})$ to be

$$\{i \mid \exists \mathsf{E} \text{ such that } \mathsf{V} \xrightarrow{i} \mathsf{E} \text{ in } \mathcal{A}\}$$

As indicated, an abstract execution has two kinds of transitions. The transition labels are variables and occurrence indices. Intuitively, a transition of the form

$$\mathsf{E} \xrightarrow{x} \mathsf{V}$$

describes the lookup of a variable $x$ in an environment described by $\mathsf{E}$, while a transition of the form

$$\mathsf{V} \xrightarrow{i} \mathsf{E}$$

describes a call from a site described by $\mathsf{V}$ to the procedure with occurrence index $i$. More precisely, the call is to a substitution instance of that procedure where the substitution is described by $\mathsf{E}$. For example, the sequence of transitions

$$\mathsf{V} \xrightarrow{i} \mathsf{E} \xrightarrow{x} \mathsf{V}'$$

represents a call to the procedure $[\![i]\!]$, followed by a lookup of an occurrence of $x$ free in $[\![i]\!]$. Any transitions from $\mathsf{V}'$, which we have not shown, would indicate the procedures to which $x$ might evaluate.

The $\Theta_{\mathcal{A}}$ and $\Psi_{\mathcal{A}}$ maps of an abstract execution $\mathcal{A}$ associate with each value state $\mathsf{V}$ in $\mathcal{A}$ an *invariance set* $\theta_{\mathsf{V}}$ and a *protocol tag* $\pi_{\mathsf{V}}$. An invariance set is a finite set of variables. These are intended to be variables whose bindings do not change across an occurrence closure evaluation, in case the result of the evaluation is a procedure. A protocol tag is a finite sequence of variables, which describes a procedure application protocol. We may write $\langle v_1, \ldots, v_n\rangle$ to indicate such a sequence of variables. For any sequence of variables $\langle v_1, \ldots, v_n\rangle$, $\lceil\langle v_1, \ldots, v_n\rangle\rceil$ is the corresponding set, $\{v_1, \ldots, v_n\}$, and $||\langle v_1, \ldots, v_n\rangle||$ is the length of the sequence, $n$. We will more fully describe how this data associated with value states is used later.

An abstract execution can be described independently of a program, but we will use abstract execution states to annotate program occurrences:

*Definition* 4.2. Let $M$ be a term. Let $\psi$ be an occurrence environment, and let $\mathcal{A}$ be an abstract execution. An *annotation map* from $(M, \psi)$ to $\mathcal{A}$ is a partial map

$$\Gamma\colon O_{M,\psi} \to S_{env} \times S_{val}$$

[4]The use of labeled transition systems to describe abstract executions is a change from Wand and Steckler [1994] and Steckler [1994]. There we wrote $\mathsf{E}(x) = \mathsf{V}$ for $\mathsf{E} \xrightarrow{x} \mathsf{V}$ and $(i, \mathsf{E}) \in \phi$ for $\mathsf{V} \xrightarrow{i} \mathsf{E}$. We believe the the use of labeled transition systems gives a clearer account of annotations that contain cycles and of the constraint generation process (see Section 5).

Such a $\Gamma$ may be undefined at constant occurrences, but we require that $\Gamma$ be otherwise total on $O_{M,\psi} - C$.

When the context permits, we may write

—$\mathsf{E}_{\Gamma,i}$ or $\mathsf{E}_i$ for the first component of $\Gamma(i)$,

—$\mathsf{V}_{\Gamma,i}$ or $\mathsf{V}_i$ for the second component of $\Gamma(i)$,

—$\theta_{\Gamma,i}$ or $\theta_i$ for $\Theta_{\mathcal{A}}(\mathsf{V}_i)$, or

—$\pi_{\Gamma,i}$ or $\pi_i$ for $\Psi_{\mathcal{A}}(\mathsf{V}_i)$.

*Definition* 4.3. Let $\mathcal{A}$ be an abstract execution. Let $M$ be a term, and let $\psi$ be an occurrence environment. An annotation map $\Gamma$ from $(M, \psi)$ to $\mathcal{A}$ is *monovariant* iff for all $i$ in $Dom(\Gamma)$, if $\mathsf{V}_{\Gamma,i} \xrightarrow{j} \mathsf{E}$ is in $\mathcal{A}$, then $\mathsf{E} = \mathsf{E}_{\Gamma,j}$.

A monovariant annotation map $\Gamma$ associates each procedure $j$ in its domain with one "abstract closure," given by the pair $(j, \mathsf{E}_{\Gamma,j})$. A monovariant annotation map thus corresponds to Shivers' well-known 0CFA control-flow analysis.

We next need to define a notion of *satisfaction* to relate concrete and abstract executions. Informally, we say that the evaluation of a program $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$ satisfies an annotation map $\Gamma$ iff whenever the input $\psi$ satisfies $\mathsf{E}_{\Gamma,i}$, then the output satisfies the value state $\mathsf{V}_{\Gamma,i}$. Moreover, similar satisfaction relations also hold at all subcomputations. In this way, we may think of the pair $(\mathsf{E}_{\Gamma,i}, \mathsf{V}_{\Gamma,i})$ as input and output assertions for a functional program, analogous to Hoare-style partial-correctness assertions for imperative programs [Hoare 1969]. Further, if $j$ is an occurrence of a procedure, then we will require that the environments $\psi$ and $\psi'$ agree on all variables in $\theta_{\Gamma,i}$, the invariance set associated with $\mathsf{V}_{\Gamma,i}$, and $\pi_{\Gamma,j}$ must be the same as $\pi_{\Gamma,i}$, the protocol tag associated with $\mathsf{V}_{\Gamma,i}$.

We formalize these intuitions as follows. Define a *protocol assignment* to be a map from occurrences to protocol tags. We write $\Pi$ for such a map.

*Definition* 4.4. We simultaneously define two satisfaction relations $\models^{\mathcal{A},\Pi}_{env}$ and $\models^{\mathcal{A},\Pi}_{val}$ inductively on the depth of occurrence environments:

(1) An occurrence environment $\psi$ satisfies an environment state $\mathsf{E}$ under an abstract execution $\mathcal{A}$ and a protocol assignment $\Pi$, written:

$$\psi \models^{\mathcal{A},\Pi}_{env} \mathsf{E}$$

iff for all $x$ and $\mathsf{V}$, if $\mathsf{E} \xrightarrow{x} \mathsf{V}$ in $\mathcal{A}$, then

(a) $x \in Dom(\psi)$ and $\psi(x) \models^{\mathcal{A},\Pi}_{val} \mathsf{V}$ and

(b) if $\psi(x) = (i, \psi')$, and $Proc(i)$, then for all $y \in \theta_{\mathsf{V}}$, $y \in Dom(\psi) \cap Dom(\psi')$, and $\psi(y) = \psi'(y)$.

(2) An occurrence closure $(i, \psi)$ satisfies a value state $\mathsf{V}$ under an abstract execution $\mathcal{A}$ and a protocol assignment $\Pi$, written:

$$(i, \psi) \models^{\mathcal{A},\Pi}_{val} \mathsf{V}$$

iff

(a) $Const(i) \vee Var(i)$ or

(b) $Proc(i)$, $\Pi(i) = \pi_{\mathsf{V}}$, and $\exists \mathsf{E}$ such that $\mathsf{V} \xrightarrow{i} \mathsf{E}$ in $\mathcal{A}$ and $\psi \models^{\mathcal{A},\Pi}_{env} \mathsf{E}$.

This definition is well founded because it is inductive in the structure of the occurrence environment $\psi$; the empty occurrence environment serves as a base case.

The first condition for $\models^{\mathcal{A},\Pi}_{env}$ requires pointwise satisfaction of the value states in the range of $\mathsf{E}$ by the values in $\psi$. The second condition says that the variables in the invariance set associated with such a value state have bindings that are invariant across environment lookups, in case a lookup returns a procedure.

For $\models^{\mathcal{A},\Pi}_{val}$, the first condition says that a value state is satisfied by any value that is not a procedure. The second condition says that for a procedure, its protocol tag must match the protocol tag associated with the value state; its occurrence index must appear as the label of a transition from the value state; furthermore, its closing environment must satisfy the environment state $\mathsf{E}$ reached by that transition. This last requirement is equivalent to saying that $\mathcal{U}[\![(i,\psi)]\!]$ is a substitution instance of $[\![i]\!]$ where the substitution satisfies $\mathsf{E}$.

The following lemmas are immediate from the definition of $\models^{\mathcal{A},\Pi}_{env}$:

Lemma 4.5. *Let $\mathcal{A}$ be an abstract execution. If $\psi$ is a scalar environment, then for any protocol assignment $\Pi$ and for all environment states $\mathsf{E}$ in $\mathcal{A}$ such that $Dom(\mathsf{E}) \subseteq Dom(\psi)$, $\psi \models^{\mathcal{A},\Pi}_{env} \mathsf{E}$.*

Lemma 4.6. *Let $\mathcal{A}$ be an abstract execution, and let $\mathsf{E}$ be an environment state in $\mathcal{A}$. Let $\psi$ be an occurrence environment, and let $\Pi$ be a protocol assignment such that $\psi \models^{\mathcal{A},\Pi}_{env} \mathsf{E}$. Suppose that $\mathsf{E}'$ is an environment state in $\mathcal{A}$ such that for all $x$ and $\mathsf{V}$, if $\mathsf{E}' \xrightarrow{x} \mathsf{V}$ in $\mathcal{A}$, then $\mathsf{E} \xrightarrow{x} \mathsf{V}$ in $\mathcal{A}$. Then $\psi \models^{\mathcal{A},\Pi}_{env} \mathsf{E}'$.*

## 5. LOCALLY CONSISTENT ANNOTATIONS

Our notion of satisfaction corresponds to what is called "concretization" in the language of abstract interpretation [Cousot and Cousot 1977]. Thus we might consider occurrence closures as points in a concrete domain, and value states as points in an abstract domain; our

$$(i, \psi) \models^{\mathcal{A},\Pi}_{val} \mathsf{V}$$

might be written $(i, \psi) \in Conc(\mathsf{V})$. However, we derive suitable $\mathsf{V}$ by applying deduction rules rather than by abstract interpretation. Such deduction rules are *local constraints* on the points in $\Gamma(i)$ and their associated data. We say that an annotation of a program is *locally consistent* iff it satisfies all the constraints for that program. In this section, we define these local constraints. In Section 6 we work out an example. Then, in Section 7, we show that any locally consistent annotation of a program soundly describes the possible executions of that program.

More formally, we state:

*Definition* 5.1. Let $M$ be a term. Let $\psi$ be an occurrence environment, and let $\mathcal{A}$ be an abstract execution. An annotation map $\Gamma$ from $(M, \psi)$ to $\mathcal{A}$ is *locally consistent* iff all the deduction rules in Figures 2 through 5 are satisfied.

In Figure 2, a double rule indicates a biimplication.

We can convey some of the intuitions behind the deduction rules. Transitions from value states are meant to give us a closure analysis: for a value state $\mathsf{V}_i$, $Dom(\mathsf{V}_i)$ should approximate the set of procedures to which $i$ may evaluate. Consider a term with occurrence index $i$. If $Proc(i)$, then for any $\psi$, $(i, \psi)$ evaluates to itself; so we have the **proc-self** rule $\mathsf{V}_i \xrightarrow{i} \mathsf{E}_i$. If $App(i)$ and $\mathsf{V}_{i.rator} \xrightarrow{j} B$, for some $B$, then by the **app-rand-bv** rule, in case $\mathsf{V}_{i.rand} \xrightarrow{k} C$, for some $k$ and $C$, also $\mathsf{V}_{j.bv} \xrightarrow{k} C$. This says that any procedure with index $k$ that may be the result of evaluating the operand $i.rand$ may be an argument to any procedure $j$ that may be the result of evaluating the operator $i.rator$. Similarly, by the **app-body-app** rule, if $App(i)$ and $\mathsf{V}_{i.rator} \xrightarrow{j} B$, then in case $\mathsf{V}_{j.body} \xrightarrow{k} C$, also $\mathsf{V}_i \xrightarrow{k} C$. This says that any procedure $k$ that may be returned by the procedure $j$ may also be the result of evaluating the application. These particular deduction rules find exact analogs in constraint-based formulations of closure analysis.

Consider some of the deduction rules for protocol tags. If the operator of an application $i$ evaluates to a procedure with occurrence index $j$, by the **app-tag-bv-rand** rule, the protocol tag for the operand $i.rand$ and the binding variable $j.bv$ have to agree. This rule forces all procedures to which $i.rator$ might evaluate to take arguments tagged with the same protocol. Similarly, if the operator evaluates to a procedure $j$, the **app-tag-body-app** rule requires the protocol tag for $j.body$ to have the same tag as the application itself. Thus for any procedure to which $i.rator$ might evaluate, its body receives the same protocol tag as the body of any other such procedure, and that tag is also the tag for the application. The **cond-tag** rule requires that the then- and else-parts of a conditional have the same protocol tag as the conditional itself.

For invariance sets, the deduction rules are less obvious. Since we have formulated the rules to deduce what variables are *not* in invariance sets, these rules tell us which variables may get rebound during an evaluation. Suppose $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$ and that $Proc(j)$ holds. We want $\theta_i$ to be a set of variables whose bindings are the same in $\psi$ and $\psi'$. With that in mind, the significance of most of the deduction rules for invariance sets should become clear in the proof of the Soundness Theorem (Theorem 7.1). Here, we describe the rules for applications, which are the most complicated. Suppose we have the evaluation:

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

What values in $\psi$ are left invariant by this calculation? $\psi'''$ is obtained from $\psi$ in two steps. First, the operator is evaluated, yielding a $\psi'$ agreeing with $\psi$ on the variables in $\theta_{i.rator}$ (by an appropriate induction hypothesis). We then evaluate $j.body$ in an extension of $\psi'$, yielding $\psi'''$, which agrees with $\psi'$ on $\theta_{j.body} - \{[\![j.bv]\!]\}$. In order that $\psi$ agrees with $\psi'''$ on the variables in $\theta_i$, we must have that $\theta_i \subseteq \theta_{i.rator} \cap (\theta_{j.body} - \{[\![j.bv]\!]\})$. This explains the rules **app-inv-rator**, **app-inv-body-app**, and **app-inv-bv-app**. The other conditions for the invariance sets are needed so that $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \models^{\mathcal{A},\Pi}_{env} \mathsf{E}_{j.body}$, in support of the induction hypothesis for the body. The details appear in the proof of Theorem 7.1.

The one protocol tag constraint, **app-tag-cons**, says that the protocol tag for the operator of an application may contain only the variables appearing in the invariance set for the operator. In other words, the only candidate dynamic variables are those that have the same bindings in the current environment as in the closing environment for the procedure to which the operator evaluates.

We sketch how to construct a locally consistent and monovariant annotation map, given a term $M$ and an occurrence environment $\psi$.

*Abstract Execution.* We need to build an abstract execution, so we construct each of the items in Definition 4.1:

(1) Generate a fresh token for each distinct lexical scope in $M$ and $\psi$. $S_{env}$ consists of these tokens.
(2) Generate a fresh token for each free variable in $M$ and $\psi$, a token for each occurrence of a binding variable and a token for each remaining occurrence, other than variables and primitive operators. $S_{val}$ consists of all these tokens.
(3) The set of transition labels $T$ consists of all variables (including binding variables) in $M$ and $\psi$ and the indices of all procedures in $M$ and $\psi$.
(4) The transition relations are computed by taking the deductive closure of the rules in Figure 2.
(5) The invariance set map $\Theta$ is defined by taking the deductive closure of the rules in Figure 3.
(6) The protocol tag map $\Psi$ is defined by taking the deductive closure of the rules in Figure 4 and solving the constraints generated by the rule in Figure 5.

Items (4) through (6) indicate the sources of transitions, invariance sets, and protocol tags for an abstract execution $\mathcal{A}$, but a solution algorithm must heed (and may take advantage of) dependencies among the deduction rules. We now briefly describe how a solution algorithm might proceed.

*Solution Algorithm.* The building of the transition relations mentioned in item (4) corresponds to a *closure analysis* [Sestoft 1991]. As we perform the closure analysis, we may simultaneously apply the rules in Figure 5, part of item (6). Those rules build equivalence classes of protocol tags; we defer solving for the actual tags.

Note, however, that rule **proc-inv** in Figure 3 is *not* monotone: it has a negation in its hypothesis. That means that the closure analysis has to be finished before performing the invariance analysis mentioned in item (5). Were these analyses intermixed, enlarging the domain of an environment state in the closure analysis might result in *removing* an assertion about an invariance set.

In the last step, *constraint solution*, mentioned in item (6), we choose a protocol tag for each equivalence class of tags. Equivalence classes of protocol tags induce equivalence classes of occurrences: for any two occurrences $i$ and $j$, if $[\pi_i] = [\pi_j]$, then $[i] = [j]$. The rule **app-tag-cons** shown in Figure 5 indicates what an appropriate tag is. For a given equivalence class of occurrences $C$, we may choose any ordering on any subset of

$$MaxInv_C = \bigcap \{\theta_{i.rator} \mid i.rator \in C\}$$

$$Var(i) \Longrightarrow \mathsf{E}_i \xrightarrow{[\![i]\!]} \mathsf{V}_i \qquad \textbf{var}$$

$$PrimApp(i) \Longrightarrow \begin{array}{c} \mathsf{E}_i \xrightarrow{x} \mathsf{V} \\ \hline\hline \mathsf{E}_{i.rand} \xrightarrow{x} \mathsf{V} \end{array} \qquad \textbf{primapp-rand-inh}$$

$$Proc(i) \Longrightarrow$$

$$\left[\begin{array}{ll}
\mathsf{V}_i \xrightarrow{i} \mathsf{E}_i & \textbf{proc-self} \\
\mathsf{E}_{i.body} \xrightarrow{[\![i.bv]\!]} \mathsf{V}_{i.bv} & \textbf{proc-body-ext} \\
\begin{array}{c} \mathsf{E}_i \xrightarrow{x} \mathsf{V} \\ \hline\hline \mathsf{E}_{i.body} \xrightarrow{x} \mathsf{V} \end{array} \quad x \neq [\![i.bv]\!] & \textbf{proc-body-inh}
\end{array}\right.$$

$$App(i) \Longrightarrow$$

$$\left[\begin{array}{ll}
\begin{array}{c} \mathsf{E}_i \xrightarrow{x} \mathsf{V} \\ \hline\hline \mathsf{E}_{i.rator} \xrightarrow{x} \mathsf{V} \end{array} & \textbf{app-rator-inh} \\
\begin{array}{c} \mathsf{E}_i \xrightarrow{x} \mathsf{V} \\ \hline\hline \mathsf{E}_{i.rand} \xrightarrow{x} \mathsf{V} \end{array} & \textbf{app-rand-inh} \\
\dfrac{\mathsf{V}_{i.rator} \xrightarrow{j} \mathsf{F} \quad \mathsf{V}_{i.rand} \xrightarrow{k} \mathsf{G}}{\mathsf{V}_{j.bv} \xrightarrow{k} \mathsf{G}} & \textbf{app-rand-bv} \\
\dfrac{\mathsf{V}_{i.rator} \xrightarrow{j} \mathsf{F} \quad \mathsf{V}_{j.body} \xrightarrow{k} \mathsf{G}}{\mathsf{V}_i \xrightarrow{k} \mathsf{G}} & \textbf{app-body-app}
\end{array}\right.$$

$$Cond(i) \Longrightarrow$$

$$\left[\begin{array}{ll}
\begin{array}{c} \mathsf{E}_i \xrightarrow{x} \mathsf{V} \\ \hline\hline \mathsf{E}_{i.test} \xrightarrow{x} \mathsf{V} \end{array} & \textbf{cond-test-inh} \\
\begin{array}{c} \mathsf{E}_i \xrightarrow{x} \mathsf{V} \\ \hline\hline \mathsf{E}_{i.then} \xrightarrow{x} \mathsf{V} \end{array} & \textbf{cond-then-inh} \\
\begin{array}{c} \mathsf{E}_i \xrightarrow{x} \mathsf{V} \\ \hline\hline \mathsf{E}_{i.else} \xrightarrow{x} \mathsf{V} \end{array} & \textbf{cond-else-inh} \\
\dfrac{\mathsf{V}_{i.then} \xrightarrow{j} \mathsf{F}}{\mathsf{V}_i \xrightarrow{j} \mathsf{F}} & \textbf{cond-then-cond} \\
\dfrac{\mathsf{V}_{i.else} \xrightarrow{j} \mathsf{F}}{\mathsf{V}_i \xrightarrow{j} \mathsf{F}} & \textbf{cond-else-cond}
\end{array}\right.$$

Fig. 2. Deduction rules for transitions.

$$Proc(i) \Longrightarrow \frac{x \notin Dom(\mathsf{E}_i)}{x \notin \theta_i} \qquad \textbf{proc-inv}$$

$$App(i) \Longrightarrow$$

$$\left[\begin{array}{ll}
\dfrac{x \notin \theta_{i.rator}}{x \notin \theta_i} & \textbf{app-inv-rator} \\[2ex]
\dfrac{\mathsf{V}_{i.rator} \xrightarrow{j} \mathsf{F} \quad x \notin \theta_{i.rator}}{x \notin \theta_{j.bv}} & \textbf{app-inv-rator-bv} \\[2ex]
\dfrac{\mathsf{V}_{i.rator} \xrightarrow{j} \mathsf{F} \quad x \notin \theta_{i.rand}}{x \notin \theta_{j.bv}} & \textbf{app-inv-rand-bv} \\[2ex]
\dfrac{\mathsf{V}_{i.rator} \xrightarrow{j} \mathsf{F} \quad x \notin \theta_{j.body}}{x \notin \theta_i} & \textbf{app-inv-body-app} \\[2ex]
\dfrac{\mathsf{V}_{i.rator} \xrightarrow{j} \mathsf{F}}{[\![j.bv]\!] \notin \theta_{j.bv}} & \textbf{app-inv-bv} \\[2ex]
\dfrac{\mathsf{V}_{i.rator} \xrightarrow{j} \mathsf{F}}{[\![j.bv]\!] \notin \theta_i} & \textbf{app-inv-bv-app}
\end{array}\right.$$

$$Cond(i) \Longrightarrow$$

$$\left[\begin{array}{ll}
\dfrac{x \notin \theta_{i.then}}{x \notin \theta_i} & \textbf{cond-inv-then} \\[2ex]
\dfrac{x \notin \theta_{i.else}}{x \notin \theta_i} & \textbf{cond-inv-else}
\end{array}\right.$$

Fig. 3. Deduction rules for invariance sets.

$$App(i) \Longrightarrow$$

$$\left[\begin{array}{ll}
\dfrac{\mathsf{V}_{i.rator} \xrightarrow{j} \mathsf{F}}{\pi_{j.bv} = \pi_{i.rand}} & \textbf{app-tag-bv-rand} \\[2ex]
\dfrac{\mathsf{V}_{i.rator} \xrightarrow{j} \mathsf{F}}{\pi_{j.body} = \pi_i} & \textbf{app-tag-body-app}
\end{array}\right.$$

$$Cond(i) \Longrightarrow \pi_i = \pi_{i.then} = \pi_{i.else} \qquad \textbf{cond-tag}$$

Fig. 4. Deduction rules for protocol tags.

$$App(i) \Longrightarrow \frac{x \notin \theta_{i.rator}}{x \notin \lceil \pi_{i.rator} \rceil} \qquad \textbf{app-tag-cons}$$

Fig. 5. Protocol tag constraint.

If a variable does not occur free in some procedure within $C$, there is no reason to make it dynamic. Therefore, in practice, we choose any ordering on any subset of

$$MaxInv_C \cap \bigcup\{FV([\![i]\!]) \mid i \in C \wedge Proc(i)\}$$

In a compiler, the choice of subset may involve performance considerations, which we do not address here.

*Complexity.* To determine the complexity of finding a locally consistent annotation, we may translate the rules in Figures 2 to 4 to the language of *conditional constraints* of Palsberg and Schwartzbach [1994, Chapter 5]. A constraint in their system is of the forms $c \in X$, $X \subseteq Y$, or $c \in X \Longrightarrow Y \subseteq Z$. Let $n$ be the number of occurrences in a program. If the number of constraints is $O(n^2)$, then solving the constraints can be done in time $O(n^3)$. A program with $n$ distinct occurrences generates $O(n^2)$ conditional constraints under a translation (not shown here). By this reduction, our deduction rules may be solved in time $O(n^3)$. The only rule not translatable into a conditional constraint, **app-tag-cons**, requires us to take $O(n)$ intersections of the complements of invariance sets of size $O(n)$. Since each intersection can be done in time $O(n)$, by using a bit-vector representation of sets, we still have an overall $O(n^3)$ time bound.

Finally for this section, we mention:

LEMMA 5.2. *Let $M$ be a term. Let $\psi$ be an occurrence environment, and let $\mathcal{A}$ be an abstract execution. Let $\Gamma$ be a locally consistent annotation from $(M, \psi)$ to $\mathcal{A}$. Let $i$ be the occurrence index of a procedure in $O_{M,\psi}$. For any occurrence $m$ of $[\![i.bv]\!]$ free in $i.body$, $\mathsf{V}_{\Gamma,m} = \mathsf{V}_{\Gamma,i.bv}$, $\theta_{\Gamma,m} = \theta_{\Gamma,i.bv}$, and $\pi_{\Gamma,m} = \pi_{\Gamma,i.bv}$.*

PROOF. By considering the deduction rules in Figure 2.

By **var**, $\mathsf{E}_m \xrightarrow{[\![m]\!]} \mathsf{V}_m$. From the rules, transitions from environment states propagate from subterms to terms, so also $\mathsf{E}_{i.body} \xrightarrow{[\![m]\!]} \mathsf{V}_m$. By **proc-body-ext**, $\mathsf{E}_{i.body} \xrightarrow{[\![i.bv]\!]} \mathsf{V}_{i.bv}$. $[\![m]\!] = [\![i.bv]\!]$, so by the determinism of transitions in $\mathcal{A}$, $\mathsf{V}_m = \mathsf{V}_{i.bv}$. The result for invariance sets and protocol tags follows from Definition 4.1. □

This lemma says that certain distinct occurrences *must* share value states. Some of the rules in Figure 2 *allow* environment states to be shared by more than one occurrence. For instance, from the **cond-test-inh** rule, the value states for a conditional and its test-part have the same domains, and they take the same transitions; so a single value state may be used for both the conditional and the test. Considered another way, the environment states for the conditional and the test are extensionally equal functions.

## 6. ANNOTATING AN EXAMPLE PROGRAM

Consider the open term

$\mathbf{let}\ f = (\lambda x.\lambda y.z\ (\mathbf{succ}\ x))$
$\mathbf{in}\ \ f\ c$

We desugar the **let** to obtain a term in $\Lambda_{in}$ :

$(\lambda f.f\ c)\ (\lambda x.\lambda y.z\ (\mathbf{succ}\ x))$

| occurrence index $i$ | subterm $[\![i]\!]$ | predicate | $\mathsf{E}_i$ | $\mathsf{V}_i$ |
|---|---|---|---|---|
| $\epsilon$ | the entire term | *App* | $e_0$ | $v_0$ |
| $rator$ | $\lambda f.f\ c$ | *Proc* | $e_0$ | $v_1$ |
| $rator.bv$ | binding $f$ | — | — | $v_f$ |
| $rator.body$ | $f\ c$ | *App* | $e_f$ | $v_2$ |
| $rator.body.rator$ | $f$ | *Var* | $e_f$ | $v_f$ |
| $rator.body.rand$ | $c$ | *Const* | $e_f$ | $v_3$ |
| $rand$ | $\lambda x.\lambda y.z$ (**succ** $x$) | *Proc* | $e_0$ | $v_4$ |
| $rand.bv$ | binding $x$ | — | — | $v_x$ |
| $rand.body$ | $\lambda y.z$ (**succ** $x$) | *Proc* | $e_x$ | $v_5$ |
| $rand.body.bv$ | binding $y$ | — | — | $v_y$ |
| $rand.body.body$ | $z$ (**succ** $x$) | *App* | $e_y$ | $v_6$ |
| $rand.body.body.rator$ | $z$ | *Var* | $e_y$ | $v_z$ |
| $rand.body.body.rand$ | **succ** $x$ | *PrimApp* | $e_y$ | $v_7$ |
| $rand.body.body.rand.rator$ | **succ** | — | — | — |
| $rand.body.body.rand.rand$ | $x$ | *Var* | $e_y$ | $v_x$ |

Fig. 6. Occurrences and annotations.

If we evaluate this term in an environment with a binding for $z$, that binding is available at the call site $f\ c$. We shall see that our analysis and closure conversion algorithm make $z$ a dynamic variable at that site.

In Figure 6, we present a table describing each occurrence in the term. There we write $\epsilon$ to indicate the empty string.

*Environment States.* As we suggested in our sketch of the solution algorithm, we generate an environment state for each distinct scope in the program. So we generate four environment states, $e_0$, $e_f$, $e_x$, and $e_y$: $e_0$ for the the outermost scope; $e_f$ for the scope inside the binding $f$; $e_x$ for the scope inside the binding $x$; and $e_y$ for the scope inside the binding $y$.

For each occurrence $i$ in the program (other than binding variables and primitive operators), we define $\mathsf{E}_i$ to be the environment state for the scope containing $i$. For instance, $\mathsf{E}_\epsilon$, $\mathsf{E}_{rator}$, and $\mathsf{E}_{rand}$ are all defined to be $e_0$. Figure 6 indicates the environment states associated with occurrences.

*Value States.* We first generate a value state for each free variable in the program and a value state for each binder. This gives us four value states, $v_z$, $v_f$, $v_x$, and $v_y$. For each occurrence $i$ of a variable in the program, either free or bound, we define $\mathsf{V}_i$ to be the value state $v_{[\![i]\!]}$. So for instance, we assign $\mathsf{V}_{rator.body.rator} = v_f$ and $\mathsf{V}_{rand.body.body.rator} = v_z$. We also generate a value state for each remaining occurrence in the program, other than primitive operators. For each such remaining occurrence $i$, we define $\mathsf{V}_i$ to be $v_n$, for some integer $n$. Figure 6 indicates the value states associated with occurrences.

*Closure Analysis: Protocol Tag Equivalence Classes.* Now we can apply the rules in Figure 2 and Figure 4. Because environment states are shared among occurrences in the same scope, each of the **-inh** rules in Figure 2, except **proc-body-inh**, is satisfied automatically. Therefore, we do not consider those rules further.

Some rules have no premises, so we may apply them immediately. By **var**, we have

(1) $\mathsf{E}_{rator.body.rator} \xrightarrow{f} \mathsf{V}_{rator.body.rator}$
(2) $\mathsf{E}_{rand.body.body.rator} \xrightarrow{z} \mathsf{V}_{rand.body.body.rator}$
(3) $\mathsf{E}_{rand.body.body.rand.rand} \xrightarrow{x} \mathsf{V}_{rand.body.body.rand.rand}$

so we add the transitions

(4) $e_f \xrightarrow{f} v_f$
(5) $e_y \xrightarrow{z} v_z$
(6) $e_x \xrightarrow{x} v_x$

By **proc-self** we have

(7) $\mathsf{V}_{rator} \xrightarrow{rator} \mathsf{E}_{rator}$
(8) $\mathsf{V}_{rand} \xrightarrow{rand} \mathsf{E}_{rand}$
(9) $\mathsf{V}_{rand.body} \xrightarrow{rand.body} \mathsf{E}_{rand.body}$

So we add

(10) $v_1 \xrightarrow{rator} e_0$
(11) $v_4 \xrightarrow{rand} e_0$
(12) $v_5 \xrightarrow{rand.body} e_x$

By **proc-body-ext**:

(13) $\mathsf{E}_{rator.body} \xrightarrow{f} \mathsf{V}_{rator.bv}$
(14) $\mathsf{E}_{rand.body} \xrightarrow{x} \mathsf{V}_{rand.bv}$
(15) $\mathsf{E}_{rand.body.body} \xrightarrow{y} \mathsf{V}_{rand.body.bv}$

We add

(16) $\mathsf{E}_{e_f} \xrightarrow{f} \mathsf{V}_{v_f}$
(17) $\mathsf{E}_{e_x} \xrightarrow{x} \mathsf{V}_{v_x}$
(18) $\mathsf{E}_{e_y} \xrightarrow{y} \mathsf{V}_{v_y}$

Note that (16) is the same as (4), and (17) is the same as (6).
From (7), we may already conclude that

(19) $\pi_{rator.bv} = \pi_{rand}$ **[app-tag-bv-rand]**
(20) $\pi_{rator.body} = \pi_{\epsilon}$ **[app-tag-body-app]**

An occurrence $i$ of a variable or binding variable has a value state $v_{[\![i]\!]}$, so we will write $\pi_{[\![i]\!]}$ for its protocol tag and $\theta_{[\![i]\!]}$ for its invariance set. Accordingly, we write (19) as

(21) $\pi_f = \pi_{rand}$

Proceeding with the closure analysis, we now consider the rules with premises. We first apply the **proc-body-inh** rule:

(22) $e_x \xrightarrow{z} v_z$ $\quad [(5),\ \mathsf{E}_{rand.body.body} = e_y,\ \mathsf{E}_{rand.body} = e_x]$
(23) $e_y \xrightarrow{x} v_x$ $\quad [(6),\ \mathsf{E}_{rand.body} = e_x,\ \mathsf{E}_{rand.body.body} = e_y]$
(24) $e_0 \xrightarrow{z} v_z$ $\quad [(22),\ \mathsf{E}_{rand.body} = e_x,\ \mathsf{E}_{rand} = e_0]$
(25) $e_f \xrightarrow{z} v_z$ $\quad [(24),\ \mathsf{E}_{rator} = e_0,\ \mathsf{E}_{rator.body} = e_f]$

We have applied to exhaustion those rules with transitions from environment states in their premises (those with names ending in **-inh**). The remaining rules cannot deduce such transitions, so we will not have to examine these rules again.

Continuing, by **app-rand-bv** we have

(26) $\mathsf{V}_{rator.bv} \xrightarrow{rand} \mathsf{E}_{rand}$ $\quad [(7),\ (8)]$

which is the same as

(27) $v_f \xrightarrow{rand} e_0$

We may write this again as

(28) $\mathsf{V}_{rator.body.rator} \xrightarrow{rand} \mathsf{E}_{rand}$

so by **app-body-app**, we have

(29) $\mathsf{V}_{rator.body} \xrightarrow{rand.body} \mathsf{E}_{rand.body}$ $\quad [(9),\ (28)]$
(30) $\mathsf{V}_{\epsilon} \xrightarrow{rand.body} \mathsf{E}_{rand.body}$ $\quad [(7),\ (29)]$

We may write these last two transitions as

(31) $v_2 \xrightarrow{rand.body} e_x$
(32) $v_0 \xrightarrow{rand.body} e_x$

We have now applied all the deduction rules in Figure 2 to exhaustion, completing the closure analysis. Note that $Dom(\mathsf{E}_{\epsilon}) = Dom(e_0) = \{z\}$. By Lemma 4.5, any scalar environment with a binding for $z$ will satisfy $\mathsf{E}_{\epsilon}$.

We can still make more deductions for protocol tags. From (28), we also have

(33) $\pi_{rand.bv} = \pi_{rator.body.rand}$ $\quad$ [**app-tag-bv-rand**]
(34) $\pi_{rand.body} = \pi_{rator.body}$ $\quad$ [**app-tag-body-app**]

Let us rewrite (33) as

(35) $\pi_x = \pi_{rator.body.rand}$

Now we have applied the rules in Figure 4 to exhaustion. We compute the equivalence classes of protocol tags:

$$\{\pi_\epsilon, \pi_{rator.body}, \pi_{rand.body}\}$$
$$\{\pi_f, \pi_{rand}\}$$
$$\{\pi_x, \pi_{rator.body.rand}\}$$

*Invariance Sets.* The rules tell us which variables do *not* belong to invariance sets. The set of all variables in the program is $\{f, v, x, z\}$ ($y$ occurs only as a binder).

We have

(36) $f, v, x \notin \theta_{rator}$ [**proc-inv**, $\mathsf{E}_{rator} = e_0$, (24)]
(37) $f, v, x \notin \theta_{rand}$ [**proc-inv**, $\mathsf{E}_{rand} = e_0$, (24)]
(38) $f, v \notin \theta_{rand.body}$ [**proc-inv**, $\mathsf{E}_{rand.body} = e_x$, (6) or (17), (22)]

Since the domains of environment states were computed during closure analysis, we will not need to use rule **proc-inv** again.

Continuing, we have

(39) $f, v, x \notin \theta_\epsilon$ [**app-inv-rator**, (36)]
(40) $f, v, x \notin \theta_{rator.bv}$ [**app-inv-rator-bv**, (7), (36)]
(41) $f, v, x \notin \theta_{rator.body.rator}$ [$\theta_{rator.body.rator} = \theta_f = \theta_{rator.bv}$, (40)]
(42) $f, v, x \notin \theta_{rator.body}$ [**app-inv-rator**, (41)]
(43) $f, v, x \notin \theta_{rand.bv}$ [**app-inv-rator-bv**, (28), (41)]

Invariance sets only matter for the operators of applications. From our deductions here, we assign

(44) $\theta_{rator} = \{z\}$ [(36)]
(45) $\theta_{rator.body.rator} = \theta_f = \{z\}$ [(41)]
(46) $\theta_{rand.body.body.rator} = \theta_z = \{f, v, x, z\}$ [no constraints]

*Constraint Solution.* Finally, we assign protocol tags. Within equivalence classes, the protocol tag depends on the invariance sets for application operators and the free variables of procedures.

$\pi_{rator}$ has no equality constraints, and $\theta_{rator} = \{z\}$ (44). By the rule **app-tag-cons**, we might choose to assign $\pi_{rator} = \langle z \rangle$. However, $z$ is not free in $[\![rator]\!] = \lambda f.fc$. So instead, we assign $\pi_{rator} = \langle\,\rangle$.

Now, $\theta_{rator.body.rator} = \{z\}$ (45), and $\pi_{rator.body.rator} = \pi_f = \pi_{rator.bv}$. Also, $\pi_{rand}$ is in the same equivalence class as $\pi_f$. Neither *rator.bv* nor *rand* is an application operator, but $[\![rand]\!]$ is a procedure. $z$ occurs free in $[\![rand]\!]$, so we assign $\pi_f = \langle z \rangle$. We also assign $\langle z \rangle$ to $\pi_{rand}$.

The other application operator is *rand.body.body.rator*. There are no equality constraints on its protocol tag, which is $\pi_z$. Since there are no procedures required

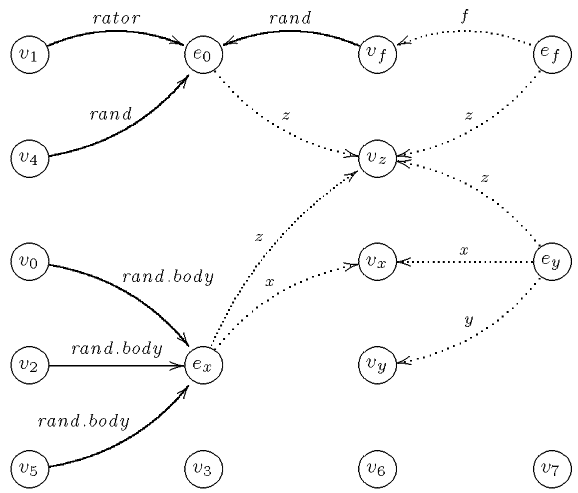


Fig. 7. Closure analysis.

to have the same protocol tag, we assign $\pi_z = \langle\,\rangle$.

The transformation depends on the protocol tags for application operators and procedures. We have assigned tags to all such occurrences, except the procedure *rand.body*. None of the tags in the equivalence class $[\pi_{rand.body}]$ is associated with an application operator. Since we do not know that *rand.body* is able to pick up its free variables at an application site, we assign $\pi_{rand.body} = \langle\,\rangle$. The other elements of the equivalence class, $\pi_\epsilon$ and $\pi_{rator.body}$, are also assigned the empty sequence.

*Summary of results.* We now summarize the results of the analysis.

In Figure 7, we show the transitions of the abstract execution. Solid arrows indicate transitions from value states to environment states; dotted arrows indicate transitions from environment states to value states. Note that, of $v_f$, $v_x$, $v_y$, and $v_z$, there is a transition only from $v_f$ (27). The way to interpret this is that $f$ may be bound to a procedure, but $x$, $y$, and $z$ cannot (given an environment satisfying $\mathsf{E}_\epsilon$). The lone transition from $v_f$ is labeled with *rand*, so $f$ may only be bound to the procedure with index *rand*, namely, $\lambda x.\lambda y.z$ $(\textbf{succ}\ x)$. The target of that transition is $e_0$, which describes the environment closing the procedure.

Figure 8 summarizes the results for invariance sets and protocol tags. Observe that for each of the three application operators, $\epsilon$, *rator.body.rator*, and *rand.body.body.rator*, the protocol tag contains only elements in the invariance set.

## 7. SOUNDNESS

Now we show that any monovariant and locally consistent annotation of a program makes the annotations valid. The Soundness Theorem says that if the environment $\psi$ — the input — satisfies $\mathsf{E}_i$, the environment state for $i$, and $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$, then $(j, \psi')$ — the output — satisfies $\mathsf{V}_i$, the value state for $i$; moreover, similar satisfaction relations hold for every subcomputation.

| occurrence index $i$ | $\theta_i$ | $\pi_i$ |
|---|---|---|
| $\epsilon$ | $\{z\}$ | $\langle\,\rangle$ |
| $rator$ | $\{z\}$ | $\langle\,\rangle$ |
| $rator.bv$ | $\{z\}$ | $\langle z\rangle$ |
| $rator.body$ | $\{z\}$ | $\langle\,\rangle$ |
| $rator.body.rator$ | $\{z\}$ | $\langle z\rangle$ |
| $rator.body.rand$ | $\{f,v,x,z\}$ | $\langle\,\rangle$ |
| $rand$ | $\{z\}$ | $\langle z\rangle$ |
| $rand.bv$ | $\{z\}$ | $\langle\,\rangle$ |
| $rand.body$ | $\{x,z\}$ | $\langle\,\rangle$ |
| $rand.body.bv$ | $\{f,v,x,z\}$ | $\langle\,\rangle$ |
| $rand.body.body$ | $\{f,v,x,z\}$ | $\langle\,\rangle$ |
| $rand.body.body.rator$ | $\{f,v,x,z\}$ | $\langle\,\rangle$ |
| $rand.body.body.rand$ | $\{f,v,x,z\}$ | $\langle\,\rangle$ |
| $rand.body.body.rand.rator$ | — | — |
| $rand.body.body.rand.rand$ | $\{f,v,x,z\}$ | $\langle\,\rangle$ |

Fig. 8. Invariance sets and protocol tags.

THEOREM 7.1 (SOUNDNESS). *Let $\mathcal{A}$ be an abstract execution. Let $M$ be a term, and let $\psi$ be an occurrence environment. Let $\Gamma$ be a monovariant and locally consistent annotation map from $(M,\psi)$ to $\mathcal{A}$, and let $\Pi$ be the protocol assignment defined by $\forall i,\ \Pi(i) = \pi_{\Gamma,i}$. Let $i$ be an occurrence in $M$ and suppose $\psi \models^{\mathcal{A},\Pi}_{env} \mathsf{E}_{\Gamma,i}$. Suppose $(i,\psi) \underset{oc}{\Longrightarrow} (j,\psi')$. Then for any subproof $(k,\psi'') \underset{oc}{\Longrightarrow} (m,\psi''')$ of this derivation*

*(1) $\psi'' \models^{\mathcal{A},\Pi}_{env} \mathsf{E}_{\Gamma,k}$,*

*(2) $(m,\psi''') \models^{\mathcal{A},\Pi}_{val} \mathsf{V}_{\Gamma,k}$, and*

*(3) if $Proc(m)$, then $\forall x \in \theta_{\Gamma,k}$, $x \in Dom(\psi'') \cap Dom(\psi''')$, and $\psi''(x) = \psi'''(x)$.*

PROOF. Induction on the size of the derivation that $(i,\psi) \underset{oc}{\Longrightarrow} (j,\psi')$.

The base cases are all straightforward. For each case in the induction step, we first show that the consequent holds at the root of the derivation tree. Next we show that the premise about $\models^{\mathcal{A},\Pi}_{env}$ holds for all immediate subproofs of the root, so that the consequent holds at all proper subproofs.

*Case PrimApp(i).* For any primitive operator, the result will be an occurrence closure where the index is a constant occurrence, and the occurrence environment is empty; see the evaluation rules in Figure 1. So suppose we have the evaluation $(i,\psi) \underset{oc}{\Longrightarrow} (c,[\,])$. Since $Const(c)$, by the definition of $\models^{\mathcal{A},\Pi}_{val}$, $(c,[\,]) \models^{\mathcal{A},\Pi}_{val} \mathsf{V}_i$. Also since $Const(c)$, the invariant holds trivially.

By the evaluation rules for primitive operators, there is one immediate subproof, $(i.rand,\psi) \underset{oc}{\Longrightarrow} (j,\psi')$, where $Const(j)$. Since $\Gamma$ is locally consistent, if $\mathsf{E}_{i.rand} \xrightarrow{x} \mathsf{V}$, then also $\mathsf{E}_i \xrightarrow{x} \mathsf{V}$, by the **primapp-rand-inh** rule. Since $\psi \models^{\mathcal{A},\Pi}_{env} \mathsf{E}_i$, by Lemma 4.6, also $\psi \models^{\mathcal{A},\Pi}_{env} \mathsf{E}_{i.rand}$. By the induction hypothesis at $i.rand$, the consequent holds at all proper subproofs.

*Case App(i).* Suppose we have the evaluation

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

Then we can obtain the desired results as follows:

(1) $\psi \models^{\mathcal{A},\Pi}_{env} \mathsf{E}_{i.rator}$ [**app-rator-inh**, Lemma 4.6]

(2) $\psi \models^{\mathcal{A},\Pi}_{env} \mathsf{E}_{i.rand}$ [**app-rand-inh**, Lemma 4.6]

(3) $(j, \psi') \models^{\mathcal{A},\Pi}_{val} \mathsf{V}_{i.rator}$ [IH re $\models^{\mathcal{A},\Pi}_{val}$ at $i.rator$]

(4) $x \in \theta_{i.rator} \Longrightarrow x \in Dom(\psi) \cap Dom(\psi'),\ \psi(x) = \psi'(x)$
[$Proc(j)$, IH for invariance at $i.rator$]

(5) $\mathsf{V}_{i.rator} \xrightarrow{j} \mathsf{E}_j$ [$Proc(j)$, (3), monovariance]

(6) $\psi' \models^{\mathcal{A},\Pi}_{env} \mathsf{E}_j$ [(3), (5), definition of $\models^{\mathcal{A},\Pi}_{env}$]

(7) $(k, \psi'') \models^{\mathcal{A},\Pi}_{val} \mathsf{V}_{i.rand}$ [IH re $\models^{\mathcal{A},\Pi}_{val}$ at $i.rand$]

(8) if $Proc(k)$ then $\mathsf{V}_{i.rand} \xrightarrow{k} \mathsf{E}_k$ [(7), monovariance]

(9) if $Proc(k)$ then $\psi'' \models^{\mathcal{A},\Pi}_{env} \mathsf{E}_k$ [(7), (8), definition of $\models^{\mathcal{A},\Pi}_{env}$]

(10) if $Proc(k)$ then $x \in \theta_{i.rand} \Longrightarrow x \in Dom(\psi) \cap Dom(\psi''),\ \psi(x) = \psi''(x)$
[IH for invariance at $i.rand$]

(11) if $Proc(k)$ then $\mathsf{V}_{j.bv} \xrightarrow{k} \mathsf{E}_k$ [(5), (8), **app-rand-bv**)]

(12) $\pi_{i.rand} = \pi_{j.bv}$ [(5), **app-tag-bv-rand**]

(13) $\pi_{j.body} = \pi_i$ [(5), **app-tag-body-app**]

(14) $\theta_{j.bv} \subseteq \theta_{i.rator}$ [(5), **app-inv-rator-bv**]

(15) $\theta_{j.bv} \subseteq \theta_{i.rand}$ [(5), **app-inv-rand-bv**]

(16) $[\![j.bv]\!] \notin \theta_{j.bv}$ [(5), **app-inv-bv**]

(17) $\theta_i \subseteq \theta_{j.body}$ [(5), **app-inv-body-app**]

(18) $\theta_i \subseteq \theta_{i.rator}$ [**app-inv-rator**]

(19) $[\![j.bv]\!] \notin \theta_i$ [(5), **app-inv-bv-app**]

(20) if $Proc(k)$ then $x \in (\theta_{i.rator} \cap \theta_{i.rand}) \Longrightarrow x \in Dom(\psi') \cap Dom(\psi'')$,
$\psi'(x) = \psi''(x)$
[(4), (10)]

(21) if $Proc(k)$ then $x \in \theta_{j.bv} \Longrightarrow x \in Dom(\psi') \cap Dom(\psi''),\ \psi'(x) = \psi''(x)$
[(14), (15), (20)]

(22) if $Proc(k)$ then $x \in \theta_{j.bv} \Longrightarrow x \in Dom(\psi'[[\![j.bv]\!] \mapsto (k, \psi'')]) \cap Dom(\psi'')$,
$\psi'[[\![j.bv]\!] \mapsto (k, \psi'')](x) = \psi''(x)$
[(16), (21)]

(23) $(k, \psi'') \models^{\mathcal{A},\Pi}_{val} \mathsf{V}_{j.bv}$ [(7), (8), (9), (11), (12)]

(24) if $\mathsf{E}_{j.body} \xrightarrow{x} \mathsf{V}$ then $\mathsf{E}_j \xrightarrow{x} \mathsf{V},\ x \neq [\![j.bv]\!]$
[**proc-body-inh**]

(25) $\mathsf{E}_{j.body} \xrightarrow{[\![j.bv]\!]} \mathsf{V}_{j.bv}$ [**proc-body-ext**]

(26) $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \models^{\mathcal{A},\Pi}_{env} \mathsf{E}_{j.body}$ [(6), (22), (23), (24), (25)]

(27) $(m, \psi''') \models^{\mathcal{A},\Pi}_{val} \mathsf{V}_{j.body}$ [IH re $\models^{\mathcal{A},\Pi}_{val}$ at $j.body$]

(28) if $Proc(m)$ then $\mathsf{V}_{j.body} \xrightarrow{m} \mathsf{E}_m$ [(27), monovariance]

(29) if $Proc(m)$ then $\psi''' \models^{\mathcal{A},\Pi}_{env} \mathsf{E}_m$ [(27), (28), definition of $\models^{\mathcal{A},\Pi}_{env}$]

(30) if $Proc(m)$ then $\mathsf{V}_i \xrightarrow{m} \mathsf{E}_m$ [(5), (28), **app-body-app**]

(31) $(m, \psi''') \models^{\mathcal{A},\Pi}_{val} \mathsf{V}_i$ [(13), (27), and (30)]

(32) if $Proc(m)$ then $x \in \theta_{j.body} \Longrightarrow x \in Dom(\psi'[[\![j.bv]\!] \mapsto (k, \psi'')]) \cap Dom(\psi''')$,
$\psi'[[\![j.bv]\!] \mapsto (k, \psi'')](x) = \psi'''(x)$
[IH for invariance at $j.body$]

(33) if $Proc(m)$ then $x \in (\theta_{j.body} - \{[\![j.bv]\!]\}) \Longrightarrow x \in Dom(\psi') \cap Dom(\psi''')$,
$\psi'(x) = \psi'''(x)$
[(32)]

(34) if $Proc(m)$ then $x \in (\theta_{i.rator} \cap (\theta_{j.body} - \{[\![j.bv]\!]\})) \Longrightarrow$
$x \in Dom(\psi) \cap Dom(\psi''')$,
$\psi(x) = \psi'''(x)$
[(4), (33)]

(35) if $Proc(m)$ then $x \in \theta_i \Longrightarrow x \in Dom(\psi) \cap Dom(\psi'''),\ \psi(x) = \psi'''(x)$
[(17), (18), (19), (34)]

The preceding steps establish the theorem consequent at the root of the derivation tree. We also want to establish the consequent at all subproofs of the derivation tree. There are three immediate subproofs of the root of the proof tree. By step (1), $\psi \models_{env}^{\mathcal{A},\Pi} \mathsf{E}_{i.rator}$ and by step (2), $\psi \models_{env}^{\mathcal{A},\Pi} \mathsf{E}_{i.rand}$. Therefore, the consequent holds at all subproofs of the evaluations of the operator and the operand. The other immediate subproof evaluates the body of the procedure that is the result of evaluating the operator. Similarly, by step (26), $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \models_{env}^{\mathcal{A},\Pi} \mathsf{E}_{j.body}$, so the consequent holds at all subproofs of the evaluation of the procedure body.

*Case Cond(i).* There are two subcases, depending on the result of the test. We show only the case where the test is true.

Suppose we have the evaluation

$$\frac{(i.test, \psi) \underset{oc}{\Longrightarrow} (j, \psi'),\ [\![j]\!] = \mathbf{true} \quad (i.then, \psi) \underset{oc}{\Longrightarrow} (k, \psi'')}{(i, \psi) \underset{oc}{\Longrightarrow} (k, \psi'')}$$

By the **cond-then-inh** rule, if $\mathsf{E}_{i.then} \xrightarrow{x} \mathsf{V}$, for some $\mathsf{V}$, then $\mathsf{E}_i \xrightarrow{x} \mathsf{V}$. By assumption, $\psi \models_{env}^{\mathcal{A},\Pi} \mathsf{E}_i$, so by Lemma 4.6, also $\psi \models_{env}^{\mathcal{A},\Pi} \mathsf{E}_{i.then}$. By the induction hypothesis at $i.then$ we have

$$(k, \psi'') \models_{val}^{\mathcal{A},\Pi} \mathsf{V}_{i.then}$$

By the rule **cond-tag**, $\pi_i = \pi_{i.then}$, and by rule **cond-then-inh**, if $\mathsf{V}_{i.then} \xrightarrow{m} B$, then $\mathsf{V}_i \xrightarrow{m} B$. By the definition of $\models_{val}^{\mathcal{A},\Pi}$, we also have

$$(k, \psi'') \models_{val}^{\mathcal{A},\Pi} \mathsf{V}_i$$

Also by the induction hypothesis, if $Proc(k)$, then $\forall x \in \theta_{i.then}, x \in Dom(\psi) \cap Dom(\psi'')$ and $\psi(x) = \psi''(x)$. By the rule **cond-inv-then**, $\theta_i \subseteq \theta_{i.then}$; hence if $Proc(k)$, then $\forall x \in \theta_i, x \in Dom(\psi) \cap Dom(\psi'')$, and $\psi(x) = \psi''(x)$.

There are two immediate subproofs, one for the evaluation of the test, the other for the evaluation of the then-part. We already showed $\psi \models_{env}^{\mathcal{A},\Pi} \mathsf{E}_{i.then}$, so the consequent holds at all subproofs of the evaluation of the then-part. Since $\Gamma$ is locally consistent, by the rule **cond-test-inh**, if $\mathsf{E}_{i.test} \xrightarrow{x} \mathsf{V}$, then $\mathsf{E}_i \xrightarrow{x} \mathsf{V}$; so by Lemma 4.6, $\psi \models_{env}^{\mathcal{A},\Pi} \mathsf{E}_{i.test}$. Therefore, the consequent holds at all subproofs of the evaluation of the test. □

Note that for each use of the Figure 2 **-inh** rules in the proof, we relied only on the "upward" direction. Including the "downward" direction in those rules has the effect of expanding the domains of environment states. We want the domains of environment states to be as large as possible, because that allows larger invariance sets in the **proc-inv** rule in Figure 3.

## 8. AN OUTPUT LANGUAGE

So far, we have shown how to annotate programs and proved that our annotations are sound. Now we are ready to consider the transformation enabled by the annotations.

The closure conversion transformation produces terms in an output language that

$$x \underset{t}{\Longrightarrow} x \qquad c \underset{t}{\Longrightarrow} c \qquad \lambda x.Q \underset{t}{\Longrightarrow} \lambda x.Q$$

$$\mathbf{true} \underset{t}{\Longrightarrow} \mathbf{true} \qquad \mathbf{false} \underset{t}{\Longrightarrow} \mathbf{false}$$

$$\frac{M \underset{t}{\Longrightarrow} c_n}{\mathbf{succ}\ M \underset{t}{\Longrightarrow} c_{n+1}} \qquad \frac{M \underset{t}{\Longrightarrow} c_n}{\mathbf{pred}\ M \underset{t}{\Longrightarrow} c_{n-1}}$$

$$\frac{M \underset{t}{\Longrightarrow} c_0}{\mathbf{zero?}\ M \underset{t}{\Longrightarrow} \mathbf{true}} \qquad \frac{M \underset{t}{\Longrightarrow} c_n, \quad n \neq 0}{\mathbf{zero?}\ M \underset{t}{\Longrightarrow} \mathbf{false}}$$

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

$$\frac{M \underset{t}{\Longrightarrow} \mathbf{true} \quad N \underset{t}{\Longrightarrow} Q}{\mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ P \underset{t}{\Longrightarrow} Q} \qquad \frac{M \underset{t}{\Longrightarrow} \mathbf{false} \quad P \underset{t}{\Longrightarrow} Q}{\mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ P \underset{t}{\Longrightarrow} Q}$$

$$\frac{\forall i, 1 \leq i \leq n, M_i \underset{t}{\Longrightarrow} V_i}{[M_1, \ldots, M_n] \underset{t}{\Longrightarrow} [V_1, \ldots, V_n]} \qquad \frac{M \underset{t}{\Longrightarrow} [V_1, \ldots, V_n, \ldots, V_m]}{\#n\ M \underset{t}{\Longrightarrow} V_n}$$

Fig. 9. Rules for evaluating $\Lambda_{clos}$ terms.

includes records. The output language $\Lambda_{clos}$ is given by the grammar

$$\begin{aligned} M \ ::= \ & x \mid c \mid \mathbf{true} \mid \mathbf{false} \mid \lambda x.M \mid \mathit{PrimOp}\ M \mid MM \mid \\ & \mathbf{if}\ M\ \mathbf{then}\ M\ \mathbf{else}\ M \mid [M, \ldots, M] \mid \#n\ M \end{aligned}$$

where *PrimOp* is defined as in Section 3 and $n$ is a positive integer. The productions for records and field selection are new; otherwise, the grammar is identical to that for $\Lambda_{in}$. We redefine a value to be any constant, boolean value, variable, procedure, or a record of values. Values in $\Lambda_{clos}$ are given by the grammar

$$\mathit{Val} \ ::= \ x \mid c \mid \mathbf{true} \mid \mathbf{false} \mid \lambda x.M \mid [V_1, \ldots, V_n]$$

where the $V$'s themselves range over values. Note that by this definition, the empty record is a value.

In Figure 9, we give the rules for evaluating terms in $\Lambda_{clos}$. Note that any value self-evaluates.

## 9. AN EQUATIONAL REASONING SYSTEM

Here we describe a system for reasoning equationally about terms in $\Lambda_{clos}$. Our motivation is to simplify the proof of the Correctness Theorem (Theorem 11.2).

Let $=_w$ be the smallest binary relation on terms in $\Lambda_{clos}$ closed under the rules in Figure 10. If $M =_w N$, we say that $M$ and $N$ are *weakly equivalent*.

THEOREM 9.1 (COEVALUATION). *If $M =_w N$ then $M \underset{t}{\Longrightarrow} V$ iff $N \underset{t}{\Longrightarrow} V$.*

PROOF. Induction on the definition of $=_w$. □

The converse of Theorem 9.1 is false. Consider the terms $\Omega = (\lambda x.xx)(\lambda x.xx)$ and $\Omega' = (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)$. Neither term reduces to a value. An easy

$$\frac{M = N}{M =_w N} \quad (\alpha)$$

$$(\lambda x.M)V =_w M[V/x] \quad V \text{ a value} \quad (\beta)$$

$$\frac{M =_w N}{N =_w M} \quad (\textbf{sym})$$

$$\frac{M =_w P \quad P =_w N}{M =_w N} \quad (\textbf{trans})$$

$$\frac{M \in \mathit{PrimOp} \quad N =_w P}{MN =_w MP} \quad (\textbf{primop})$$

$$\frac{M =_w M' \quad N =_w N' \quad P =_w P'}{\textbf{if } M \textbf{ then } N \textbf{ else } P =_w \textbf{if } M' \textbf{ then } N' \textbf{ else } P'} \quad (\textbf{cond})$$

$$\frac{M =_w N \quad Z =_w Z'}{MZ =_w NZ'} \quad (\textbf{cong})$$

$$\frac{\forall i,\ 1 \le i \le n,\ P_i =_w Q_i}{[P_i, \ldots, P_n] =_w [Q_i, \ldots, Q_n]} \quad (\textbf{rec})$$

$$\frac{M =_w [M_1, \ldots, M_n, \ldots, M_m]}{\#n\ M =_w M_n} \quad (\textbf{select})$$

Fig. 10. Closure conditions for $=_w$.

induction on the rules for $=_w$ shows that for any pair of $\Lambda_{clos}$ terms $M$ and $N$, if $M =_w N$, then $M =_\beta N$ (treating **if** $-$ **then** $-$ **else** $-$, record constructors $[-, \ldots, -]$, and record selection operators as constants). Since $\Omega$ and $\Omega'$ are not $\beta$-convertible, they are not $=_w$.

## 10. THE CLOSURE CONVERSION TRANSFORMATION

Once we have annotated a program according to the deduction rules, we can transform it according to the algorithm in Figure 11. In the case for $\mathit{Proc}(i)$, the application of $(\lambda\vec{u}.\Phi(i.\mathit{body}))$ is a desugaring of the **let**'s that appeared in our introductory examples.

Suppose we have an annotation map $\Gamma$ from a term $M$ and an environment to an abstract execution. The transformation $\Phi_\Gamma$ is a map from $O_M$, the set of occurrence indices in $M$, to $\Lambda_{clos}$, the language of output terms. The formalism is

$$\Phi_\Gamma : O_M \to \Lambda_{clos}$$

Usually the annotation map $\Gamma$ will be understood from context, so we will write simply $\Phi$, as in Figure 11. For a procedure occurrence, the algorithm uses its associated protocol tag to determine which variables are included in the resulting closure. For an application occurrence, the protocol tag associated with the operator is used to determine which dynamic variables, if any, should be inserted as extra arguments and in what order.

We extend the transformation to occurrence closures by giving a map

$$\hat{\Phi} : O_M \times O_\psi \to \Lambda_{clos}$$

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

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

$$\begin{aligned} Proc(i) \Longrightarrow \Phi(i) = {} & [(\lambda e, v_1, \cdots, v_n[\![i.bv]\!].(\lambda \vec{u}.\Phi(i.body))\ (\#1\ e)\ \cdots\ (\#p\ e)), [\vec{u}]] \\ & \text{where } \langle v_1, \ldots, v_n \rangle = \pi_i, \\ & e \text{ is fresh,} \\ & \lceil \vec{u} \rceil = FV([\![i]\!]) - \{v_1, \ldots, v_n\}, \text{ and} \\ & ||\vec{u}|| = m \end{aligned}$$

$$PrimApp(i) \Longrightarrow \Phi(i) = [\![i.rator]\!]\ \Phi(i.rand)$$

$$\begin{aligned} App(i) \Longrightarrow \Phi(i) = {} & \mathbf{app}\ \Phi(i.rator)\ v_1\ \cdots\ v_n\ \Phi(i.rand) \\ & \text{where } \langle v_1, \ldots, v_n \rangle = \pi_{i.rator} \end{aligned}$$

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

Fig. 11. The closure conversion transformation.

defined as

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

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

We now apply the transformation to the annotated term from Section 6. We have

$$\begin{aligned} & \mathbf{app}\,[(\lambda e, f.\,\mathbf{app}\ f\ z\ c), [\,]] \\ & \quad [(\lambda e', z, x.[(\lambda e'', y.(\lambda x, z.\,\mathbf{app}\ z\ (\mathbf{succ}\ x))\ (\#1\ e'')\ (\#2\ e'')), [x, z]]), [\,]] \end{aligned}$$

As expected, $z$ is inserted as a dynamic variable at the call site $f\ c$.

## 11. CORRECTNESS

How do we know that the closure conversion transformation produces correct code? In words: when we evaluate a closure-converted program, the result is the transform of the original answer.

The Correctness Theorem states this property formally. Before we can prove that result, we need the following:

Lemma 11.1. *For an occurrence environment $\psi$, if $x \in Dom(\psi)$, then $\hat{\Phi}(\psi(x))$ is a value in $\Lambda_{clos}$.*

Proof. The proof follows by induction on the depth of occurrence environments. Let $\psi(x) = (j, \psi')$. The only difficult case is for procedures.

Suppose $Proc(j)$. We have $\hat{\Phi}(j, \psi') = \Phi(j)\{\hat{\Phi} \circ \psi'\}$. Suppose $\pi_j = \langle v_1, \ldots, v_n \rangle$. Then

$$\Phi(j)\{\hat{\Phi} \circ \psi'\} = [(\lambda e, \vec{v}, [\![j.bv]\!].(\lambda \vec{u}.\Phi(j.body))\ (\#1\ e) \cdots (\#p\ e)), [\vec{u}]]\{\hat{\Phi} \circ \psi'\}$$

where $e$ is fresh, $\lceil \vec{u} \rceil = FV([\![j]\!]) - \{v_1, \ldots, v_n\}$, and $m = ||\vec{u}||$.

Applying the substitution to the record means applying it to each record field. The first field is a procedure, so we get a substitution instance of the procedure, which is a value. Applying the substitution to the second field, we get either an empty record, which is a value, or by the induction hypothesis, a record of values, which is also a value. Hence, the entire record is a value. □

Now we can show the Correctness Theorem.

THEOREM 11.2 (CORRECTNESS). *Let $\mathcal{A}$ be an abstract execution. Let $M$ be a term, and let $\psi$ be an occurrence environment. Let $\Gamma$ be a monovariant and locally consistent annotation map from $(M, \psi)$ to $\mathcal{A}$, and let $\Pi$ be the protocol assignment defined by $\forall i,\ \Pi(i) = \pi_{\Gamma,i}$. Let $i$ be an occurrence index in $M$, and suppose $\psi \models^{\mathcal{A},\Pi}_{env} \mathsf{E}_{\Gamma,i}$. If*

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

*then*

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

We illustrate this theorem by the commutative diagram:

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

PROOF. Follows by induction on the size of the derivation that $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$. The only difficult case is for applications. Suppose $App(i)$, and we have the derivation

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

We want to show that $\hat{\Phi}(i, \psi) \underset{t}{\Longrightarrow} \hat{\Phi}(m, \psi''')$. We use our equational reasoning system to find a term that coevaluates with the transform, and we show that the term evaluates to $\hat{\Phi}(m, \psi''')$.

Suppose $\pi_{i.rator} = \langle v_1, \ldots, v_n \rangle$. We have

$$\begin{aligned}
\hat{\Phi}(i, \psi) &= \Phi(i)\{\hat{\Phi} \circ \psi\} \\
&= (\mathbf{app}\ \Phi(i.rator)\ v_1\ \cdots\ v_n\ \Phi(i.rand))\{\hat{\Phi} \circ \psi\} \\
&= \mathbf{app}\ \hat{\Phi}(i.rator, \psi)\ \hat{\Phi}(\psi(v_1))\ \cdots\ \hat{\Phi}(\psi(v_n))\ \hat{\Phi}(i.rand, \psi)
\end{aligned}$$

We sketch what the proof tree for the evaluation of $\hat{\Phi}(i, \psi)$ should look like in Figure 12. Since we have not yet shown that the evaluation is valid, a question mark appears to the left of the proof tree. In order to use the application rule, $\mathbf{app}\ \hat{\Phi}(i.rator, \psi)\ \hat{\Phi}(\psi(v_1))\ \cdots\ \hat{\Phi}(\psi(v_n))$ should evaluate to some procedure, which we have written as $\lambda x.M$ in the proof tree.

Rather than work our way through the steps in this proof tree, we will reason equationally, using the $=_w$ relation. Also, instead of reasoning with $\hat{\Phi}(i, \psi)$ itself, we present a term that co-evaluates with $\hat{\Phi}(i, \psi)$, and we work with that term.

$$\mathbf{app}\ \hat\Phi(i.rator,\psi)\ \hat\Phi(\psi(v_1))\ \cdots\ \hat\Phi(\psi(v_n))\ \hat\Phi(i.rand,\psi)$$
$$\mathbf{app}\ \hat\Phi(i.rator,\psi)\ \hat\Phi(\psi(v_1))\ \cdots\ \hat\Phi(\psi(v_n))$$
$$\mathbf{app}\ \hat\Phi(i.rator,\psi)$$
$$\mathbf{app}$$
$$\Downarrow$$
$$\mathbf{app}$$
$$\hat\Phi(i.rator,\psi)$$
$$\Downarrow \text{ induction hypothesis at } i.rator$$
$$\hat\Phi(j,\psi')$$
$$((\#1\ r)\ (\#2\ r))[\hat\Phi(j,\psi')/r]$$
$$\Downarrow \cdots$$
$$\cdots$$
$$\cdots$$
$$?$$
$$(\lambda x.M)$$
$$\hat\Phi(i.rand,\psi)$$
$$\Downarrow \text{ induction hypothesis at } i.rand$$
$$\hat\Phi(k,\psi'')$$
$$M[\hat\Phi(k,\psi'')/x]$$
$$\Downarrow \cdots$$
$$\hat\Phi(m,\psi''')$$
$$\hat\Phi(m,\psi''')$$

Fig. 12. Evaluating an application.

Consider the proof tree in Figure 12. By an easy induction on the definition of $\underset{t}{\Longrightarrow}$, only values may appear on the right of $\underset{t}{\Longrightarrow}$, so $\hat\Phi(j,\psi')$ and $\hat\Phi(k,\psi'')$ must be values that self-evaluate. Therefore, we can replace $\hat\Phi(i.rator,\psi)$ by $\hat\Phi(j,\psi')$, and $\hat\Phi(i.rand,\psi)$ by $\hat\Phi(k,\psi'')$, with no effect on the final result.

So we have

$$\hat\Phi(i,\psi) \underset{t}{\Longrightarrow} \hat\Phi(m,\psi''')$$
$$\text{iff}$$
$$\mathbf{app}\ \hat\Phi(j,\psi')\ \hat\Phi(\psi(v_1))\ \cdots\ \hat\Phi(\psi(v_n))\ \hat\Phi(k,\psi'') \underset{t}{\Longrightarrow} \hat\Phi(m,\psi''')$$

We can now use equational reasoning to simplify the term to evaluate. By the $\beta$-rule for $=_w$, we have

$$\begin{array}{rcl}\mathbf{app}\ \hat\Phi(j,\psi') & = & (\lambda r.(\#1\ r)\ (\#2\ r))\ \hat\Phi(j,\psi') \\ & =_w & ((\#1\ r)\ (\#2\ r))[\hat\Phi(j,\psi')/r] \\ & = & (\#1\ \hat\Phi(j,\psi'))\ (\#2\ \hat\Phi(j,\psi'))\end{array}$$

since $\hat\Phi(j,\psi')$ is a value.

By Theorem 7.1, $(j,\psi') \models^{\mathcal{A},\Pi}_{val} \mathsf{V}_{i.rator}$, so $\Pi(j) = \pi_j = \pi_{i.rator} = \langle v_1,\ldots,v_n\rangle$. From the occurrence closure derivation, we know that $Proc(j)$, so $\Phi(j)$ is a closure record. Simplifying further, we have

$$
\begin{array}{ll}
 & (\#1\ \hat{\Phi}(j,\psi'))\ (\#2\ \hat{\Phi}(j,\psi')) \\
= & (\#1\ \Phi(j)\{\hat{\Phi}\circ\psi'\})\ (\#2\ \Phi(j)\{\hat{\Phi}\circ\psi'\}) \\
=_w & (\lambda e, v_1,\cdots,v_n,[\![j.bv]\!].(\lambda\vec{u}.\Phi(j.body))\ (\#1\ e)\cdots(\#p\ e))\{\hat{\Phi}\circ\psi'\}\ [\vec{u}]\{\hat{\Phi}\circ\psi'\} \\
= & (\lambda e.(\lambda v_1,\cdots,v_n,[\![j.bv]\!].(\lambda\vec{u}.\Phi(j.body))\ (\#1\ e)\cdots(\#p\ e))\{\hat{\Phi}\circ\psi'\})\ [\vec{u}]\{\hat{\Phi}\circ\psi'\} \\
=_w & (\lambda v_1,\cdots,v_n,[\![j.bv]\!].(\lambda\vec{u}.\Phi(j.body))\ (\#1\ e)\cdots(\#p\ e))\{\hat{\Phi}\circ\psi'\}[[\vec{u}]\{\hat{\Phi}\circ\psi'\}/e] \\
= & (\lambda v_1,\cdots,v_n,[\![j.bv]\!].(\lambda\vec{u}.\Phi(j.body))\ (\#1\ [\vec{u}])\cdots(\#p\ [\vec{u}]))\{\hat{\Phi}\circ\psi'\}
\end{array}
$$

where $e$ is fresh, $\lceil\vec{u}\rceil = FV([\![j]\!]) - \{v_1,\ldots,v_n\}$, and $p = \|\vec{u}\|$.

So far we have shown

$$
\begin{array}{ll}
 & \mathbf{app}\ \hat{\Phi}(j,\psi') \\
=_w & (\#1\hat{\Phi}(j,\psi'))\ (\#2\hat{\Phi}(j,\psi')) \\
=_w & (\lambda v_1,\cdots,v_n,[\![j.bv]\!].(\lambda\vec{u}.\Phi(j.body))\ (\#1\ [\vec{u}])\cdots(\#p\ [\vec{u}]))\{\hat{\Phi}\circ\psi'\}
\end{array}
$$

Let $N$ be the term $\lambda[\![j.bv]\!].(\lambda\vec{u}.\Phi(j.body))\ (\#1\ [\vec{u}])\cdots(\#p\ [\vec{u}])$. By the **trans** rule and $n+1$ applications of the **cong** rule, we have

$$
\begin{array}{c}
\mathbf{app}\ \hat{\Phi}(j,\psi')\ \hat{\Phi}(\psi(v_1))\ \cdots\ \hat{\Phi}(\psi(v_n))\ \hat{\Phi}(k,\psi'') \\
=_w \\
(\lambda v_1,\cdots,v_n.N)\{\hat{\Phi}\circ\psi'\}\ \hat{\Phi}(\psi(v_1))\ \cdots\ \hat{\Phi}(\psi(v_n))\ \hat{\Phi}(k,\psi'')
\end{array}
$$

By Lemma 11.1, each term in $\hat{\Phi}(\psi(v_1))\ \cdots\ \hat{\Phi}(\psi(v_n))$ is a value, and we can repeatedly invoke the $\beta$ and **cong** rules for $=_w$, as follows:

$$
\begin{array}{ll}
 & (\lambda v_1,\cdots,v_n.N)\{\hat{\Phi}\circ\psi'\}\ \hat{\Phi}(\psi(v_1))\ \cdots\ \hat{\Phi}(\psi(v_n)) \\
= & (\lambda v_1'.(\lambda v_2,\cdots,v_n.N)[v_1'/v_1])\{\hat{\Phi}\circ\psi'\}\ \hat{\Phi}(\psi(v_1))\ \cdots\ \hat{\Phi}(\psi(v_n)) \\
= & (\lambda v_1'.(\lambda v_2,\cdots,v_n.N)[v_1'/v_1]\{\hat{\Phi}\circ\psi'\})\ \hat{\Phi}(\psi(v_1))\ \cdots\ \hat{\Phi}(\psi(v_n)) \\
=_w & (\lambda v_2,\cdots,v_n.N)[v_1'/v_1]\{\hat{\Phi}\circ\psi'\}[\hat{\Phi}(\psi(v_1))/v_1']\ \hat{\Phi}(\psi(v_2))\ \cdots\ \hat{\Phi}(\psi(v_n)) \\
= & (\lambda v_2,\cdots,v_n.N)\{\hat{\Phi}\circ\psi'[v_1\mapsto\psi(v_1)]\}\ \hat{\Phi}(\psi(v_2))\ \cdots\ \hat{\Phi}(\psi(v_n)) \\
 & \cdots \\
=_w & N\{\hat{\Phi}\circ\psi'[v_1,\ldots,v_n\mapsto\psi(v_1),\ldots,\psi(v_n)]\} \\
= & N\{\hat{\Phi}\circ\psi'[v_1,\ldots,v_n\mapsto\psi'(v_1),\ldots,\psi'(v_n)]\} \\
= & N\{\hat{\Phi}\circ\psi'\}
\end{array}
$$

The first step is an explicit $\alpha$-conversion where $v_1'$ is fresh. Each $=_w$ step represents a use of the $\beta$-rule plus some number of uses of the **cong** rule. Each $=$ representing syntactic equality can be replaced by $=_w$, so by repeated application of the **trans** rule, the first term is weakly equivalent to the last.

In the last two steps lies the key to lightweight closure conversion; here is where we use our invariance sets. Since $(i.rator,\psi)\underset{oc}{\Longrightarrow}(j,\psi')$ and $Proc(j)$, by Theorem 7.1, $\forall x\in\theta_{i.rator}$, $x\in Dom(\psi)\cap Dom(\psi')$, and $\psi(x)=\psi'(x)$. By the **app-tag-cons** rule, since $\pi_{i.rator}=\langle v_1,\ldots,v_n\rangle$, then $\{v_1,\ldots,v_n\}\subseteq\theta_{i.rator}$. So in the next-to-last step, in the extended $\psi'$, we can replace $\psi(v_1),\ldots,\psi(v_n)$ by $\psi'(v_1),\ldots,\psi'(v_n)$.

Another $\alpha$-conversion allows the substitution to be moved inside the body of the procedure:

$$\begin{array}{l} = N\{\hat{\Phi} \circ \psi'\} \\ = (\lambda[\![j.bv]\!].(\lambda\vec{u}.\Phi(j.body))\ (\#1\ [\vec{u}]) \cdots (\#p\ [\vec{u}]))\{\hat{\Phi} \circ \psi'\} \\ = (\lambda x'.((\lambda\vec{u}.\Phi(j.body))\ (\#1\ [\vec{u}]) \cdots (\#p\ [\vec{u}]))[x'/[\![j.bv]\!]])\{\hat{\Phi} \circ \psi'\} \\ = (\lambda x'.(\lambda\vec{u}.\Phi(j.body))[x'/[\![j.bv]\!]]\ (\#1\ [\vec{u}]) \cdots (\#p\ [\vec{u}]))\{\hat{\Phi} \circ \psi'\} \\ = (\lambda x'.((\lambda\vec{u}.\Phi(j.body))[x'/[\![j.bv]\!]]\ (\#1\ [\vec{u}]) \cdots (\#p\ [\vec{u}]))\{\hat{\Phi} \circ \psi'\}) \end{array}$$

Since $\lceil\vec{u}\rceil \subseteq FV([\![j]\!])$, we know $[\![j.bv]\!]$ is not in $\vec{u}$. Therefore, the substitution of fresh variable $x'$ for $[\![j.bv]\!]$ possibly affects only the inner procedure.

By **cong** and repeated application of **trans**, we have

$$\begin{array}{ll} & \mathbf{app}\ \hat{\Phi}(j, \psi')\ \hat{\Phi}(\psi(v_1))\ \cdots\ \hat{\Phi}(\psi(v_n))\ \hat{\Phi}(k, \psi'') \\ =_w & (\lambda x'.((\lambda\vec{u}.\Phi(j.body))[x'/[\![j.bv]\!]]\ (\#1\ [\vec{u}]) \cdots (\#p\ [\vec{u}]))\{\hat{\Phi} \circ \psi'\})\ \hat{\Phi}(k, \psi'') \end{array}$$

Since $\hat{\Phi}(k, \psi'')$ is a value, we can use the $\beta$-rule again:

$$\begin{array}{ll} & (\lambda x'.((\lambda\vec{u}.\Phi(j.body))[x'/[\![j.bv]\!]]\ (\#1\ [\vec{u}]) \cdots (\#p\ [\vec{u}]))\{\hat{\Phi} \circ \psi'\})\ \hat{\Phi}(k, \psi'') \\ =_w & ((\lambda\vec{u}.\Phi(j.body))[x'/[\![j.bv]\!]]\ (\#1\ [\vec{u}]) \cdots (\#p\ [\vec{u}]))\{\hat{\Phi} \circ \psi'\}[\hat{\Phi}(k, \psi'')/x'] \\ = & (\lambda\vec{u}.\Phi(j.body))\{\hat{\Phi} \circ \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]\} \\ & \qquad (\#1\ [\vec{u}])\{\hat{\Phi} \circ \psi'\} \cdots (\#p\ [\vec{u}])\{\hat{\Phi} \circ \psi'\} \\ = & (\lambda\vec{u}'.\Phi(j.body)[\vec{u}'/\vec{u}])\{\hat{\Phi} \circ \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]\} \\ & \qquad (\#1\ [\vec{u}])\{\hat{\Phi} \circ \psi'\} \cdots (\#p\ [\vec{u}])\{\hat{\Phi} \circ \psi'\} \\ = & (\lambda\vec{u}'.\Phi(j.body)[\vec{u}'/\vec{u}]\{\hat{\Phi} \circ \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]\}) \\ & \qquad (\#1\ [\vec{u}])\{\hat{\Phi} \circ \psi'\} \cdots (\#p\ [\vec{u}])\{\hat{\Phi} \circ \psi'\} \\ = & (\lambda\vec{u}'.\Phi(j.body)[\vec{u}'/\vec{u}]\{\hat{\Phi} \circ \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]\}) \\ & \qquad (\#1\ [\vec{u}]\{\hat{\Phi} \circ \psi'\}) \cdots (\#p\ [\vec{u}]\{\hat{\Phi} \circ \psi'\}) \\ =_w & (\lambda\vec{u}'.\Phi(j.body)[\vec{u}'/\vec{u}]\{\hat{\Phi} \circ \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]\})\ \hat{\Phi}(\psi'(u_1)) \cdots \hat{\Phi}(\psi'(u_m)) \end{array}$$

Each variable in $\vec{u}'$ is fresh. The last step follows by repeated application of the **select** rule, also using the **cong** and **trans** rules.

By Lemma 11.1, each of $\hat{\Phi}(\psi'(u_1)) \cdots \hat{\Phi}(\psi'(u_m))$ is a value. So by repeated application of the $\beta$-rule, and using the **cong** and **trans** rules, we have

$$\begin{array}{ll} & (\lambda\vec{u}'.\Phi(j.body)[\vec{u}'/\vec{u}]\{\hat{\Phi} \circ \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]\})\ \hat{\Phi}(\psi'(u_1)) \cdots \hat{\Phi}(\psi'(u_m)) \\ =_w & \Phi(j.body)[\vec{u}'/\vec{u}]\{\hat{\Phi} \circ \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]\}[\hat{\Phi}(\psi'(u_1)) \cdots \hat{\Phi}(\psi'(u_m))/\vec{u}'] \\ = & \Phi(j.body)[\vec{u}'/\vec{u}]\{\hat{\Phi} \circ \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]\}[\vec{u}\{\hat{\Phi} \circ \psi'\}/\vec{u}'] \\ = & \Phi(j.body)\{\hat{\Phi} \circ \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]\} \end{array}$$

But this last term is the same as

$$\hat{\Phi}(j.body, \psi'[[\![j.bv]\!] \mapsto (k, \psi'')])$$

With much effort, we have shown that

$$\begin{array}{ll} & \mathbf{app}\ \hat{\Phi}(j, \psi')\ \hat{\Phi}(\psi(v_1))\ \cdots\ \hat{\Phi}(\psi(v_n))\ \hat{\Phi}(k, \psi'') \\ =_w & \Phi(j.body, \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]) \end{array}$$

By Theorem 7.1,

$$\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \models^{\mathcal{A},\Pi}_{env} \mathsf{E}_{j.body}$$

So by the induction hypothesis at $j.body$

$$\Phi(j.body, \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]) \underset{t}{\Longrightarrow} \hat{\Phi}(m, \psi''')$$

By Theorem 9.1

$$\mathbf{app}\ \hat{\Phi}(j, \psi')\ \hat{\Phi}(\psi(v_1))\ \cdots\ \hat{\Phi}(\psi(v_n))\ \hat{\Phi}(k, \psi'') \underset{t}{\Longrightarrow} \hat{\Phi}(m, \psi''')$$

Hence

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

□

Corollary 11.3. *Let $\mathcal{A}$ be an abstract execution. Let $M$ be a term, and let $\psi$ be an occurrence environment. Let $\Gamma$ be a monovariant and locally consistent annotation map from $(M, \psi)$ to $\mathcal{A}$, and let $\Pi$ be the protocol assignment defined by $\forall i,\ \Pi(i) = \pi_{\Gamma,i}$. Let $i$ be an occurrence index, and let $\psi$ be an occurrence environment such that $\psi \models^{\mathcal{A},\Pi}_{env} \mathsf{E}_{\Gamma,i}$. If*

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

*and $j$ is the occurrence index of a constant $c$, then*

$$\hat{\Phi}(i, \psi) \underset{t}{\Longrightarrow} c$$

## 12. RELATED WORK

The use of input-output assertions to verify programs began with Floyd [1967]. Like Floyd, our notion of soundness uses input and output assertions (our environment states and value states) that are valid for all subcomputations. Our environment and value states also correspond to the pre- and postconditions used by Hoare [1969] to assert that a program meets a specification. Like Hoare, we do not assume termination of annotated programs; our states are *partial* correctness assertions.

Kildall [1973] described the iterative solution of dataflow equations for a compiler. Aho et al. [1986, Chapter 10, section 10.11] suggest a general framework for solving dataflow equations for programs with basic blocks.

Many researchers have now considered how to perform *closure analysis*, a phrase coined by Sestoft. A closure analysis computes a set of procedures to which an expression may evaluate. In our analysis, the transitions from a value state associated with an occurrence represent the set of procedures to which that occurrence may evaluate. Sestoft [1988] originally used a closure analysis to prove the correctness of a transformation that replaces parameter passing by assignment to global variables. Bondorf [1991] adapted Sestoft's analysis for a subset of Scheme. Shivers' dissertation presented two analyses called 0CFA and 1CFA for a version of Scheme, using an abstract interpretation [Shivers 1991]. 0CFA is directly comparable to our analysis; 1CFA is a finer analysis that indexes abstract closures by their call sites. Ayers [1993] used similar techniques to compute a highly optimized version of 0CFA. Palsberg and Schwartzbach [1995] use a constraint-based closure analysis similar to ours to support *safety analysis*. Stefanescu and Zhou [1994] have formulated a generalized closure analysis framework using abstract interpretation that can handle both mono- and polyvariance. Sabry and Felleisen [1994] compare the results of closure analysis for programs written in direct style with the results from analyzing their continuation-passing style transforms.

Flanagan and Felleisen [1995] have formulated closure analysis using A-normal forms; this technique may offer a way to avoid some of the complexity associated with our occurrence closures. Their closure analysis supported a "soft-typing" transformation, in which some operations that check the types of their arguments at run time are replaced by nonchecking equivalents.

Heintze [1992] analyzed programs using constraints on sets of program values in his dissertation for logic, imperative, and functional languages. His focus was on the algorithmics of solving constraints, rather than using solutions to justify program transformations. Wand [1993] introduced the idea of showing that any solution to a set of constraints yields a correct transformation, in the context of partial evaluation.

The idea of approximating an occurrence closure by an abstract closure comes from Jones [1981]; this is the extension of the idea of record types used in Jones and Muchnick [1982] to an occurrence closure evaluator.

Wand [1992] gave an alternative proof of correctness for a closure conversion algorithm by semantic methods. There, the correctness proof was that, under certain restrictions, if the closure-converted term produced a constant, then the original term produced the same constant.

Landin was the first to describe the use of closures to represent higher-order functions [Landin 1964]. Many compiler writers have described their closure conversion algorithms; see Steele [1978], Kranz [1988], and Appel [1992]. Fradet and Le Métayer [1991] describe how closure creation may be avoided in some cases. Shao and Appel [1994] use linked closures and allocation of closures in registers to reduce heap usage and memory traffic in the SML/NJ compiler. Starting from the work described here, Hannan [1995] has shown how to annotate programs for lightweight closure conversion using a type system described in the ELF theorem prover. Minamide et al. [1996] describe a closure conversion algorithm for ML programs that produces well-typed terms in which closures have existential type.

We have used techniques related to those described in this article for other analyses. The "Ultra-$\beta$" transformation in Steckler [1994, Chapter 5] eliminates redundant binding of procedure parameters; the "selective thunkification" transformation in Steckler [1994, Chapter 6] and Steckler and Wand [1994a] prevents thunking of procedure arguments that are certain to be evaluated under call-by-name; and the analysis in [Steckler 1996] detects local channels in a concurrent, distributed language.

## 13. CONCLUSIONS

We have presented a method for proving the correctness of an optimized closure conversion transformation. Our analysis relies on applying deduction rules to provide program annotations. Some components of our annotations, e.g., the closure analysis given by the transitions of an abstract execution, might be made the subject of an abstract interpretation. Other components, e.g., our invariance sets, do not seem easily expressible in that framework. These other portions, which do not represent individual program states, but *relate* program states, seem to fall into F. Nielson's category of "second-order" analyses [Nielson 1985]. Because we have used similar techniques in a number of other analyses, we think the methods here are of wide applicability.

ACKNOWLEDGMENTS

Thanks to Richard Kelsey for discussing escape analysis and its possible applications. Robert Muller and Jens Palsberg made useful comments on earlier versions of this work. The anonymous reviewers made helpful, detailed comments. Some of the diagrams were typeset using Kristoffer Rose and Ross Moore's XY-pic macros for TEX (see `http://www.brics.dk/~krisrose/Xy-pic.html`).

REFERENCES

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass.

APPEL, A. W. 1992. *Compiling with Continuations.* Cambridge University Press, Cambridge, England.

APPEL, A. W. AND JIM, T. 1989. Continuation-passing, closure-passing style. In *Conference Record of the 16th ACM Symposium on Principles of Programming Languages.* ACM, New York, 293–302.

AUGUSTSSON, L. 1984. A compiler for lazy ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming.* ACM, New York, 218–227.

AYERS, A. E. 1993. Abstract analysis and optimization of Scheme. Ph.D. thesis, MIT, Cambridge, Mass.

BONDORF, A. 1991. Automatic autoprojection of higher-order recursive equations. *Sci. of Comput. Program. 17,* 1-3 (Dec.), 3–34.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages.* ACM, New York, 238–252.

FLANAGAN, C. AND FELLEISEN, M. 1995. Set-based analysis for full scheme and its use in soft-typing. Tech. Rep. COMP TR95-253, Dept. of Computer Science, Rice Univ., Houston, Tex. Oct.

FLOYD, R. W. 1967. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Mathematics.* American Mathematical Society, Providence, R.I.

FRADET, P. AND LE MÉTAYER, D. 1991. Compilation of functional languages by program transformation. *ACM Trans. Program. Lang. Syst. 13,* 1 (Jan.), 21–51.

HANNAN, J. 1995. Type systems for closure conversion. In *Participants' Proceedings of the Workshop on Types for Program Analysis.* Computer Science Dept., Aarhus Univ., Denmark, 48–62. Available as DAIMI PB-493.

HEINTZE, N. 1992. Set based program analysis. Ph.D. thesis, Carnegie-Mellon Univ., Pittsburgh, Pa.

HOARE, C. A. R. 1969. An axiomatic basis for computer programming. *Commun. ACM 12,* 576–580, 583.

JONES, N. D. 1981. Flow analysis of lambda expressions (preliminary version). In *The 8th International Colloquium on Automata, Languages, and Programming.* Lecture Notes in Computer Science, vol. 115. Springer-Verlag, Berlin, 114–128.

JONES, N. D. AND MUCHNICK, S. S. 1982. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Conference Record of the 9th ACM Symposium on Principles of Programming Languages.* ACM, New York, 66–74.

KILDALL, G. 1973. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages.* ACM, New York, 194–206.

KRANZ, D. A. 1988. Orbit: An optimizing compiler for Scheme. Ph.D. thesis, Yale Univ., New Haven, Conn.

LANDIN, P. J. 1964. The mechanical evaluation of expressions. *Comput. J. 6,* 4, 308–320.

MINAMIDE, Y., MORRISETT, G., AND HARPER, R. 1996. Typed closure conversion. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages*. ACM, New York, 271–283.

NIELSON, F. 1985. Program transformations in a denotational setting. *ACM Trans. Program. Lang. Syst. 7,* 3 (July), 359–379.

PALSBERG, J. AND SCHWARTZBACH, M. I. 1994. *Object-Oriented Type Systems*. Wiley Professional Computing. Wiley, Chichester.

PALSBERG, J. AND SCHWARTZBACH, M. I. 1995. Safety analysis versus type inference. *Inf. Comput. 118,* 1 (Apr.), 128–141.

PLOTKIN, G. D. 1975. Call-by-name, call-by-value and the $\lambda$-calculus. *Theor. Comput. Sci. 1*, 125–159.

SABRY, A. AND FELLEISEN, M. 1994. Is continuation-passing useful for data flow analysis? In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*. ACM, New York, 1–12.

SESTOFT, P. 1988. Replacing function parameters by global variables. M.S. thesis, DIKU, Univ. of Copenhagen.

SESTOFT, P. 1991. Analysis and efficient implementation of functional programs. Ph.D. thesis, DIKU, Univ. of Copenhagen, Denmark.

SHAO, Z. AND APPEL, A. W. 1994. Space-efficient closure representations. Tech. Rep. CS-TR-454-94, Princeton Univ., Princeton, N.J. Mar.

SHIVERS, O. 1991. Control-flow analysis of higher-order languages. Ph.D. thesis, Carnegie-Mellon Univ., Pittsburgh, Pa.

STECKLER, P. 1996. Detecting local channels in Distributed Poly/ML. LFCS Rep. ECS-LFCS-96-340, Univ. of Edinburgh, Scotland. Jan.

STECKLER, P. AND WAND, M. 1994a. Selective thunkification. In *Proceedings of the 1st International Static Analysis Symposium*, B. L. Charlier, Ed. Lecture Notes in Computer Science, vol. 864. Springer-Verlag, Berlin, 162–178.

STECKLER, P. AND WAND, M. 1994b. Tracking available values for lightweight closures (summary). In *Proceedings of the Atlantique Workshop on Semantics Based Program Manipulation*, N. Jones and C. Talcott, Eds. DIKU, Univ. of Copenhagen, Denmark, 63–70. Available as DIKU Rep. No. 94/12.

STECKLER, P. A. 1994. Correct higher-order program transformations. Ph.D. thesis, Northeastern Univ., Boston, Mass.

STEELE, G. L. 1978. Rabbit: A compiler for Scheme. M.S. thesis, MIT, Cambridge, Mass. Available as MIT Artificial Intelligence Laboratory Tech. Rep. 474.

STEFANESCU, D. AND ZHOU, Y. 1994. An equational framework for the flow analysis of higher-order functions. In *Proceedings of the 1994 ACM Symposium on Lisp and Functional Programming*. ACM, New York, 318–327.

WAND, M. 1992. Correctness of procedure representations in higher-order assembly language. In *Proceedings of the 7th International Conference on the Mathematical Foundations of Programming Semantics*, S. Brookes, Ed. Lecture Notes in Computer Science, vol. 598. Springer-Verlag, Berlin, 294–311.

WAND, M. 1993. Specifying the correctness of binding-time analysis. *J. of Funct. Program. 3,* 3, 365–387.

WAND, M. AND STECKLER, P. 1994. Selective and lightweight closure conversion. In *Conference Record of the 21st ACM Symposium on Principles of Programming Languages*. ACM, New York, 435–445.

Received December 1995; revised June 1996; accepted September 1996