Selective and Lightweight Closure Conversion

PAUL A. STECKLER
University of Edinburgh
and
MITCHELL WAND
Northeastern University

We consider the problem of selective and 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 each of its possible call sites. We formulate the flow analysis as a deductive system that generates a labelled 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 post-conditions*; 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

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 the conferences *POPL'94* [Wand and Steckler 1994] and *Atlantique* [Steckler and Wand 1994b], and in the first-named author's Ph.D. dissertation [Steckler 1994].
Authors' addresses: P.A. Steckler, Laboratory for the Foundations of Computer Science, Department of Computer Science, James Clerk Maxwell Building, King's Buildings, University of Edinburgh, Edinburgh EH9 3JZ, Scotland; email: steck@dcs.ed.ac.uk; M. Wand, College of Computer Science, Cullinane Hall, Northeastern University, Boston, MA 02115, U.S.A.; email: wand@ccs.neu.edu.

program that are justified by those propositions.

For first-order languages, the justification of such optimizations or transformations has been investigated for many years and is reasonably well-understood. For higher-order languages such as Scheme, 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 λ-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 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 is sometimes called *closure-passing style* [Appel and Jim 1989].

For example, we might convert

$$\begin{array}{l}\textbf{let } f = \lambda xy.\lambda z.(x - y - z)\\ \quad\; g = \lambda h.h\ 3\\ \textbf{in } g(f\ 9\ 7)\end{array}$$

to

$$\begin{array}{l}\textbf{let } f = \lambda xy.[(\lambda ez.\,\textbf{destr}\ e\ (\lambda xy.(x - y - z))), [x, y]]\\ \quad\; g = \lambda h.\,\textbf{app}\ h\ 3\\ \textbf{in } g(f\ 9\ 7)\end{array}$$

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

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

Closure conversion is a crucial step in the compilation process for higher-order languages. 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 -1. Closure conversion may produce a transform whose result is not convertible with the result of the original program. Let us closure convert just the inner procedure in

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

to get

$$\begin{array}{l}\textbf{let } z = 3\\ \textbf{in } \textbf{let } f = \lambda x.[\lambda ey.\,\textbf{destr}\ e\ \lambda xz.x + y + z, [x, z]]\\ \quad\;\; \textbf{in } \textbf{app}\ f\ 4\end{array}$$

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

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

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

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

In general, one might have many different procedure-calling protocols coexisting in the same program. For example, procedures imported from some library might have a different protocol from those defined in the client program, and the compiler must keep these separate. Another example is an interpreter based on denotational semantics, in which one might have radically different representations for environments, ordinary procedures, and continuations, though each of these domains consists of mathematical functions.

Second, 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, others we believe represent a new approach.

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 whether to use ordinary procedure application or closure application. In the case of closure application, the application protocol also indicates which variables may be made dynamic, and in what order those variables appear as arguments. Second, the dynamic variables at a call site must have 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:

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

Variables x and q are free in the procedure f, and y and r are 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. The innermost **let** rebinds q and r, so we have to put those variables in the closures for f and g, respectively.

After closure conversion, we might have:

$$
\begin{array}{l}
\textbf{let } x = \ldots \\
\quad\;\; y = \ldots \\
\textbf{in let } f = [(\lambda exyv.\, \textbf{destr } e\ (\lambda q.\ \ldots\ x\ \ldots\ q\ \ldots), [q]] \\
\qquad\quad\; g = [(\lambda exyz.\, \textbf{destr } e\ (\lambda r.\ \ldots\ y\ \ldots\ r\ \ldots), [r]] \\
\quad \textbf{in let } q = \ldots \\
\qquad\quad\;\;\; r = \ldots \\
\qquad \textbf{in } \textbf{app}\,(\textbf{if } (\textbf{zero? } s) \textbf{ then } f \textbf{ else } g)\ x\ y\ c
\end{array}
$$

Other choices are possible, subject to the condition that f and g obey the same application protocol. For instance, we could have left both f and g as ordinary procedures; or the order of the dynamic variables could have been reversed; 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. As we saw for q and r in the last example, 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 in the scope of that variable. Even though the variable is visible at the call site, it may be bound to the wrong value. Consider:

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

Variable x is in scope at the call site (yf). But there are two calls to g, both of which invoke the scope for x. For the left-hand call, x is bound to c_1; for the right-hand call, x is bound to c_2. During the left-hand call to g, y is bound to a copy of f with its free x bound to c_2. That is, f has escaped from the other invocation of x's scope. Therefore, the dataflow analysis needs to assure that x is not considered a dynamic variable.

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

$$\begin{array}{l}\textbf{let } x = c_1 \\ \textbf{in let } g = (\lambda y.\ \ldots\ x\ \ldots) \\ \quad \textbf{in letrec } f = (\lambda hn.\,\textbf{if } (\textbf{zero? } n) \textbf{ then } (h\ c_2) \textbf{ else } (f\ h\ (\textbf{pred } n)) \\ \qquad \textbf{in } f\ g\ c_3\end{array}$$

The variable x is free in the procedure g, so ordinarily x would appear in g's closure. Now consider the call site $(h\ c_2)$. 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_3. 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:

$$\begin{array}{l}\textbf{let } x = c_1 \\ \textbf{in let } g = [(\lambda exy.\,\textbf{destr } e\ (\lambda.\ \ldots\ x\ \ldots), [\,]] \\ \quad \textbf{in letrec } f = (\lambda hn.\,\textbf{if } (\textbf{zero? } n) \textbf{ then} (\textbf{app } h\ c_2) \textbf{ else} (f\ h\ (\textbf{pred } n)) \\ \qquad \textbf{in } f\ g\ c_3\end{array}$$

and our analysis can give this.

The rest of the paper is organized as follows:

— In section 3, we describe the input language, Λ_{in}, and give an evaluator for terms in the language. We also describe a data structure called an *occurrence*

[1] We can define a procedure in our input language that returns the least fixed point of a recursive function, so we can consider the **letrec** construct in this example as syntactic sugar. Plotkin gives such a procedure Z as $\lambda f.(\lambda x.f(\lambda y.(xx)y))\ (\lambda x.f(\lambda y.(xx)y))$ [Plotkin 1975].

closure that represents Λ_{in} terms, and give an evaluator for such data structures. We present two theorems that relate the evaluators.

— In section 4, we present an equational reasoning system that will be used to simplify the proof of the correctness of the closure conversion transformation.

— In section 5, we describe the syntax and semantics of the annotations given to programs by our analysis. Program annotations are given as the states of a labelled transition system.

— In section 6, we show how program annotations may be derived as the solution to a deductive system generated from an input program text.

— 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 and extend the equational reasoning system to handle terms in the language. We also show how the application of closure records in the output language simulates procedure application.

— In section 9, we give the closure conversion transformation.

— In section 10, we give a correctness result for the transformation.

— In section 11, we present related work.

— Finally, in section 12, we present conclusions.

3. LANGUAGES

The language framework given in this section consists of a source language and its evaluator; some notation for describing occurrences of terms and their closing environments; and a fine-grained evaluator that preserves occurrence and environment source information.

3.1 The source language

We present the source language on which the transformation will operate. The language Λ_{in} is an untyped λ-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. Sometimes we will use the word "constant" to refer either to an integer constant or a boolean constant.

The grammar above describes the abstract structure of Λ_{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.

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

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

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

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

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

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

Fig. 1. Rules for the term evaluator

To emphasize that Λ_{in} may be considered a programming language, we will usually refer to λ-abstractions in Λ_{in} as "procedures." Procedures are fully curried, but we will sometimes write $\lambda\vec{x}.M$ as a notational convenience for $\lambda x_1. \ldots .\lambda x_n.M$. 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 α-congruent, that is, the terms are syntactically identical, except for possibly different names for their bound variables.

Values in Λ_{in} are given by the grammar:

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

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

To evaluate programs in Λ_{in}, we can use the call-by-value *term evaluator* shown in Figure 1. The evaluation relation $\underset{t}{\Longrightarrow}$ is a relation between Λ_{in} terms and values.

3.2 Occurrences of terms

A fundamental drawback of the term evaluator is that substitution discards source information about terms: Given a substituted term, we cannot refer back to the parse tree node that produced it. To solve this problem, we build an *occurrence evaluator* that simulates the term evaluator while preserving term source information.

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

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

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 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 $i.rand$. 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 1. *The following definitions are mutually referential:*

(1) An occurrence closure (i, ψ) *is a pair consisting of an occurrence index and an occurrence environment.*

(2) An occurrence environment ψ *is a finite map from variables to occurrence closures such that* $\forall x \in Dom(\psi)$, *if* $\psi(x) = (i, \psi')$ *then* $[\![i]\!] \in Val$.

This definition is well-founded because the empty map, $\emptyset$, is an occurrence environment. $\psi[x \mapsto (i, \psi')]$ is an extended occurrence environment that agrees on all variables with ψ, except that the extended environment maps x to (i, ψ'). We will sometimes write $\psi[x_1, \ldots, x_n \mapsto v_1, \ldots, v_n]$ to indicate the multiply-extended environment $\psi[x_1 \mapsto v_1] \cdots [x_n \mapsto v_n]$.

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

DEFINITION 2. *An occurrence environment* ψ *is a* scalar environment *iff* $\forall x \in Dom(\psi)$, $\psi(x) = (j, \psi')$ *implies* $[\![j]\!]$ *is a scalar value.*

In particular, the empty environment $\emptyset$ is a scalar environment.

[2]Other approaches used to specify occurrences include generating a unique label for each occurrence [Sestoft 1991], and α-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 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 [Flanagan and Felleisen 1995].

In a chain-free occurrence environment, a single lookup gets the value of a variable. Without the chain-free condition, one might have to follow a chain of lookups to get a variable's value.

DEFINITION 3. *An occurrence environment* ψ *is* chain-free *iff* $\forall x \in Dom(\psi)$, *if* $\psi(x) = (i, \psi')$ *then*

(1) *if* i *is the occurrence index of a variable, then* $[\![i]\!] \notin Dom(\psi')$, *and*
(2) *if* i *is the occurrence index of a procedure, then* ψ' *is chain-free*

Since occurrence environments are finitely deep, this definition is well-founded.

Observe:

LEMMA 1. *If* ψ *is a scalar environment, then* ψ *is chain-free.*

PROOF. By the definitions of scalar environment and chain-freedom. □

When evaluating an occurrence closure (i, ψ), the occurrence evaluator may manipulate occurrences that originate in the environment ψ, 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.

Let us define a set of indices for occurrences that originate in environments. Let K be an infinite set of symbols, disjoint from Σ. An occurrence environment ψ is an *initial environment* iff $\forall x \in Dom(\psi)$, if $\psi(x) = (k, \psi')$ then $k \in K$ and ψ' is an initial environment. This definition is well-founded since it makes the empty environment initial. An occurrence index that has a prefix $k \in K$ 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 infinite and regular set of all such *constant occurrences*. C is disjoint from the set 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:

$$O = C + ((K + \epsilon)\ (rator + rand + body + test + then + else)^*\ (bv + \epsilon))$$

3.3 Relating the term and occurrence evaluators

The occurrence evaluator uses environment extension to simulate the substitutions of the term evaluator. The evaluation relation $\underset{oc}{\Longrightarrow}$ defines a relation between

[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.

occurrence closures that represent arbitrary terms and occurrence closures representing the values of those terms. The rules for the occurrence evaluator are given in Figure 2.

We use the predicates

$$Const,\ Var,\ Proc,\ PrimApp,\ App,\ Cond$$

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 $[\![\cdot]\!]$, 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 $\mathcal{U}[\![\cdot]\!]$ defined by:

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

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

Suppose

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

Then

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

3.5 Simulation and adequacy

The occurrence evaluator was designed to simulate the operation of the call-by-value term evaluator. If we can evaluate an occurrence closure using the occurrence

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

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

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

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

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

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

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

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

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

Fig. 2. Rules for the occurrence evaluator

evaluator, then we can evaluate the term represented by the occurrence closure using the term evaluator; we formalize this notion below. This strategy is easily generalized for "natural-style" term evaluators. For instance, in [Steckler 1994], we built an occurrence evaluator that simulates a call-by-name term evaluator.

To prove the claim of simulation, we need the following result.

LEMMA 2. *If i is the occurrence index of a value, then for any occurrence environment ψ, $\mathcal{U}[\![(i, \psi)]\!]$ is a value.*

PROOF. By induction on the depth of ψ. □

Now we can show that the occcurence evaluator simulates the term evaluator:

THEOREM 1 (SIMULATION). *If $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$ then $\mathcal{U}[\![(i, \psi)]\!] \underset{t}{\Longrightarrow} \mathcal{U}[\![(j, \psi')]\!]$.*

The following commutative diagram illustrates the theorem:

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

PROOF. By induction on the definition of $\underset{oc}{\Longrightarrow}$. □

Why does the converse of the Simulation Theorem fail? There are infinitely many occurrence closures that represent a term. Suppose $[\![(i, \psi)]\!] \underset{t}{\Longrightarrow} [\![(j, \psi')]\!]$. There must be a different occurrence closure, say (k, ψ''), such that $[\![(k, \psi'')]\!] = [\![(j, \psi')]\!]$. So also $[\![(i, \psi)]\!] \underset{t}{\Longrightarrow} [\![(k, \psi'')]\!]$. Since occurrence closure evaluation is deterministic, (i, ψ) might evaluate to (j, ψ') or to (k, ψ''), but not to both.

We would still like to show some weakened converse, to indicate that the actions of the occurrence evaluator are not arbitrary. For instance, if the $\underset{oc}{\Longrightarrow}$ relation were empty, the Simulation Theorem would be true trivially. We shall show that the occurrence evaluator is adequate for the term evaluator.

We first need a technical lemma.

LEMMA 3. *Suppose ψ is chain-free and $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$. If $Var(j)$, then $[\![j]\!] \notin Dom(\psi')$. If $Proc(j)$, then ψ' is chain-free.*

PROOF. By induction on the size of the derivation that $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$. We show only the non-trivial cases.

Base cases

case $Var(i) \wedge [\![i]\!] \in Dom(\psi)$

Then $(i, \psi) \underset{t}{\Longrightarrow} \psi([\![i]\!])$. Suppose $\psi([\![i]\!]) = (j, \psi')$. If $Var(j)$, by the definition of chain-freedom, $[\![j]\!] \notin Dom(\psi')$. On the other hand, if $Proc(j)$, by the definition of chain-freedom, ψ' is chain-free.

Induction

case $App(i)$

So we have an evaluation:

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

By the induction hypothesis at $i.rator$, ψ' is chain-free. By the induction hypothesis at $i.rand$, if $Var(k)$, then $[\![k]\!] \notin Dom(\psi'')$; in case $Proc(k)$, ψ'' is chain-free. Therefore, $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')]$ is also chain-free. So we can invoke the induction hypothesis at $j.body$, and conclude that if $Var(m)$, then $[\![m]\!] \notin Dom(\psi''')$; if $Proc(m)$, then ψ''' is chain-free.

case $Cond(i)$

We show only the subcase where the test is true; the other subcase is similar. Suppose:

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

By the induction hypothesis, if $Var(k)$, $[\![k]\!] \notin Dom(\psi'')$; if $Proc(k)$, then ψ'' is chain-free. $\square$

Now we prove the partial converse to the Simulation Theorem:

THEOREM 2 (ADEQUACY). *Let ψ be a chain-free occurrence environment. If $\mathcal{U}[\![(i, \psi)]\!] \underset{t}{\Longrightarrow} M$ then there are j, ψ' such that $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$ and $M = \mathcal{U}[\![(j, \psi')]\!]$.*

PROOF. Induction on the size of the derivation that $\mathcal{U}[\![(i, \psi)]\!] \underset{t}{\Longrightarrow} M$. Most of the cases are straightforward; we show only the primop application and application cases, which show the use of the chain-free condition.

Suppose $PrimApp(i)$.

Then

$$\begin{aligned}\mathcal{U}[\![(i,\psi)]\!] &= [\![i]\!]\{\mathcal{U}[\![\cdot]\!]\circ\psi\} \\ &= ([\![i.rator]\!]\ [\![i.rand]\!])\{\mathcal{U}[\![\cdot]\!]\circ\psi\} \\ &= [\![i.rator]\!]\ [\![i.rand]\!]\{\mathcal{U}[\![\cdot]\!]\circ\psi\} \\ &= [\![i.rator]\!]\ \mathcal{U}[\![(i.rand,\psi)]\!]\end{aligned}$$

and the term evaluation derivation is of the form

$$\frac{\mathcal{U}[\![(i.rand,\psi)]\!] \underset{t}{\Longrightarrow} N}{\mathcal{U}[\![(i,\psi)]\!] \underset{t}{\Longrightarrow} M}$$

for constants M and N, where the relation between them depends on the primop $[\![i.rator]\!]$.

By the induction hypothesis, there are j and ψ' such that $(i.rand,\psi) \underset{oc}{\Longrightarrow} (j,\psi')$ and $\mathcal{U}[\![(j,\psi')]\!] = N$.

Since (j,ψ') appears on the right-hand side of $\underset{oc}{\Longrightarrow}$, by an easy induction on the definition of $\underset{oc}{\Longrightarrow}$, $[\![j]\!]$ is a value. $[\![j]\!]$ cannot be a procedure, else $\mathcal{U}[\![(j,\psi')]\!]$ would also be a procedure. Assume $Var(j)$. Since ψ is chain-free, by Lemma 3, $[\![j]\!] \notin Dom(\psi')$. Then $\mathcal{U}[\![(j,\psi')]\!] = [\![j]\!]$, a variable, and we have a contradiction.

So $[\![j]\!]$ must be a constant; in fact, $[\![j]\!] = N$. By the appropriate occurrence evaluator rule, $(i,\psi) \underset{oc}{\Longrightarrow} (c,\emptyset)$, for some constant c. As desired,

$$\begin{aligned}\mathcal{U}[\![(c,\emptyset)]\!] &= [\![c]\!]\{\mathcal{U}[\![\cdot]\!]\circ\emptyset\} \\ &= [\![c]\!]\end{aligned}$$

But $[\![c]\!] = M$, since we apply the same primop in the occurrence and term evaluators.

Suppose $App(i)$.

Then

$$\begin{aligned}\mathcal{U}[\![(i,\psi)]\!] &= [\![i]\!]\{\mathcal{U}[\![\cdot]\!]\circ\psi\} \\ &= ([\![i.rator]\!]\ [\![i.rand]\!])\{\mathcal{U}[\![\cdot]\!]\circ\psi\} \\ &= \mathcal{U}[\![(i.rator,\psi)]\!]\ \mathcal{U}[\![(i.rand,\psi)]\!]\end{aligned}$$

and the evaluation derivation is of the form

$$\frac{\mathcal{U}[\![(i.rator,\psi)]\!] \underset{t}{\Longrightarrow} \lambda x.Q \quad \mathcal{U}[\![(i.rand,\psi)]\!] \underset{t}{\Longrightarrow} N \quad Q[N/x] \underset{t}{\Longrightarrow} M}{\mathcal{U}[\![(i,\psi)]\!] \underset{t}{\Longrightarrow} M}$$

By the induction hypothesis at $i.rator$, there are j and ψ' such that $(i.rator,\psi) \underset{oc}{\Longrightarrow} (j,\psi')$ and $\mathcal{U}[\![(j,\psi')]\!] = \lambda x.Q$.

$[\![j]\!]$ is a value, since it appears on the right-hand side of an occurrence closure evaluation. Now, $[\![j]\!]$ cannot be a constant, else the unwind $\mathcal{U}[\![(j,\psi')]\!]$ would be the same constant. Similarly, by the chain-free condition and Lemma 3, if we assume

$Var(j)$ is a variable, then $[\![j]\!] \notin Dom(\psi')$, and the unwind would be the variable $[\![j]\!]$. Therefore, $[\![j]\!]$ must a procedure.

By the induction hypothesis at $i.rand$, there are k and ψ'' such that $(i.rand, \psi) \underset{oc}{\Longrightarrow} (k, \psi'')$ and $\mathcal{U}[\![(k, \psi'')]\!] = N$.

So we have

$$\begin{aligned}
\mathcal{U}[\![(j, \psi')]\!] &= [\![j]\!]\{\mathcal{U}[\![\cdot]\!] \circ \psi'\} \\
&= (\lambda[\![j.bv]\!].[\![j.body]\!])\{\mathcal{U}[\![\cdot]\!] \circ \psi'\} \\
&= \lambda x'.[\![j.body]\!][x'/[\![j.bv]\!]]\{\mathcal{U}[\![\cdot]\!] \circ \psi'\} \\
&= \lambda x.Q
\end{aligned}$$

where x' is fresh.

To use the application rule, we need to evaluate the substituted body:

$$\begin{aligned}
Q[N/x] &= [\![j.body]\!][x'/[\![j.bv]\!]]\{\mathcal{U}[\![\cdot]\!] \circ \psi'\}[N/x'] \\
&= [\![j.body]\!]\{\mathcal{U}[\![\cdot]\!] \circ \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]\} \\
&= \mathcal{U}[\![(j.body, \psi'[[\![j.bv]\!] \mapsto (k, \psi'')])]\!]
\end{aligned}$$

To see that the second equality holds, observe that for any variable x, where $x \neq [\![j.bv]\!]$, we substitute $\mathcal{U}[\![(\psi'(x))]\!]$ on both sides of the equation. For $[\![j.bv]\!]$, we substitute N on the left-hand side, and $\mathcal{U}[\![(k, \psi'')]\!]$ on the right-hand side. But $\mathcal{U}[\![(k, \psi'')]\!] = N$, as we showed above. On the last line, we use the definition of unwinding.

To invoke the induction hypothesis at $j.body$, we have to show that $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')]$ is chain-free. By Lemma 3, ψ' is chain-free. Also by Lemma 3, if $Var(k)$, then $[\![k]\!] \notin Dom(\psi'')$; if $Proc(k)$, then ψ'' is chain-free. So $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')]$ is chain-free.

By the induction hypothesis at $j.body$, there are m and ψ''' such that

$$(j.body, \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]) \underset{oc}{\Longrightarrow} (m, \psi''')$$

and

$$\mathcal{U}[\![(m, \psi''')]\!] = M$$

By the occurrence evaluator rule for applications, $(i, \psi) \underset{oc}{\Longrightarrow} (m, \psi''')$. $\square$

4. AN EQUATIONAL REASONING SYSTEM

Here we describe a system for reasoning equationally about terms in Λ_{in}. Our motivation is to simplify the proof of Theorem 6 in Section 10. We will later extend the equational system so that it may be applied to terms in the output language. When we need the extended relation, we will add appropriate rules.

Let $=_w$ be the smallest binary relation on terms in Λ_{in} closed under the rules in Figure 3. If $M =_w N$, we say that M and N are *weakly equivalent.*

THEOREM 3 (CO-EVALUATION). *If $M =_w N$ then $M \underset{t}{\Longrightarrow} V$ iff $N \underset{t}{\Longrightarrow} V$.*

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

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

$$\frac{M =_w N}{N =_w M} \qquad (\mathbf{sym})$$

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

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

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

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

Fig. 3. Closure conditions for $=_w$

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

When we extend the equational reasoning system in Section 8, we will prove a new co-evaluation result by extending the proof just given.

The converse of Theorem 3 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 induction on the rules for $=_w$ shows that for any pair of Λ_{in} terms M and N, if $M =_w N$, then $M =_\beta N$ (treating **if - then - else -** as a constant). Since Ω and Ω' are not β-convertible, they are not $=_w$.

5. ANNOTATIONS

A *labelled transition system* consists of a set of states S, a set of transition labels T, and a T-indexed family of transition relations [Milner 1989]. For an element t of T, we write $\xrightarrow{t} \subseteq S \times S$ for the t-member of that family.

In preparation for the closure conversion transformation, we construct for any program an *abstract execution*. An abstract execution consists of a labelled transition system satisfying some restrictions, to be explained presently, and some additional data.

In an abstract execution, the states of the labelled transition system are divided into two sets, called *environment propositions* and *value propositions*. We use A to range over environment propositions, and $\mathcal{P}$ to range over value propositions. In addition to the labelled transition system, an abstract execution contains two maps that associate with each value proposition an *invariance set* and a *protocol tag*. An invariance set θ is a finite set of variables. These are intended to be variables whose bindings do not change across an occurrence closure evaluation. A protocol tag π is an element of $\{id\} \cup \{cl_\sigma \mid \sigma \in Var^*\}$, describing a procedure application protocol. When describing protocol tags, we may write $\langle v_1, \ldots, v_n \rangle$ to indicate a sequence of variables. If σ is a sequence of variables, then $\lceil \sigma \rceil$ is the corresponding set consisting of all variables in the sequence. We may write the invariance set and protocol tag components of a value proposition $\mathcal{P}$ as $\theta_\mathcal{P}$ and $\pi_\mathcal{P}$, respectively. We will more fully describe how these components of value propositions are used later.

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

$$A \xrightarrow{x} \mathcal{P}$$

describes the lookup of a variable x in an environment described by A, while a transition of the form

$$\mathcal{P} \xrightarrow{i} A$$

describes a call from a site described by $\mathcal{P}$ 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 A. We further require that an abstract execution be deterministic. That is, for any A and x, there is at most one $\mathcal{P}$ such that $A \xrightarrow{x} \mathcal{P}$; for any $\mathcal{P}$ and i, there is at most one A such that $\mathcal{P} \xrightarrow{i} A$.

An abstract execution may be described independently of a particular program, but we will use abstract execution states to annotate program occurrences. An *annotation map* $\Gamma_\mathcal{E}$ maps each program occurrence i to a pair $(A_i, \mathcal{P}_i)$, where A_i and $\mathcal{P}_i$ are an environment proposition and a value proposition in the abstract execution $\mathcal{E}$. Given an annotation map, we may simply write θ_i instead of $\theta_{\mathcal{P}_i}$ for the invariance set of $\mathcal{P}_i$, and π_i instead of $\pi_{\mathcal{P}_i}$ for the protocol tag of $\mathcal{P}_i$.

The use of labelled transition systems to describe abstract executions is a change from [Wand and Steckler 1994] and [Steckler 1994]. There we wrote $A(x) = \mathcal{P}$ for $A \xrightarrow{x} \mathcal{P}$ and $(i, A) \in \phi$ for $\mathcal{P} \xrightarrow{i} A$. Because transitions are deterministic, we may think of environment propositions as functions. We believe the the use of labelled transition systems gives a clearer account of annotations that contain cycles and also of the constraint generation process (see Section 6, below).

DEFINITION 4. *Let $\mathcal{E}$ be an abstract execution. An annotation map $\Gamma_\mathcal{E}$ is* monovariant *iff for any transition in $\mathcal{E}$ of the form $\mathcal{P} \xrightarrow{i} A$, where $\mathcal{P} = \mathcal{P}_j$ for some program occurrence j, $A = A_i$.*

A monovariant annotation map associates each procedure i with one "abstract closure," given by the pair (i, A_i).

We would like to say that our program annotations are sound in some sense. Informally, we say that the evaluation of a program $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$ satisfies an annotation map iff in the case that the input ψ satisfies A_i, then the output satisfies the value proposition $\mathcal{P}_i$ and, moreover, similar satisfaction relations also hold at all subcomputations. In this way, we may think of the pair $(A_i, \mathcal{P}_i)$ as input and output assertions for a functional program, analogous to Hoare-style partial correctness assertions for imperative programs [Hoare 1969]. Moreover, if j is an occurrence of a procedure, then we will require that the environments ψ and ψ' agree on all variables in θ, the invariance set component of $\mathcal{P}_i$, and π_j must be the same as π_i, the protocol tag component of $\mathcal{P}_i$.

We formalize these intuitions as follows:

Define a *protocol assignment* to be a map from occurrences to protocol tags. We write Π for such a map.

Definition 5. *The following definitions are mutually recursive.*

(1) An occurrence environment ψ satisfies an environment proposition A under an abstract execution $\mathcal{E}$ and protocol assignment Π, as follows:

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

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

(a) $x \in Dom(\psi)$ and $\psi(x) \models^{\mathcal{E},\Pi}_{val} \mathcal{P}$, and

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

(2) An occurrence closure (i, ψ) satisfies a value proposition $\mathcal{P} = (\theta, \pi)$ under an abstract execution $\mathcal{E}$ and protocol assignment Π, as follows:

$$(i, \psi) \models^{\mathcal{E},\Pi}_{val} \mathcal{P}$$

iff

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

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

Since occurrence closures are finite structures, these definitions are well-founded.

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

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

As we mentioned, an environment proposition acts functionally. For an environment proposition A in an abstract execution $\mathcal{E}$, define $Dom(A)$ to be:

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

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

LEMMA 4. *Let $\mathcal{E}$ be an abstract execution. If ψ is a scalar environment, then for any protocol assignment Π and for all environment propositions A in $\mathcal{E}$ such that $Dom(A) \subseteq Dom(\psi)$, $\psi \models^{\mathcal{E},\Pi}_{env} A$.*

LEMMA 5. *Let $\mathcal{E}$ be an abstract execution and let A be an environment proposition in $\mathcal{E}$. Let ψ be an occurrence environment and let Π be a protocol assignment such that $\psi \models^{\mathcal{E},\Pi}_{env} A$. Suppose that A' is an environment proposition in $\mathcal{E}$ such that for all x and $\mathcal{P}$, if $A' \xrightarrow{x} \mathcal{P}$ in $\mathcal{E}$, then $A \xrightarrow{x} \mathcal{P}$ in $\mathcal{E}$. Then $\psi \models^{\mathcal{E},\Pi}_{env} A'$.*

6. LOCALLY CONSISTENT ANNOTATIONS

An abstract execution contains two kinds of states, environment propositions and value propositions, which we use to annotate programs. Our definitions in the preceding section tell us what these annotations will look like, and the semantics just given tells us when these propositions are true. Note that we have carefully defined the semantics of propositions independently of any method used to annotate programs. In this section, we show how to associate annotations with programs to achieve the soundness goals mentioned in Section 5.

Recall, an annotation map associates each program occurrence i with a pair $(A_i, \mathcal{P}_i)$. We obtain such a map in three steps. In the first phase, *closure analysis*, we obtain the deductive closure of the rules in the system shown in Figure 4. The deductive closure of these rules tells us what the transitions of the abstract execution will be, and which states of the abstract execution are associated with which program occurrences.

In second next step, *constraint generation*, we take deductive closures of the rules in Figures 5 and 6. These sets of rules have no mutual dependencies, therefore they may be applied in parallel. The rules in Figure 5 give us the *complement* of the invariance set component for the value proposition at each occurrence. The universe of variables is of course limited to the variables appearing in the program. From the rules in Figure 6, we obtain equality constraints on protocol tags. Note that not all occurrences have their protocol tags constrained. Equality is an equivalence relation, so the deductive closure yields equivalence classes of occurrences. Each element of such a class is constrained to have the same protocol tag as all other elements of the class.

In the last step, *constraint solution*, we choose a protocol tag for each equivalence class. We may choose a protocol tag of id or cl_σ, for an appropriate σ. The rule **app-tag-cons** shown in Figure 7 indicates what an appropriate σ is. For a given

$Var(i) \Longrightarrow A_i \xrightarrow{[\![i]\!]} \mathcal{P}_i$	**var**
$PrimApp(i) \Longrightarrow \dfrac{A_{i.rand} \xrightarrow{x} \mathcal{P}}{A_i \xrightarrow{x} \mathcal{P}}$	**primapp**
$Proc(i) \Longrightarrow$	
$\mathcal{P}_i \xrightarrow{i} A_i$	**proc-self**
$A_{i.body} \xrightarrow{[\![i.bv]\!]} \mathcal{P}_{i.bv}$	**proc-body-ext**
$\dfrac{A_{i.body} \xrightarrow{x} \mathcal{P}}{A_i \xrightarrow{x} \mathcal{P}} \quad x \neq [\![i.bv]\!]$	**proc-body-inh**
$App(i) \Longrightarrow$	
$\dfrac{A_{i.rator} \xrightarrow{x} \mathcal{P}}{A_i \xrightarrow{x} \mathcal{P}}$	**app-rator-inh**
$\dfrac{A_{i.rand} \xrightarrow{x} \mathcal{P}}{A_i \xrightarrow{x} \mathcal{P}}$	**app-rand-inh**
$\dfrac{\mathcal{P}_{i.rator} \xrightarrow{j} B \quad \mathcal{P}_{i.rand} \xrightarrow{k} C}{\mathcal{P}_{j.bv} \xrightarrow{k} C}$	**app-rand-bv**
$\dfrac{\mathcal{P}_{i.rator} \xrightarrow{j} B \quad \mathcal{P}_{j.body} \xrightarrow{k} C}{\mathcal{P}_i \xrightarrow{k} C}$	**app-body-app**
$Cond(i) \Longrightarrow$	
$\dfrac{A_{i.test} \xrightarrow{x} \mathcal{P}}{A_i \xrightarrow{x} \mathcal{P}}$	**cond-test-inh**
$\dfrac{A_{i.then} \xrightarrow{x} \mathcal{P}}{A_i \xrightarrow{x} \mathcal{P}}$	**cond-then-inh**
$\dfrac{A_{i.else} \xrightarrow{x} \mathcal{P}}{A_i \xrightarrow{x} \mathcal{P}}$	**cond-else-inh**
$\dfrac{\mathcal{P}_{i.then} \xrightarrow{j} B}{\mathcal{P}_i \xrightarrow{j} B}$	**cond-then-cond**
$\dfrac{\mathcal{P}_{i.else} \xrightarrow{j} B}{\mathcal{P}_i \xrightarrow{j} B}$	**cond-else-cond**

Fig. 4. Deduction rules for transitions

$$
\begin{array}{ll}
Proc(i) \Longrightarrow \dfrac{x \notin Dom(A_i)}{x \notin \theta_i} & \textbf{proc-inv} \\
App(i) \Longrightarrow & \\
\left[\begin{array}{c}
\dfrac{x \notin \theta_{i.rator}}{x \notin \theta_i} \\
\dfrac{\mathcal{P}_{i.rator} \xrightarrow{j} B \quad x \notin \theta_{i.rand}}{x \notin \theta_{j.bv}} \\
\dfrac{\mathcal{P}_{i.rator} \xrightarrow{j} B \quad x \notin \theta_{j.body}}{x \notin \theta_i} \\
\dfrac{\mathcal{P}_{i.rator} \xrightarrow{j} B \quad x \notin \theta_{i.rator}}{x \notin \theta_{j.bv}} \\
\dfrac{\mathcal{P}_{i.rator} \xrightarrow{j} B}{[\![j.bv]\!] \notin \theta_{j.bv}} \\
\dfrac{\mathcal{P}_{i.rator} \xrightarrow{j} B}{[\![j.bv]\!] \notin \theta_i}
\end{array}\right.
&
\begin{array}{l}
\textbf{app-inv-rator} \\[2ex]
\textbf{app-inv-rand-bv} \\[2ex]
\textbf{app-inv-body-app} \\[2ex]
\textbf{app-inv-rator-bv} \\[2ex]
\textbf{app-inv-bv} \\[2ex]
\textbf{app-inv-bv-app}
\end{array} \\
Cond(i) \Longrightarrow & \\
\left[\begin{array}{c}
\dfrac{x \notin \theta_{i.then}}{x \notin \theta_i} \\
\dfrac{x \notin \theta_{i.else}}{x \notin \theta_i}
\end{array}\right.
&
\begin{array}{l}
\textbf{cond-inv-then} \\[2ex]
\textbf{cond-inv-else}
\end{array}
\end{array}
$$

Fig. 5. Deduction rules for invariance sets

$$
\begin{array}{ll}
App(i) \Longrightarrow & \\
\left[\begin{array}{c}
\dfrac{\mathcal{P}_{i.rator} \xrightarrow{j} B}{\pi_{j.bv} = \pi_{i.rand}} \\
\dfrac{\mathcal{P}_{i.rator} \xrightarrow{j} B}{\pi_{j.body} = \pi_i}
\end{array}\right.
&
\begin{array}{l}
\textbf{app-tag-bv-rand} \\[2ex]
\textbf{app-tag-body-app}
\end{array} \\
Cond(i) \Longrightarrow \ \pi_i = \pi_{i.then} = \pi_{i.else} & \textbf{cond-tag}
\end{array}
$$

Fig. 6. Deduction rules for protocol tags

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

Fig. 7. Protocol tag constraint

equivalence class C, choose any ordering on any subset of

$$\bigcap\{\theta_i \mid i \text{ an application operator}, \ i \in C\}$$

If all the deduction rules in Figures 4 through 7 hold for an annotation map $\Gamma_\mathcal{E}$, we say that $\Gamma_\mathcal{E}$ is *locally consistent.*

In the resulting abstract execution, the transitions of the form $\mathcal{P} \xrightarrow{j} A$ give us a (stylized form of) *closure analysis* [Sestoft 1991]. For a value proposition $\mathcal{P}$ in an abstract execution $\mathcal{E}$, define $Dom(\mathcal{P})$ to be:

$$\{j \mid \exists A \text{ such that } \mathcal{P} \xrightarrow{j} A \text{ in } \mathcal{E}\}$$

If $\Gamma_\mathcal{E}$ associates an occurrence i with a value proposition $\mathcal{P}_i$, then $Dom(\mathcal{P}_i)$ describes the set of procedures to which i may evaluate.

Now we can convey some of the intuition behind the deduction rules. Consider a term with occurrence index i. If $Proc(i)$, we have the rule $\mathcal{P}_i \xrightarrow{i} A_i$, since $[\![i]\!]$ evaluates to itself. If $App(i)$ and $\mathcal{P}_{i.rator} \xrightarrow{j} B$, for some B, then by the rules, in case $\mathcal{P}_{i.rand} \xrightarrow{k} C$, for some k and C, also $\mathcal{P}_{j.bv} \xrightarrow{k} C$. This says that for 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, if $App(i)$ and $\mathcal{P}_{i.rator} \xrightarrow{j} B$, then in case $\mathcal{P}_{j.body} \xrightarrow{k} C$, also $\mathcal{P}_i \xrightarrow{k} C$. 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 other constraint-based formulations of closure analysis.

For invariance sets, the deduction rules require a bit more complex motivation. Suppose $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$ and that $Proc(j)$. We want θ_i to be a set of variables whose bindings are the same in ψ and ψ'. 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 4). 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 ψ are left invariant by this calculation? ψ''' is obtained from ψ in two steps: First, the operator is evaluated, yielding a ψ' agreeing with ψ on the variables in $\theta_{i.rator}$ (by an appropriate induction hypothesis). We then evaluate $j.body$ in an extension of ψ', yielding ψ''', which agrees with ψ' on $\theta_{j.body} - [\![j.bv]\!]$. In order that ψ agrees with ψ''' on the variables in θ_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 $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \models^{\mathcal{E},\Pi}_{env} A_{j.body}$, in support of the induction hypothesis for the body. The details appear in the proof of Theorem 4.

7. SOUNDNESS

Now we show that any monovariant and locally consistent annotation of a program makes the annotations true, in the following sense:

THEOREM 4 (SOUNDNESS). *Let $\mathcal{E}$ be an abstract execution. Let $\Gamma_{\mathcal{E}}$ be a monovariant and locally consistent annotation map, and let Π be the protocol assignment defined by $\forall i,\ \Pi(i) = \pi_i$. Let i be an occurrence index and ψ an occurrence environment such that $\psi \models^{\mathcal{E},\Pi}_{env} A_i$. Suppose $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$. Then for any subproof $(k, \psi'') \underset{oc}{\Longrightarrow} (m, \psi''')$ of this derivation:*

(1) $\psi'' \models^{\mathcal{E},\Pi}_{env} A_k$,

(2) $(m, \psi''') \models^{\mathcal{E},\Pi}_{val} \mathcal{P}_k$, and

(3) if $Proc(m)$, then $\forall x \in \theta_k,\ x \in Dom(\psi'') \cap Dom(\psi''')$, 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 $\models^{\mathcal{E},\Pi}_{env}$ 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, and the occurrence environment is empty; see the evaluation rules in Figure 2. So suppose we have the evaluation $(i, \psi) \underset{oc}{\Longrightarrow} (c, \emptyset)$. Since $Const(c)$, by the definition of $\models^{\mathcal{E},\Pi}_{val}$, $(c, \emptyset) \models^{\mathcal{E},\Pi}_{val} \mathcal{P}_i$. 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 Γ is locally consistent, if $A_{i.rand} \xrightarrow{x} \mathcal{P}$, then also $A_i \xrightarrow{x} \mathcal{P}$, by the **primapp** rule. Since $\psi \models^{\mathcal{E},\Pi}_{env} A_i$, by Lemma 5, also $\psi \models^{\mathcal{E},\Pi}_{env} A_{i.rand}$. 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)	$\psi \models^{\mathcal{E},\Pi}_{env} A_{i.rator}$	[**app-rator-inh**, Lemma 5]
(2)	$\psi \models^{\mathcal{E},\Pi}_{env} A_{i.rand}$	[**app-rand-inh**, Lemma 5]
(3)	$(j, \psi') \models^{\mathcal{E},\Pi}_{val} \mathcal{P}_{i.rator}$	[IH re $\models^{\mathcal{E},\Pi}_{val}$ at $i.rator$]
(4)	$x \in \theta_{i.rator} \Longrightarrow x \in Dom(\psi) \cap Dom(\psi'),\ \psi(x) = \psi'(x)$	[$Proc(j)$, IH re invariance at $i.rator$]
(5)	$\mathcal{P}_{i.rator} \xrightarrow{j} A_j$	[$Proc(j)$, (3), monovariance]
(6)	$\psi' \models^{\mathcal{E},\Pi}_{env} A_j$	[(3), (5), def. of $\models^{\mathcal{E},\Pi}_{env}$]
(7)	$(k, \psi'') \models^{\mathcal{E},\Pi}_{val} \mathcal{P}_{i.rand}$	[IH re $\models^{\mathcal{E},\Pi}_{val}$ at $i.rand$]
(8)	if $Proc(k)$ then $\mathcal{P}_{i.rand} \xrightarrow{k} A_k$	[(7), monovariance]
(9)	if $Proc(k)$ then $\psi'' \models^{\mathcal{E},\Pi}_{env} A_k$	[(7), (8), def. of $\models^{\mathcal{E},\Pi}_{env}$]
(10)	if $Proc(k)$ then $x \in \theta_{i.rand} \Longrightarrow x \in Dom(\psi) \cap Dom(\psi''),\ \psi(x) = \psi''(x)$	[IH re invariance at $i.rand$]
(11)	if $Proc(k)$ then $\mathcal{P}_{j.bv} \xrightarrow{k} A_k$	[(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) $(k, \psi'') \models^{\mathcal{E},\Pi}_{val} \mathcal{P}_{j.bv}$ [(7), (8), (9), (11), (12)]

(24) if $A_{j.body} \xrightarrow{x} \mathcal{P}$ then $A_j \xrightarrow{x} \mathcal{P},\ x \neq [\![j.bv]\!]$
[**proc-body-inh**]

(25) $A_{j.body} \xrightarrow{[\![j.bv]\!]} \mathcal{P}_{j.bv}$ [**proc-body-ext**]

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

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

(28) if $Proc(m)$ then $\mathcal{P}_{j.body} \xrightarrow{m} A_m$ [(27), monovariance]

(29) if $Proc(m)$ then $\psi''' \models^{\mathcal{E},\Pi}_{env} A_m$ [(27), (28), def. of $\models^{\mathcal{E},\Pi}_{env}$]

(30) if $Proc(m)$ then $\mathcal{P}_i \xrightarrow{m} A_m$ [(5), (28), **app-body-app**]

(31) $(m, \psi''') \models^{\mathcal{E},\Pi}_{val} \mathcal{P}_i$ [(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 re 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)]

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

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), $\psi \models^{\mathcal{E},\Pi}_{env} A_{i.rator}$ and by step (2), $\psi \models^{\mathcal{E},\Pi}_{env} A_{i.rand}$. 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), $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \models^{\mathcal{E},\Pi}_{env} A_{j.body}$, 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:

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

Since Γ is locally consistent. By the **cond-then-inh** rule, if $A_{i.then} \xrightarrow{x} \mathcal{P}$, for some $\mathcal{P}$, then $A_i \xrightarrow{x} \mathcal{P}$. By assumption, $\psi \models^{\mathcal{E},\Pi}_{env} A_i$, so by Lemma 5, also $\psi \models^{\mathcal{E},\Pi}_{env} A_{i.then}$. By the induction hypothesis at $i.then$:

$$(k, \psi'') \models^{\mathcal{E},\Pi}_{val} \mathcal{P}_{i.then}$$

By the rule **cond-tag**, $\pi_i = \pi_{i.then}$, and by rule **cond-then-inh**, if $\mathcal{P}_{i.then} \xrightarrow{m} B$, then $\mathcal{P}_i \xrightarrow{m} B$. By the definition of $\models^{\mathcal{E},\Pi}_{val}$, also

$$(k, \psi'') \models^{\mathcal{E},\Pi}_{val} \mathcal{P}_i$$

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 $\psi \models^{\mathcal{E},\Pi}_{env} A_{i.then}$, so the consequent holds at all subproofs of the evaluation of the then-part. Since Γ is locally consistent, by the rule **cond-test-inh**, if $A_{i.test} \xrightarrow{x} \mathcal{P}$, then $A_i \xrightarrow{x} \mathcal{P}$; so by Lemma 5, $\psi \models^{\mathcal{E},\Pi}_{env} A_{i.test}$. Therefore, the consequent holds at all subproofs of the evaluation of the test. □

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

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

Fig. 8. Additional evaluation rules for Λ_{clos}

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 includes records. The output language Λ_{clos} is given by the grammar:

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

where *PrimOp* is defined as in Section 3. The productions for records and the **destr** operator are new; otherwise, the grammar is identical to that for Λ_{in}. We redefine a value to be any constant, boolean value, variable, procedure, or a record of values. Values in Λ_{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 8, we give two new rules for evaluating terms in Λ_{clos}. These rules supplement the rules given for the term evaluator in Section 3 to give an evaluator for terms in Λ_{clos}. As for the input language evaluator, the evaluator for Λ_{clos} has the property that any value self-evaluates.

For purposes of the **destr** rule, we may write $\lambda.M$ for the term M. Therefore, if N evaluates to the empty record [ ], and M evaluates to V, then **destr** $N\ \lambda.M$ evaluates to V.

We perform closure conversion as a source-to-source transformation. Our closures are records containing two fields. We can define combinators that act as projection functions to get at each field:

$$\begin{aligned} \mathbf{fst} &= (\lambda r.\,\mathbf{destr}\ r\ (\lambda xy.x)) \\ \mathbf{snd} &= (\lambda r.\,\mathbf{destr}\ r\ (\lambda xy.y)) \end{aligned}$$

We now need to extend our equational reasoning system to handle records. In Figure 9, we give additional closure conditions for the $=_w$ relation. Redefine $=_w$ to be the smallest binary relation on terms in Λ_{clos}, closed under the rules in Figures 3

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

$$\frac{\begin{array}{l} M =_w [\ \overbrace{V_1, \ldots, V_n}^{\text{values}}\ ] \\ N =_w (\lambda x_1 \ldots x_n.P) \\ P[V_1/x_1, \ldots, V_n/x_n] =_w R \end{array}}{\mathbf{destr}\ MN =_w R} \quad \textbf{(destr)}$$

Fig. 9. Additional closure conditions for $=_w$ and Λ_{clos}

and 9.

The co-evaluation property is maintained for the extended relation:

Theorem 5. *For $M, N \in \Lambda_{clos}$, if $M =_w N$ then $M \underset{t}{\Longrightarrow} V$ iff $N \underset{t}{\Longrightarrow} V$.*

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

To suggest how closure records effectively represent the procedures from which they are derived, we show that applying a closure record to a value simulates the application of the original procedure to the same value.

Let $F = \lambda x.M$, and let F_{clos} be the closure record $[(\lambda ex.\,\mathbf{destr}\ e\ (\lambda\vec{u}.M)), [\vec{u}]]$, where $\lceil\vec{u}\rceil = FV(\lambda x.M)$. We show that for any value V, $\mathbf{app}\ F_{clos}\ V =_w F\ V$.

Note that this closure record is hand-built, as it were; our closure transformation was not used. We cannot invoke the transformation, since we have not even annotated the program. Here, we are just demonstrating the operation of closure records. (If we had annotated the program, it is unlikely that the transformation would be the identity on the procedure body M.)

On the right-hand side, we can use the β rule to show that $F\ V =_w M[V/x]$.

On the left-hand side, we can reason as follows:

$$\begin{array}{lll} & (\lambda r.(\mathbf{fst}\ r)(\mathbf{snd}\ r))\ F_{clos}\ V & \\ =_w & (\mathbf{fst}\ F_{clos})\ (\mathbf{snd}\ F_{clos})\ V & (\beta),\ \textbf{(cong)} \\ = & ((\lambda r.\,\mathbf{destr}\ r\ (\lambda xy.x))\ F_{clos})\ ((\lambda r.\,\mathbf{destr}\ r\ (\lambda xy.y))\ F_{clos})\ V & \\ =_w & (\mathbf{destr}\ F_{clos}\ (\lambda xy.x))\ (\mathbf{destr}\ F_{clos}\ (\lambda xy.y))\ V & (\beta),\ \textbf{(cong)} \end{array}$$

The β steps are justified by the fact that F_{clos} is a value.

Using the definition of F_{clos}, we have:

$$
\begin{array}{lll}
& (\mathbf{destr}\ F_{clos}\ (\lambda xy.x))\ (\mathbf{destr}\ F_{clos}\ (\lambda xy.y))\ V & \\
=_w & (\lambda ex.\,\mathbf{destr}\ e\ (\lambda\vec{u}.M))\ [\vec{u}]\ V & (\mathbf{destr}),\ (\mathbf{cong}) \\
=_w & (\lambda x.\,\mathbf{destr}\,[\vec{u}]\ (\lambda\vec{u}.M))\ V & (\beta),\ (\mathbf{cong}) \\
=_w & (\mathbf{destr}\,[\vec{u}]\ (\lambda\vec{u}.M))[V/x] & (\beta) \\
= & \mathbf{destr}\,[\vec{u}]\ (\lambda\vec{u}.M)[V/x] & \\
= & \mathbf{destr}\,[\vec{u}]\ (\lambda\vec{u}'.M[\vec{u}'/\vec{u}])[V/x] & \vec{u}'\ \text{fresh} \\
= & \mathbf{destr}\,[\vec{u}]\ (\lambda\vec{u}'.M[\vec{u}'/\vec{u}][V/x]) & \\
=_w & M[\vec{u}'/\vec{u}][V/x][\vec{u}/\vec{u}'] & (\mathbf{destr}) \\
= & M[V/x] &
\end{array}
$$

In the last few steps, where we move the substitution $[V/x]$ toward the procedure body M, we are able to do because x was the binding variable in the original procedure, so $x \notin \lceil\vec{u}\rceil$, hence the substitution affects only the procedure $\lambda\vec{u}.M$. We α-convert to avoid capturing any free variables in V. To see that the final equality holds, observe that on the left-hand side, any variable among $\vec{u}$ is substituted for itself.

Wherever $=$ appears in this last series of equations, we may substitute $=_w$ by the α rule. So by the symmetry and transitivity of $=_w$, we have

$$\mathbf{app}\ F_{clos}\ V =_w F\ V$$

as desired. Then by Lemma 5,

$$\mathbf{app}\ F_{clos}\ V \underset{t}{\Longrightarrow} W \text{ iff } F\ V \underset{t}{\Longrightarrow} W$$

In this way, application of the closure record simulates application of the procedure it represents.

9. THE CLOSURE CONVERSION TRANSFORMATION

Once we have annotated a program so that the local constraints are satisfied, we can transform it according to the algorithm in Figure 10. Suppose we have an abstract execution $\mathcal{E}$ and an annotation map $\Gamma_{\mathcal{E}}$ for a program. The transformation $\Phi_{\Gamma_{\mathcal{E}}}$ is a map from O, the set of occurrence indices, to Λ_{clos}, the language of output terms. The formalism is:

$$\Phi_{\Gamma_{\mathcal{E}}} : O \to \Lambda_{clos}$$

Usually the annotation map $\Gamma_{\mathcal{E}}$ will be understood from context, so we will write simply Φ, as in Figure 10. For a procedure occurrence, the algorithm uses its associated protocol tag to determine whether or not to create an explicit closure, and if so, which variables are included in the closure. Recall, **app** is defined as the procedure $\lambda r.(\mathbf{fst}\ r)(\mathbf{snd}\ r)$. For an application occurrence, the protocol tag associated with the operator is used to determine whether **app** should be inserted, and if so, which dynamic variables, if any, should be inserted as extra arguments and in what order.

We extend the transformation to occurrence closures:

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

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

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

$$Proc(i) \wedge \pi_i = id \Longrightarrow \Phi(i) = \lambda[\![i.bv]\!].\Phi(i.body)$$

$$\begin{aligned} &Proc(i) \wedge \pi_i = cl_{\langle v_1,\ldots,v_n\rangle} \Longrightarrow \\ &\qquad \Phi(i) = [(\lambda e v_1 \ldots v_n [\![i.bv]\!].\, \mathbf{destr}\ e\ (\lambda\vec{u}.\Phi(i.body))), [\vec{u}]] \\ &\qquad\qquad \text{where } e \text{ fresh} \\ &\qquad\qquad \text{and } \lceil\vec{u}\rceil = FV([\![i]\!]) - \{v_1,\ldots,v_n\} \end{aligned}$$

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

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

$$App(i) \wedge \pi_{i.rator} = cl_{\langle v_1,\ldots,v_n\rangle} \Longrightarrow \Phi(i) = \mathbf{app}\ \Phi(i.rator)\ v_1\ \ldots\ v_n\ \Phi(i.rand)$$

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

Fig. 10. The closure conversion transformation

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

10. 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:

LEMMA 6. *For an occurrence environment ψ, if $x \in Dom(\psi)$, then $\hat{\Phi}(\psi(x))$ is a value in Λ_{clos}.*

PROOF. Induction on the depth of occurrence environments.

Let $\psi(x) = (j, \psi')$. The only difficult case is for procedures.

Suppose $Proc(j)$.

We have $\hat{\Phi}(j, \psi') = \Phi(j)\{\hat{\Phi} \circ \psi'\}$. There are two possibilities for $\Phi(j)$, depending on the protocol tag for j:

$$\Phi(j)\{\hat{\Phi}\circ\psi'\} = \begin{cases} (\lambda[\![j.bv]\!].\Phi(j.body))\{\hat{\Phi}\circ\psi'\} & \text{if } \pi_j = id \\ \\ [\lambda e\vec{v}[\![j.bv]\!].\,\mathbf{destr}\ e\ \lambda\vec{u}.\Phi(j.body),[\vec{u}]]\{\hat{\Phi}\circ\psi'\} & \text{if } \pi_j = cl_{\langle v_1,\ldots,v_n\rangle} \\ \quad \text{where } e \text{ fresh} & \\ \quad \text{and } \lceil\vec{u}\rceil = FV([\![j]\!]) - \{v_1,\ldots,v_n\} & \end{cases}$$

In the first case, we have a substitution instance of a procedure, which is a value. In the second case, observe that 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, by the induction hypothesis, we get either an empty record, which is a value, or a record of values, which is also a value. Hence, the entire record is a value. □

Now we can show:

Theorem 6 (Correctness). *Let $\mathcal{E}$ be an abstract execution, and let $\Gamma_{\mathcal{E}}$ be a monovariant and locally consistent annotation map. Let Π be the protocol assignment defined by $\forall i, \Pi(i) = \pi_i$. Let i be an occurrence index, and ψ an occurrence environment such that $\psi \models^{\mathcal{E},\Pi}_{env} A_i$. If*

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

then

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

We can illustrate this theorem by another 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. 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 need to consider the effect of the operator's protocol tag on the transformation. If the operator tag is id, then we can use a straightforward induction. If the operator tag is π_σ, then we can use our equational reasoning system to find a term that co-evaluates with the transform, and show that the term evaluates to $\hat{\Phi}(m,\psi''')$.

Suppose $\pi_{i.rator} = id$. Then $\Phi(i,\psi) = \Phi(i.rator,\psi)\ \Phi(i.rand,\psi)$. Since the transform is an application, we can just follow the application rule in the term evaluator.

By Theorem 4, $\psi \models^{\mathcal{E},\Pi}_{env} A_{i.rator}, A_{i.rand}$. We can invoke the induction hypothesis at $i.rator$:

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

and at $i.rand$:

$$\hat{\Phi}(i.rand,\psi) \underset{t}{\Longrightarrow} \hat{\Phi}(k,\psi'')$$

Also by Theorem 4, $(j,\psi') \models^{\mathcal{E},\Pi}_{val} \mathcal{P}_{i.rator}$. Since $Proc(j)$, we know $\Pi(j) = \pi_j = \pi_{i.rator} = id$.

Therefore:

$$\begin{aligned}
\Phi(j,\psi') &= \Phi(j)\{\hat{\Phi}\circ\psi'\} && \text{definition of } \hat{\Phi}\\
&= (\lambda[\![j.bv]\!].\Phi(j.body))\{\hat{\Phi}\circ\psi'\} && \text{definition of } \Phi,\ \pi_j = id\\
&= (\lambda x'.\Phi(j.body)[x'/[\![j.bv]\!]])\{\hat{\Phi}\circ\psi'\} && \alpha\text{-conversion}\\
&= \lambda x'.\Phi(j.body)[x'/[\![j.bv]\!]]\{\hat{\Phi}\circ\psi'\} && x' \text{ fresh}
\end{aligned}$$

We now want evaluate the substituted body of the procedure. Observe:

$$\begin{aligned}
\Phi(j.body)[x'/[\![j.bv]\!]]\{\hat{\Phi}\circ\psi'\}[\hat{\Phi}(k,\psi'')/x'] &= \Phi(j.body)\{\hat{\Phi}\circ\psi'[[\![j.bv]\!]\mapsto(k,\psi'')]\}\\
&= \hat{\Phi}(j.body,\psi'[[\![j.bv]\!]\mapsto(k,\psi'')])
\end{aligned}$$

By Theorem 4,

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

so by the induction hypothesis at $j.body$:

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

Then by the application rule for terms:

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

Now suppose $\pi_{i.rator} = cl_{\langle v_1,\ldots,v_n\rangle}$. Our goal once more is to show that $\hat{\Phi}(i,\psi) \underset{t}{\Longrightarrow} \hat{\Phi}(m,\psi''')$. We have:

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

$$
\begin{array}{l}
\textbf{app}\ \hat\Phi(i.rator,\psi)\ \hat\Phi(\psi(v_1))\ \ldots\ \hat\Phi(\psi(v_n))\ \hat\Phi(i.rand,\psi) \\
\quad \textbf{app}\ \hat\Phi(i.rator,\psi)\ \hat\Phi(\psi(v_1))\ \ldots\ \hat\Phi(\psi(v_n)) \\
\qquad\qquad \textbf{app}\ \hat\Phi(i.rator,\psi) \\
\qquad\qquad\quad \textbf{app} \\
\qquad\qquad\quad \Downarrow \\
\qquad\qquad\quad \textbf{app} \\
\qquad\qquad\quad \hat\Phi(i.rator,\psi) \\
\qquad\qquad\quad \Big\Downarrow \text{ induction hypothesis at } i.rator \\
\qquad \ldots \qquad \hat\Phi(j,\psi') \\
\qquad\qquad\quad ((\textbf{fst}\ r)\ (\textbf{snd}\ r))[\hat\Phi(j,\psi')/r] \\
? \qquad\qquad\quad \Big\Downarrow \ \ldots \\
\qquad\qquad\quad \ldots \\
\qquad\qquad \ldots \\
\quad (\lambda x.M) \\
\quad \hat\Phi(i.rand,\psi) \\
\quad \Big\Downarrow \text{ induction hypothesis at } i.rand \\
\quad \hat\Phi(k,\psi'') \\
\quad M[\hat\Phi(k,\psi'')/x] \\
\quad \Big\Downarrow \ \ldots \\
\quad \hat\Phi(m,\psi''') \\
\hat\Phi(m,\psi''')
\end{array}
$$

Fig. 11. Evaluating an application where the operator tag is $cl_{\langle v_1,\ldots,v_n\rangle}$

We sketch what the proof tree for the evaluation of $\hat\Phi(i,\psi)$ should look like in Figure 11. 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, **app** $\hat\Phi(i.rator,\psi)\ \hat\Phi(\psi(v_1))\ \ldots\ \hat\Phi(\psi(v_n))$ 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. Instead of beginning our chain of reasoning with $\hat\Phi(i,\psi)$ itself, we present a term that co-evaluates with $\hat\Phi(i,\psi)$, and work with that term.

Consider the proof tree in Figure 11. 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:

$$
\begin{array}{c}
\hat\Phi(i,\psi) \underset{t}{\Longrightarrow} \hat\Phi(m,\psi''') \\
\text{iff} \\
\textbf{app}\ \hat\Phi(j,\psi')\ \hat\Phi(\psi(v_1))\ \ldots\ \hat\Phi(\psi(v_n))\ \hat\Phi(k,\psi'') \underset{t}{\Longrightarrow} \hat\Phi(m,\psi''')
\end{array}
$$

We can now use equational reasoning to simplify the term to evaluate. By the

β-rule for $=_w$, we have

$$\begin{aligned}\mathbf{app}\ \hat{\Phi}(j,\psi') &= (\lambda r.(\mathbf{destr}\ r\ \lambda xy.x)\ (\mathbf{destr}\ r\ \lambda xy.y))\ \hat{\Phi}(j,\psi')\\ &=_w ((\mathbf{destr}\ r\ \lambda xy.x)\ (\mathbf{destr}\ r\ \lambda xy.y))[\hat{\Phi}(j,\psi')/r]\\ &= (\mathbf{destr}\ \hat{\Phi}(j,\psi')\ \lambda xy.x)\ (\mathbf{destr}\ \hat{\Phi}(j,\psi')\ \lambda xy.y)\end{aligned}$$

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

By Lemma 4, $(j,\psi') \models^{\mathcal{E},\Pi}_{val} \mathcal{P}_{i.rator}$, so $\Pi(j) = \pi_j = \pi_{i.rator} = cl_{\langle v_1,\ldots,v_n\rangle}$. Therefore, $\Phi(j)$ is a closure record. Simplifying further, we have:

$$\begin{aligned}&(\mathbf{destr}\ \hat{\Phi}(j,\psi')\ \lambda xy.x)\ (\mathbf{destr}\ \hat{\Phi}(j,\psi')\ \lambda xy.y)\\ =\ &(\mathbf{destr}\ \Phi(j)\{\hat{\Phi}\circ\psi'\}\ \lambda xy.x)\ (\mathbf{destr}\ \Phi(j)\{\hat{\Phi}\circ\psi'\}\ \lambda xy.y)\\ =_w\ &(\lambda e v_1\ldots v_n[\![j.bv]\!].\,\mathbf{destr}\ e\ \lambda\vec{u}.\Phi(j.body))\{\hat{\Phi}\circ\psi'\}\ [\vec{u}]\{\hat{\Phi}\circ\psi'\}\\ =\ &(\lambda e.(\lambda v_1\ldots v_n[\![j.bv]\!].\,\mathbf{destr}\ e\ \lambda\vec{u}.\Phi(j.body))\{\hat{\Phi}\circ\psi'\})\ [\vec{u}]\{\hat{\Phi}\circ\psi'\}\\ =_w\ &(\lambda v_1\ldots v_n[\![j.bv]\!].\,\mathbf{destr}\ e\ \lambda\vec{u}.\Phi(j.body))\{\hat{\Phi}\circ\psi'\}[[\vec{u}]\{\hat{\Phi}\circ\psi'\}/e]\\ =\ &(\lambda v_1\ldots v_n[\![j.bv]\!].\,\mathbf{destr}[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))\{\hat{\Phi}\circ\psi'\}\end{aligned}$$

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

So far we have shown:

$$\begin{aligned}&\mathbf{app}\ \hat{\Phi}(j,\psi')\\ =_w\ &(\mathbf{destr}\ \hat{\Phi}(j,\psi')\ \lambda xy.x)\ (\mathbf{destr}\ \hat{\Phi}(j,\psi')\ \lambda xy.y)\\ =_w\ &(\lambda v_1\ldots v_n[\![j.bv]\!].\,\mathbf{destr}[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))\{\hat{\Phi}\circ\psi'\}\end{aligned}$$

By the **trans** rule and $n+1$ applications of the **cong** rule, we have:

$$\mathbf{app}\ \hat{\Phi}(j,\psi')\ \hat{\Phi}(\psi(v_1))\ \ldots\ \hat{\Phi}(\psi(v_n))\ \hat{\Phi}(k,\psi'')$$
$$=_w$$
$$(\lambda v_1\ldots v_n[\![j.bv]\!].\,\mathbf{destr}[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))\{\hat{\Phi}\circ\psi'\}\ \hat{\Phi}(\psi(v_1))\ \ldots\ \hat{\Phi}(\psi(v_n))\ \hat{\Phi}(k,\psi'')$$

Now let N be $\lambda[\![j.bv]\!].\,\mathbf{destr}[\vec{u}]\ \lambda\vec{u}.\Phi(j.body)$, so we may abbreviate the last term as $(\lambda v_1\ldots v_n.N)\{\hat{\Phi}\circ\psi'\}\ \hat{\Phi}(\psi(v_1))\ \ldots\ \hat{\Phi}(\psi(v_n))\ \hat{\Phi}(k,\psi'')$.

By Lemma 6, each term in $\hat{\Phi}(\psi(v_1))\ \ldots\ \hat{\Phi}(\psi(v_n))$ is a value, and we can repeatedly invoke the β and **cong** rules for $=_w$, as follows:

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

The first step is an explicit α-conversion where v_1' is fresh. Each $=_w$ step represents a use of the β 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 4, $\forall x \in \theta_{i.rator}$, $x \in Dom(\psi) \cap Dom(\psi')$, and $\psi(x) = \psi'(x)$. By the **app-tag-cons** rule, since $\pi_{i.rator} = cl_{\langle v_1,\ldots,v_n \rangle}$, $\{v_1, \ldots, v_n\} \subseteq \theta_{i.rator}$. So in the next-to-last step, in the extended ψ', we can replace $\psi(v_1), \ldots, \psi(v_n)$ by $\psi'(v_1), \ldots, \psi'(v_n)$.

Another α-conversion allows the substitution to be moved inside the body of the procedure:

$$\begin{aligned}
&= N\{\hat{\Phi} \circ \psi'\} \\
&= (\lambda[\![j.bv]\!].\ \mathbf{destr}\,[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))\{\hat{\Phi} \circ \psi'\} \\
&= (\lambda x'.(\mathbf{destr}\,[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))[x'/[\![j.bv]\!]])\{\hat{\Phi} \circ \psi'\} \\
&= (\lambda x'.\ \mathbf{destr}\,[\vec{u}]\ (\lambda\vec{u}.\Phi(j.body))[x'/[\![j.bv]\!]])\{\hat{\Phi} \circ \psi'\} \\
&= (\lambda x'.\ \mathbf{destr}\,[\vec{u}]\{\hat{\Phi} \circ \psi'\}\ (\lambda\vec{u}.\Phi(j.body))[x'/[\![j.bv]\!]]\{\hat{\Phi} \circ \psi'\})
\end{aligned}$$

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:

$$\begin{aligned}
&\mathbf{app}\ \hat{\Phi}(j, \psi')\ \hat{\Phi}(\psi(v_1))\ \ldots\ \hat{\Phi}(\psi(v_n))\ \hat{\Phi}(k, \psi'') \\
=_w\ &(\lambda x'.\ \mathbf{destr}\,[\vec{u}]\{\hat{\Phi} \circ \psi'\}\ (\lambda\vec{u}.\Phi(j.body))[x'/[\![j.bv]\!]]\{\hat{\Phi} \circ \psi'\})\ \hat{\Phi}(k, \psi'')
\end{aligned}$$

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

$$\begin{aligned}
&(\lambda x'.\ \mathbf{destr}\,[\vec{u}]\{\hat{\Phi} \circ \psi'\}\ (\lambda\vec{u}.\Phi(j.body))[x'/[\![j.bv]\!]]\{\hat{\Phi} \circ \psi'\})\ \hat{\Phi}(k, \psi'') \\
=_w\ &(\mathbf{destr}\,[\vec{u}]\{\hat{\Phi} \circ \psi'\}\ (\lambda\vec{u}.\Phi(j.body))[x'/[\![j.bv]\!]]\{\hat{\Phi} \circ \psi'\})[\hat{\Phi}(k, \psi'')/x'] \\
=\ &\mathbf{destr}\,[\vec{u}]\{\hat{\Phi} \circ \psi'\}\ (\lambda\vec{u}.\Phi(j.body))[x'/[\![j.bv]\!]]\{\hat{\Phi} \circ \psi'\}[\hat{\Phi}(k, \psi'')/x'] \\
=\ &\mathbf{destr}\,[\vec{u}]\{\hat{\Phi} \circ \psi'\}\ (\lambda\vec{u}.\Phi(j.body))\{\hat{\Phi} \circ \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]\}
\end{aligned}$$

By Lemma 6, $[\vec{u}]\{\hat{\Phi} \circ \psi'\}$ is a record of values. Therefore we can simplify the last term using the **destr** rule for $=_w$. First we α-convert so we can move the substitution inside the procedure body. In the procedure, each u in $\vec{u}$ is replaced by a fresh variable:

$$\begin{aligned}
&\mathbf{destr}\,[\vec{u}]\{\hat{\Phi} \circ \psi'\}\ (\lambda\vec{u}.\Phi(j.body))\{\hat{\Phi} \circ \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]\} \\
=\ &\mathbf{destr}\,[\vec{u}]\{\hat{\Phi} \circ \psi'\}\ (\lambda\vec{u}'.\Phi(j.body)[\vec{u}'/\vec{u}])\{\hat{\Phi} \circ \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]\} \\
=\ &\mathbf{destr}\,[\vec{u}]\{\hat{\Phi} \circ \psi'\}\ (\lambda\vec{u}'.\Phi(j.body)[\vec{u}'/\vec{u}]\{\hat{\Phi} \circ \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]\}) \\
=_w\ &(\Phi(j.body)[\vec{u}'/\vec{u}]\{\hat{\Phi} \circ \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]\})[\vec{u}\{\hat{\Phi} \circ \psi'\}/\vec{u}'] \\
=\ &\Phi(j.body)\{\hat{\Phi} \circ \psi'[[\![j.bv]\!] \mapsto (k, \psi'')]\} \\
=\ &\hat{\Phi}(j.body, \psi'[[\![j.bv]\!] \mapsto (k, \psi'')])
\end{aligned}$$

With much effort, we have shown that

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

We already showed in the preceding subcase that $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \models^{\mathcal{E},\Pi}_{env}$

$A_{j.body}$. 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 5

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

hence

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

□

Corollary 1. *Let $\mathcal{E}$ be an abstract execution, and let $\Gamma_{\mathcal{E}}$ be a monovariant and locally consistent annotation map. Let Π be the protocol assignment defined by $\forall i, \Pi(i) = \pi_i$. Let i be an occurrence index, and ψ an occurrence environment such that $\psi \models^{\mathcal{E},\Pi}_{env} A_i$. If*

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

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

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

11. RELATED WORK

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

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

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 proposition associated with an occurrence represent the set of procedures to which that occurrence may evaluate. Sestoft [Sestoft 1988] originally used a closure analysis to prove the correctness of a transformation that replaces parameter-passing by assignment to global variables. Bondorf adapted Sestoft's analysis for a subset of Scheme [Bondorf 1991]. In Shivers' dissertation, he 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 used similar techniques to compute a highly-optimized version of 0CFA [Ayers 1993]. Palsberg and Schwartzbach

[Palsberg and Schwartzbach 1995] use a constraint-based closure analysis similar to ours to support *safety analysis*. Stefanescu and Zhou have formulated a generalized closure analysis framework using abstract interpretation that can handle both mono- and polyvariance [Stefanescu and Zhou 1994]. Sabry and Felleisen compare the results of closure analysis for programs written in direct style with the results from analyzing their continuation-passing style transforms [Sabry and Felleisen 1994]. Flanagan and Felleisen 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 [Flanagan and Felleisen 1995].

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

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.

In [Wand 1992], Wand 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; Appel 1992]. Fradet and Le Métayer describe how closure creation may be avoided in some cases [Fradet and Le Métayer 1991]. Most recently, starting from the work described here, Hannan has shown how to annotate programs for selective and lightweight closure conversion using a type system described in the ELF theorem prover [Hannan 1995]. Minamide, Morrisett, and Harper describe a closure conversion algorithm for ML programs that produces well-typed terms in which closures have existential type [Minamide et al. 1996].

12. 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]. We have used similar techniques in other analyses [Steckler 1994, Chapters 5,6][Steckler and Wand 1994a], so we think the methods here are of wide applicability.

REFERENCES

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

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

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

AUGUSTSSON, L. 1984. A compiler for lazy ML. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, pp. 218–27.

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

BONDORF, A. 1991. Automatic autoprojection of higher-order recursive equations. *Science of Computer Programming 17,* 1-3 (December), 3–34.

FLANAGAN, C. AND FELLEISEN, M. 1995. Set-based analysis for full scheme and its use in soft-typing. Tech. Rep. COMP TR95-253 (October), Department of Computer Science, Rice University.

FLOYD, R. W. 1967. Assigning meanings to programs. In *Proc. Symp. on Appl. Math.* American Mathematical Society.

FRADET, P. AND LE MÉTAYER, D. 1991. Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems 13,* 1 (January), 21–51.

HANNAN, J. 1995. Type systems for closure conversion. In *Types for Program Analysis*, Aarhus, Denmark, pp. 48–62. Computer Science Department, Aarhus University. Available as DAIMI PB-493.

HEINTZE, N. 1992. Set based program analysis. Ph. D. thesis, Carnegie-Mellon University.

HOARE, C. A. R. 1969. An axiomatic basis for computer programming. *Comm. ACM 12,* 576–80, 583.

JONES, N. D. 1981. Flow analysis of lambda expressions. In *International Colloquium on Automata, Languages, and Programming.*

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 9th ACM Symposium on Principles of Programming Languages*, pp. 66–74.

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

KRANZ, D. A. 1988. Orbit: An optimizing compiler for Scheme. Ph. D. thesis, Yale University.

LANDIN, P. J. 1964. The mechanical evaluation of expressions. *Computer J. 6,* 4, 308–20.

MILNER, R. 1989. *Communication and Concurrency.* International Series in Computer Science. Prentice-Hall.

MINAMIDE, Y., MORRISETT, G., AND HARPER, R. 1996. Typed closure conversion. In *Conference Record of 23rd ACM Symposium on Principles of Programming Languages*, pp. [to appear].

NIELSON, F. 1985. Program transformations in a denotational setting. *ACM Transactions on Programming Languages 7,* 3 (July), 359–79.

PALSBERG, J. AND SCHWARTZBACH, M. I. 1995. Safety analysis versus type inference. *Information and Computation 118,* 1 (April), 128–41.

PLOTKIN, G. D. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science 1*, 125–59.

SABRY, A. AND FELLEISEN, M. 1994. Is continuation-passing useful for data flow analysis? In *Proc. ACM SIGPLAN '94 Conf. on Programming Language Design and Implementation*, pp. 1–12.

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

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

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

STECKLER, P. AND WAND, M. 1994a. Selective thunkification. In B. L. CHARLIER (Ed.), *Proceedings of the 1st International Static Analysis Symposiu m*, Volume 864 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, and New York, pp. 162–78. Springer-Verlag.

STECKLER, P. AND WAND, M. 1994b. Tracking available values for lightweight closures (summary). In N. JONES AND C. TALCOTT (Eds.), *Proc. Atlantique Workshop on Semantics Based Program Manipulation*, pp. 63–70. Available as DIKU Report No. 94/12, University of Copenhagen.

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

STEELE, G. L. 1978. Rabbit: A compiler for Scheme. M.S. thesis, MIT. MIT Artificial Intelligence Laboratory Technical Report 474.

STEFANESCU, D. AND ZHOU, Y. 1994. An equational framework for the flow analysis of higher-order functions. In *Proc. 1994 ACM Symposium on Lisp and Functional Programming*, pp. 318–27.

WAND, M. 1992. Correctness of procedure representations in higher-order assembly language. In S. BROOKES (Ed.), *Proceedings Mathematical Foundations of Programming Semantics '91*, Volume 598 of *Lecture Notes in Computer Science*, pp. 294–311. Berlin, Heidelberg, and New York: Springer-Verlag.

WAND, M. 1993. Specifying the correctness of binding-time analysis. *Journal of Functional Programming 3,* 3, 365–87.

WAND, M. AND STECKLER, P. 1994. Selective and lightweight closure conversion. In *Conference Record of 21st ACM Symposium on Principles of Programming Languages*, pp. 435–45.

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.

Received December 1995.