CORRECT HIGHER-ORDER PROGRAM TRANSFORMATIONS

A Thesis
Presented to the Faculty of the Graduate School
of the College of Computer Science
of Northeastern University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Paul A. Steckler
July 1994

Abstract

We present a method for proving the correctness of compiler optimizations for higher-order programming languages. Programs are annotated with propositions derived as the solutions to dataflow constraints formulated as local consistency conditions for each parse tree node. We can prove that any such solution yields sound annotations, that is, the propositions are true. Each compiler optimization is presented as a source-to-source transformation. Using the annotations and additional constraint information, we can prove that the optimization is correct with respect to some criterion. The correctness criterion is similar for each transformation. The particular optimizations we exhibit are *selective and lightweight closure conversion* (constructing source-level closures for procedures, with two variations), *Ultra-β* (a generalization of copy propagation to higher-order languages), and *selective thunkification* (transforming call-by-name programs into call-by-value equivalents).

Acknowledgements

I would like to thank my advisor, Mitch Wand, for suggesting the line of research that led to this dissertation. His insight and patience have been invaluable.

I would also like to thank the other members of my dissertation committee, Karl Lieberherr, Bob Muller, and Jens Palsberg, for their helpful comments on earlier drafts.

Many present and former professors at the College of Computer Science at Northeastern University have been helpful to me as I pursued my doctorate. In particular, I would like to acknowledge the influence of Professors Cindy Brown, Larry Finkelstein, Richard Kelsey, Andy Klapper, Luc Longpré, and Alan Selman.

Contents

List of Figures

Chapter 1

Introduction

Compilers perform a variety of code transformations to generate a high-quality target program from a source program. Some of these transformations, such as parsing, require understanding only of the source program syntax. Other transformations require deeper, semantic analysis of the source program, based on how information will be propagated when the program is run. Any code transformation should be correct, in the sense that the effects of the transformation are precisely understood. Because computer programs and their execution are complex, designing correct code transformations is a nontrivial task. In this dissertation, we present a method for analyzing and transforming higher-order programs so that the transformations can be proved correct.

We apply our techniques to four different program transformations. Two of the transformations are optimized versions of a transformation that a compiler for a higher-order language needs to perform in any case; the other two are optimizations that yield better code than would be generated otherwise. To analyze programs, we compute an approximation of the data that can pass through individual program points, and annotate the program with that information. We compute such dataflows by solving systems of constraints associated with each program point. Each transformation has its own supporting analysis, although there is some overlap in the kinds of dataflow information computed. The transformations are guided by the information provided by the computed annotations. Our transformations are *source-to-source*, so that the result of a transformation is a program in a language similar to the source language. Using a similar correctness criterion in each case, we are able to show that the transformations meet that criterion.

1.1 Optimizing code transformations

A programming language compiler typically processes a program in phases. As presented in the standard textbook on compiler construction, the "Dragon Book," a compiler *front end* performs lexical analysis, syntactic analysis (parsing) and semantic analysis (type-checking, e.g.), followed by intermediate code generation and optimization; in the *back end*, machine code is generated [3, Chapter 1]. The boundaries between phases are sometimes blurry, but it is useful to view a compiler in terms of such an organization.

Optimizing code transformations are usually performed on intermediate code. A compiler designer has a range of choices for an intermediate language.[1] The source language used in this dissertation, Λ_{in}, may be considered (1) as a programming language in its own right, or (2) as an intermediate language produced by a compiler front end. Since our transformations are source-to-source, they can take place in the front end, at any time after a program is parsed, in the case of (1), or, more conventionally, following intermediate code generation, in the case of (2).

1.2 Transforming higher-order programming languages

The source language for our transformations, Λ_{in}, is higher-order, meaning that procedures can take other procedures as arguments, and return procedures as results. Well-known higher-order programming languages include Common Lisp [52], Scheme [14], and ML [40]. Languages that lack this capability are called first-order. Some languages, such as Pascal [27] and C [31], have restricted higher-order capabilities. The control-flow graph for a first-order program is textually apparent, since at every call site, the name of the called procedure is given:

$$\ldots\ myproc(x)\ \ldots$$

At the call site shown in this program fragment, the only procedure that may be called is *myproc*. By contrast, in a higher-order language, any procedure that happens to flow to a call site may be called. Since a called procedure may have been passed in or returned, its identity may be unknown:

$$\ldots\ ((f\ x)\ y)\ \ldots$$

[1] See [5, Chapter 1] for a discussion of some of the considerations in choosing a suitable intermediate language.

In this fragment, f is a variable that is bound to some procedure. The call $(f\ x)$ returns some other procedure, which is then applied to y. It is not apparent from the program text which procedures are actually called. Therefore, computing dataflow information for a program written in a higher-order language is a problem of a different nature than flow-analyzing a comparable first-order program.

The literature of analyses and program transformations for first-order languages is well established; the comparable literature for higher-order languages, while growing, is still relatively thin. The Dragon Book suggests a staged organization for a code optimizer [3, Figure 10.3]. The first stage in that organization is control-flow analysis, that is, constructing the control-flow graph. As suggested, it is not obvious how to do so.[2] The Dragon Book does not discuss the particular problems of program transformations for higher-order languages. Some transformations for first-order languages may also be applied to higher-order languages. We will perform two such transformations: In both cases, the higher-order setting requires a more complex analysis than in the first-order case. We will also present two variations of a transformation that is applicable only to higher-order languages. Because we are able to use it to support a range of different transformations, we believe our method is generally applicable to higher-order language transformations.

For simplicity, our source language has no assignment construct. Variable binding occurs only when procedures are applied to arguments. There are few real-world languages that lack assignments, although one can program using just the functional core of a language with imperative features. Many consider Scheme and ML "mostly functional" languages, where the imperative features are built on a core of functional constructs. Our main purpose in using such a language is to lessen the number of syntactic constructs used and simplify the model of computation, making our proofs shorter and simpler. Adding imperative features would be straightforward; the immediate goal is to establish the utility of our approach.

1.3 Abstract interpretation

The theoretical programming language community has used the method of abstract interpretation, developed by Cousot and Cousot, as a mathematically well-founded tool for performing program

[2] An unusual approach to the control-flow problem was taken by the Self '91 compiler for the Self object-oriented programming language [12]. In Self, the type of an *object* associated with a variable may not be known at compile time, because of control-flow issues similar to those we have discussed. Instead of statically computing a complete control-flow graph and propagating types along the edges, the Self '91 compiler generated code for likely types only. If an unlikely type was encountered, a generated stub invoked the compiler at run-time. Only a portion of a program's control-flow graph was built statically.

analyses [17]. Abstract interpretation has been applied to higher-order languages, so we need to contrast it with our approach.

In fact, there is more than one style of abstract interpretation. In the Cousots' original work, each state of an operational semantics for a flowchart language included an environment component. For each program point, their abstract interpretation effectively computed an approximation of the set of possible environments for states at that point. Since the environments in the set are the kinds of things actually computed by the operational semantics, this approach is sometimes called a *collecting interpretation*.

Some more recent abstract interpretations have a more denotational flavor. A traditional denotational semantics[3] for a programming language maps syntactic constructs of the language to values in some *domain*, a partially-ordered set with sufficient structure to allow interpretation of recursive definitions.[4] Values in such a domain correspond to program answers. In a denotational-style abstract interpretation, a non-standard semantics maps syntactic constructs to values in an *abstract* domain, which represents the information sought in a particular analysis. Values in the abstract domain are not necessarily sets of elements in the standard domain, as in a collecting interpretation. The Burn, Hankin, and Abramsky strictness analysis [11] is a well-known example of the denotational approach.

For any abstract interpretation, the abstract values computed should accurately capture the actual values or the actual properties of programs. In other words, the abstract interpretation should be correct. How to demonstrate correctness depends on the particular method used. For instance, the Cousots showed the consistency of their collecting interpretation by exhibiting a pair of functions, called the *abstraction* and *concretization* functions, which related abstract sets of environments to the sets of environments computed during actual execution. Burn, Hankin, and Abramsky were able to show that if their non-standard semantics indicated that a procedure was strict in one of its arguments, then in the standard semantics, in fact, the procedure was strict in that argument.

[3] See [54] or [21], for explanations of denotational semantics.

[4] Domain theory was developed largely by Dana Scott (see [46]). See [2] for the relevance of domains to abstract interpretation, and [60] for a more thorough discussion of domains.

1.4 Features of the method

We highlight some of the techniques used in our method and compare them with other approaches, including abstract interpretation.

1.4.1 Computing annotations

Rather than using a collecting interpretation or non-standard semantics, we compute our annotations by solving constraints formulated as *local consistency conditions* for each program point. Such constraints directly correspond to the use of "dataflow equations" for analyses of first-order languages [3, Section 10.5]. Of course, the analysis for a first-order program will not include the kinds of constraints we have. For instance, in each of our analyses, we set up constraints on the sets of procedures that can be passed to and returned from other procedures. As in dataflow analyses for first-order languages, we solve dataflow constraints by iterating over tentative, progressively better solutions until a fixed point is reached.

When we say program points, we just mean parse tree nodes. In the first-order case, dataflow equations are typically associated with program points at the boundaries of basic blocks, that is, at edges in the control-flow graph [3, Section 10.5]. Therefore, in the first-order case, solving dataflow equations means computing the information that is propagated along the edges in an existing control-flow graph. Solving constraints in our higher-order setting requires *iteratively* building the control-flow graph. Dataflow information can be propagated along the edges of the control-flow graph as it is built. For some annotations, it may be more efficient to propagate information along the edges of the completed graph.[5]

One of the attractive features of our method is the simplicity of the annotations and the constraints on them. Generally, we annotate programs with pieces of syntax derived directly from the program itself. For example, in one of our analyses (Chapter 4), a component of the annotations associated with program points is simply a set of variable pairs; this is typical. Solving our constraints means solving only some simple set inclusions.[6] A practical benefit of such syntactic annotations is that the implementation of our analyses does not require extraordinary mathematical sophistication on the part of the compiler writer.

[5] For example, see the SOLVE-PROTOCOL-TAGS subroutine in Chapter 3.

[6] Heintze has also modelled program analysis as the solution of constraints on sets, [23], but with a focus on solution algorithms, rather than on using solutions as the basis of program transformations.

Some components of our annotations represent aspects of program state (e.g., our *flows* and several kinds of *tags*), and so might be made the subject of an abstract interpretation, while other portions (e.g., our *invariance* sets and relations), 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 [42].

1.4.2 Verifying annotations

How do we know our annotations are correct?

For each analysis, we provide a formal semantics which relates entities seen during program execution, such as environments and values, to program annotations. Such a semantics serves the the same purpose as, for instance, the Cousots' abstraction and concretization functions, because it is used to show that annotations are related to actual program behavior. For example, we may deduce certain consequences for the evaluation of a term at a parse tree node that has the "sets of variable pairs" annotation mentioned above, given the formal semantics of such an annotation.

Of course, we have written our local consistency conditions so the computed annotations will be meaningful. As an execution model for programs, we provide a specialized version of a *structured operational semantics*. Suppose for some program we can solve the local consistency conditions to give an annotated version of the program. If certain initial conditions are met, we can show that when we run the program using the operational semantics, the entities actually seen during execution *satisfy* the program's annotations. The required notions of satisfaction are just what is given by the semantics of annotations. If the satisfaction relations hold for every subcomputation, we say the annotations are *sound*. We feel that this notion of correctness is intuitive. For each analysis, we prove such a soundness result.

A compiler writer does not have to be concerned about how the annotations have been shown correct. Once the analysis has been proven correct, the compiler writer can treat the annotations as a black box, and instead concentrate on the constraint solver and transformation. As we have suggested, solving the constraints is not difficult, and the transformations are straightforward.

1.4.3 Analysis and transformation are considered together

Many program analyses have been presented as bare analyses, without a particular transformation in mind. An important aspect of our method is that an analysis and a transformation are considered as one problem to be solved. To perform a transformation, we design an analysis in anticipation of the transformation. The information provided by the analysis determines the transformation, but we cannot prove the correctness of a transformation based on the annotations alone. As the following chapters will show, the soundness proof for an analysis and the correctness proof for the transformation it justifies are closely tied.[7] This closeness suggests that an analysis and a transformation ought to be considered together, so that ours is the right approach.

1.5 The transformations

We now give examples of each of the four transformations. Higher-order languages can be thought of as syntactically sugared versions of the λ-calculus of Church [13] with various extra features. The source language in these examples is the λ-calculus with constants, similar to the Λ_{in} language actually used in the transformations. We will write M as a metavariable to indicate an arbitrary λ-term. We use **let** as syntactic sugar for procedure application in the usual way, so that **let** $x = M$ **in** N is equivalent to $(\lambda x.N)M$. Each transformation produces terms in an output language which is an enhanced version of the common input language. Hence, in some of the transformed example programs below, there are syntactic constructs such as records, which do not appear in source programs.

1.5.1 Selective and lightweight closure conversion

A compiler for a lexically-scoped higher-order programming language typically represents a procedure by a record consisting of a piece of code and a local environment containing the values of the free variables of the original procedure. This data structure is called a *closure*.[8] In closure conversion, these data structures are built at the source-language level. Procedure creation is replaced by closure creation, and procedure application is replaced by invocation of the code part of the closure

[7] See Section 3.11 for a discussion of this issue.

[8] Compilers for C and Pascal do not need to represent procedures as closures, because of the limitations on their higher-order capabilities.

on the local environment and the actual parameter. We present two versions of closure conversion in Chapters 3 and 4. Even though a compiler for a higher-order language needs to create procedure representations of some kind, until the work described in this dissertation, the correctness of such a transformation had not been proved.

For example, we might convert

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

to

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

where **app** is defined by

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

The procedure f takes two arguments and returns a procedure of one argument. In the closure-converted version of f, the returned procedure is represented by a record. The transformed version of g takes a procedure represented as a closure record and applies the code part of the closure to the local environment (the record of free variables) and the actual parameter. **destr** destructures the local environment and binds the pieces to x and y, recreating the desired bindings for the procedure body $(x - y - z)$.

Notice that g did not become a closure record under the transformation. Since some procedures may become closures and others may not, we say the transformation is *selective*. Since the **app** procedure is used to apply closure records, there are two basic protocols for procedure application; a given call site obeys one of them. The dataflow analysis is responsible for assuring that all procedures flowing to a call site expect to use the same protocol.

As an optimization, we can transform the program

$$\begin{array}{l}\textbf{let } x = c_1\\ \textbf{in let } f = \lambda y.x + y + z\\ \quad \textbf{in } f\ c_2\end{array}$$

to

$$\begin{array}{l}\textbf{let } x = c_1\\ \textbf{in let } f = [(\lambda exy.\,\textbf{destr}\ e\ (\lambda z.x + y + z)), [z]]\\ \quad \textbf{in } f\ x\ c_2\end{array}$$

Instead of putting x in the local environment of the closure record, we supply it as an extra argument at the call site. Since the closure record omits some of the procedure's free variables, we say the closure is *lightweight*. Such a closure requires less memory than its "fullweight" counterpart.[9]

Our analysis is clever enough to track data around loops. Let us add a constructor **letrec**, which allows us to create recursive procedures:

$$\begin{array}{l}\lambda x.\ \ldots\ \textbf{letrec } f = (\lambda n.\textbf{if } (\textbf{zero?}\ n)\ \textbf{then } n\ \textbf{else } (+\ x\ (f\ (-\ n\ 1))))\\ \qquad\quad \textbf{in } f\ c\end{array}$$

Note that x is free in the procedure. Closure converting f, we get:

$$\begin{array}{l}\lambda x.\ \ldots\ \textbf{letrec } f = [\lambda en.\,\textbf{destr}\ e\ (\lambda x.\textbf{if } (\textbf{zero?}\ n)\ \textbf{then } n\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } (+\ x\ (\textbf{app } f\ (-\ n\ 1))), [x]]\\ \qquad\quad \textbf{in app } f\ c\end{array}$$

Since x is not rebound across loop iterations, we can leave it out of the closure:

$$\begin{array}{l}\lambda x.\ \ldots\ \textbf{letrec } f = [\lambda exn.\,\textbf{destr}\ e\ (\lambda.\textbf{if } (\textbf{zero?}\ n)\ \textbf{then } n\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } (+\ x\ (\textbf{app } f\ x\ (-\ n\ 1))), []]\\ \qquad\quad \textbf{in app } f\ x\ c\end{array}$$

At the two sites where f may be applied, x is added as an extra argument. Note that we have written the second argument to **destr** as a procedure without arguments. That argument should

[9] In a sense, lightweight closure conversion is a finer-grained version of λ-*lifting* [6]. Some procedures have *all* their free variables supplied as extra arguments at their call sites. Such procedures do not require closures at all, and so can be "lifted" to an outermost lexical scope. We do not avoid closure creation, and do not lift procedures.

be understood as just the body of the "procedure." This is a convention we will use whenever we encounter a closure record with an empty record of free variables.

Higher-order procedures can escape the bindings of their free variables. It is possible for a procedure to be called at a site where its free variables are in scope, but where the variables have different bindings than they had where the procedure was defined. Therefore, another task for the dataflow analysis is to make sure that any variables left out of a closure will have the right values at the closure's call sites.

In Chapter 4, we give lightweightness a twist: Instead of supplying at call sites the same variables that we omit from closures, we may supply different variables, which we know to have the same values as the omitted variables. If we can do so, we say the values of the omitted variables are *available* at the call sites. The omitted variables may not be in scope or have the wrong values at some or all of those sites, but each omitted variable has some representative variable in scope at the call site which is bound to the correct, needed value.

1.5.2 Ultra-β

If a variable can be shown to be a copy of another variable, the copy can be replaced by the original, and the binding creating the copy eliminated. Eliminating binding operations may reduce the execution time of the transformed program. The analysis for performing *copy propagation* on programs written in a typical first-order language is well understood; see [3, Section 10.7]. The fundamental idea is to track the copies present on entry to and exit from basic blocks. In a higher-order language, as we shall see in Chapter 5, the basic idea is the same, but the analysis is considerably more involved. Since our input language has no assignments, copies are established only by the binding of formal parameters to the values of actual arguments.

In the simplest case, replacing a copy variable by its original corresponds to the β-rule of λ-calculus:

$$(\lambda x.x)\ y \longrightarrow_{\beta} y$$

Inside the body of the procedure, x is a copy of y, so we can replace the entire expression by y.

Now consider a more complicated program:

$$\begin{array}{l}\textbf{let } y = \ldots \\ \textbf{in let } f = \lambda x.\ \ldots\ x\ \ldots \\ \quad \textbf{in if } (z = 0) \textbf{ then } f\ y \textbf{ else } \ldots\end{array}$$

Inside the body of f, the reference to x is a copy of y. Suppose we naively replace the definition of f with just the body of f, with y substituted for x. Now suppose the evaluation of the body of f raises an exception or loops. Since the then-part of the conditional is not necessarily evaluated, the naive transformation may introduce unwanted consequences.

Instead, we transform to:

$$\begin{array}{l}\textbf{let } y = \ldots \\ \textbf{in let } f = \textbf{thunk}(\ldots\ y\ \ldots) \\ \quad \textbf{in if } (z = 0) \textbf{ then } (\textbf{run } f) \textbf{ else } \ldots\end{array}$$

By removing the formal argument, the procedure f becomes a thunk (a procedure without parameters). At the call site in the then-part of the conditional, we introduce a **run** as an extra piece of syntax. Running the thunk causes the body of the thunk to be evaluated. Since the thunk takes no arguments, the operand y at the call site is no longer needed. If the then-part is never evaluated, then the thunk is never run, so any undesirable consequences resulting from evaluating the body of f are avoided.

Of course, the dataflow patterns that establish that a variable is a copy of another can be as complex as one likes (again, flow around loops provides a challenge), and our analysis should be able to detect as many as possible.

Recognizing the kinship of such a transformation to β-reduction, Shivers called his similar transformation for Scheme programs Super-β [49, Chapter 10]. In his dissertation, Shivers discussed some of the issues involved in his transformation, but did not describe his algorithm in detail. We describe our transformation completely, and prove it correct; hence the name, Ultra-β.

1.5.3 Selective thunkification

Most programming languages are evaluated call-by-value. That is, an argument to a procedure is evaluated when the procedure is applied. Under the alternative strategy, call-by-name (sometimes called "lazy evaluation"), evaluation of procedure arguments is delayed until the argument is actually needed. It is well known that a call-by-name evaluation strategy can be simulated by call-by-value by enclosing all procedure arguments in thunks. Each time an argument is needed, its generated thunk is run. This is potentially wasteful, since a given thunk may have its body evaluated several times. Therefore, it is better to avoid generating thunks when possible. On the other hand, we do not want to avoid thunks in all cases, because in the original call-by-name program, an argument may not be evaluated at all. If we force the evaluation of an argument by not thunkifying it, the same program that ran perfectly well under call-by-name may loop or raise an exception under call-by-value. We perform *selective* thunkification, so that some arguments at call sites are thunkified, and others not. Because some arguments may not be thunkified, the transformed program may run faster than a naive, all-thunks transformation. This transformation is the subject of Chapter 6.

If we are certain to evaluate an argument under call-by-name, then we should avoid thunkifying that argument. Our *strictness analysis* determines that some arguments are certain to be evaluated. So the program

$$(\lambda x.x)\ c$$

should not be changed for running call-by-value. But for the program

$$(\mathbf{if}\ (z = 0)\ \mathbf{then}\ (\lambda x.y)\ \mathbf{else}\ (\lambda z.z))\ M$$

our strictness analysis cannot determine whether the term M will be evaluated, so the transform should be:

$$(\mathbf{if}\ (z = 0)\ \mathbf{then}\ (\lambda x.c)\ \mathbf{else}\ (\lambda y.\,\mathbf{run}\ y))\ (\mathbf{thunk}\ M)$$

In the closure conversion and Ultra-β transformations, there are issues of protocol agreement for procedure application. In selective thunkification, variable references use a protocol. So in this last example, the variable reference y became **run** y under the transformation. Since each variable reference in a transformed program uses a specific protocol, a variable must be bound only to ordinary values, otherwise only to thunks. Our dataflow analysis assures that the correct variable reference protocol is always observed.

1.6 Overview of the dissertation

The remainder of the dissertation is organized as follows:

- In Chapter 2, we describe a call-by-value language framework used to support the closure conversion and Ultra-β analyses and transformations. The framework includes a formal description of the input language and two different evaluators. We also present an equational reasoning system that simplifies a long proof in Chapter 3.
- In Chapters 3–6, we present the transformations described in the last section. In Chapter 6, the language framework is modified to handle call-by-name evaluation.
- In Chapter 7, we conclude with reflections on what has been accomplished, and possibilities for future work.

Each of Chapters 3–6, describing the transformations, follows a similar pattern. First, we give the syntax and semantics of the annotations that are used in the particular analysis. Next, we present a set of local consistency conditions on annotations. For each set of local constraints, we provide a solution algorithm (at varying levels of detail). We then prove a *Soundness Theorem*, which states that any solution to the local consistency conditions makes the annotations true. After describing the particular transformation, we prove a *Correctness Theorem*, which states that any solution to the local consistency conditions also justifies the transformation.

Chapter 2

A language framework

In this chapter, we present a first version of a language framework on which we will build our analyses and transformations. This framework consists of a source language and its evaluator; some notation for describing occurrences of terms and environments for closing occurrences; a fine-grained evaluator that preserves occurrence and environment source information; and an equational reasoning system.

2.1 The source language

We present the source language on which each transformation will operate. Our input language, Λ_{in}, is an untyped λ-calculus with constants, boolean values, conditionals, and primitive operators, given by the following grammar:

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

where

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

The metavariable c ranges over an infinite set of constants indexed by the integers. Sometimes we will use the words "ordinary constant" to refer to one of these constants, but often we will use the word "constant" to refer to either one of these constants or a boolean value.

To emphasize that Λ_{in} may be considered a programming language, we will usually refer to

λ-abstractions 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 syntactically equal, that is, the terms are identical, with possibly different names for their bound variables.

The grammars we write describe the abstract structure of terms, rather than their concrete syntax. Since the grouping of terms is ambiguous by the above grammar, we will often include parentheses when writing terms in Λ_{in}. For the same reason, we will use parentheses when writing terms described by other grammars.

For now, a *value* is any variable, constant, boolean value, or procedure in Λ_{in}. Values are given by the grammar:

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

where the metavariable M ranges over terms in Λ_{in}. Each transformation will produce terms in its own output language. In subsequent chapters, we will need to expand the definition of values to include certain terms in the various output languages. A *scalar value* is a boolean value or ordinary 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 on terms and their values. This evaluator, and the others we will show later, are examples of *structured operational semantics.* An evaluation derivation using such an evaluator takes the form of a proof tree. Later, in Chapter 6, we will look at a term evaluator that uses a call-by-name strategy.

From these evaluation rules, it is clear that any value in Λ_{in} evaluates to itself.

2.2 Occurrences of terms

For each analysis in the following chapters, we want to distinguish occurrences of terms. A fundamental drawback of the term evaluator is that substitution discards source information about terms. 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

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

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

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

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

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

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

Figure 1: Rules for the term evaluator

indices for source programs are finite strings over the alphabet

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

An occurrence index for a source program term describes the path from the root of the parse tree to the occurrence of the term.[1] 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

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

If M is a term with occurrence index i, then $[\![i]\!] = M$. For terms with subterms, we may separately bracket the indices for the subterms. For instance, an application $[\![i]\!]$ may also be written as $[\![i.rator]\!]\ [\![i.rand]\!]$. Similarly, for a procedure with index i, we may write $\lambda[\![i.bv]\!].[\![i.body]\!]$.

[1] Other approaches used to specify occurrences include generating a unique label for each occurrence [48], and α-converting a program so that each binding occurrence is unique, and using the binding variable as a token [43]. 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 know the indices of its subterms.

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.*

So:

Lemma 1 *The empty environment* $\emptyset$ *is a scalar environment.*

Proof. Immediate from the definition of scalar environment. ∎

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

Proof. By the definitions of scalar environment and chain-free. ∎

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 regular and infinite set of strings, disjoint from the set of program occurrences. 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. That is, for any ordinary constant or boolean value 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 refer to constants.

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

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

2.3 Relating the term and occurrence evaluators

The occurrence evaluator uses occurrence closures to simulate the substitutions of the term evaluator. The evaluation relation $\underset{oc}{\Longrightarrow}$ defines a relation between occurrence closures that represent arbitrary terms and the occurrence closures representing the values of those terms. The rules for the occurrence evaluator are given in Figure 2.

We use the predicates

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

to test the syntactic category of occurrences of terms in Λ_{in}. The predicate *Const* is true for an occurrence of a scalar value.

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

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

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

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

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

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

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

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

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

Figure 2: Rules for the occurrence evaluator

2.3.1 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.

Examples

Suppose

$$\begin{array}{rcl} [\![i]\!] &=& x \\ [\![j]\!] &=& (\lambda z.z), \text{ and} \\ \psi &=& \emptyset[x \mapsto (j,\emptyset)] \end{array}$$

Then

$$\begin{array}{rcl} \mathcal{U}[\![(i,\psi)]\!] &=& [\![i]\!]\{\mathcal{U}[\![\cdot]\!] \circ \psi\} \\ &=& x\{\mathcal{U}[\![\cdot]\!] \circ \emptyset[x \mapsto (j,\emptyset)]\} \\ &=& \mathcal{U}[\![(j,\emptyset)]\!] \\ &=& [\![j]\!]\{\mathcal{U}[\![\cdot]\!] \circ \emptyset\} \\ &=& (\lambda z.z)\{\mathcal{U}[\![\cdot]\!] \circ \emptyset\} \\ &=& (\lambda z.z) \end{array}$$

In this example, ψ is only one level deep.

In the following example, we have a deeper occurrence environment. Suppose

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

Then

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

2.3.2 Simulation and adequacy

The occurrence evaluator was written so that its operation simulates the operation of the call-by-value term evaluator. If we can evaluate an occurrence closure using the occurrence evaluator, we can evaluate the term represented by the occurrence closure using the term evaluator; we formalize this notion below. This strategy can be generalized. In Chapter 6, we will build an occurrence evaluator that simulates a call-by-name term evaluator.

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

Lemma 3 *If i is the occurrence index of a value, then for any occurrence environment ψ, $\mathcal{U}[\![(i,\psi)]\!]$ is a value.*

Proof. Induction on the depth of ψ.

Base case $\psi = \emptyset$

We have

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

By assumption, $[\![i]\!]$ is a value.

Induction $\psi \neq \emptyset$

We have to consider the syntactic category of $[\![i]\!]$.

case $Const(i)$

Then

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

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

In this case

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

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

We have

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

By Definition 1, the index-part of the occurrence closure $\psi([\![i]\!])$ is the index of a value, so $\mathcal{U}[\![\psi([\![i]\!])]\!]$ is a value by the induction hypothesis.

case $Abs(i)$

So

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

which is a substitution instance of a procedure. ∎

By this lemma, any occurrence closure in the range of an occurrence environment represents a value. Therefore, we will sometimes informally refer to such occurrence closures as "values."

Now we can show:

Theorem 1 (Simulation) *If* $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$ *then* $\mathcal{U}\llbracket(i, \psi)\rrbracket \underset{t}{\Longrightarrow} \mathcal{U}\llbracket(j, \psi')\rrbracket$.

The following diagram illustrates the theorem:

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

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

Base cases

case $Var(i) \wedge \llbracket i\rrbracket \notin Dom(\psi)$

Then $(i, \psi) \underset{oc}{\Longrightarrow} (i, \emptyset)$, and

$$\begin{array}{rcl}
\mathcal{U}\llbracket(i, \psi)\rrbracket & = & \llbracket i\rrbracket\{\mathcal{U}\llbracket\cdot\rrbracket \circ \psi\} \\
 & = & \llbracket i\rrbracket \\
 & \underset{t}{\Longrightarrow} & \llbracket i\rrbracket \\
 & = & \llbracket i\rrbracket\{\mathcal{U}\llbracket\cdot\rrbracket \circ \emptyset\} \\
 & = & \mathcal{U}\llbracket(i, \emptyset)\rrbracket
\end{array}$$

case $Var(i) \wedge \llbracket i\rrbracket \in Dom(\psi)$

Then $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$ where $(j, \psi') = \psi(\llbracket i\rrbracket)$, and

$$\begin{array}{rcl}
\mathcal{U}\llbracket(i, \psi)\rrbracket & = & \llbracket i\rrbracket\{\mathcal{U}\llbracket\cdot\rrbracket \circ \psi\} \\
 & = & \mathcal{U}\llbracket\psi(\llbracket i\rrbracket)\rrbracket \\
 & = & \mathcal{U}\llbracket(j, \psi')\rrbracket \\
 & \underset{t}{\Longrightarrow} & \mathcal{U}\llbracket(j, \psi')\rrbracket
\end{array}$$

where the last step follows from Lemma 3 and the fact that values self-evaluate.

case $Const(i)$

Then $(i, \psi) \underset{oc}{\Longrightarrow} (i, \emptyset)$, and

$$\begin{array}{rcl} \mathcal{U}[\![(i,\psi)]\!] & = & [\![i]\!]\{\mathcal{U}[\![\cdot]\!] \circ \psi\} \\ & = & [\![i]\!] \\ & \underset{t}{\Longrightarrow} & [\![i]\!] \\ & = & \mathcal{U}[\![(i,\emptyset)]\!] \end{array}$$

case $Abs(i)$

We have $(i, \psi) \underset{oc}{\Longrightarrow} (i, \psi)$, and by Lemma 3, $\mathcal{U}[\![(i,\psi)]\!]$ is a value which evaluates to itself.

Induction

case $PrimApp(i)$

By one of the rules involving primitive operators, there is a derivation that $(i, \psi) \underset{oc}{\Longrightarrow} (c, \emptyset)$, where $Const(c)$. That derivation has a subproof $(i.rand, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$, where $Const(j)$. The relation between $[\![j]\!]$ and $[\![c]\!]$ depends on the operator involved.

We wish to evaluate $\mathcal{U}[\![(i,\psi)]\!]$. We have:

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

The third step holds because the substitution has no effect on the operator.

By the induction hypothesis

$$\begin{array}{rcl} \mathcal{U}[\![(i.rand,\psi)]\!] & \underset{t}{\Longrightarrow} & \mathcal{U}[\![(j,\psi')]\!] \\ & = & [\![j]\!]\{\mathcal{U}[\![\cdot]\!] \circ \psi'\} \\ & = & [\![j]\!] \end{array}$$

So by the appropriate term evaluation rule, depending on the operator, $\mathcal{U}[\![(i,\psi)]\!] \underset{t}{\Longrightarrow} c = \mathcal{U}[\![(c,\emptyset)]\!]$.

case $App(i)$

Suppose we have the derivation:

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

By the definition of unwinding

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

By the induction hypothesis, we have:

$$\begin{array}{rl}
\mathcal{U}[\![(i.rator,\psi)]\!] \underset{t}{\Longrightarrow} \mathcal{U}[\![(j,\psi')]\!] & \text{IH at } i.rator \\
\mathcal{U}[\![(i.rand,\psi)]\!] \underset{t}{\Longrightarrow} \mathcal{U}[\![(k,\psi'')]\!] & \text{IH at } i.rand
\end{array}$$

Now, by replacing the binding variable with a fresh variable x', we have

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

To use the application rule for terms, we want to evaluate the body of this procedure with the value of the operand substituted for the binding variable. But

$$\begin{aligned}
& [\![j.body]\!][x'/[\![j.bv]\!]]\{\mathcal{U}[\![\cdot]\!]\circ\psi'\}[\mathcal{U}[\![(k,\psi'')]\!]/x'] \\
= \ & [\![j.body]\!]\{\mathcal{U}[\![\cdot]\!]\circ\psi'[[\![j.bv]\!] \mapsto (k,\psi'')]\} \\
= \ & \mathcal{U}[\![(j.body,\psi'[[\![j.bv]\!] \mapsto (k,\psi'')])]\!] \\
\underset{t}{\Longrightarrow} \ & \mathcal{U}[\![(m,\psi''')]\!]
\end{aligned}$$

where the evaluation step holds by the induction hypothesis at $j.body$. To see that the first equality holds, consider the effects of the substitutions on both sides of the equation. For $[\![j.bv]\!]$, we substitute $\mathcal{U}[\![(k,\psi'')]\!]$, since x' is fresh and not in the domain of ψ'. For any other variable x, $x \neq [\![j.bv]\!]$, we substitute $\mathcal{U}[\![\psi'(x)]\!]$. The next equality holds by the definition of unwinding. By the rule for applications, then, $\mathcal{U}[\![(i,\psi)]\!] \underset{t}{\Longrightarrow} \mathcal{U}[\![(m,\psi''')]\!]$.

case $Cond(i)$

There are two subcases here, depending on the result of the evaluation of the test. We show only one of the subcases; the other is similar.

Suppose we have the derivation:

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

We want to show that $\mathcal{U}[\![(i,\psi)]\!] \underset{t}{\Longrightarrow} \mathcal{U}[\![(k,\psi'')]\!]$. We have

$$\begin{array}{ll} & \mathcal{U}[\![(i,\psi)]\!] \\ = & [\![i]\!]\{\mathcal{U}[\![\cdot]\!] \circ \psi\} \\ = & (\mathbf{if}\ [\![i.test]\!]\ \mathbf{then}\ [\![i.then]\!]\ \mathbf{else}\ [\![i.else]\!])\{\mathcal{U}[\![\cdot]\!] \circ \psi\} \\ = & \mathbf{if}\ \mathcal{U}[\![(i.test,\psi)]\!]\ \mathbf{then}\ \mathcal{U}[\![(i.then,\psi)]\!]\ \mathbf{else}\ \mathcal{U}[\![(i.else,\psi)]\!] \end{array}$$

By the induction hypothesis,

$$\mathcal{U}[\![(i.test,\psi)]\!] \underset{oc}{\Longrightarrow} \mathcal{U}[\![(j,\psi')]\!]$$

and

$$\mathcal{U}[\![(i.then,\psi)]\!] \underset{oc}{\Longrightarrow} \mathcal{U}[\![(k,\psi'')]\!]$$

Since $\mathcal{U}[\![(j,\psi')]\!] = \mathbf{true}$, by the appropriate term evaluation rule for conditionals,

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

∎

The converse of the theorem is false, which we can show by counterexample. Suppose $Var(i)$ and $Var(j)$, and $\psi(\llbracket i \rrbracket) = (j, \emptyset)$. Then

$$\begin{array}{rcl} \mathcal{U}\llbracket (i, \psi) \rrbracket & = & \llbracket i \rrbracket \{ \mathcal{U}\llbracket \cdot \rrbracket \circ \psi \} \\ & = & \mathcal{U}\llbracket \psi(\llbracket i \rrbracket) \rrbracket \\ & = & \mathcal{U}\llbracket (j, \emptyset) \rrbracket \\ & = & \llbracket j \rrbracket \{ \mathcal{U}\llbracket \cdot \rrbracket \circ \emptyset \} \\ & = & \llbracket j \rrbracket \\ & \underset{t}{\Longrightarrow} & \llbracket j \rrbracket \\ & = & \mathcal{U}\llbracket (i, \psi) \rrbracket \end{array}$$

But $(i, \psi) \underset{oc}{\not\Longrightarrow} (i, \psi)$.

Why does the converse fail? There are infinitely many occurrence closures that represent a term. Intuitively, there is more information in an occurrence closure than in its unwind. The greater amount of information in the evaluation of an occurrence closure can imply the lesser amount in the evaluation of its unwind, but the reverse cannot be true.

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 (and many theorems we will prove subsequently).

When designing a denotational semantics for a language that has an existing operational semantics, a design goal is that the denotational semantics and operational semantics co-designate a value for terminating programs (see [60, Section 11.4]). If we can do so, we say that the denotational semantics is *adequate* for the operational semantics.

Given an adequate denotational semantics, we can reason about programs denotationally, knowing that our conclusions are sound for the operational semantics. Here, we consider the term evaluator as primary, and show that our occurrence evaluator is adequate for the term evaluator. Therefore, we can reason using the occurrence evaluator and make sound conclusions about the evaluation of terms.

The Adequacy Theorem requires a technical lemma.

Lemma 4 *Suppose ψ is chain-free and $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$. If $Var(j)$, then $\llbracket j \rrbracket \notin Dom(\psi')$. If $Abs(j)$, then ψ' is chain-free.*

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

Base cases

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

Then $(i, \psi) \underset{t}{\Longrightarrow} (i, \emptyset)$.

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

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

case $Const(i)$

Then $(i, \psi) \underset{t}{\Longrightarrow} (i, \emptyset)$.

case $Abs(i)$

Then $(i, \psi) \underset{t}{\Longrightarrow} (i, \psi)$. By assumption, ψ is chain-free.

Induction

case $PrimApp(i)$

Then $(i, \psi) \underset{oc}{\Longrightarrow} (c, \emptyset)$, where $Const(c)$.

case $App(i)$

So we have an evaluation:

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

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 $Abs(k)$, 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 $[\![m]\!] \notin Dom(\psi''')$ and if $Abs(m)$, then ψ''' is chain-free.

case $Cond(i)$

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

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

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

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 $\exists j, \psi'$ such that $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$ and $M = \mathcal{U}[\![(j, \psi')]\!]$.*

Proof. Induction on the size of the derivation that $\mathcal{U}[\![(i, \psi)]\!] \underset{t}{\Longrightarrow} M$.

Base cases

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

Then

$$\begin{aligned} \mathcal{U}[\![(i, \psi)]\!] &= [\![i]\!]\{\mathcal{U}[\![\cdot]\!] \circ \psi\} \\ &= [\![i]\!] \\ &\underset{t}{\Longrightarrow} [\![i]\!] \end{aligned}$$

We have $(i, \psi) \underset{oc}{\Longrightarrow} (i, \emptyset)$ and

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

as desired.

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

By Lemma 3, $\mathcal{U}[\![(i, \psi)]\!]$ is a value, which self-evaluates. We have $(i, \psi) \underset{oc}{\Longrightarrow} \psi([\![i]\!])$, and

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

case *Const*(i)

Then

$$\begin{array}{rcl} \mathcal{U}[\![(i,\psi)]\!] & = & [\![i]\!]\{\mathcal{U}[\![\cdot]\!]\circ\psi\} \\ & = & [\![i]\!] \\ & \underset{t}{\Longrightarrow} & [\![i]\!] \end{array}$$

and $(i,\psi) \underset{oc}{\Longrightarrow} (i,\emptyset)$. As desired,

$$\begin{array}{rcl} \mathcal{U}[\![(i,\emptyset)]\!] & = & [\![i]\!]\{\mathcal{U}[\![\cdot]\!]\circ\emptyset\} \\ & = & [\![i]\!] \end{array}$$

case *Abs*(i)

Then $\mathcal{U}[\![(i,\psi)]\!]$ is a substitution instance of a procedure, which self-evaluates, and $(i,\psi) \underset{oc}{\Longrightarrow} (i,\psi)$.

Induction

case *PrimApp*(i)

Then

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

and the evaluation derivation is

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

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

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 4, $[\![j]\!] \notin Dom(\psi')$. 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,

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

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

case $App(i)$

Then

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

and the evaluation derivation must be

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

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 Lemma 4, 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

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

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 4, ψ' is chain-free. Also by Lemma 4, if $Var(k)$, then $[\![k]\!] \notin Dom(\psi'')$; if $Abs(k)$, 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

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

By the occurrence evaluator rules, $(i, \psi) \underset{oc}{\Longrightarrow} (m, \psi''')$.

case $Cond(i)$

Again, we show only one subcase. We have

$$\begin{aligned} \mathcal{U}[\![(i, \psi)]\!] &= [\![i]\!]\{\mathcal{U}[\![\cdot]\!] \circ \psi\} \\ &= (\textbf{if } [\![i.test]\!] \textbf{ then } [\![i.then]\!] \textbf{ else } [\![i.else]\!])\{\mathcal{U}[\![\cdot]\!] \circ \psi\} \\ &= \textbf{if } \mathcal{U}[\![(i.test, \psi)]\!] \textbf{ then } \mathcal{U}[\![(i.then, \psi)]\!] \textbf{ else } \mathcal{U}[\![(i.else, \psi)]\!] \end{aligned}$$

Suppose we have the evaluation:

$$\frac{\mathcal{U}[\![(i.test, \psi)]\!] \underset{t}{\Longrightarrow} \textbf{true} \qquad \mathcal{U}[\![(i.then, \psi)]\!] \underset{oc}{\Longrightarrow} M}{\mathcal{U}[\![(i, \psi)]\!] \underset{oc}{\Longrightarrow} M}$$

By the induction hypothesis at $i.test$, there are j and ψ' such that $(i.test, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$, and $\mathcal{U}[\![(j, \psi')]\!] = \mathbf{true}$. Now, $[\![j]\!]$ must be a value. But $[\![j]\!]$ cannot be a procedure, else the unwind would be a procedure. If $[\![j]\!]$ were a variable, then by Lemma 4, $[\![j]\!] \notin Dom(\psi')$, so the unwind would be just $[\![j]\!]$. So $[\![j]\!]$ must be the constant $\mathbf{true}$.

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

So by the appropriate conditional rule in the occurrence evaluator,

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

■

2.4 An equational reasoning system

Here we describe a system for reasoning equationally about terms in Λ_{in}. Our motivation is to simplify the presentation of a proof in Chapter 3. In that chapter, we will need to extend the equational system so that it may be applied to terms in an output language. When we need the extended relation, we will add new closure conditions to the rules presented now.

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.*

Weakly-equivalent terms co-evaluate.

Theorem 3 (Co-evaluation) *If $M =_w N$ then $M \underset{t}{\Longrightarrow} V$ iff $N \underset{t}{\Longrightarrow} V$.*

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

Base cases

case $M = N$ by the α rule

Use an easy induction on the structure of terms.

case $M = N$ by the β rule

M must be of the form $(\lambda x.P)Q$, where Q a value, and so $N = P[Q/x]$. Then $(\lambda x.P) \underset{t}{\Longrightarrow} (\lambda x.P)$,

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

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

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

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

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

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

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

Figure 3: Closure conditions for $=_w$

and $Q \underset{t}{\Longrightarrow} Q$. By the evaluation rule for applications, $(\lambda x.P)\ Q \underset{t}{\Longrightarrow} V$ iff $P[Q/x] \underset{t}{\Longrightarrow} V$.

Induction

case $M =_w N$ by **sym**

So we have a proof that $N =_w M$. Then $N \underset{t}{\Longrightarrow} V$ iff $M \underset{t}{\Longrightarrow} V$, by the induction hypothesis.

case $M =_w N$ by **trans**

We must have proofs that $M =_w P$ and $P =_w N$, for some term P. By the induction hypothesis, $M \underset{t}{\Longrightarrow} V$ iff $P \underset{t}{\Longrightarrow} V$ iff $N \underset{t}{\Longrightarrow} V$. So $M \underset{t}{\Longrightarrow} V$ iff $N \underset{t}{\Longrightarrow} V$.

case $M =_w N$ by **primop**

Let $M = PQ$ and let $N = PQ'$. We have a proof that $M = N$, so it must be that $Q =_w Q'$. By the induction hypothesis, $Q \underset{t}{\Longrightarrow} V$ iff $Q' \underset{t}{\Longrightarrow} V$. If Q and Q' co-evaluate to an ordinary constant, then by the relevant evaluation rule, PQ and PQ' co-evaluate to a constant. Otherwise, if Q and Q' do not evaluate, or they evaluate to a value other than an ordinary constant, then neither PQ nor PQ' evaluates.

case $M =_w N$ by **cond**

Suppose $M = \textbf{if } P \textbf{ then } Q \textbf{ else } R =_w \textbf{if } P' \textbf{ then } Q' \textbf{ else } R' = N$. We must have proofs that $P =_w P'$, $Q =_w Q'$, and $R =_w R'$. By the induction hypothesis,

$$\begin{aligned} P &\underset{t}{\Longrightarrow} V_1 \text{ iff } P' \underset{t}{\Longrightarrow} V_1 \\ Q &\underset{t}{\Longrightarrow} V_2 \text{ iff } Q' \underset{t}{\Longrightarrow} V_2, \text{ and} \\ R &\underset{t}{\Longrightarrow} V_3 \text{ iff } R' \underset{t}{\Longrightarrow} V_3 \end{aligned}$$

Therefore, exactly one of the following holds:

1. $P, P' \underset{t}{\Longrightarrow}$ **true** and $Q, Q' \underset{t}{\Longrightarrow} V_2$, and both conditionals evaluate to V_2

2. $P, P' \underset{t}{\Longrightarrow}$ **false** and $R, R' \underset{t}{\Longrightarrow} V_3$, and both conditionals evaluate to V_3

3. Neither conditional may be evaluated

case $M =_w N$ by **cong**

Suppose $M = PZ =_w P'Z' = N$. Then we have proofs that $P =_w P'$ and $Z =_w Z'$. By the induction hypothesis, $P \underset{t}{\Longrightarrow} V$ iff $P' \underset{t}{\Longrightarrow} V$. If V is not a procedure, then neither application evaluates. Suppose V is a procedure. By the induction hypothesis, $Z \underset{t}{\Longrightarrow} W$ iff $Z' \underset{t}{\Longrightarrow} W$. By the application rule for terms, $PZ \underset{t}{\Longrightarrow} X$ iff $P'Z' \underset{t}{\Longrightarrow} X$. ∎

When we extend the equational reasoning system in Chapter 3, we will prove a new co-evaluation result by building on the proof just given.

We conjecture that the converse of Theorem 3 is false, though we do not prove it here. Because it contains the β rule, the $=_w$ relation implies a notion of reduction [8]. A proof that the converse is false might take the tack of showing that this notion of reduction is a conservative extension [33] to the notion of reduction β for the pure λ-calculus. If so, then the notion of reduction implied by $=_w$ is Church-Rosser (confluent). That done, we would exhibit two terms M and N that cannot be evaluated, so that $M \underset{t}{\Longrightarrow} V$ iff $N \underset{t}{\Longrightarrow} V$. If M and N have no common reduct, it must be that $M \neq_w N$.

2.5 Related work

The input language is based on the λ-calculus of Church [13]. An exhaustive treatment of the untyped λ-calculus is found in Barendregt [8]. The programming language Lisp [39] and its many progeny, such as Scheme [14], and ML [40], are essentially sugared versions of the λ-calculus that use call-by-value evaluation, and add various pragmatic features and extensions.

Call-by-value evaluation rules for terms were given by Plotkin [44].

Closure evaluators have a long history in programming language theory. One of the earliest and best-known is the SECD machine of Landin [36]. In the same spirit as our simulation theorem, Plotkin related computations of the SECD machine to the values produced by a call-by-value term evaluator [44, Theorem 1]. Plotkin's "Real" function maps SECD closures to λ-terms, exactly as our unwinding function $\mathcal{U}[\![\cdot]\!]$. Jones used a call-by-value closure evaluator to perform program dataflow analyses [29]. After Plotkin, Jones used a Real function to map his closures to λ-terms. In discussing [29, Lemma 1.5], Jones noted that his closure evaluator effectively manipulates occurrences.

Closure evaluators are kin to calculi of explicit substitutions, in which substitutions are kept separate from terms across rewrite rules. Lescanne has provided an overview of several calculi with explicit substitutions [38], including that of Abadi et al. [1]. In general, calculi of explicit substitutions have not been developed with the goal of supporting compiler dataflow analysis and optimizations.

Plotkin developed an equational theory λ_V to prove the co-evaluation of substitution instances of an arbitrary program context, when the substituted terms are provably equal in the theory [44]. This is the notion of *observational equivalence*. Since a program context may be empty, provably equal terms themselves co-evaluaute. Similarly, Gunter presented an equational reasoning system for terms in the simply-typed λ-calculus; equal terms are proved to have the same interpretation in both a set model and a type frame [22, Chapter 2].

Chapter 3

Selective and Lightweight Closure Conversion

In this chapter,[1] we use our language framework to study a version of the closure conversion transformation. In Chapter 1, we saw some examples of the transformation. We now present some more complex examples, followed by the technical details of the transformation.

As mentioned, our version of closure conversion has two distinctive features. Some procedures may be transformed into explicit closures, while others may remain ordinary procedures. The transformation is in this way *selective*. If a free variable of a procedure is in scope at all the call sites of the procedure, we may be able to supply it as an extra argument at the call site, so we can leave that variable out of the closure. A closure with one or more omitted free variables is called *lightweight*; a variable supplied as an extra argument at a call site is a *dynamic* variable.

3.1 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

[1] A preliminary version of this chapter appeared as [59].

be made dynamic, and in what order those variables appear as arguments. Second, the dynamic variables at a call site must have the same value as they have at the definition sites for the procedures that flow to that site. As the following examples should suggest, how to perform the dataflow analysis is not obvious.

Consider:

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

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:

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

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 may 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 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:

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

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 $(\lambda z.zx)$, which 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:[2]

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

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 should detect 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:

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

[2] We mentioned in Chapter 1 that **letrec** allows us to create recursive procedures. We can define a term in Λ_{in}, known as applicative-order Y, which achieves this effect. Applying Y to a recursive function returns the function's fixed point. Applicative-order Y is the term $\lambda f.(\lambda x.f(\lambda y.(x\ x)\ y))\ (\lambda x.f(\lambda y.(x\ x)\ y))$. Plotkin calls this procedure Z [44].

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

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

Figure 4: Additional evaluation rules for Λ_{clos}

3.2 An output language

As the examples have shown, the closure conversion transform produces terms in an output language that includes records. The output language Λ_{clos} is given by the grammar:

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

where *PrimOp* is defined as in Chapter 2. 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 with all its fields values in Λ_{clos}. Values in Λ_{clos} are given by the grammar:

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

Note that by this definition, the empty record is a value.

In Figure 4, we give two new rules for evaluating terms in Λ_{clos}. These rules supplement the rules given for the term evaluator in Chapter 2 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:

$$\textbf{fst} = (\lambda r.\ \textbf{destr}\ r\ (\lambda xy.x))$$
$$\textbf{snd} = (\lambda r.\ \textbf{destr}\ r\ (\lambda xy.y))$$

$$
\begin{array}{l}
\mathbf{fst}\ [M, N] \\
\quad (\lambda r.\mathbf{destr}\ r\ (\lambda xy.x)) \\
\quad \Downarrow \\
\quad (\lambda r.\mathbf{destr}\ r\ (\lambda xy.x)) \\
\\
\quad [M, N] \\
\qquad M \\
\qquad \Downarrow \ \cdots \\
\qquad V_1 \\
\\
\qquad N \\
\qquad \Downarrow \ \cdots \\
\qquad V_2 \\
\quad [V_1, V_2] \\
\\
\quad \mathbf{destr}\ [V_1, V_2]\ (\lambda xy.x) \\
\qquad [V_1, V_2] \\
\qquad \Downarrow \ \cdots \\
\qquad [V_1, V_2] \\
\\
\qquad (\lambda xy.x) \\
\qquad \Downarrow \\
\qquad (\lambda xy.x) \\
\\
\qquad x[V_1/x, V_2/y] = V_1 \\
\qquad \Downarrow \\
\qquad V_1 \\
\quad V_1 \\
V_1
\end{array}
$$

Figure 5: Verifying the operation of **fst**

In Figure 5, we show a proof tree to verify the operation of **fst** ; the operation of **snd** is similar.[3]

We need to extend our equational reasoning system to handle records. In Figure 6, 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 and 6.

The co-evaluation property is maintained for the extended relation. We can use the proof of Theorem 3 to bootstrap this result.

Theorem 4 (Co-evaluation) *If* $M =_w N$ *then* $M \underset{t}{\Longrightarrow} V$ *iff* $N \underset{t}{\Longrightarrow} V$.

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

Base cases

The base cases are the same as in Theorem 3.

[3] This vertical style of presentation for a proof tree was suggested by [37].

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

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

Figure 6: Additional closure conditions for $=_w$ and Λ_{clos}

Induction

The induction-step cases for the **sym**, **trans**, **primop**, and **cond** rules are the same as in Theorem 3.

case $M =_w N$ by **rec**

So M and N must be records with an equal number n of fields, and $\forall i,\ 1 \leq i \leq n,\ M_i =_w N_i$. By the induction hypothesis, each pair of corresponding fields co-evaluates to the same value, else neither field evaluates. By the evaluation rule for records, $M \underset{t}{\Longrightarrow} V$ iff $N \underset{t}{\Longrightarrow} V$.

case $M =_w N$ by **destr**

Then $M = \textbf{destr}\ PQ$, and there are subproofs:

$$\begin{array}{rcll} P & =_w & [V_1, \ldots, V_n] & \text{for values } V_1, \ldots, V_n \\ Q & =_w & (\lambda x_1 \ldots x_n.S) & \\ S[V_1/x_1, \ldots, V_n/x_n] & =_w & N & \end{array}$$

Since $V_1, \ldots, V_n$ are values,

$$[V_1, \ldots, V_n] \underset{t}{\Longrightarrow} [V_1, \ldots, V_n]$$

so by the induction hypothesis,

$$P \underset{t}{\Longrightarrow} [V_1, \ldots, V_n]$$

Similarly, we know that

$$(\lambda x_1 \ldots x_n.S) \underset{t}{\Longrightarrow} (\lambda x_1 \ldots x_n.S)$$

so by the induction hypothesis, we have

$$Q \underset{t}{\Longrightarrow} (\lambda x_1 \ldots x_n.S)$$

By the induction hypothesis

$$S[V_1/x_1, \ldots, V_n/x_n] \underset{t}{\Longrightarrow} V \text{ iff } N \underset{t}{\Longrightarrow} V$$

So by the evaluation rule for **destr**, $M \underset{t}{\Longrightarrow} V$ iff $N \underset{t}{\Longrightarrow} V$. ∎

3.3 A language for annotations

Each transformation we perform has its own language for annotating programs. In general, our annotations are propositions about some aspect of a program. For the current closure conversion transformation, we annotate each occurrence i with an *environment proposition* A_i and a *value proposition* $\mathcal{P}_i$ such that if we run the occurrence evaluator on the occurrence closure (i, ψ), where the input ψ satisfies A_i, then the output satisfies the value proposition $\mathcal{P}_i$. If our annotations achieve this goal, we say that the annotations are *sound*. In this way, the pair $(A_i, \mathcal{P}_i)$ is the functional analog to a Hoare-style partial correctness assertion [24].

Let us informally describe the annotations that support the closure conversion transformation and their intended semantics. A value proposition is a triple consisting of

1. A *flow*, which describes the set of procedures to which an occurrence might evaluate,

2. A *protocol tag*, which describes the protocol used by all the procedures to which an occurrence might evaluate, and

3. An *invariance set*, is a set of variables whose values are left invariant by evaluation of the occurrence, in case the result is a procedure.

For an occurrence of an open term, the three components of the occurrence's associated value proposition depend on the inputs to the term. We use an environment proposition as a set of assumptions about the value propositions for the free variables of an occurrence. Therefore, an environment proposition is a map from variables to value propositions.

That said, we can define an *abstract closure* as a pair consisting of the occurrence index of a procedure and an environment proposition.

Now we can define the components of a value proposition somewhat more formally:

1. A *flow* is a finite set of abstract closures.

2. A *protocol tag* is an element of $\{id\} \cup \{cl_\sigma \mid \sigma \in Var^*\}$.

3. An *invariance set* is a finite set of variables.

For notation, we indicate a flow by the symbol ϕ, a protocol tag by π, and an invariance set by θ. By (2), above, in the protocol tag cl_σ, σ is a sequence of variables. We write $\langle v_1, \ldots, v_n \rangle$ to indicate a sequence of variables. If $\vec{v}$ is a sequence of variables, then $\lceil \vec{v} \rceil$ is the corresponding set consisting of all variables in the sequence.

Environment propositions and value propositions can be viewed as the nodes of trees, which may be infinite. For an environment proposition A, its children are $\{A(x) \mid x \in Dom(A)\}$. For a value proposition $\mathcal{P} = (\phi, \pi, \theta)$, its children are the environment propositions in the set $\{A \mid (i, A) \in \phi\}$. There is no ordering on the children of nodes. Because the children of a value proposition are determined by its flow component only, for the purposes of this tree construction, we regard value propositions as identical if their flows are identical.

In a given tree, a subtree may appear more than once. For example, suppose that for an environment proposition A, $A(x) = \mathcal{P} = (\phi, \pi, \theta)$, and $(i, A) \in \phi$. In that case, $\mathcal{P}$ has A as a parent and as a child.

Now suppose that in a given tree, the number of abstract closures is finite, so that there are only finitely many different possible flows. In that case, there are only finitely many different possible value proposition nodes in the tree. Then by the above construction, even if the tree is infinite, there are only finitely many different subtrees. Hence, the tree is a *regular tree* [16, Section 4].

3.4 The semantics of annotations

In introducing our annotations, we described informally their intended semantics, and mentioned two notions of satisfaction, one for environment propositions, another for value propositions. Here we give a formal semantics of annotations, describing two relations corresponding to those two notions of satisfaction.

One more definition is needed. Define a *protocol assignment* to be a map from occurrences to protocol tags. We write Π for such a map.

Definition 4 *The following definitions are mutually recursive.*

1. *An occurrence environment ψ satisfies an environment proposition A under a protocol assignment Π, as follows:*
$$\psi \models^{\Pi}_{env} A$$
iff $\forall x \in Dom(A)$
   (a) $x \in Dom(\psi)$ *and* $\psi(x) \models^{\Pi}_{val} A(x)$, *and*
   (b) *if* $A(x) = (\phi, \pi, \theta)$, $\psi(x) = (i, \psi')$, *and* $Abs(i)$, *then* $\forall y \in \theta$, $y \in Dom(\psi) \cap Dom(\psi')$, *and* $\psi(y) = \psi'(y)$

2. *An occurrence closure (i, ψ) satisfies a value proposition (ϕ, π, θ) under a protocol assignment Π, as follows:*
$$(i, \psi) \models^{\Pi}_{val} (\phi, \pi, \theta)$$
iff
   (a) $Const(i) \vee Var(i)$, *or*
   (b) $Abs(i)$ *and* $\Pi(i) = \pi$ *and* $\exists A$ *such that* $(i, A) \in \phi$ *and* $\psi \models^{\Pi}_{env} A$

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

For $\models^{\Pi}_{env}$, the first condition requires pointwise satisfaction of the propositions in the range of A by the values in ψ. The second condition says that an invariance set θ is a set of variables whose bindings are invariant across variable lookup in case the lookup returns a procedure.

For $\models^{\Pi}_{val}$, 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 in an abstract closure listed in the flow ϕ, and furthermore that its closing environment satisfies the environment proposition in the abstract closure. This is roughly equivalent to saying that $\mathcal{U}[\![(i, \psi)]\!]$ is a substitution instance of $[\![i]\!]$ where the substitution satisfies A.

Observe:

Lemma 5 *If ψ is a scalar environment, then for all environment propositions A such that $Dom(A) \subseteq Dom(\psi)$, $\psi \overset{\Pi}{\models}_{env} A$.*

Proof. By the definition of $\overset{\Pi}{\models}_{env}$.

We can partially order the set of value propositions by saying $(\phi, \pi, \theta) \leq (\phi', \pi', \theta')$ iff $\phi \subseteq \phi'$, $\pi = \pi'$, and $\theta' \subseteq \theta$.

Lemma 6

1. *If $(i, \psi) \overset{\Pi}{\models}_{val} \mathcal{P}$, and $\mathcal{P} \leq \mathcal{P}'$, then $(i, \psi) \overset{\Pi}{\models}_{val} \mathcal{P}'$.*
2. *If $\psi \overset{\Pi}{\models}_{env} A$ and $\forall x \in Dom(A')$, $A(x) \leq A'(x)$, then $\psi \overset{\Pi}{\models}_{env} A'$.*

Proof. Immediate from the preceding semantics of annotations and the ordering on value propositions. ∎

3.5 Local consistency

We have two kinds of annotations, environment propositions and value propositions. Our definitions in the preceding section tell us what they will look like, and the semantics just given tells us how these propositions are made true by environments (for environment propositions) or values (for value propositions). 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 goal mentioned in Section 3.3.

We wish to find an *annotation map* that associates with each program occurrence i an environment proposition A_i, and a value proposition $\mathcal{P}_i = (\phi_i, \pi_i, \theta_i)$. We will use the symbol Γ for an annotation map, so we can write $\Gamma(i) = (A_i, \mathcal{P}_i)$. We obtain such a map from a solution to the *local consistency conditions* given in Figure 7. If the local conditions hold for every occurrence in the domain of an annotation map Γ, we say that Γ is *locally-consistent.*

For each occurrence index i in the domain of an annotation map Γ, $\Gamma(i)$ is a pair of trees rooted at A_i and $\mathcal{P}_i$. As we have indicated, if the number of abstract closures in such trees is finite, the trees are regular. We are interested in a special case of this property. We say that an annotation map Γ is *monovariant* iff $\forall i \in Dom(\Gamma)$, $\Gamma(i) = (A_i, \mathcal{P}_i)$, and for all flows ϕ in all value proposition nodes in the trees rooted at A_i and $\mathcal{P}_i$, $(j, B) \in \phi$ implies $B = A_j$. Therefore, if Γ is monovariant, then $\forall i \in Dom(\Gamma)$, $\Gamma(i) = (A_i, \mathcal{P}_i)$ implies that A_i and $\mathcal{P}_i$ are the roots of regular trees.

The flow portions of the local conditions in Figure 7 are the basis of a *closure analysis*. That is, a solution to the local conditions associates each occurrence i with a flow ϕ_i, where the set of abstract closures in ϕ_i describes the set of procedures to which $[\![i]\!]$ may evaluate. If the monovariance condition holds, then each procedure has one abstract closure as its representative. Also, for every application site, there is an associated protocol described by the protocol tag for the operator. All procedures that actually flow to a call site should have the same protocol tag as the operator.

Now we can convey some intuition behind the constraints on the flows and protocol tags. Consider a term with occurrence index i. If $[\![i]\!]$ is a procedure, we have the constraint $\{(i, A_i)\} \subseteq \phi_i$, which says that $[\![i]\!]$ may evaluate to itself. For an application $[\![i]\!]$ we have the constraints, that for all abstract closures (j, B) in $\phi_{i.rator}$, $\mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}$ and $\mathcal{P}_{j.body} \leq \mathcal{P}_i$. Restricting these two constraints to flows, by the definition of the ordering on value propositions, we have $\phi_{i.rand} \subseteq \phi_{j.bv}$ and $\phi_{j.body} \subseteq \phi_i$. The first of these says that any procedure obtained by evaluating the operand may be an argument to the operator $[\![j]\!]$; the second says that any procedure returned from $[\![j]\!]$ may be a result of the application itself. Restricting these two constraints to protocol tags, we have $\pi_{i.rand} = \pi_{j.bv}$ and $\pi_{j.body} = \pi_i$. So all procedures that may be arguments to $[\![j]\!]$ must agree with the protocol for the binding variable of $[\![j]\!]$, and all procedures that may be returned from $[\![j]\!]$ must agree with the protocol for the application itself.

For a conditional, we have the constraints $\mathcal{P}_{i.then} \leq \mathcal{P}_i$ and $\mathcal{P}_{i.else} \leq \mathcal{P}_i$. Together, these say that any procedures that may be returned by the then-part or the else-part may be the result of the conditional itself, and that procedures that may be returned from either branch must agree with the protocol for the conditional.

For invariance sets, the local conditions are a bit more involved. If we have that $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$, and j is the index of a procedure, we want θ_i to be a set of variables whose bindings are the same in ψ and ψ'. With that aim in mind, the significance of most of the constraints on the invariance sets should become clear in our proof of the Soundness Theorem (Theorem 5). Here, we wish to give some intuition for the constraints for ordinary applications, where the constraints are the most

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

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

$$PrimApp(i) \Longrightarrow A_{i.rand} = A_i$$

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

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

Figure 7: Local consistency conditions for annotations

complicated. Suppose we have the evaluation proof:

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

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 $\theta_{j.body} - [\![j.bv]\!]$. So 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 constraints $\theta_i \subseteq \theta_{i.rator}$, $\theta_i \subseteq \theta_{j.body}$, and $[\![j.bv]\!] \notin \theta_i$. The other conditions for the invariance sets are needed so that $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \overset{\Pi}{\models} A_{j.body}$, in support of the induction hypothesis for the body. The details appear in the proof of Theorem 5.

3.6 A solution algorithm

Using the algorithm in Figure 8, we can always obtain a monovariant, locally-consistent annotation. We assume that all procedures will be transformed into closures. For purposes of the algorithm, let $Vars$ be the set of all variables that occur in the program to be analyzed, and let $ScopeVars_i$ be the set of variables in lexical scope for a term with occurrence index i. The predicate $BVar$ tests whether an occurrence is the binding variable of some procedure.

To assure soundness using this algorithm, we impose the restriction that program inputs are scalar values (booleans or ordinary constants). By Lemma 5, such an environment satisfies any assumptions we make about the value propositions for the program's free variables.

The algorithm makes two other pragmatic concessions. First, it does not solve for environment propositions as such. Instead, the algorithm makes sure that each binding variable's value proposition is propagated to free occurrences of that variable in the scope of the binder. Free variables of the program are initialized with an empty flow and an empty invariance set. Since a free variable can never be bound to a procedure, its protocol tag is arbitrary. Second, since we do not solve for environment propositions, the flows in the algorithm are just sets of indices, rather than abstract closures. We highlight this difference by using *flow* instead of the Greek ϕ in the presentation of the algorithm.

In the subroutine SOLVE-PROTOCOL-TAGS, we partition occurrence indices into equivalence classes using the standard disjoint-set forest operations MAKE-SET and UNION.[4] Each equivalence class represents a set of occurrences that will be given the same protocol tag. For each class, we take the intersection of the θ's associated with all call site operators in the class to obtain the largest possible set of dynamic variables for those call sites. In this way, lightweightness is maximized, given the solution to the flows. If one is content with "fullweight" closures, this subroutine can be skipped, and assign all procedures and all call sites a protocol tag of $cl_{\langle\rangle}$.

What is the running time of the presented solution algorithm? Let n be the number of nodes in the parse tree for the program to be analyzed. Therefore, there are maximally n variables in the program. For each iteration of the main loop of the algorithm (steps **(2)** to **(10)** in Figure 8), we either add at least one index to a flow, or remove at least one variable from an invariance set. After initialization, the smallest flow is the empty set, and the largest invariance set has all variables in

[4] See [15, Chapter 22] for an overview of disjoint-set forest algorithms and their analysis.

```
SOLVE-CONSTRAINTS

INITIALIZE-SETS
repeat
    done ← true
    ∀i such that App(i)
        if SOLVE-APP-CONSTRAINTS(i) = false then
            done ← false
    ∀i such that Cond(i)
        if SOLVE-COND-CONSTRAINTS(i) = false then
            done ← false
until done = true
SOLVE-PROTOCOL-TAGS
```

Figure 8: A constraint solution algorithm

the program. When we are done iterating, for each of n nodes, there are maximally n indices in the flow for the node, and minimally the empty invariance set. During each iteration, we visit $O(n)$ nodes, and the number of iterations is $O(n^2)$. So the running time is:

$$O(n) * O(n^2) * \text{cost of visiting a node}$$

Each of the flows and invariance sets is of size $O(n)$, so if we use a bit-vector representation of sets, the time to take each union and intersection is $O(n)$. When visiting a conditional node, we take two unions and two intersections. When visiting an application node, we take $O(n)$ unions and intersections. Therefore, the cost of visiting a node is $O(n^2)$, and the running time for the main loop is $O(n^5)$.

Does the protocol tag solution subroutine given in Figure 10 add to the asymptotic running time? There are n MAKE-SET operations, and $O(n^2)$ UNION operations. To see this, observe that there are $O(n)$ applications in a program. For each application, there are $O(n)$ indices in the flow for the operator. For each such index, we call UNION twice. Therefore, the number of UNION's in the loop for applications is $O(n^2)$. In the loop for procedures, there are $O(n)$ UNION's, since the total number of variables in procedure bodies is $O(n)$. The loop for conditionals executes $O(n)$ times, and there are two UNION's for each iteration. So there are $O(n^2)$ disjoint-set operations; there are n elements in the forest of disjoint sets. By the famous result of Tarjan [55], the disjoint-set operations in steps **(1)** to **(11)** take time $O(n^2 * \alpha(n^2, n))$, where α is the extremely slow-growing inverse-Ackermann function. Next, we need to consider the calculation of the dynamic variable vectors in steps **(12)** to **(15)**. The loop in those steps is executed $O(n)$ times. But the forest represents a partition of the

```
SOLVE-APP-CONSTRAINTS(i)

retval ← true
if θ_i ⊈ θ_{i.rator} then
    θ_i ← θ_i ∩ θ_{i.rator}
    retval ← false
∀j ∈ flow_{i.rator}
    if SET-LEQ-ORDER(i.rand, j.bv) = false then
        ∀k such that [[k]] ∈ FV([[j.body]]) and [[k]] = [[j.bv]]
            flow_k ← flow_{j.bv}; θ_k ← θ_{j.bv}
        retval ← false
    if SET-LEQ-ORDER(j.body, i) = false then
        retval ← false
    if θ_{j.bv} ⊈ θ_{i.rator} then
        θ_{j.bv} ← θ_{j.bv} ∩ θ_{i.rator}
        ∀k such that [[k]] ∈ FV([[j.body]]) and [[k]] = [[j.bv]]
            θ_k ← θ_{j.bv}
        retval ← false
    if [[j.bv]] ∈ θ_i then
        θ_i ← θ_i − {[[j.bv]]}
        retval ← false
return retval
```

```
SOLVE-COND-CONSTRAINTS(i)

retval ← true
if SET-LEQ-ORDER(i.then, i) then
    retval ← false
if SET-LEQ-ORDER(i.else, i) then
    retval ← false
return retval
```

Figure 9: Subroutines for the solution algorithm

INITIALIZE-SETS

$$\forall i, \mathit{flow}_i \leftarrow \begin{cases} \{i\} & \text{if } Abs(i) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\forall i, \theta_i \leftarrow \begin{cases} ScopeVars_i & \text{if } Abs(i) \\ Vars - \{[\![i]\!]\} & \text{if } BVar(i) \\ Vars - \{[\![j]\!]\} & \text{if } Var(i), BVar(j), \\ & [\![i]\!] = [\![j]\!], \text{ and} \\ & i \text{ a free occurrence in the scope of } [\![j]\!] \\ Vars & \text{otherwise} \end{cases}$$

SET-LEQ-ORDER(i, j)

$\mathit{retval} \leftarrow$ **true**
if $\mathit{flow}_i \not\subseteq \mathit{flow}_j$ **then**
  $\mathit{flow}_j \leftarrow \mathit{flow}_j \cup \mathit{flow}_i$
  $\mathit{retval} \leftarrow$ **false**
if $\theta_j \not\subseteq \theta_i$ **then**
  $\theta_j \leftarrow \theta_j \cap \theta_i$
  $\mathit{retval} \leftarrow$ **false**
return retval

SOLVE-PROTOCOL-TAGS

$\forall i$, MAKE-SET(i)
$\forall i$ such that $Abs(i)$
  $\forall j$ such that j a free occurrence of $[\![i.bv]\!]$ in $[\![i.body]\!]$
    UNION$(j, i.bv)$
$\forall i$ such that $App(i)$
  $\forall j \in \mathit{flow}_{i.rator}$
    UNION$(i.rand, j.bv)$
    UNION$(j.body, i)$
$\forall i$ such that $Cond(i)$
  UNION$(i.then, i)$
  UNION$(i.else, i)$
$\forall$ sets C
  let σ be some ordering on $\bigcap\{\theta_{i.rator} \mid i.rator \in C\}$
  $\forall i \in C$
    $\pi_i \leftarrow cl_\sigma$

Figure 10: More subroutines

occurrence indices, so there are $O(n)$ intersections taken during the execution of the loop. Again, we can represent the θ's as bit vectors. Therefore, the amortized cost of those intersections is $O(n^2)$. The amortized cost of the assignments in step **(15)** is $O(n)$, since each occurrence is assigned a protocol tag just once. So the time upper bound for the subroutine is $O(n^2 * \alpha(n^2, n))$, less than the bound for the rest of the algorithm. Therefore, the entire algorithm has an asymptotic running time of $O(n^5)$.

In our presentation, we have chosen clarity over performance. One could use clever data structures to reduce the running time. For instance, Palsberg and Schwartzbach [43] use a graph with bit vectors at each node to perform a closure analysis, with an $O(n^3)$ running time. Their sets of procedure tokens correspond to the flows in our algorithm. It would be straightforward to adapt their algorithm to handle our invariance sets by adding another bit vector so that the flows and invariance set constraints are solvable in $O(n^3)$ time. Since the protocol tag solution subroutine has a smaller time upper bound, with such an approach, the entire algorithm would run in time $O(n^3)$.

3.7 Soundness

Now we show that *any* solution to the local conditions — whether produced by our algorithm or not — makes the annotations true, in the following sense:

Theorem 5 (Soundness) *Let Γ 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 $\psi \overset{\Pi}{\models}_{env} A_i$. Suppose $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$. Then for any subproof $(k, \psi'') \underset{oc}{\Longrightarrow} (m, \psi''')$ of this derivation:*

1. *$\psi'' \overset{\Pi}{\models}_{env} A_k$,*

2. *$(m, \psi''') \overset{\Pi}{\models}_{val} (\phi_k, \pi_k, \theta_k)$, and*

3. *if $Abs(m)$, 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 $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$.

Base cases

For the base cases, there are no proper subproofs, so we show that the consequents hold only at the root of the derivation tree. In each case, the consequent $\psi \models^{\Pi}_{env} A_i$ holds by assumption.

case *Var*(i)

By the local consistency of Γ, $A_i(\llbracket i \rrbracket) = (\phi_i, \pi_i, \theta_i)$, so $\llbracket i \rrbracket \in Dom(A_i)$. Since $\psi \models^{\Pi}_{env} A_i$, $\llbracket i \rrbracket \in Dom(\psi)$, so $(i, \psi) \underset{oc}{\Longrightarrow} \psi(\llbracket i \rrbracket)$. Since $\psi \models^{\Pi}_{env} A_i$, $\psi(\llbracket i \rrbracket) \models^{\Pi}_{val} A_i(\llbracket i \rrbracket)$, so $\psi(\llbracket i \rrbracket) \models^{\Pi}_{val} (\phi_i, \pi_i, \theta_i)$.

Let $\psi(\llbracket i \rrbracket) = (j, \psi')$. Since $\psi \models^{\Pi}_{env} A_i$, if $Abs(j)$ then $\forall x \in \theta_i$, $x \in Dom(\psi) \cap Dom(\psi')$, and $\psi(x) = \psi'(x)$.

case *Const*(i)

So $(i, \psi) \underset{oc}{\Longrightarrow} (i, \emptyset)$. From the definition of $\models^{\Pi}_{val}$, $(i, \emptyset) \models^{\Pi}_{val} (\phi_i, \pi_i, \theta_i)$. Since $Const(i)$, the invariant holds trivially.

case *Abs*(i)

We have $(i, \psi) \underset{oc}{\Longrightarrow} (i, \psi)$. First we show that $(i, \psi) \models^{\Pi}_{val} (\phi_i, \pi_i, \theta_i)$. By the definition of $\models^{\Pi}_{val}$, this holds if $Abs(i)$ (true by assumption), $\Pi(i) = \pi_i$ (true by the construction of Π), and if there exists an environment proposition A such that $(i, A) \in \phi_i$ and $\psi \models^{\Pi}_{env} A$. By the local consistency of Γ, $\{(i, A_i)\} \subseteq \phi_i$, and by assumption, $\psi \models^{\Pi}_{env} A_i$.

Now we show the invariant holds. Again by the local consistency of Γ, $x \in \theta_i$ implies $x \in Dom(A_i)$. Since $\psi \models^{\Pi}_{env} A_i$, $Dom(A_i) \subseteq Dom(\psi)$, so $\forall x \in \theta_i$, $x \in Dom(\psi)$, and trivially, $\psi(x) = \psi(x)$.

Induction

For each case in the induction step, we first show that the consequents hold at the root of the derivation tree. Next we show that the premise about $\models^{\Pi}_{env}$ holds for all immediate subproofs of the root, so that the consequents hold for 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. So suppose we have the evaluation $(i,\psi) \underset{oc}{\Longrightarrow} (c,\emptyset)$. Since $Const(c)$, by the definition of $\overset{\Pi}{\models}_{val}$, $(c,\emptyset) \overset{\Pi}{\models}_{val} (\phi_i, \pi_i, \theta_i)$. 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, $A_{i.rand} = A_i$, so $\psi \overset{\Pi}{\models}_{env} A_{i.rand}$. Therefore, by the induction hypothesis at $i.rand$, all the consequents hold for all proper subproofs.

case $App(i)$

Suppose we have the evaluation proof:

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

Then we can obtain the desired results as follows:

(1) $\psi \overset{\Pi}{\models}_{env} A_{i.rator}, A_{i.rand}$
[$A_{i.rator} = A_{i.rand} = A_i$]

(2) $(j,\psi') \overset{\Pi}{\models}_{val} (\phi_{i.rator}, \pi_{i.rator}, \theta_{i.rator})$
[IH re $\overset{\Pi}{\models}_{val}$ at $i.rator$]

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

(4) $(j, A_j) \in \phi_{i.rator}$
[definition of $\overset{\Pi}{\models}_{val}$, monovariance, (2)]

(5) $\psi' \overset{\Pi}{\models}_{env} A_j$
[by (2), (4), definition of $\overset{\Pi}{\models}_{val}$]

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

(7) if $Abs(k)$ then $x \in \theta_{i.rand} \Longrightarrow x \in Dom(\psi) \cap Dom(\psi''),$
$\psi(x) = \psi''(x)$

[IH re invariance at $i.rand$]

(8) $\phi_{i.rand} \subseteq \phi_{j.bv}$

$[\mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}]$

(9) $\phi_{j.body} \subseteq \phi_i$

$[\mathcal{P}_{j.body} \leq \mathcal{P}_i]$

(10) $\pi_{i.rand} = \pi_{j.bv}$

$[\mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}]$

(11) $\pi_{j.body} = \pi_i$

$[\mathcal{P}_{i.body} \leq \mathcal{P}_i]$

(12) $\theta_{j.bv} \subseteq \theta_{i.rator}$

[invariance set constraints]

(13) $\theta_{j.bv} \subseteq \theta_{i.rand}$

$[\mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}]$

(14) $[\![j.bv]\!] \notin \theta_{j.bv}$

[invariance set constraints]

(15) $\theta_i \subseteq \theta_{j.body}$

$[\mathcal{P}_{i.body} \leq \mathcal{P}_i]$

(16) $\theta_i \subseteq \theta_{i.rator}$

[invariance set constraints]

(17) $[\![j.bv]\!] \notin \theta_i$

[invariance set constraints]

(18) if $Abs(k)$ then $x \in (\theta_{i.rator} \cap \theta_{i.rand}) \Longrightarrow x \in Dom(\psi') \cap Dom(\psi''),$
$$\psi'(x) = \psi''(x)$$
[by (3), (7)]

(19) if $Abs(k)$ then $x \in \theta_{j.bv} \Longrightarrow x \in Dom(\psi') \cap Dom(\psi''),$
$$\psi'(x) = \psi''(x)$$
[by (12), (13), (18)]

(20) if $Abs(k)$ then $x \in \theta_{j.bv} \Longrightarrow x \in Dom(\psi'[[\![j.bv]\!] \mapsto (k, \psi'')]) \cap Dom(\psi''),$
$$\psi'[[\![j.bv]\!] \mapsto (k, \psi'')](x) = \psi''(x)$$
[by (14), (19)]

(21) $(k, \psi'') \models^{\Pi}_{val} (\phi_{j.bv}, \pi_{j.bv}, \theta_{j.bv})$
[by (6), (8), (10)]

(22) $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \models^{\Pi}_{env} A_j[[\![j.bv]\!] \mapsto (\phi_{j.bv}, \pi_{j.bv}, \theta_{j.bv})]$
[by (5), (20), (21)]

(23) $A_{j.body} = A_j[[\![j.bv]\!] \mapsto (\phi_{j.bv}, \pi_{j.bv}, \theta_{j.bv})]$
[definition of locally-consistent annotation]

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

(25) $(m, \psi''') \models^{\Pi}_{val} (\phi_{j.body}, \pi_{j.body}, \theta_{j.body})$
[IH re $\models^{\Pi}_{val}$ at $j.body$]

(26) if $Abs(m)$ 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 $j.body$]

(27) $(m, \psi''') \models^{\Pi}_{val} (\phi_i, \pi_i, \theta_i)$
[(9), (11), and (25)]

(28) if $Abs(m)$ then $x \in (\theta_{j.body} - [\![j.bv]\!]) \Longrightarrow x \in Dom(\psi') \cap Dom(\psi'''),$
$$\psi'(x) = \psi'''(x)$$
[by (26)]

(29) if $Abs(m)$ then $x \in (\theta_{i.rator} \cap (\theta_{j.body} - [\![j.bv]\!])) \Longrightarrow x \in Dom(\psi) \cap Dom(\psi'''),$
$$\psi(x) = \psi'''(x)$$
[by (3), (28)]

(30) if $Abs(m)$ then $x \in \theta_i \Longrightarrow x \in Dom(\psi) \cap Dom(\psi'''),$
$$\psi(x) = \psi'''(x)$$
[by (15), (16), (17), (29)]

There are three immediate subproofs of the root of the proof tree. By step (1), $\psi \overset{\Pi}{\models}_{env} A_{i.rator}, A_{i.rand}$. Therefore, the consequents hold for 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 (24), $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \overset{\Pi}{\models}_{env} A_{j.body}$, so the consequents hold for 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, we have $A_{i.then} = A_i$, so $\psi \overset{\Pi}{\models}_{env} A_{i.then}$. By the induction hypothesis at $i.then$:

$$(k, \psi'') \overset{\Pi}{\models}_{val} (\phi_{i.then}, \pi_{i.then}, \theta_{i.then})$$

By the local constraints on annotations, $\mathcal{P}_{i.then} \leq \mathcal{P}_i$, so by Lemma 6,

$$(k, \psi'') \overset{\Pi}{\models}_{val} (\phi_i, \pi_i, \theta_i)$$

Also by the induction hypothesis, if $Abs(k)$, then $\forall x \in \theta_{i.then}, x \in Dom(\psi) \cap Dom(\psi'')$ and $\psi(x) = \psi''(x)$. By the local constraints, $\mathcal{P}_{i.then} \leq \mathcal{P}_i$, so $\theta_i \subseteq \theta_{i.then}$, hence if $Abs(k)$, then $\forall x \in \theta_i, x \in$

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

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

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

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

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

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

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

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

Figure 11: The closure conversion transformation

$Dom(\psi) \cap Dom(\psi'')$, 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 $\psi \models^{\Pi}_{env} A_{i.then}$, so the consequents hold for all subproofs of the evaluation of the then-part. Since Γ is locally-consistent, we also have $A_{i.test} = A_i$, so $\psi \models^{\Pi}_{env} A_{i.test}$. Therefore, the consequents hold for all subproofs of the evaluation of the test. ∎

3.8 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 11. The transformation Φ is a map from O, the set of occurrence indices, to Λ_{clos}, the language of output terms. The formalism is:

$$\Phi : O \rightarrow \Lambda_{clos}$$

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. For all other syntactic categories of occurrences, the transformation algorithm is just called recursively on subterm occurrences.

We extend the transformation to occurrence closures:

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

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

3.9 Simulation of procedure application

To suggest how closure records act like 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 $G = \lambda x.M$, and let G_{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, $\mathbf{app}\ G_{clos}\ V =_w G\ V$.

Note that the closure record is hand-built, as it were; the transformation Φ 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 $G\ V =_w M[V/x]$.

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

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

The β step is justified by the fact that G_{clos} is a value.

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

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

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

$$\mathbf{app}\ G_{clos}\ V =_w G\ V$$

as desired. Then by Lemma 4,

$$\mathbf{app}\ G_{clos}\ V \underset{t}{\Longrightarrow} W \text{ iff } G\ V \underset{t}{\Longrightarrow} W$$

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

3.10 Correctness

How do we know that the closure conversion transformation produces correct code? A compiler that implements lexical scoping creates closures so that procedures use the bindings for their free variables that existed where the procedures were closed. Our occurrence evaluator manipulates occurrence closures, which, like closures for procedures, contain environment information. Since in the occurrence evaluator all terms are represented by occurrence closures, the occurrence evaluator already correctly implements lexical scoping. We want transformed programs to obey lexical scoping, so the transformation produces closures, but just for procedures. Therefore, our correctness criterion relates the answers from running untransformed occurrence closure programs to the answers from running the transforms of those programs.

Suppose we run the occurrence evaluator on (i, ψ), and the result is (j, ψ'). If we apply the $\hat{\Phi}$ transform to (i, ψ), the result is a *term* in Λ_{clos} — we run the term evaluator on transformed programs. Since the occurrence evaluator already observes lexical scoping, we want, roughly, to have the transformed program evaluate to the same answer as the original program. But the original answer was an occurrence closure, not a term.

What term should we expect as the result of evaluating the transformed program? One possibility is that the new answer is $\mathcal{U}[\![(j, \psi')]\!]$, the unwind of the original answer. After a bit of reflection, we see that that cannot be right: If $[\![i]\!]$ is a procedure with a protocol tag of $cl_{\langle\rangle}$, then $\hat{\Phi}(i, \psi)$ is a closure record that self-evaluates, while the unwind cannot be a closure record. The other way we know how to get a term from an occurrence closure is by applying $\hat{\Phi}$; this possibility turns out to be correct. 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 7 *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.

Base case $\psi = \emptyset$

True trivially.

Induction $\psi \neq \emptyset$

If $x \in Dom(\psi)$, then $\psi(x) = (j, \psi')$ for some j and ψ', so

$$\begin{aligned} \hat{\Phi}(\psi(x)) &= \hat{\Phi}(j, \psi') \\ &= \Phi(j)\{\hat{\Phi} \circ \psi'\} \end{aligned}$$

From the definition of occurrence environment, $[\![j]\!]$ is a value in Λ_{in}, that is, a variable, constant, or procedure.

case $Var(j)$

Then:

$$\begin{aligned} \Phi(j)\{\hat{\Phi} \circ \psi'\} &= [\![j]\!]\{\hat{\Phi} \circ \psi'\} \\ &= \hat{\Phi}(\psi'([\![j]\!])) \end{aligned}$$

which is a value by the induction hypothesis.

case $Const(j)$

In this case:

$$\begin{aligned} \Phi(j)\{\hat{\Phi} \circ \psi'\} &= [\![j]\!]\{\hat{\Phi} \circ \psi'\} \\ &= [\![j]\!] \end{aligned}$$

since the substitution does not affect the constant.

case $Abs(j)$

There are two possibilities for the transform, 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 Γ 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 $\psi \stackrel{\Pi}{\models}_{env} A_i$. 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 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 $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$.

Base cases

case $Var(i)$

By the local consistency of Γ, $A_i([\![i]\!]) = (\phi_i, \pi_i, \theta_i)$, so $[\![i]\!] \in Dom(A_i)$. Since $\psi \stackrel{\Pi}{\models}_{env} A_i$, $[\![i]\!] \in Dom(\psi)$.

Then $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$, where $(j, \psi') = \psi([\![i]\!])$, and:

$$\begin{array}{rcl}
\hat{\Phi}(i, \psi) & = & \Phi(i)\{\hat{\Phi} \circ \psi\} \\
 & = & [\![i]\!]\{\hat{\Phi} \circ \psi\} \\
 & = & \hat{\Phi}(\psi([\![i]\!])) \\
 & = & \hat{\Phi}(j, \psi') \\
 & \underset{t}{\Longrightarrow} & \hat{\Phi}(j, \psi')
\end{array}$$

As explanation for the evaluation step, we have by Lemma 7 that $\hat{\Phi}(j, \psi')$ is a value, so it self-evaluates.

case $Const(i)$

Then $(i, \psi) \underset{oc}{\Longrightarrow} (i, \emptyset)$. Transforming the program, we have:

$$\begin{aligned} \hat{\Phi}(i, \psi) &= \Phi(i)\{\hat{\Phi} \circ \psi\} \\ &= [\![i]\!]\{\hat{\Phi} \circ \psi\} \\ &= [\![i]\!] \\ &\underset{t}{\Longrightarrow} [\![i]\!] \end{aligned}$$

Transforming the evaluation result, we get

$$\begin{aligned} \hat{\Phi}(i, \emptyset) &= \Phi(i)\{\hat{\Phi} \circ \emptyset\} \\ &= [\![i]\!]\{\hat{\Phi} \circ \emptyset\} \\ &= [\![i]\!] \end{aligned}$$

case $Abs(i)$

Then $(i, \psi) \underset{oc}{\Longrightarrow} (i, \psi)$, so we want to show that $\hat{\Phi}(i, \psi)$ self-evaluates. Since $Abs(i)$, the transform depends on the protocol tag π_i.

If $\pi_i = id$, then $\hat{\Phi}(i, \psi)$ is a procedure, which self-evaluates.

If $\pi_i = cl_{\langle v_1, \ldots, v_n \rangle}$, then $\Phi(i, \psi) = \Phi(i)\{\hat{\Phi} \circ \psi\}$ is a record. The first field in the record is a substitution instance of a procedure, and the second field is either the empty record, or, by Lemma 7, a record of values. Both fields are values, hence the entire record is a value, which self-evaluates.

Induction

case $PrimApp(i)$

Then $(i, \psi) \underset{oc}{\Longrightarrow} (c, \emptyset)$ for some constant c. We want to show that $\hat{\Phi}(i, \psi) \underset{t}{\Longrightarrow} \hat{\Phi}(c, \emptyset)$. We have:

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

Regardless of the particular operator, there must be a subproof $(i.rand, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$, for some

j and ψ', where $Const(j)$. The relation between the constants c and $[\![j]\!]$ depends on the operator. By assumption, $\psi \models^{\Pi}_{env} A_i$, and by the local consistency of Γ, $A_{i.rand} = A_i$, so $\psi \models^{\Pi}_{env} A_{i.rand}$. By the induction hypothesis, $\hat{\Phi}(i.rand, \psi) \underset{t}{\Longrightarrow} \hat{\Phi}(j, \psi')$. Since $Const(j)$, by a now-familiar argument, $\hat{\Phi}(j, \psi') = [\![j]\!]$.

By the appropriate rule in the term evaluator,

$$\begin{array}{rcl}\hat{\Phi}(i, \psi) & \underset{t}{\Longrightarrow} & c \\ & = & [\![c]\!] \\ & = & \hat{\Phi}(c, \emptyset)\end{array}$$

case $App(i)$

Suppose we have the derivation

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

We want to show that $\hat{\Phi}(i, \psi) \underset{t}{\Longrightarrow} \hat{\Phi}(m, \psi''')$. As in the case for procedures, we need to consider the effect of a protocol tag on the transform.

Suppose $\pi_{i.rator} = id$. Then:

$$\begin{array}{rcll}\Phi(i, \psi) & = & \Phi(i)\{\hat{\Phi} \circ \psi\} & \\ & = & (\Phi(i.rator)\ \Phi(i.rand))\{\hat{\Phi} \circ \psi\} & \text{definition of } \Phi \\ & = & \Phi(i.rator)\{\hat{\Phi} \circ \psi\}\ \Phi(i.rand)\{\hat{\Phi} \circ \psi\} & \text{definition of substitution} \\ & = & \Phi(i.rator, \psi)\ \Phi(i.rand, \psi) & \text{definition of } \hat{\Phi}\end{array}$$

We have a term that is an application, so we can just follow the application rule in the term evaluator.

First we evaluate the operator. By assumption, $\psi \models^{\Pi}_{env} A_i$, and by the local consistency of Γ, $A_{i.rator} = A_{i.rand} = A_i$. Therefore, $\psi \models^{\Pi}_{env} A_{i.rator}, A_{i.rand}$, so we can invoke the induction hypothesis at $i.rator$:

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

Since $\psi \models^{\Pi}_{env} A_{i.rator}$ and $(i.rator, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$, by Theorem 5, $(j, \psi') \models^{\Pi}_{val} (\phi_{i.rator}, \pi_{i.rator}, \theta_{i.rator})$. By the above derivation, we know $Abs(()j)$, so by the definition of $\models^{\Pi}_{val}$, $\Pi(j) = \pi_{i.rator} = id$. But by the definition of Π, $\Pi(j) = \pi_j$. Therefore, we may write:

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

We can move the substitution inside the body of the procedure because the fresh variable x' is not in the range of ψ, and the fresh binder cannot capture any other variable.

Next we evaluate the operand by invoking the induction hypothesis at $i.rand$:

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

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

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

Consider the first equality. On both sides of the equation, we have two substitution instances of the same term. We show that the substitutions are identical. Note that x' is fresh, so it is neither in the domain nor free in the range of $\hat{\Phi}\circ\psi'$. For a variable y, where $y \neq [\![j.bv]\!]$, the first and last substitutions on the left-hand side have no effect, so we substitute $\hat{\Phi}(\psi'(y))$ on both sides. For $[\![j.bv]\!]$, the middle substitution on the left-hand side has no effect, and we substitute $\hat{\Phi}(k,\psi'')$ on both sides. The second equality proceeds from the definition of $\hat{\Phi}$.

In order to use the induction hypothesis for evaluating the substituted body, we need to show that

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

Since the evaluation of the procedure body is a subproof of the derivation for the application itself, the result holds by Theorem 5. 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:

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

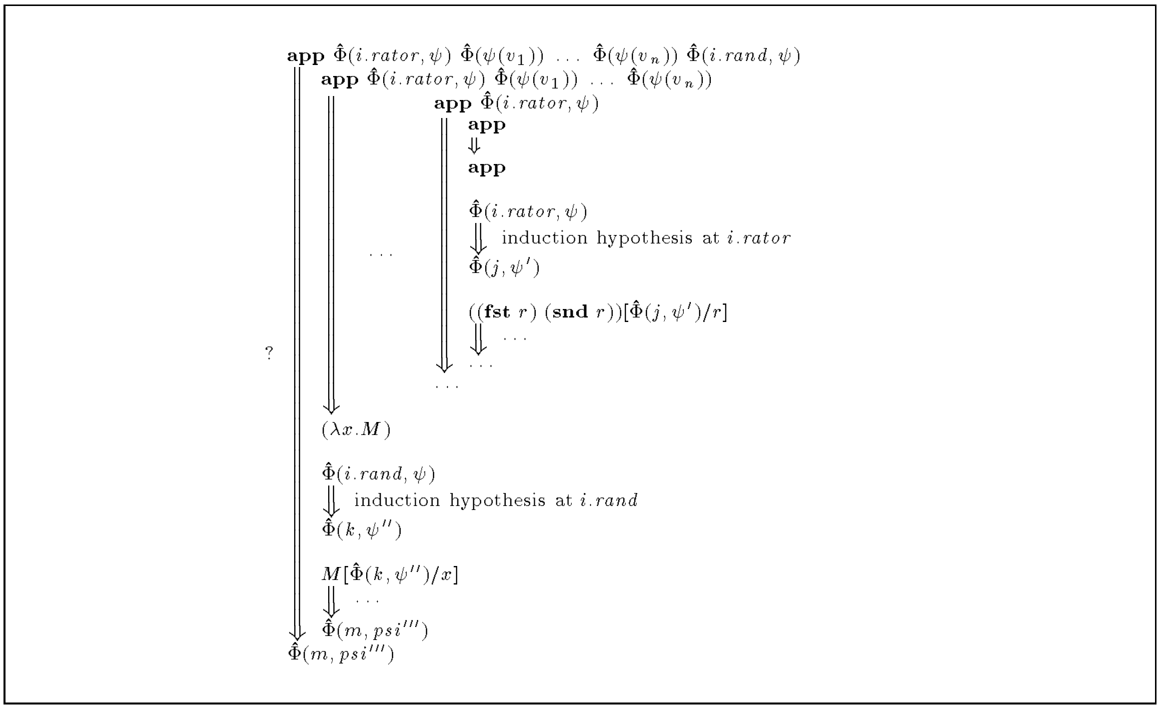

Figure 12: Evaluating an application where the operator tag is $cl_{\langle v_1,\ldots,v_n\rangle}$

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:

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

We sketch what we think the proof tree for the evaluation of $\hat{\Phi}(i,\psi)$ should look like in Figure 12. Since we have not yet shown that the evaluation is valid, a question mark appears to the left of the proof tree. In order to use the application rule, **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 12. If it were true that $\hat{\Phi}(j,\psi')$ and $\hat{\Phi}(k,\psi'')$ each self-evaluates, we could replace $\Phi(i.rator,\psi)$ by $\hat{\Phi}(j,\psi')$, and $\Phi(i.rand,\psi)$ by $\hat{\Phi}(k,\psi'')$ with no effect on the final result of the evaluation. 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.

So it must be that:

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

We can now use equational reasoning to simplify the term to evaluate. This strategy is useful since by Lemma 4, two weakly equivalent terms evaluate to the same value (or neither evaluates). 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.

Simplifying further, we have:

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

By Lemma 5, $(j,\psi') \models^{\Pi}_{val} (\phi_{i.rator},\pi_{i.rator},\theta_{i.rator})$, so $\Pi(j)=\pi_j=\pi_{i.rator}=th$. Therefore, $\Phi(j)$ is a closure record. In the first $=_w$ step, we use two applications of the **destr** rule and the **cong** rule to get at the fields in the closure record. The second $=_w$ step uses the β-rule. On the fourth line, we rely on the fact that e is fresh, and so not in the domain of ψ'. In the last step, we explicitly replace the single occurrence of e by $[\vec{u}]$.

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:

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

By Lemma 7, 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:

$$\begin{array}{ll} & (\lambda v_1 \ldots v_n [\![j.bv]\!].\,\mathbf{destr}\,[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))\{\hat{\Phi}\circ\psi'\}\ \hat{\Phi}(\psi(v_1))\ \ldots\ \hat{\Phi}(\psi(v_n)) \\ = & (\lambda v_1'.(\lambda v_2 \ldots v_n [\![j.bv]\!].\,\mathbf{destr}\,[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))[v_1'/v_1])\{\hat{\Phi}\circ\psi'\}\ \hat{\Phi}(\psi(v_1))\ \ldots \\ = & \lambda v_1'.(\lambda v_2 \ldots v_n [\![j.bv]\!].\,\mathbf{destr}\,[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))[v_1'/v_1]\{\hat{\Phi}\circ\psi'\}\ \hat{\Phi}(\psi(v_1))\ \ldots \\ =_w & (\lambda v_2 \ldots v_n [\![j.bv]\!].\,\mathbf{destr}\,[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))[v_1'/v_1]\{\hat{\Phi}\circ\psi'\}[\hat{\Phi}(\psi(v_1))/v_1']\ \ldots \\ = & (\lambda v_2 \ldots v_n [\![j.bv]\!].\,\mathbf{destr}\,[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))\{\hat{\Phi}\circ\psi'[v_1\mapsto\psi(v_1)]\}\ \hat{\Phi}(\psi(v_2))\ \ldots \\ & \ldots \\ =_w & (\lambda[\![j.bv]\!].\,\mathbf{destr}\,[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))\{\hat{\Phi}\circ\psi'[v_1,\ldots,v_n\mapsto\psi(v_1),\ldots,\psi(v_n)]\} \\ = & (\lambda[\![j.bv]\!].\,\mathbf{destr}\,[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))\{\hat{\Phi}\circ\psi'[v_1,\ldots,v_n\mapsto\psi'(v_1),\ldots,\psi'(v_n)]\} \\ = & (\lambda[\![j.bv]\!].\,\mathbf{destr}\,[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))\{\hat{\Phi}\circ\psi'\} \end{array}$$

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.

To see that the equality after the first $=_w$ step holds, consider a free variable x, where $x \neq v_1$. On both sides of the equation, we substitute $\hat{\Phi}(\psi'(x))$. For a free v_1, on the left-hand side, we substitute $\hat{\Phi}(\psi(v_1))$, while on the right-hand side, we substitute $\hat{\Phi}(\psi'[v_1\mapsto\psi(v_1)](v_1)) = \hat{\Phi}(\psi(v_1))$.

In the last two steps lies the key to lightweight closure conversion; here is where we use our invariance sets. Now, by assumption, $\psi \models^{\Pi}_{env} A_i$, but $A_{i.rator} = A_i$ by the local consistency of Γ. Since $(i.rator,\psi) \underset{oc}{\Longrightarrow} (j,\psi')$ and $Abs(j)$, by Theorem 5, $\forall x \in \theta_{i.rator}$, $x \in Dom(\psi) \cap Dom(\psi')$, and $\psi(x) = \psi'(x)$. By the local constraints, 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 $\psi'(v_1),\ldots,\psi'(v_n)$. In the last step, we use the fact that the extensions to ψ' have no effect, allowing us to remove them.

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

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

Since $\lceil\vec{u}\rceil \subseteq FV([\![j]\!])$, we know $[\![j.bv]\!]$ is not any of the $\vec{u}$. 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:

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

On the right-hand side of the first equality, we move the substitution toward the procedure, which holds because x' may occur only in the procedure. To see that the second equality holds, observe that for a free variable x, $x \neq [\![j.bv]\!]$, we substitute $\hat{\Phi}(\psi'(x))$ on both sides, while for a free $[\![j.bv]\!]$, we substitute $\hat{\Phi}(k,\psi'')$.

By Lemma 7, $[\vec{u}]\{\hat{\Phi}\circ\psi'\}$ 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:

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

To see that the equality after the $=_w$ step holds, we can extend our familiar argument to handle the vector $\vec{u}$. On both sides of the equation, for any free u among the $\vec{u}$, we substitute $\hat{\Phi}(\psi'(u))$.

Note that u cannot be $[\![j.bv]\!]$. For any other free variable x, which may be $[\![j.bv]\!]$, we substitute $\hat{\Phi}(\psi'[[\![j.bv]\!] \mapsto (k, \psi'')](x))$ on both sides. In the last step, we use the definition of $\hat{\Phi}$.

With much effort, we have shown that

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

As we showed for the previous subcase, we know by Theorem 5 that $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \overset{\Pi}{\models}_{env} A_{j.body}$, so by the induction hypothesis at $j.body$

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

Then by Theorem 4

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

hence

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

case $Cond(i)$

We have:

$$\begin{array}{ll} & \hat{\Phi}(i, \psi) \\ = & \Phi(i)\{\hat{\Phi} \circ \psi\} \\ = & (\mathbf{if}\ \Phi(i.test)\ \mathbf{then}\ \Phi(i.then)\ \mathbf{else}\ \Phi(i.else))\{\hat{\Phi} \circ \psi\} \\ = & \mathbf{if}\ \hat{\Phi}(i.test, \psi)\ \mathbf{then}\ \hat{\Phi}(i.then, \psi)\ \mathbf{else}\ \hat{\Phi}(i.else, \psi) \end{array}$$

There are two subcases, depending on the result of the test. For brevity, we show only one subcase; the other is similar.

Suppose:

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

By assumption $\psi \models^{\Pi}_{env} A_i$, and by the local consistency of Γ, $A_{i.test} = A_{i.then} = A_i$. Therefore, we can invoke the induction hypothesis to show:

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

and

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

Also:

$$\begin{aligned} \hat{\Phi}(j, \psi') &= \Phi(j)\{\hat{\Phi} \circ \psi'\} \\ &= [\![j]\!]\{\hat{\Phi} \circ \psi'\} \\ &= \textbf{true}\{\hat{\Phi} \circ \psi'\} \\ &= \textbf{true} \end{aligned}$$

By the evaluation rule for conditionals, where the test evaluates to **true**,

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

∎

As for Theorem 1, the converse fails due to a choice-of-representative problem. Suppose $\hat{\Phi}(i, \psi) \underset{t}{\Longrightarrow} \hat{\Phi}(j, \psi')$. Let k be an occurrence index such that $[\![k]\!] = [\![j]\!]$, where $j \neq k$. Since the two terms are identical, we can annotate them so that the protocol tags associated with their corresponding occurrences are the same. Then $\hat{\Phi}(i, \psi) \underset{t}{\Longrightarrow} \hat{\Phi}(k, \psi') = \hat{\Phi}(j, \psi')$. Since the evaluation of occurrence closures is deterministic, it may be that $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$ or $(i, \psi) \underset{oc}{\Longrightarrow} (k, \psi')$, but not both.

3.11 Factoring the constraints

After we presented the system of local constraints (Figure 7), we prefaced the Soundness Theorem (Theorem 5) by stating that any solution to the constraints makes the annotations true. However, we did not use all of the constraints in proving the theorem. In this section, we consider how the constraints can be factored into those needed for soundness, and those needed to show the correctness of the transformation.

All but one of the constraints were used in proving soundness, namely

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

This constraint assures that the dynamic variables inserted at a call site are a subset of the invariance set associated with the operator. That property has no bearing on soundness, since dynamic variables are inserted by the transformation. For the proof of the Correctness Theorem (Theorem 6), we use this constraint to show that the bindings for dynamic variables are invariant across the evaluation of the operator in an application.

This view of the constraints suggests that any sound-in-fact annotation plus some other constraints yields correctness. Suppose we were able to annotate a program using some arbitrary method, so that we knew that the Soundness Theorem held. In that case, the additional constraint just mentioned would allow us to prove the Correctness Theorem. For the analyses in subsequent chapters, the constraints can also be factored into a large number that support a Soundness Theorem, and a smaller number that support a Correctness Theorem. Nonetheless, this factorization is conceptual — it is difficult to imagine how to make the Soundness Theorem hold without solving constraints similar to those we have given. In another sense, then, all the constraints are needed to prove the Correctness Theorem.

3.12 Related work

Our program analysis and closure conversion algorithm draw on several sources.

The use of input-output assertions to verify programs began with Floyd [19]. Like Floyd, in our soundness theorems in this and later chapters, we show that input and output assertions (our environment propositions and value propositions) 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 [24]. 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 [32]. In the "Dragon Book", Aho et al. suggest a general framework for solving dataflow equations for programs with basic blocks [3, 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 flow component of a value proposition associated with an operator at a call site represents the set of procedures that may be called at that site.

Sestoft [47] 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 [9]. In Shivers' dissertation, he presented two analyses called 0CFA and 1CFA for a version of Scheme, using an abstract interpretation [49]. 0CFA is directly comparable to our analysis; in essence, 1CFA indexes abstract closures by their call sites. More recently, Ayers used similar techniques to compute a highly-optimized version of 0CFA [7]. Palsberg and Schwartzbach [43] 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 [53]. 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 [45].

Heintze analyzed programs using constraints on sets of program values in his dissertation [23] 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 [57, 58].

The idea of approximating an occurrence closure by an abstract closure comes from [29]; this is the extension of the idea of record types used in [30] to an occurrence closure evaluator.

In [56], 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 [36]. Many compiler writers have described their closure conversion algorithms; see [51, 34, 5]. Fradet and Le Métayer describe how closure creation may be avoided in some cases [20].

Chapter 4

Tracking available values

4.1 Supplying available values at call sites

[1]In the preceding chapter, we demonstrated a method for proving the correctness of a closure conversion transformation, which allowed us in some cases to supply free variables of a procedure as extra arguments at the procedure's call sites. We can generalize that result so that if the values of free variables are *available* as the bindings of other variables at call sites, those other variables can be supplied as extra arguments. The new analysis subsumes the previous one, in that any dynamic variable may still be considered dynamic in the new analysis. Our general approach will be the same; the low-level details of the annotations and the transformation require updating.

4.2 An example

Consider the program:

$$\begin{aligned}\lambda xa.\mathbf{let}\ & f = (\lambda y.(\lambda z.(+\ x\ y\ z)))\\ & g = (\lambda h.(h\ 3))\\ \mathbf{in}\ & (g\ (f\ a))\end{aligned}$$

[1]A summary of this chapter appeared as [50].

The procedure $(\lambda z.(+\ x\ y\ z))$ has free variables x and y. The only call site for this procedure is at $(h\ 3)$ in the body of g. At that site, x is in scope, and has the same binding as at the procedure's definition site. Using the analysis from the preceding chapter, we can make x a dynamic variable, and omit it from the closure for the procedure. The other free variable, y, is not in scope at the call site, so the old analysis requires that it be placed in the closure for the procedure.

However, the new analysis recognizes that the *value* of y at the procedure's definition site is available as the value of a at the call site. Therefore, we can insert a as an extra argument at the call site, and omit y from the closure. Assuming that only that procedure becomes an explicit closure, our revised transformation produces:

$$\begin{aligned}\lambda xa.\mathbf{let}\ f &= (\lambda y.[(\lambda exyz.\,\mathbf{destr}\ e\ \lambda.(+\ x\ y\ z)), []])\\ g &= (\lambda h.(\mathbf{app}\ h\ x\ a\ 3))\\ \mathbf{in}\ &(g\ (f\ a))\end{aligned}$$

As we have seen before, closure records may have an empty record of free variables.

4.3 Revising the annotations; invariance relations

Most of the machinery we constructed for the previous analysis is still valid. We will continue to use Λ_{in} as our source language, and Λ_{clos} as the language for transformed programs.

Recall the invariance sets from the previous analysis, which were sets of variables. If we had a solution to the local consistency conditions, then we were able to prove that if we had the evaluation $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$, and j was the index of a procedure, then for each variable x in θ_i, the invariance set for i, $\psi(x) = \psi'(x)$. For the new analysis, we replace invariance sets with *invariance relations*, which are binary relations on the set of all variables. We reformulate the local consistency conditions so that any solution of them ensures that if $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$ and j is the index of a procedure, then for each pair (x, y) in θ_i, the invariance relation for i, $\psi(x) = \psi'(y)$. Therefore, if we have a pair (x, y) in the invariance relation for the operator of an application, we can supply x as an extra argument at the call site and leave y out of closures that flow to that call site.

4.4 A new semantics of annotations

In the previous analysis, we formalized two notions of satisfaction. The first notion specified a satisfaction relation on environments and proposition environments, the other gave a satisfaction relation on values and propositions. The first of these definitions needs modification for the new analysis. The second definition is as before, except that θ is an invariance relation rather than an invariance set. Of course, since these two definitions are mutually recursive, the meaning of both has in fact changed.

A value proposition $\mathcal{P}$ is now a triple consisting of a flow ϕ (a set of abstract closures), a protocol tag π (describing an application protocol), and an invariance relation θ. Protocol tags still range over the set $\{id\} \cup \{cl_\sigma \mid \sigma \in Var^*\}$. As before, a protocol assignment Π is a map from occurrence indices to protocol tags. An environment proposition A is still a finite map from variables to value propositions.

We now say:

1. An occurrence environment ψ satisfies an environment proposition A under a protocol assignment Π, as follows:
$$\psi \overset{\Pi}{\models}_{env} A$$
iff $\forall x \in Dom(A)$
   (a) $x \in Dom(\psi)$ and $\psi(x) \overset{\Pi}{\models}_{val} A(x)$, and
   (b) if $A(x) = (\phi, \pi, \theta)$, $\psi(x) = (i, \psi')$, and $Abs(i)$, then $\forall (y, z) \in \theta$, $y \in Dom(\psi)$, $z \in Dom(\psi')$, and $\psi(y) = \psi'(z)$

   and, almost as before,

2. An occurrence closure (i, ψ) satisfies a proposition (ϕ, π, θ) under a protocol assignment Π, as follows:
$$(i, \psi) \overset{\Pi}{\models}_{val} (\phi, \pi, \theta)$$
iff
   (a) $Const(i) \vee Var(i)$, or
   (b) $Abs(i)$, $\Pi(i) = \pi$ and there exists A such that $(i, A) \in \phi$ and $\psi \overset{\Pi}{\models}_{env} A$.

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

$$Abs(i) \Longrightarrow \begin{cases} A_{i.body} = A_i[[\![i.bv]\!] \mapsto \mathcal{P}_{i.bv}] \\ \{(i, A_i)\} \subseteq \phi_i, \text{ and} \\ (x, y) \in \theta_i \Longrightarrow x = y \wedge x \in Dom(A_i) \end{cases}$$

$$PrimApp(i) \Longrightarrow A_{i.rand} = A_i$$

$$App(i) \Longrightarrow \begin{cases} A_{i.rator} = A_{i.rand} = A_i, \\ \theta_i \subseteq \theta_{i.rator}, \\ (x, y) \in \theta_{i.rator} \cap \theta_{i.rand} \Longrightarrow (x, x) \in \theta_{i.rator} \\ \pi_{i.rator} = cl_\sigma \Longrightarrow \forall y \in \lceil \sigma \rceil,\ \exists x \text{ such that } (x, y) \in \theta_{i.rator}, \text{ and} \\ \forall (j, B) \in \phi_{i.rator} \\ \quad \begin{cases} Abs(j), \\ \mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}, \\ \mathcal{P}_{j.body} \leq \mathcal{P}_i, \\ \theta_{j.bv} \subseteq \theta_{i.rator}, \\ (x, y) \in \theta_{j.bv} \cup \theta_i \Longrightarrow x \neq [\![j.bv]\!], \text{ and} \\ (x, y) \in \theta_{i.rator} \cap (\theta_{j.body} - \{([\![j.bv]\!], \_)\}) \Longrightarrow (x, x) \in \theta_{i.rator} \end{cases} \end{cases}$$

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

Figure 13: Local consistency conditions for available value analysis

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

The ordering on value propositions is now:

$$(\phi, \pi, \theta) \leq (\phi', \pi', \theta') \text{ iff } \phi \subseteq \phi' \wedge \pi = \pi' \wedge \theta' \subseteq \theta$$

Lemma 6, p. 48, can be restated here, using the current definition of a value proposition.

A new set of local consistency conditions is shown in Figure 13. For convenience, we write $\{(x, \_)\}$ for the set of all pairs of variables with x in the first position. The constraints on flows are identical to those in the earlier analysis. For protocol tags, the constraints are the same as before, except for tags associated with applications. Before, we had the constraint

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

and now we have

$$\pi_{i.rator} = cl_\sigma \Longrightarrow \forall y \in \lceil \sigma \rceil,\ \exists x \text{ such that } (x, y) \in \theta_{i.rator}$$

The old constraint said that if lightweight closures are applied at a call site, the set of supplied dynamic variables must be a subset of the invariance set associated with the operator. The new constraint says that if we apply lightweight closures at a call site, each dynamic variable x must have a representative element (x, y) in the invariance relation associated with the operator. Of course, the constraints on invariance relations are significantly different than those for invariance sets. An iterative algorithm similar to that used to solve the earlier set of constraints can be used to solve the new constraints.

We retain the definitions of annotation map, monovariance and local consistency from the preceding chapter.

4.5 Soundness of the annotations

We can now state a soundness theorem for the available value analysis:

Theorem 7 (Soundness) *Let Γ be a monovariant, 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 $\psi \overset{\Pi}{\models}_{env} A_i$. Suppose $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$. Then for any subproof $(k, \psi'') \underset{oc}{\Longrightarrow} (m, \psi''')$ of this derivation:*

1. *$\psi'' \overset{\Pi}{\models}_{env} A_k$,*
2. *$(m, \psi''') \overset{\Pi}{\models}_{val} (\phi_k, \pi_k, \theta_k)$, and*
3. *if $Abs(j)$ then $\forall (x, y) \in \theta_k$, $x \in Dom(\psi'')$, $y \in Dom(\psi''')$, and $\psi''(x) = \psi'''(y)$.*

For consequent (1) (value satisfaction), note that the textual definition for a value satisfying a proposition has not changed, and does not involve the θ-component of the proposition. Therefore, the significant changes to the proof involve consequent (2) (invariance).

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

Base cases

For the base cases, there are no proper subproofs. In each base case, the consequent $\psi \overset{\Pi}{\models}_{env} A_i$ holds by assumption.

case $Var(i)$

By the local consistency of Γ, $A_i(\llbracket i \rrbracket) = (\phi_i, \pi_i, \theta_i)$, so $\llbracket i \rrbracket \in Dom(A_i)$. Since $\psi \models^{\Pi}_{env} A_i$, $\llbracket i \rrbracket \in Dom(\psi)$. So $(i, \psi) \underset{oc}{\Longrightarrow} \psi(\llbracket i \rrbracket)$. Also since $\psi \models^{\Pi}_{env} A_i$, $\psi(\llbracket i \rrbracket) \models^{\Pi}_{val} (\phi_i, \pi_i, \theta_i)$.

Let $\psi(\llbracket i \rrbracket) = (j, \psi')$. Since $\psi \models^{\Pi}_{env} A_i$, if $Abs(j)$, then $\forall (x, y) \in \theta_i$, $x \in Dom(\psi)$, $y \in Dom(\psi')$, and $\psi(x) = \psi'(y)$.

case $Const(i)$

We have $(i, \psi) \underset{oc}{\Longrightarrow} (i, \emptyset)$. Immediately from the definition of $\models^{\Pi}_{val}$, $(i, \emptyset) \models^{\Pi}_{val} (\phi_i, \pi_i, \theta_i)$. Since $Const(i)$, the invariant holds trivially.

case $Abs(i)$

So $(i, \psi) \underset{oc}{\Longrightarrow} (i, \psi)$. First we show that $(i, \psi) \models^{\Pi}_{val} (\phi_i, \pi_i, \theta_i)$. By the definition of $\models^{\Pi}_{val}$, this holds if $\Pi(i) = \pi_i$ (true by the definition of Π), and if there is some A such that $(i, A) \in \phi_i$ and $\psi \models^{\Pi}_{env} A$. The annotation map is locally-consistent, so $\{(i, A_i)\} \subseteq A_i$, and by assumption, $\psi \models^{\Pi}_{env} A_i$. By the local consistency conditions, $(x, y) \in \theta_i$ implies $x = y$ and $x \in Dom(A_i)$. Since $\psi \models^{\Pi}_{env} A_i$, $Dom(A_i) \subseteq Dom(\psi)$, and trivially, $\psi(x) = \psi(x)$.

Induction

For each case in the induction step, we first show that the consequents hold at the root of the proof tree. Then we show that the premise about $\models^{\Pi}_{env}$ holds for all immediate subproofs of the root, so that the consequents hold for all proper subproofs.

case $PrimApp(i)$

Then $(i, \psi) \underset{oc}{\Longrightarrow} (c, \emptyset)$, for some constant c. So by the definition of $\models^{\Pi}_{val}$, $(c, \emptyset) \models^{\Pi}_{val} (\phi_i, \pi_i, \theta_i)$. Since $Const(c)$, the invariant holds trivially.

By the evaluation rules for primitive operators, there is one immediate subproof, $(i.rand, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$, for some j and ψ', where $Const(j)$. Since Γ is locally-consistent, $A_{i.rand} = A_i$, so $\psi \models^{\Pi}_{env} A_{i.rand}$. Therefore, by the induction hypothesis at $i.rand$, all the consequents hold for all proper subproofs.

case $App(i)$

Suppose we have the evaluation proof:

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

Then:

(1) $\psi \overset{\Pi}{\models}_{env} A_{i.rator}, A_{i.rand}$

$[A_{i.rator} = A_{i.rand} = A_i]$

(2) $(j, \psi') \overset{\Pi}{\models}_{val} (\phi_{i.rator}, \pi_{i.rator}, \theta_{i.rator})$

[IH re $\overset{\Pi}{\models}_{val}$ at $i.rator$]

(3) $(x, y) \in \theta_{i.rator} \Longrightarrow x \in Dom(\psi),\ y \in Dom(\psi'),$

$$\psi(x) = \psi'(y)$$

[$Abs(j)$, IH re invariance at $i.rator$]

(4) $(j, A_j) \in \phi_{i.rator}$

[definition of $\overset{\Pi}{\models}_{val}$, monovariance, (2)]

(5) $\psi' \overset{\Pi}{\models}_{env} A_j$

[by (2), (4), definition of $\overset{\Pi}{\models}_{val}$]

(6) $(k, \psi'') \overset{\Pi}{\models}_{val} (\phi_{i.rand}, \pi_{i.rand}, \theta_{i.rand})$

[IH re $\overset{\Pi}{\models}_{val}$ at $i.rand$]

(7) if $Abs(k)$ then $(x, y) \in \theta_{i.rand} \Longrightarrow x \in Dom(\psi),\ y \in Dom(\psi''),$

$$\psi(x) = \psi''(y)$$

[IH re invariance at $i.rand$]

(8) $\phi_{i.rand} \subseteq \phi_{j.bv}$

$[\mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}]$

(9) $\phi_{j.body} \subseteq \phi_i$

$[\mathcal{P}_{i.body} \leq \mathcal{P}_i]$

(10) $\pi_{i.rand} = \pi_{j.bv}$

$[\mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}]$

(11) $\pi_{j.body} = \pi_i$

$[\mathcal{P}_{i.body} \leq \mathcal{P}_i]$

(12) $\theta_{j.bv} \subseteq \theta_{i.rator}$

[invariance relation constraints]

(13) $\theta_{j.bv} \subseteq \theta_{i.rand}$

$[\mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}]$

(14) $(x, y) \in \theta_{j.bv} \Longrightarrow x \neq [\![j.bv]\!]$

[invariance relation constraints]

(15) $\theta_i \subseteq \theta_{j.body}$

$[\mathcal{P}_{i.body} \leq \mathcal{P}_i]$

(16) $\theta_i \subseteq \theta_{i.rator}$

[invariance relation constraints]

(17) $(x, y) \in \theta_i \Longrightarrow x \neq [\![j.bv]\!]$

[invariance relation constraints]

(18) $(x, y) \in (\theta_{i.rator} \cap \theta_{i.rand}) \Longrightarrow (x, x) \in \theta_{i.rator}$

[invariance relation constraints]

(19) $$(x, y) \in (\theta_{i.rator} \cap \theta_{i.rand}) \Longrightarrow x \in Dom(\psi) \cap Dom(\psi'), \quad \psi(x) = \psi'(x)$$

[by (3), (18)]

(20) if $Abs(k)$ then $(x, y) \in (\theta_{i.rator} \cap \theta_{i.rand}) \Longrightarrow x \in Dom(\psi) \cap Dom(\psi'),$
$$y \in Dom(\psi') \cap Dom(\psi''),$$
$$\psi(x) = \psi'(x) = \psi'(y) = \psi''(y)$$
[by (3), (7), (19)]

(21) if $Abs(k)$ then $(x, y) \in \theta_{j.bv} \Longrightarrow x \in Dom(\psi) \cap Dom(\psi'),$
$$y \in Dom(\psi') \cap Dom(\psi''),$$
$$\psi(x) = \psi'(x) = \psi'(y) = \psi''(y)$$
[by (12), (13), (20)]

(22) if $Abs(k)$ then $(x, y) \in \theta_{j.bv} \Longrightarrow x \in Dom(\psi'[[\![j.bv]\!] \mapsto (k, \psi'')]),$
$$y \in Dom(\psi''),$$
$$\psi'[[\![j.bv]\!] \mapsto (k, \psi'')](x) = \psi''(y)$$
[by (14), (21)]

(23) $(k, \psi'') \overset{\Pi}{\models}_{val} (\phi_{j.bv}, \pi_{j.bv}, \theta_{j.bv})$
[by (6), (8), (10)]

(24) $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \overset{\Pi}{\models}_{env} A_j[[\![j.bv]\!] \mapsto (\phi_{j.bv}, \pi_{j.bv}, \theta_{j.bv})]$
[by (5), (22), (23)]

(25) $A_{j.body} = A_j[[\![j.bv]\!] \mapsto (\phi_{j.bv}, \pi_{j.bv}, \theta_{j.bv})]$
[definition of consistent annotation]

(26) $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \overset{\Pi}{\models}_{env} A_{j.body}$
[(24), (25)]

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

(28) $(m, \psi''') \overset{\Pi}{\models}_{val} (\phi_i, \pi_i, \theta_i)$
[(9), (11), and (27)]

$$\text{(29)}\quad \text{if } Abs(m) \text{ then } (x,y) \in \theta_{j.body} \Longrightarrow \begin{array}{l} x \in Dom(\psi'[[\![j.bv]\!] \mapsto (k,\psi'')]), \\ y \in Dom(\psi'''), \\ \psi'[[\![j.bv]\!] \mapsto (k,\psi'')](x) = \psi'''(y) \end{array}$$

[IH re invariance at $j.body$]

$$\text{(30)}\quad \text{if } Abs(m) \text{ then } (x,y) \in (\theta_{j.body} - \{([\![j.bv]\!],\_)\}) \Longrightarrow \begin{array}{l} x \in Dom(\psi'),\ y \in Dom(\psi'''), \\ \psi'(x) = \psi'''(y) \end{array}$$

[by (29)]

$$\text{(31)}\quad \text{if } Abs(m) \text{ then } (x,y) \in (\theta_{i.rator} \cap (\theta_{j.body} - \{([\![j.bv]\!],\_)\})) \Longrightarrow \begin{array}{l} x \in Dom(\psi) \cap Dom(\psi'), \\ y \in Dom(\psi') \cap Dom(\psi'''), \\ \psi(x) = \psi'(y), \\ \psi'(x) = \psi'''(y) \end{array}$$

[by (3), (30)]

$$\text{(32)}\quad (x,y) \in (\theta_{i.rator} \cap (\theta_{j.body} - \{([\![j.bv]\!],\_)\})) \Longrightarrow (x,x) \in \theta_{i.rator}$$

[invariance relation constraints]

$$\text{(33)}\quad (x,y) \in (\theta_{i.rator} \cap (\theta_{j.body} - \{([\![j.bv]\!],\_)\})) \Longrightarrow \begin{array}{l} x \in Dom(\psi) \cap Dom(\psi'), \\ \psi(x) = \psi'(x) \end{array}$$

[by (3), (32)]

$$\text{(34)}\quad \text{if } Abs(m) \text{ then } (x,y) \in (\theta_{i.rator} \cap (\theta_{j.body} - \{([\![j.bv]\!],\_)\})) \Longrightarrow \begin{array}{l} x \in Dom(\psi),\ y \in Dom(\psi'''), \\ \psi(x) = \psi'''(y) \end{array}$$

[by (31), (33)]

$$\text{(35)}\quad \text{if } Abs(m) \text{ then } (x,y) \in \theta_i \Longrightarrow \begin{array}{l} y \in Dom(\psi) \cap Dom(\psi'''), \\ \psi(x) = \psi'''(y) \end{array}$$

[by (15), (16), (17), (34)]

There are three immediate subproofs of the root of the proof tree. By step (1), $\psi \overset{\Pi}{\models}_{env} A_{i.rator}, A_{i.rand}$, so the consequents hold for 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), $\psi'[[\![j.bv]\!] \mapsto (k,\psi'')] \overset{\Pi}{\models}_{env} A_{j.body}$, so the consequents hold for all subproofs of the evaluation of the procedure body.

case $Cond(i)$

This time, we show only the subcase where the test is false.

Suppose we have:

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

By assumption, $\psi \overset{\Pi}{\models}_{env} A_i$, and since Γ is locally-consistent, $A_{i.else} = A_i$, so $\psi \overset{\Pi}{\models}_{env} A_{i.else}$. So by the induction hypothesis at $i.else$

$$(k, \psi'') \overset{\Pi}{\models}_{val} (\phi_{i.else}, \pi_{i.else}, \theta_{i.else})$$

By the constraints on annotations, $\mathcal{P}_{i.else} \leq \mathcal{P}_i$, so by Lemma 6,

$$(k, \psi'') \overset{\Pi}{\models}_{val} (\phi_i, \pi_i, \theta_i)$$

Also by the induction hypothesis, if $Abs(k)$, then $\forall (x, y) \in \theta_{i.else}$, $x \in Dom(\psi)$, $y \in Dom(\psi'')$, and $\psi(x) = \psi''(y)$. Since the annotation map is locally-consistent, $\mathcal{P}_{i.else} \leq \mathcal{P}_i$, so $\theta_i \subseteq \theta_{i.else}$, hence if $Abs(k)$, then $\forall (x, y) \in \theta_i$, $x \in Dom(\psi)$, $y \in Dom(\psi'')$, and $\psi(x) = \psi''(y)$.

There are two immediate subproofs, one for the evaluation of the test, the other for the evaluation of the else-part. We showed $\psi \overset{\Pi}{\models}_{env} A_{i.else}$, so the consequents hold for all subproofs of the evaluation of the else-part. Since Γ is locally-consistent, we also have $A_{i.test} = A_i$, so $\psi \overset{\Pi}{\models}_{env} A_{i.test}$. Therefore, the consequents hold for all subproofs of the evaluation of the test. ∎

4.6 Invariance relations subsume invariance sets

If we compare the soundness result just given with Theorem 5 from Chapter 3, we see how the new analysis improves on the old one. Suppose $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$. If in the previous analysis, we had a variable x in the invariance set θ_i, that implied that $\psi(x) = \psi'(x)$. For the same program, the current analysis allows us to place the pair (x, x) in the invariance relation θ_i, which also implies that $\psi(x) = \psi'(x)$. For some programs, the current analysis may allow us to place pairs (y, z) in θ_i, where $y \neq z$: consider the example at the beginning of this chapter. Therefore, the current analysis allows us to omit the same variables from closures as before, and possibly others.

$$\begin{array}{l}
\mathcal{D}_{Map}(\Gamma) = \Gamma', \\
\quad \text{where } Dom(\Gamma) = Dom(\Gamma'), \text{ and} \\
\qquad\qquad \forall i \in Dom(\Gamma),\ \Gamma(i) = (A, \mathcal{P}) \Longrightarrow \Gamma'(i) = (\mathcal{D}_{Env}(A), \mathcal{D}_{Val}(\mathcal{P})) \\
\\
\mathcal{D}_{Env}(A) = A', \\
\quad \text{where } Dom(A) = Dom(A'), \text{ and} \\
\qquad\qquad \forall x \in Dom(A),\ A(x) = \mathcal{P} \Longrightarrow A'(x) = \mathcal{D}_{Val}(\mathcal{P}) \\
\\
\mathcal{D}_{Val}(\phi, \pi, \theta) = (\phi', \pi', \theta') \\
\quad \text{where } \phi' = \{(j, \mathcal{D}_{Env}(B)) \mid (j, B) \in \phi\}, \\
\qquad\qquad \pi' = \pi, \text{ and} \\
\qquad\qquad \theta' = \{(x, x) \mid x \in \theta\}
\end{array}$$

Figure 14: Annotation translations

We can prove this subsumption claim. To do so, define the mutually recursive translations $\mathcal{D}_{Map}$, $\mathcal{D}_{Env}$, and $\mathcal{D}_{Val}$ shown in Figure 14. $\mathcal{D}_{Map}$ is a translation from annotation maps as described in the preceding chapter to our current annotation maps. Similarly, $\mathcal{D}_{Env}$ is a translation from our earlier environment propositions to our current environment propositions, and $\mathcal{D}_{Val}$ is a translation from our earlier value propositions to our current value propositions.

Recall that environment propositions and value propositions may describe infinite trees. The translations in Figure 14 map infinite trees to infinite trees. The translations can be considered a set of mutually recursive equations to be satisfied, rather than an algorithm. If an annotation map Γ is monovariant, however, then $\forall i \in Dom(\Gamma)$, $\Gamma(i) = (A_i, \mathcal{P}_i)$ implies the trees rooted at A_i and $\mathcal{P}_i$ are regular. In that case, the number of subtrees is finite, so the number of different nodes is finite; hence, the number of equations to be solved is finite. Therefore, in the case of monovariance, we can consider a subsumption translation as a *system of regular equations* [16, Section 4.2]. Such a system has a unique solution.

We can show:

Lemma 8 *If $\mathcal{P} \leq \mathcal{P}'$, then $\mathcal{D}_{Val}(\mathcal{P}) \leq \mathcal{D}_{Val}(\mathcal{P}')$.*

Proof. By the definition of $\mathcal{D}_{Val}$.

Let $\mathcal{P} = (\phi, \pi, \theta)$, and $\mathcal{P}' = (\phi', \pi', \theta')$, and suppose $\mathcal{P} \leq \mathcal{P}'$. Then $\phi \subseteq \phi'$, $\pi = \pi'$, and $\theta' \subseteq \theta$.

Now let $\mathcal{D}_{Val}(\mathcal{P}) = (\phi_{\mathcal{D}}, \pi_{\mathcal{D}}, \theta_{\mathcal{D}})$, and $\mathcal{D}_{Val}(\mathcal{P}') = (\phi'_{\mathcal{D}}, \pi'_{\mathcal{D}}, \theta'_{\mathcal{D}})$. By the definition of $\mathcal{D}_{Val}$:

- $\phi_{\mathcal{D}} = \{(j, \mathcal{D}_{Env}(B)) \mid (j, B) \in \phi\}$,
- $\phi'_{\mathcal{D}} = \{(j, \mathcal{D}_{Env}(B)) \mid (j, B) \in \phi'\}$,
- $\pi_{\mathcal{D}} = \pi$,
- $\pi'_{\mathcal{D}} = \pi'$,
- $\theta_{\mathcal{D}} = \{(x, x) \mid x \in \theta\}$, and
- $\theta'_{\mathcal{D}} = \{(x, x) \mid x \in \theta'\}$

Therefore, $\phi_{\mathcal{D}} \subseteq \phi'_{\mathcal{D}}$, $\pi_{\mathcal{D}} = \pi_{\mathcal{D}'}$, and $\theta'_{\mathcal{D}} \subseteq \theta_{\mathcal{D}}$. By the ordering on value propositions, $(\phi_{\mathcal{D}}, \pi_{\mathcal{D}}, \theta_{\mathcal{D}}) \leq (\phi'_{\mathcal{D}}, \pi'_{\mathcal{D}}, \theta'_{\mathcal{D}})$. ∎

Now we can prove the subsumption claim.

Theorem 8 (Subsumption) *Let Γ be an annotation map that is locally-consistent with respect to the conditions given in Figure 7, and let $\Gamma' = \mathcal{D}_{Map}(\Gamma)$. Then Γ' is locally-consistent with respect to the conditions given in Figure 13.*

Proof. By showing the local consistency of Γ' for each node $i \in Dom(\Gamma')$.

We refer to the constraints in Figure 7 as the "old constraints", and those in Figure 13 as the "new constraints."

For all $i \in Dom(\Gamma)$, $\Gamma(i) = (A_i, \mathcal{P}_i)$, so by the definition of $\mathcal{D}_{Map}$, $\Gamma'(i) = (\mathcal{D}_{Env}(A_i), \mathcal{D}_{Val}(\mathcal{P}_i))$. Therefore, for all $i \in Dom(\Gamma')$, the environment proposition associated with i is $A'_i = \mathcal{D}_{Env}(A_i)$, and the value proposition associated with i is $(\phi'_i, \pi'_i, \theta'_i) = \mathcal{P}'_i = \mathcal{D}_{Val}(\mathcal{P}_i)$.

case $Var(i)$

By the old constraints, $A_i([\![i]\!]) = \mathcal{P}_i$. Since $[\![i]\!] \in Dom(A_i)$, by the definition of $\mathcal{D}_{Env}$, $[\![i]\!] \in Dom(A'_i)$, and $A'_i([\![i]\!]) = \mathcal{D}_{Val}(A_i([\![i]\!]))$. Therefore, $A'_i([\![i]\!]) = \mathcal{D}_{Val}(\mathcal{P}_i) = \mathcal{P}'_i$. Hence, the new constraint $A'_i([\![i]\!]) = \mathcal{P}'_i$ is satisfied.

case $Const(i)$

By the old constraints, $\theta_i = \emptyset$. By the definition of $\mathcal{D}_{Val}$, $\theta'_i = \emptyset$, satisfying the only new constraint for this case.

case $Abs(i)$

There are three old and three new constraints, which may be considered in pairs.

Under the old constraints, $A_{i.body} = A_i[[\![i.bv]\!] \mapsto \mathcal{P}_{i.bv}]$, so $\mathcal{D}_{Env}(A_{i.body}) = \mathcal{D}_{Env}(A_i[[\![i.bv]\!] \mapsto \mathcal{P}_{i.bv}])$. The comparable new constraint is $A'_{i.body} = A'_i[[\![i.bv]\!] \mapsto \mathcal{P}'_{i.bv}]$. But $\mathcal{D}_{Env}(A_{i.body}) = A'_{i.body}$, and $\mathcal{D}_{Env}(A_i[[\![i.bv]\!] \mapsto \mathcal{P}_{i.bv}]) = \mathcal{D}_{Env}(A_i)[[\![i.bv]\!] \mapsto \mathcal{D}_{Val}(\mathcal{P}_{i.bv})] = A'_i[[\![i.bv]\!] \mapsto \mathcal{P}'_{i.bv}]$.

By the old constraints, $\{(i, A_i)\} \subseteq \phi_i$. The comparable new constraint is $\{(i, A'_i)\} \subseteq \phi'_i$. By the definition of $\mathcal{D}_{Val}$, $\phi'_i = \{(j, \mathcal{D}_{Env}(B)) \mid (j, B) \in \phi_i\}$. Therefore, $\{(i, \mathcal{D}_{Env}(A_i))\} = \{(i, A'_i)\} \subseteq \phi'_i$.

Last, we have the old constraint $\theta_i \subseteq Dom(A_i)$. The comparable new constraint is $(x, y) \in \theta'_i \implies x = y \wedge x \in Dom(A'_i)$. Now, by the definition of $\mathcal{D}_{Val}$, $\theta'_i = \{(x, x) \mid x \in \theta_i\}$, so the first conjunct of the new constraint is satisfied. By the definition of $\mathcal{D}_{Env}$, $Dom(A_i) = Dom(A'_i)$, satisfying the second conjunct.

case $PrimApp(i)$

The only old constraint here is $A_{i.rand} = A_i$. Therefore, $\mathcal{D}_{Env}(A_{i.rand}) = \mathcal{D}_{Env}(A_i)$. The only new constraint is $A'_{i.rand} = A'_i$. But $\mathcal{D}_{Env}(A_{i.rand}) = A'_{i.rand}$ and $\mathcal{D}_{Env}(A_i) = A'_i$.

case $App(i)$

Again here, we can consider the old and constraints in pairs, with exceptions.

Under the old constraints, $A_{i.rator} = A_{i.rand} = A_i$. So $\mathcal{D}_{Env}(A_{i.rator}) = \mathcal{D}_{Env}(A_{i.rand}) = \mathcal{D}_{Env}(A_i)$. Therefore, $A'_{i.rator} = A'_{i.rand} = A'_i$, satisfying the comparable new constraint.

Under the old constraints, $\theta_i \subseteq \theta_{i.rator}$. The comparable new constraint is $\theta'_i \subseteq \theta'_{i.rator}$. Now, $\theta'_i = \{(x, x) \mid x \in \theta_i\}$ and $\theta'_{i.rator} = \{(x, x) \mid x \in \theta_{i.rator}\}$. By this construction, the new constraint is satisfied.

One of the new contraints has no counterpart in the old constraints. Consider the new constraint $(x, y) \in \theta'_{i.rator} \cap \theta'_{i.rand} \implies (x, x) \in \theta'_{i.rator}$. Since under the $\mathcal{D}_{Val}$ translation $\theta'_{i.rator}$ may contain only such "diagonal" elements, the constraint holds as a tautology.

Consider the old constraint, $\pi_{i.rator} = cl_\sigma \implies \lceil\sigma\rceil \subseteq \theta_{i.rator}$. The comparable new constraint is $\pi'_{i.rator} = cl'_\sigma \implies \forall y \in \lceil\sigma\rceil, \exists x$ such that $(x, y) \in \theta'_{i.rator}$. Now, by the definition of $\mathcal{D}_{Val}$,

$\pi_{i.rator} = \pi'_{i.rator}$. Suppose $\pi_{i.rator} = id$. Then both the old and new constraints are satisfied trivially. Now suppose $\pi_{i.rator} = cl_\sigma$. By the old constraint, $\lceil\sigma\rceil \subseteq \theta_{i.rator}$. By the definition of $\mathcal{D}_{Val}$, $\theta'_{i.rator} = \{(x,x) \mid x \in \theta_{i.rator}\}$. Therefore, $\forall x \in \lceil\sigma\rceil$, $(x,x) \in \theta'_{i.rator}$, satisfying the new constraint.

Next, consider the constraints conditioned on the flow for the operator. By the definition of $\mathcal{D}_{Val}$, $\phi'_{i.rator} = \{(j, \mathcal{D}_{Env}(B)) \mid (j,B) \in \phi_{i.rator}\}$. Therefore, the old and new flows contain abstract closures with the same indices, so we can examine sets of old and new constraints in pairs.

The first set of old constraints here is $\forall (j,B) \in \phi_{i.rator},\ Abs(j)$. By the construction of $\phi'_{i.rator}$ just given, if $(j,B) \in \phi_{i.rator}$, then $(j, \mathcal{D}_{Env}(B)) \in \phi'_{i.rator}$. Therefore, the set of new constraints, $\forall (j,B) \in \phi'_{i.rator},\ Abs(j)$, is satisfied.

Next, we have the old constraints, $\forall (j,B) \in \phi_{i.rator}, \mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}$. The comparable set of new constraints is, $\forall (j,B) \in \phi'_{i.rator}, \mathcal{P}'_{i.rand} \leq \mathcal{P}'_{j.bv}$. But $\mathcal{P}'_{i.rand} = \mathcal{D}_{Val}(\mathcal{P}_{i.rand})$, and $\forall (j,B) \in \phi'_{i.rator}$, $\mathcal{P}'_{j.bv} = \mathcal{D}_{Val}(\mathcal{P}_{j.bv})$. The result holds by Lemma 8. A similar argument can be made for the old constraints, $\forall (j,B) \in \phi_{i.rator}, \mathcal{P}_{j.body} \leq \mathcal{P}_i$, and the comparable new constraints, $\forall (j,B) \in \phi'_{i.rator}$, $\mathcal{P}'_{j.body} \leq \mathcal{P}'_i$.

For the old constraints, $\forall (j,B) \in \phi_{i.rator}, \theta_{j.bv} \subseteq \theta_{i.rator}$, the comparable set of new constraints is, $\forall (j,B) \in \phi'_{i.rator}, \theta'_{j.bv} \subseteq \theta'_{i.rator}$. We can use the same reasoning here as above, where we concluded that $\theta'_i \subseteq \theta'_{i.rator}$ followed from the comparable inclusion on invariance sets.

Next, we have the old constraints, $\forall (j,B) \in \phi_{i.rator}, [\![j.bv]\!] \notin \theta_{j.bv} \cup \theta_i$. The comparable new constraints are $\forall (j,B) \in \phi'_{i.rator}, (x,y) \in \theta'_{j.bv} \cup \theta'_i \implies x \neq [\![j.bv]\!]$. Since the old constraints hold by assumption, $\theta'_{j.bv} = \{(x,x) \mid x \in \theta_{j.bv}\}$, and $\theta'_i = \{(x,x) \mid x \in \theta_i\}$, so the new constraints are satisfied.

Finally, we have the new constraints $\forall (j,B) \in \phi'_{i.rator}, (x,y) \in \theta'_{i.rator} \cap (\theta'_{j.body} - \{([\![j.bv]\!], \_)\}) \implies (x,x) \in \theta'_{i.rator}$. There are no comparable old constraints. Since $\theta'_{i.rator}$ contains only diagonal elements, the constraint holds as a tautology.

case $Cond(i)$

Under the old constraints, $A_{i.test} = A_{i.then} = A_{i.else} = A_i$, so $\mathcal{D}_{Env}(A_{i.test}) = \mathcal{D}_{Env}(A_{i.then}) = \mathcal{D}_{Env}(A_{i.else}) = \mathcal{D}_{Env}(A_i)$. Therefore, $A'_{i.test} = A'_{i.then} = A'_{i.else} = A'_i$.

We have the old constraint $\mathcal{P}_{i.then} \leq \mathcal{P}_i$. By Lemma 8, $\mathcal{D}_{Val}(\mathcal{P}_{i.then}) \leq \mathcal{D}_{Val}(\mathcal{P}_i)$, so the new constraint $\mathcal{P}'_{i.then} \leq \mathcal{P}'_i$ is satisfied. A similar argument holds for the old constraint $\mathcal{P}_{i.else} \leq \mathcal{P}_i$ and the comparable new constraint $\mathcal{P}'_{i.else} \leq \mathcal{P}'_i$. ∎

4.7 Revising the transformation; correctness

What do we have to change in the transformation? The previous transformation performed two essential tasks: forming closures, and supplying extra variable arguments at call sites. The basic job of the new transformation is the same, only some details differ.

If we have a procedure with a protocol tag of $cl_{\langle v_1,\ldots,v_n\rangle}$, that means we can omit $\{v_1,\ldots,v_n\}$ from the procedure's closure. That is as before. Now, for a call site that has a protocol tag of $cl_{\langle v_1,\ldots,v_n\rangle}$, for each v_i in $\{v_1,\ldots,v_n\}$, we need to choose a pair (u, v_i) from the invariance relation, and supply u as a dynamic variable. We know such a pair exists by the local consistency conditions.

We give the new transformation in Figure 15. We have the map:

$$\Phi \colon O \to \Lambda_{clos}$$

Again, the transform is extended to occurrence closures:

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

The reader should be aware that we are overloading the symbols Φ and $\hat{\Phi}$ — the transformations in the current chapter and Chapter 3 are different. Before proving the correctness theorem, we need a supporting lemma.

Lemma 9 *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.

Use the same reasoning as in the proof of Lemma 7. ∎

The Correctness Theorem is stated exactly as before.

Theorem 9 (Correctness) *Let Γ 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 $\psi \overset{\Pi}{\models}_{env} A_i$. If*

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

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

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

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

$$Abs(i) \wedge \pi_i = cl_{\langle v_1,\ldots,v_n\rangle} \Longrightarrow \Phi(i) = [(\lambda e v_1 \ldots v_n [\![i.bv]\!].\, \mathbf{destr}\ e\ (\lambda\vec{u}.\Phi(i.body))), [\vec{u}]] \quad \text{where } e \text{ fresh and } \vec{u} = FV([\![i]\!]) - \{v_1,\ldots,v_n\}$$

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

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

$$App(i) \wedge \pi_{i.rator} = cl_{\langle v_1,\ldots,v_n\rangle} \Longrightarrow \Phi(i) = \mathbf{app}\ \Phi(i.rator)\ w_1\ \ldots\ w_n\ \Phi(i.rand) \quad \text{where } \forall k,\ 1 \le k \le n,\ (w_k, v_k) \in \theta_{i.rator}$$

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

Figure 15: The available-value closure conversion transformation

then

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

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

For all cases save one subcase, our proof here is the same as for Theorem 6. In the subcase that $App(i)$ and $\pi_{i.rator} = cl_{\langle v_1,\ldots,v_n\rangle}$, we can slightly modify the corresponding subcase in the proof of Theorem 6.

Suppose we have the derivation:

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

Then:

$$\begin{aligned} \hat{\Phi}(i,\psi) &= \Phi(i)\{\hat{\Phi}\circ\psi\} \\ &= (\mathbf{app}\ \Phi(i.rator)\ w_1\ \ldots\ w_n\ \Phi(i.rand))\{\hat{\Phi}\circ\psi\} \end{aligned}$$

$$= \quad \mathbf{app}\ \hat{\Phi}(i.rator, \psi)\ \hat{\Phi}(\psi(w_1))\ \ldots\ \hat{\Phi}(\psi(w_n))\ \hat{\Phi}(i.rand, \psi)$$

This is the same transform as we had in the earlier proof, except that the set of dynamic variables $\{w_1\ \ldots\ w_n\}$ may be different than the set of variables that appear in the protocol tag $\pi_{i.rator} = cl_{\langle v_1,\ldots,v_n\rangle}$.

By the reasoning given in the earlier proof, we can show that this last term co-evaluates with

$$\mathbf{app}\ \hat{\Phi}(j, \psi')\ \hat{\Phi}(\psi(w_1))\ \ldots\ \hat{\Phi}(\psi(w_n))\ \hat{\Phi}(k, \psi'')$$

By replacing dynamic v's with dynamic w's, we can go through the same chain of equational reasoning to show:

$$\begin{array}{ll} & \mathbf{app}\ \hat{\Phi}(j, \psi')\ \hat{\Phi}(\psi(w_1))\ \ldots\ \hat{\Phi}(\psi(w_n))\ \hat{\Phi}(k, \psi'') \\ =_w & (\lambda v_1 \ldots v_n [\![j.bv]\!].\, \mathbf{destr}\ \vec{u}\ \lambda\vec{u}.\Phi(j.body))\{\hat{\Phi} \circ \psi'\}\ \hat{\Phi}(\psi(w_1))\ \ldots\ \hat{\Phi}(\psi(w_n))\ \hat{\Phi}(k, \psi'') \end{array}$$

The comparable equation in the proof of Theorem 6 appears on p. 71. In the next sequence of steps, we will rely on the invariance *relations* in our new analysis, where the corresponding sequence in the earlier proof relied on invariance *sets*.

By Lemma 9, each term in $\hat{\Phi}(\psi(w_1))\ \ldots\ \hat{\Phi}(\psi(w_n))$ is a value, so we can repeatedly invoke the β and **cong** rules for $=_w$:

$$\begin{array}{ll} & (\lambda v_1 \ldots v_n [\![j.bv]\!].\, \mathbf{destr}[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))\{\hat{\Phi} \circ \psi'\}\ \hat{\Phi}(\psi(w_1))\ \ldots\ \hat{\Phi}(\psi(w_n)) \\ = & (\lambda v_1'.(\lambda v_2 \ldots v_n [\![j.bv]\!].\, \mathbf{destr}[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))[v_1'/v_1])\{\hat{\Phi} \circ \psi'\}\ \hat{\Phi}(\psi(w_1))\ \ldots \\ = & \lambda v_1'.(\lambda v_2 \ldots v_n [\![j.bv]\!].\, \mathbf{destr}[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))[v_1'/v_1]\{\hat{\Phi} \circ \psi'\}\ \hat{\Phi}(\psi(w_1))\ \ldots \\ =_w & (\lambda v_2 \ldots v_n [\![j.bv]\!].\, \mathbf{destr}[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))[v_1'/v_1]\{\hat{\Phi} \circ \psi'\}[\hat{\Phi}(\psi(w_1))/v_1']\ \ldots \\ = & (\lambda v_2 \ldots v_n [\![j.bv]\!].\, \mathbf{destr}[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))\{\hat{\Phi} \circ \psi'[v_1 \mapsto \psi(w_1)]\}\ \hat{\Phi}(\psi(w_2))\ \ldots \\ & \ldots \\ =_w & (\lambda [\![j.bv]\!].\, \mathbf{destr}[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))\{\hat{\Phi} \circ \psi'[v_1, \ldots, v_n \mapsto \psi(w_1), \ldots, \psi(w_n)]\} \\ = & (\lambda [\![j.bv]\!].\, \mathbf{destr}[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))\{\hat{\Phi} \circ \psi'[v_1, \ldots, v_n \mapsto \psi'(v_1), \ldots, \psi'(v_n)]\} \\ = & (\lambda [\![j.bv]\!].\, \mathbf{destr}[\vec{u}]\ \lambda\vec{u}.\Phi(j.body))\{\hat{\Phi} \circ \psi'\} \end{array}$$

Most of the steps here are justified by arguments used in the presentation of the corresponding sequence in the earlier proof.

As in the earlier proof, the last two steps are crucial for understanding why lightweight closures work correctly. By assumption, $\psi \models^{\Pi}_{env} A_i$, and by the local consistency of Γ, $A_{i.rator} = A_i$. Since $(i.rator, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$ and $Abs(j)$, by Theorem 7, $\forall (x, y) \in \theta_{i.rator}$, $x \in Dom(\psi)$, $y \in Dom(\psi')$, and

$\psi(x) = \psi'(y)$. Now, by the local consistency of Γ, since $\pi_{i.rator} = cl_{\langle v_1,\ldots,v_n \rangle}$, for each variable v in $\{v_1, \ldots, v_n\}$, there is a w such that $(w, v) \in \theta_{i.rator}$. Those w's are the dynamic variables $w_1 \ \ldots \ w_n$ inserted at the call site by the transformation. So in the next-to-last step, in the extended ψ', we can safely replace $\psi(w_1), \ldots, \psi(w_n)$ by $\psi'(v_1), \ldots, \psi'(v_n)$. In the last step, we discard the redundant extensions to ψ'.

On the last line of the sequence just given, we have a term identical to that on the last line of the corresponding sequence in the earlier proof. Therefore, from this point forward, we can proceed as in the earlier proof. We conclude that $\hat{\Phi}(i, \psi) \underset{t}{\Longrightarrow} \hat{\Phi}(m, \psi''')$, as desired. ∎

4.8 Conclusion

In this chapter, we generalized the lightweightness feature of our earlier closure conversion transformation. This incremental improvement provides some evidence of the strength of the method used. In the following chapters, as better evidence of that strength, we will apply that method to two new and wholly different transformations.

Chapter 5

Ultra-β

In this chapter, we adapt the techniques developed for closure conversion to a different transformation. Ultra-β extends a well-known transformation for first-order programs to the higher-order setting.

5.1 Eliminating redundant bindings

If we can determine that a variable reference is a copy of another variable, the copy can be replaced by the original. In conventional compiler technology, this technique is called *copy propagation* [3, Chapter 10].

Shivers has applied this technique to Scheme [49, Chapter 10]. His "Super-β" transformation eliminates bindings in two steps. First, copies of variables are replaced with their originals. Next, a useless variable analysis determines if a binding is no longer needed. If so, the binding can be eliminated.

We adapt the techniques developed earlier to perform a similar transformation. With some hubris, we call this transformation "Ultra-β". As before, we perform a dataflow analysis by solving local consistency constraints. We show that any solution to the constraints is sound, and that the resulting transformation is correct.

5.2 Examples and discussion

As we mentioned in Chapter 1, in simple cases, replacing a copy of a variable by its original can be accomplished with β-reduction. In the program

$$\begin{array}{l} \textbf{let } x = \ldots \\ \textbf{in } \ldots \textbf{ let } y = x \\ \qquad\quad \textbf{in } \ldots y \ldots \end{array}$$

the reference to y has the same value as x.

We can remove the inner **let**, and substitute x for y:

$$\begin{array}{l} \textbf{let } x = \ldots \\ \textbf{in } \ldots x \ldots \end{array}$$

Note that the binder for the substituted variable x has to be visible at the substitution site.

An intermediate variable may establish the link between a variable and its copy:

$$\begin{array}{l} \textbf{let } x = \ldots \\ \textbf{in } \ldots \textbf{ let } y = x \\ \qquad\quad \textbf{in } \ldots y \ldots \textbf{ let } z = y \\ \qquad\qquad\qquad\qquad\quad y = a \\ \qquad\qquad\qquad\quad\;\; \textbf{in } \ldots z \ldots \end{array}$$

In this example, z becomes a copy of x via the first binding for y. That binding also makes the reference to y a copy of x. At its reference site, z is not a copy of y, so we cannot substitute y for z. But we can perform two β-reduction steps, substituting x for the references to y and z:

$$\begin{array}{l} \textbf{let } x = \ldots \\ \textbf{in } \ldots x \ldots \textbf{ let } y = a \\ \qquad\qquad\quad \textbf{in } \ldots x \ldots \end{array}$$

In the last two examples, β-reduction removed a binding **let**, which is just syntactic sugar for procedure application. After each β-reduction step, we are left with a procedure body in which a variable has been substituted for the procedure's binding variable. The actual Ultra-β transformation is designed so that if we evaluate such a procedure body in a transformed program, the procedure would have been applied in the original program. Otherwise, evaluating such a procedure body may cause undesirable consequences, such as looping or exceptions, which would not have occurred in the original program. Therefore, the actual transformation does more than simple β-reduction. We next describe how the transformation avoids undesirable consequences.

5.2.1 Removing binders

In Shivers' Super-β, when a copy variable is removed, its corresponding binder in a Scheme formal parameter list is also removed. Rewriting our first example in this chapter in Scheme, and desugaring the **let**, we have:

```
((lambda (x)
   ((lambda (y)
         ... y ...) x)) ... )
```

and the transform is:

```
((lambda (x)
   ((lambda ()
         ... x ...)) ... )
```

Removing the binder just reduces the number of arguments to the Scheme procedure; the procedure is still a procedure. In this example, the transformed procedure has zero arguments. In Scheme, it is possible to have procedures of zero arguments, so this strategy is acceptable. Super-β transforms a Scheme procedure into a procedure with possibly fewer arguments. We want Ultra-β to have a similar effect on procedures in Λ_{in}.

5.2.2 Using thunks; binding protocols

In Λ_{in}, all procedures have exactly one argument, so we need a way to describe procedures that have had their binding variable removed. To that end, we introduce the notion of parameterless procedures, called thunks,[1] into the language of transformed programs.

[1] Thunks originated in the design of the language ALGOL-60 to implement a call-by-name evaluation strategy [26]. We shall see this use in the following chapter.

In the program

$$\begin{array}{l}\lambda v.\ \ldots\ \mathbf{let}\ f = (\lambda x.\ \ldots\ x\ \ldots) \\ \qquad\quad \mathbf{in\ let}\ z = v \\ \qquad\qquad \mathbf{in}\ \ldots\ (fz)\ \ldots\end{array}$$

the reference to x inside the body of f is a copy of v, if f is ever applied. But we cannot simply remove the binding variable of f and substitute v for x in the body of f:

$$\begin{array}{l}\lambda v.\ \ldots\ \mathbf{let}\ f = (\ \ldots\ v\ \ldots) \qquad \texttt{wrong!} \\ \qquad\quad \mathbf{in\ let}\ z = v \\ \qquad\qquad \mathbf{in}\ \ldots\ f\ \ldots\end{array}$$

It is not certain that f will be applied. Suppose now that if f is applied, it just loops. It may be that f is never applied in the original program. Therefore, naively removing the binder will cause the transformed program to loop. Instead, the transform should be:

$$\begin{array}{l}\lambda v.\ \ldots\ \mathbf{let}\ f = \mathbf{thunk}(\ldots\ v\ \ldots) \\ \qquad\quad \mathbf{in\ let}\ z = v \\ \qquad\qquad \mathbf{in}\ \ldots\ \mathbf{run}\ f\ \ldots\end{array}$$

where **run** is an operator that causes the body of a thunk to be evaluated. In the transformed program, the body of f is evaluated as the body of the created thunk. Since the thunk is run at the same site where f was applied, the thunk is run only if in the original program, f would have been applied. At the call site (fz), we can remove the argument, because the thunk does not take arguments. Note that the inner **let** may be unneeded now, in case there are no other references to z. Ultra-β does not take the extra step of removing such useless bindings.

Observe that in this transformation, procedures now obey two *binding protocols*. For the transformed program to work correctly, it must be that all values flowing to a call site are ordinary procedures, or all are thunks. As in the closure conversion analysis, a dataflow analysis makes sure that all values flowing to a call site obey the same protocol. To see why the flow analysis is needed, examine:

$$
\begin{aligned}
\lambda v.\ \ldots\ &\textbf{let } f = (\lambda x.\ \ldots\ x\ \ldots) \\
&\qquad g = (\lambda y.\ \ldots\ y\ \ldots) \\
&\textbf{in let } h = (\textbf{if } (\textbf{zero?}\ t)\ \textbf{then } f \textbf{ else } g) \\
&\qquad\quad z = v \\
&\quad \textbf{in } (h\ z)
\end{aligned}
$$

At the call site $h\ z$, h is bound to either f or g. Because z is a copy of v, the references to x and y in the bodies of f and g, respectively, are also copies of v. Therefore, the transformation can generate thunks for f and g. The procedures f and g must agree on protocol, so either both should become thunks under the transformation, else both should remain ordinary procedures. In the first case, the transform is:

$$
\begin{aligned}
\lambda v.\ \ldots\ &\textbf{let } f = \textbf{thunk}(\ldots\ v\ \ldots) \\
&\qquad g = \textbf{thunk}(\ldots\ v\ \ldots) \\
&\textbf{in let } h = (\textbf{if } (\textbf{zero?}\ t)\ \textbf{then } f \textbf{ else } g) \\
&\qquad\quad z = v \\
&\quad \textbf{in run } h
\end{aligned}
$$

5.3 A new output language

As our examples showed, the Ultra-β transformation produces terms in an output language that includes thunks. A grammar that describes Λ_{thunk} is:

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

where PrimOp is defined as in Chapter 2. Values are now elements of the subset of Λ_{thunk} given by the grammar:

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

where M is any term in Λ_{thunk}.

For convenience, we will continue to use the symbol $\underset{t}{\Longrightarrow}$ to indicate the evaluation relation on terms and their values. The term evaluator for Λ_{thunk} includes all the evaluation rules for Λ_{in} in Chapter 2, Figure 1, and the additional rules shown in Figure 16.

$$\textbf{thunk}\ M \underset{t}{\Longrightarrow} \textbf{thunk}\ M$$

$$\frac{M \underset{t}{\Longrightarrow} \textbf{thunk}\ N \quad N \underset{t}{\Longrightarrow} P}{\textbf{run}\ M \underset{t}{\Longrightarrow} P}$$

Figure 16: Additional term evaluation rules for Λ_{thunk}

5.4 Annotations

Because the Ultra-β transformation is much different than the two versions of closure conversion in the last two chapters, the supporting annotations and their constraints are considerably different than before. We begin by describing the language of annotations.

Definition 5 *An* alias relation *is a binary relation on Var.*

We write $\mathbf{A}$ to indicate an alias relation.

By writing $(x, y) \in \mathbf{A}$, we mean that x and y are aliased to the same value in some occurrence environment. For each occurrence i, we will associate two alias relations, $\mathbf{A}_i^-$ and $\mathbf{A}_i^+$. If we have the occurrence evaluation $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$, we intend that $\mathbf{A}_i^-$ indicates the aliases in ψ, while $\mathbf{A}_i^+$ describes the aliases in ψ'. Abusing ordinary set notation, we will write $\mathbf{A} - s$ to mean $\mathbf{A} - \{(p, q) \mid p = s \text{ or } q = s\}$.

Definition 6 *A* binder tag *π is an element of the set $\{id, th\}$.*

If a procedure has a binder tag of th, we mean that the flow analysis indicates that its binding variable is always a copy of another variable in scope where the procedure is closed, so the transform of the procedure should be a thunk. If the flow analysis cannot make that determination, the procedure gets the binder tag id.

Definition 7 *A* binder tag assignment *Π is a map from occurrence indices to reference tags.*

As in the previous analyses, each parse tree node is annotated with an environment proposition and a value proposition. A value proposition $\mathcal{P}$ is now a 4-tuple consisting of a flow, binder tag, invariance relation, and a "+" alias relation. Therefore, $\mathcal{P}_i = (\phi_i, \pi_i, \theta_i, \mathbf{A}_i^+)$. Again, an environment proposition A is a finite map from variables to value propositions.

For each parse tree node, we also add a "-" alias relation. Therefore, an annotation map is a map from occurrence indices to triples consisting of an environment proposition, an alias relation, and a value proposition. For an annotation map Γ, we have $\Gamma(i) = (A_i, \mathbf{A}_i^-, \mathcal{P}_i)$.

5.5 The semantics of annotations

We now have three satisfaction relations.

The relation on environments and environment propositions is very much like the corresponding relation in Chapter 4. For values and value propositions, the satisfaction relation is also similar to the corresponding relations in the two closure conversion analyses, with the additional requirement that the environment part of the value satisfy the alias relation part of the proposition. An environment satisfies an alias relation if all pairs in the alias relation are indeed aliases in that environment.

Formally, we have:

1. An occurrence environment ψ satisfies an alias relation $\mathbf{A}$, as follows:

$$\psi \models_{alias} \mathbf{A}$$

   iff $(s,t) \in \mathbf{A} \Longrightarrow s,t \in Dom(\psi)$ and $\psi(s) = \psi(t)$.

2. An occurrence environment ψ satisfies an environment proposition A under a binder tag assignment Π, as follows:

$$\psi \overset{\Pi}{\models}_{env} A$$

   iff $\forall x \in Dom(A)$

   (a) $x \in Dom(\psi)$ and $\psi(x) \overset{\Pi}{\models}_{val} A(x)$, and

   (b) if $A(x) = (\phi, \theta, \pi, \mathbf{A})$ and $\psi(x) = (i, \psi')$, then $\forall (y,z) \in \theta$, $y \in Dom(\psi)$, $z \in Dom(\psi')$, and $\psi(y) = \psi'(z)$

3. An occurrence closure (i, ψ) satisfies a value proposition $(\phi, \theta, \pi, \mathbf{A})$ under a binder tag assignment Π, as follows:

$$(i, \psi) \overset{\Pi}{\models}_{val} (\phi, \theta, \pi, \mathbf{A})$$

iff $\psi \models_{alias} \mathbf{A}$, and either:

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

(b) $Abs(i)$, $\Pi(i) = \pi$, and there exists A such that $(i, A) \in \phi$ and $\psi \overset{\Pi}{\models}_{env} A$

Occurrence closures are finite, so these definitions are well-founded.

The following lemmas will be useful.

Lemma 10 *If* $\psi \models_{alias} \mathbf{A}$ *and* $\mathbf{A}' \subseteq \mathbf{A}$*, then* $\psi \models_{alias} \mathbf{A}'$*.*

Proof. Immediate from the definition of $\models_{alias}$. ∎

Lemma 11 *If* $\psi \models_{alias} \mathbf{A}, \mathbf{A}'$ *then* $\psi \models_{alias} \mathbf{A} \cup \mathbf{A}'$*.*

Proof. Immediate. ∎

Lemma 12 *If* $\psi \models_{alias} \mathbf{A}$*, and* $\mathbf{A}'$ *is the symmetric and transitive closure of* $\mathbf{A}$*, then* $\psi \models_{alias} \mathbf{A}'$*.*

Proof. By the definition of $\models_{alias}$, and the symmetry and transitivity of the equality of occurrence closures. ∎

Value propositions are ordered as follows. Say that $(\phi, \pi, \theta, \mathbf{A}) \leq (\phi', \pi', \theta', \mathbf{A}')$ iff $\phi \subseteq \phi'$, $\pi = \pi'$, $\theta' \subseteq \theta$, and $\mathbf{A}' \subseteq \mathbf{A}$. So:

Lemma 13

1. *If* $(i, \psi) \overset{\Pi}{\models}_{val} \mathcal{P}$*, and* $\mathcal{P} \leq \mathcal{P}'$*, then* $(i, \psi) \overset{\Pi}{\models}_{val} \mathcal{P}'$*.*
2. *If* $\psi \overset{\Pi}{\models}_{env} A$ *and* $\forall x \in Dom(A')$*,* $A(x) \leq A'(x)$*, then* $\psi \overset{\Pi}{\models}_{env} A'$*.*

Proof. By Lemma 10, the ordering on value propositions, and the definitions of $\overset{\Pi}{\models}_{val}$ and $\overset{\Pi}{\models}_{env}$. ∎

5.6 Local consistency conditions

The Ultra-β local consistency conditions are given in Figure 17. We write $\mathbf{STClos}\,(\mathbf{A})$ to indicate the symmetric and transitive closure of an alias relation $\mathbf{A}$. For flows, the constraints here are identical to those in the two closure conversion analyses, Chapters 3 and 4. For invariance relations, the constraints here are nearly the same as in the available value analysis, Chapter 4.[2] The constraints on alias relations are designed to assure that those relations in fact describe sets of aliases for the starting and result environments when we evaluate occurrence closures. For binder tags, the constraints assure that at a given call site, all procedures are thunks, else all are ordinary procedures. The binder tag constraints allow a procedure to be designated a thunk only if inside its body, the procedure's binding variable is a copy of another variable available to the procedure.

We can solve for the flows, alias relations, and invariance relations using an iterative algorithm. The flows and invariance relations can be initialized as for the available value analysis in Chapter 4. Initialize the "-" alias relation for the program (the parse tree root) to a relation that will be satisfied by the starting environment — the empty alias relation is satisfied by any occurrence environment, so that is always a valid choice. Initialize all other alias relations to the Cartesian product of all variables in the program. Next, iteratively apply the constraints until a solution for the flows, alias relations, and invariance relations is reached.

Finally, to solve for binder tags, partition occurrences into equivalence classes, as in the original closure conversion analysis (Chapter 3, Figure 10). Within each equivalence class, for each occurrence index i such that $Abs(i)$, test the condition

$$\exists x \text{ such that } (x, [\![i.bv]\!]) \in \mathbf{STClos}\,(\mathbf{A}^{-}_{i.body}) \wedge x \neq [\![i.bv]\!] \wedge x \in \mathit{Dom}(A_i)$$

If all procedure occurrences within a class pass the test, then give all occurrences in the class a binder tag of th. Otherwise, all occurrences in the class receive a binder tag of id.

5.7 Soundness

The Soundness Theorem for Ultra-β is similar to its previous counterparts. In the current theorem, as before, we have the premise that the starting environment satisfies the environment proposition associated with the starting occurrence. Unlike previous soundness theorems, however, we have the additional premise that the starting environment satisfies the "-" alias relation associated with the starting occurrence. The invariance consequent (*4*, below) is the same as we had in the soundness theorem for our available value analysis. Of course, the value satisfaction consequent (*3*) has

[2] In the available value analysis, there are no constraints on invariance relations in the $Const$ and $PrimApp$ cases. Also, the invariance relation constraint in the Abs case here is less restrictive than in the earlier analysis.

$$Var(i) \Longrightarrow A_i(\llbracket i \rrbracket) = \mathcal{P}_i$$

$$Const(i) \Longrightarrow \begin{cases} \theta_i = \emptyset \text{ and} \\ \mathbf{A}_i^+ = \emptyset \end{cases}$$

$$Abs(i) \Longrightarrow \begin{cases} A_{i.body} = A_i[\llbracket i.bv \rrbracket \mapsto \mathcal{P}_{i.bv}] \\ \{(i, A_i)\} \subseteq \phi_i, \\ (x,y) \in \theta_i \Longrightarrow (x = y \wedge x \in Dom(A_i)) \vee (x,y) \in \mathbf{STClos}\,(\mathbf{A}_i^-), \\ \pi_i = th \Longrightarrow \exists x \text{ such that } (x, \llbracket i.bv \rrbracket) \in \mathbf{STClos}\,(\mathbf{A}_{i.body}^-) \wedge \\ \qquad\qquad\qquad\qquad x \neq \llbracket i.bv \rrbracket \wedge x \in Dom(A_i), \text{ and} \\ \mathbf{A}_i^+ \subseteq \mathbf{A}_i^- \end{cases}$$

$$PrimApp(i) \Longrightarrow \begin{cases} A_{i.rand} = A_i, \\ \theta_i = \emptyset, \\ \mathbf{A}_{i.rand}^- \subseteq \mathbf{A}_i^-, \text{ and} \\ \mathbf{A}_i^+ = \emptyset \end{cases}$$

$$App(i) \Longrightarrow \begin{cases} A_{i.rator} = A_{i.rand} = A_i, \\ \mathbf{A}_{i.rator}^- \subseteq \mathbf{A}_i^-, \\ \mathbf{A}_{i.rand}^- \subseteq \mathbf{A}_i^-, \\ \theta_i \subseteq \theta_{i.rator}, \\ (x,y) \in \theta_{i.rator} \cap \theta_{i.rand} \Longrightarrow (x,x) \in \theta_{i.rator}, \text{ and} \\ \forall (j, B) \in \phi_{i.rator} \\ \quad \begin{cases} Abs(j), \\ \mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}, \\ \mathcal{P}_{j.body} \leq \mathcal{P}_i, \\ \theta_{j.bv} \subseteq \theta_{i.rator}, \\ (x,y) \in \theta_{j.bv} \cup \theta_i \Longrightarrow x \neq \llbracket j.bv \rrbracket, \\ (x,y) \in \theta_{i.rator} \cap (\theta_{j.body} - \{(\llbracket j.bv \rrbracket, \_)\}) \Longrightarrow (x,x) \in \theta_{i.rator}, \\ \neg\, Var(i.rand) \Longrightarrow \mathbf{A}_{j.body}^- \subseteq \mathbf{A}_{i.rator}^+ - \llbracket j.bv \rrbracket, \text{ and} \\ Var(i.rand) \Longrightarrow \mathbf{A}_{j.body}^- \subseteq (\mathbf{A}_{i.rator}^+ - \llbracket j.bv \rrbracket) \cup \\ \qquad\qquad \{(x, \llbracket j.bv \rrbracket) \mid (\llbracket i.rand \rrbracket, x) \in \theta_{i.rator}\} \end{cases} \end{cases}$$

$$Cond(i) \Longrightarrow \begin{cases} A_{i.test} = A_{i.then} = A_{i.else} = A_i, \\ \mathbf{A}_{i.test}^- \subseteq \mathbf{A}_i^-, \\ \mathbf{A}_{i.then}^- \subseteq \mathbf{A}_i^-, \\ \mathbf{A}_{i.else}^- \subseteq \mathbf{A}_i^-, \\ \mathcal{P}_{i.then} \leq \mathcal{P}_i, \text{ and} \\ \mathcal{P}_{i.else} \leq \mathcal{P}_i \end{cases}$$

Figure 17: Local consistency conditions for Ultra-β

changed, since the value proposition for the starting occurrence now includes the "+" alias relation component.[3]

Theorem 10 (Soundness) *Let Γ be a monovariant and locally-consistent annotation map and let Π be the binder tag assignment defined by $\forall i,\ \Pi(i) = \pi_i$. Let i be an occurrence index, and ψ an occurrence environment such that $\psi \overset{\Pi}{\models}_{env} A_i$ and $\psi \models_{alias} \mathbf{A}_i^-$. Suppose $(i, \psi) \underset{oc}{\Longrightarrow} (j, \psi')$. Then for any subproof $(k, \psi'') \underset{oc}{\Longrightarrow} (m, \psi''')$ of this derivation:*

1. $\psi'' \overset{\Pi}{\models}_{env} A_k$,
2. $\psi'' \models_{alias} \mathbf{A}_k^-$,
3. $(m, \psi''') \overset{\Pi}{\models}_{val} (\phi_k, \pi_k, \theta_k, \mathbf{A}_k^+)$, *and*
4. $\forall (x, y) \in \theta_k$, $x \in Dom(\psi'')$, $y \in Dom(\psi''')$, *and* $\psi''(x) = \psi'''(y)$.

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

Base cases

For the base cases, there are no proper subproofs. In each base case, consequents (*1*) and (*2*) hold by assumption.

case *Var*(i)

By the local consistency of Γ, $A_i(\llbracket i \rrbracket) = (\phi_i, \pi_i, \theta_i, \mathbf{A}_i^+)$, so $\llbracket i \rrbracket \in Dom(A_i)$. Since $\psi \overset{\Pi}{\models}_{env} A_i$, $\llbracket i \rrbracket \in Dom(\psi)$, so $(i, \psi) \underset{oc}{\Longrightarrow} \psi(\llbracket i \rrbracket)$. Also since $\psi \overset{\Pi}{\models}_{env} A_i$, $\psi(\llbracket i \rrbracket) \overset{\Pi}{\models}_{val} A_i(\llbracket i \rrbracket) = (\phi_i, \pi_i, \theta_i, \mathbf{A}_i^+)$.

Let $\psi(\llbracket i \rrbracket) = (j, \psi')$. Since $\psi \overset{\Pi}{\models}_{env} A_i$, $\forall (x, y) \in \theta_i$, $x \in Dom(\psi)$, $y \in Dom(\psi')$, and $\psi(x) = \psi'(y)$.

case *Const*(i)

We have $(i, \psi) \underset{oc}{\Longrightarrow} (i, \emptyset)$. By the local consistency of Γ, $\mathbf{A}_i^+ = \emptyset$, which is satisfied by any occurrence environment. $\llbracket i \rrbracket$ is a constant, so $(i, \emptyset) \overset{\Pi}{\models}_{val} (\phi_i, \pi_i, \theta_i, \mathbf{A}_i^+)$. By local consistency, $\theta_i = \emptyset$, so the invariant holds trivially.

[3] By the current Soundness Theorem, the alias relation portions of the Ultra-β analysis provide a correct solution to Exercise 10.36 in the "Dragon Book" [3, Chapter 10] for a higher-order language.

case $Abs(i)$

So $(i, \psi) \underset{oc}{\Longrightarrow} (i, \psi)$. By assumption, $\psi \models_{alias} \mathbf{A}_i^-$. By the local consistency of Γ, $\mathbf{A}_i^+ \subseteq \mathbf{A}_i^-$, so by Lemma 10, $\psi \models_{alias} \mathbf{A}_i^+$. By the definition of Π, $\Pi(i) = \pi_i$. By assumption, $\psi \overset{\Pi}{\models}_{env} A_i$, and by the local consistency of Γ, $\{(i, A_i)\} \subseteq \phi_i$. Therefore, $(i, \psi) \overset{\Pi}{\models}_{val} (\phi_i, \pi_i, \theta_i, \mathbf{A}_i^+)$.

To show that the invariant holds, consider a pair $(x, y) \in \theta_i$. By the local consistency conditions, either $x = y \wedge x \in Dom(A_i)$, or $(x, y) \in \mathbf{STClos}(\mathbf{A}_i^-)$.

Suppose on one hand that $x = y$ and $x \in Dom(A_i)$. By assumption $\psi \overset{\Pi}{\models}_{env} A_i$, and by the definition of $\overset{\Pi}{\models}_{env}$, $x \in Dom(\psi)$. Trivially, $\psi(x) = \psi(y)$.

Suppose on the other hand that $(x, y) \in \mathbf{STClos}(\mathbf{A}_i^-)$. By assumption, $\psi \models_{alias} \mathbf{A}_i^-$, so by Lemma 12, $\psi \models_{alias} \mathbf{STClos}(\mathbf{A}_i^-)$. By the definition of $\models_{alias}$, $\psi(x) = \psi(y)$.

Induction

For each case in the induction step, we first show that the consequents hold at the root of the proof tree. Then we show that the premises about $\overset{\Pi}{\models}_{env}$ and $\models_{alias}$ hold for all immediate subproofs of the root, so that the consequents hold for all proper subproofs.

case $PrimApp(i)$

We have $(i, \psi) \underset{oc}{\Longrightarrow} (c, \emptyset)$ for some constant c. By the local consistency of Γ, $\mathbf{A}_i^+ = \emptyset$, and trivially, any occurrence environment satisfies the empty alias relation. Since c is a constant, $(c, \emptyset) \overset{\Pi}{\models}_{val} (\phi_i, \pi_i, \theta_i, \mathbf{A}_i^+)$. By the local consistency of Γ, $\theta_i = \emptyset$, so 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, $A_{i.rand} = A_i$, so $\psi \overset{\Pi}{\models}_{env} A_{i.rand}$. Also by local consistency, $\mathbf{A}_{i.rand}^- \subseteq \mathbf{A}_i^-$, so by Lemma 10, $\psi \models_{alias} \mathbf{A}_{i.rand}^-$. Therefore, by the induction hypothesis at $i.rand$, all the consequents hold for all proper subproofs.

case $App(i)$

Suppose we have the derivation:

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

Then:

(1) $\psi \overset{\Pi}{\models}_{env} A_{i.rator}, A_{i.rand}$
$[A_{i.rator} = A_{i.rand} = A_i]$

(2) $\psi \models_{alias} \mathbf{A}^-_{i.rator}$
$[\mathbf{A}^-_{i.rator} \subseteq \mathbf{A}^-_i$, Lemma 10]

(3) $(j, \psi') \overset{\Pi}{\models}_{val} (\phi_{i.rator}, \pi_{i.rator}, \theta_{i.rator}, \mathbf{A}^+_{i.rator})$
[IH re $\overset{\Pi}{\models}_{val}$ at $i.rator$]

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

(5) $(j, A_j) \in \phi_{i.rator}$
[definition of $\overset{\Pi}{\models}_{val}$, monovariance, (3)]

(6) $\psi' \overset{\Pi}{\models}_{env} A_j$
[by definition of $\overset{\Pi}{\models}_{val}$, (3), (5)]

(7) $\psi' \models_{alias} \mathbf{A}^+_{i.rator}$
[by definition of $\overset{\Pi}{\models}_{val}$, (3)]

(8) $\psi \models_{alias} \mathbf{A}^-_{i.rand}$
$[\mathbf{A}^-_{i.rand} \subseteq \mathbf{A}^-_i$, Lemma 10]

(9) $(k, \psi'') \overset{\Pi}{\models}_{val} (\phi_{i.rand}, \pi_{i.rand}, \theta_{i.rand}, \mathbf{A}^+_{i.rand})$
[IH re $\overset{\Pi}{\models}_{val}$ at $i.rand$]

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

(11) $\psi'' \models_{alias} \mathbf{A}^{+}_{i.rand}$
[by (9), definition of $\overset{\Pi}{\models}_{val}$]

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

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

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

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

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

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

(18) $(x, y) \in \theta_{j.bv} \Longrightarrow x \neq [\![j.bv]\!]$
[invariance relation constraints]

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

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

(21) $(x, y) \in \theta_i \Longrightarrow x \neq [\![j.bv]\!]$
[invariance relation constraints]

(22) $(x, y) \in (\theta_{i.rator} \cap \theta_{i.rand}) \Longrightarrow (x, x) \in \theta_{i.rator}$
[invariance relation constraints]

(23) $$(x, y) \in (\theta_{i.rator} \cap \theta_{i.rand}) \Longrightarrow \begin{array}{l} x \in Dom(\psi) \cap Dom(\psi'), \\ \psi(x) = \psi'(x) \end{array}$$
[by (4), (22)]

(24) $$(x, y) \in (\theta_{i.rator} \cap \theta_{i.rand}) \Longrightarrow \begin{array}{l} x \in Dom(\psi) \cap Dom(\psi'), \\ y \in Dom(\psi') \cap Dom(\psi''), \\ \psi(x) = \psi'(x) = \psi'(y) = \psi''(y) \end{array}$$
[by (4), (10), (23)]

(25) $$(x, y) \in \theta_{j.bv} \Longrightarrow \begin{array}{l} x \in Dom(\psi) \cap Dom(\psi'), \\ y \in Dom(\psi') \cap Dom(\psi''), \\ \psi(x) = \psi'(x) = \psi'(y) = \psi''(y) \end{array}$$
[by (16), (17), (24)]

(26) $$(x, y) \in \theta_{j.bv} \Longrightarrow \begin{array}{l} x \in Dom(\psi'[[\![j.bv]\!] \mapsto (k, \psi'')]), \\ y \in Dom(\psi''), \\ \psi'[[\![j.bv]\!] \mapsto (k, \psi'')](x) = \psi''(y) \end{array}$$
[by (18), (25)]

(27) $\mathbf{A}^+_{j.bv} \subseteq \mathbf{A}^+_{i.rand}$
$[\mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}]$

(28) $\mathbf{A}^+_i \subseteq \mathbf{A}^+_{j.body}$
$[\mathcal{P}_{j.body} \leq \mathcal{P}_i]$

(29) $(k, \psi'') \models^{\Pi}_{val} (\phi_{j.bv}, \pi_{j.bv}, \theta_{j.bv}, \mathbf{A}^+_{j.bv})$
[by (9), (12), (14), (27)]

(30) $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \overset{\Pi}{\models}_{env} A_j[[\![j.bv]\!] \mapsto (\phi_{j.bv}, \pi_{j.bv}, \theta_{j.bv}, \mathbf{A}^+_{j.bv})]$
[by (6), (26), (29)]

(31) $A_{j.body} = A_j[[\![j.bv]\!] \mapsto (\phi_{j.bv}, \pi_{j.bv}, \theta_{j.bv}, \mathbf{A}^+_{j.bv})]$
[definition of consistent annotation]

(32) $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \overset{\Pi}{\models}_{env} A_{j.body}$
[(30), (31)]

(33) $\psi' \models_{alias} \mathbf{A}^+_{i.rator}$
[by (3), definition of $\overset{\Pi}{\models}_{val}$]

(34) $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \models_{alias} \mathbf{A}^+_{i.rator} - [\![j.bv]\!]$
[by (33)]

(35) $\neg\, Var(i.rand) \Longrightarrow \psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \models_{alias} \mathbf{A}^+_{i.rator} - [\![j.bv]\!]$
[trivially from (34)]

(36) $Var(i.rand) \Longrightarrow \psi([\![i.rand]\!]) = (k, \psi'')$
[by derivation that $(i, \psi) \underset{oc}{\Longrightarrow} (m, \psi''')$]

(37) $Var(i.rand) \wedge ([\![i.rand]\!], x) \in \theta_{i.rator} \Longrightarrow \psi'(x) = (k, \psi'')$
[by (4), (36)]

(38) $Var(i.rand) \wedge ([\![i.rand]\!], x) \in \theta_{i.rator} \Longrightarrow \psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \models_{alias} \{(x, [\![j.bv]\!])\}$
[by (37)]

(39) $Var(i.rand) \Longrightarrow$
$\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \models_{alias} (\mathbf{A}^+_{i.rator} - [\![j.bv]\!]) \cup \{(x, [\![j.bv]\!]) \mid ([\![i.rand]\!], x) \in \theta_{i.rator}\}$
[(34), (38), Lemma 11]

(40) $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \models_{alias} \mathbf{A}^-_{j.body}$
[local consistency of Γ, (35), (39)]

(41) $(m, \psi''') \models^{\Pi}_{val} (\phi_{j.body}, \pi_{j.body}, \theta_{j.body}, \mathbf{A}^{+}_{j.body})$
[IH at $j.body$]

(42) $(m, \psi''') \models^{\Pi}_{val} (\phi_i, \pi_i, \theta_i, \mathbf{A}^{+}_i)$
[(13), (15), (28), (41)]

(43) $$(x, y) \in \theta_{j.body} \implies \begin{array}{l} x \in Dom(\psi'[[\![j.bv]\!] \mapsto (k, \psi'')]), \\ y \in Dom(\psi'''), \\ \psi'[[\![j.bv]\!] \mapsto (k, \psi'')](x) = \psi'''(y) \end{array}$$
[IH re invariance at $j.body$]

(44) $$(x, y) \in (\theta_{j.body} - \{([\![j.bv]\!], \_)\}) \implies \begin{array}{l} x \in Dom(\psi'),\ y \in Dom(\psi'''), \\ \psi'(x) = \psi'''(y) \end{array}$$
[by (43)]

(45) $$(x, y) \in (\theta_{i.rator} \cap (\theta_{j.body} - \{([\![j.bv]\!], \_)\})) \implies \begin{array}{l} x \in Dom(\psi) \cap Dom(\psi'), \\ y \in Dom(\psi') \cap Dom(\psi'''), \\ \psi(x) = \psi'(y), \\ \psi'(x) = \psi'''(y) \end{array}$$
[by (4), (44)]

(46) $(x, y) \in (\theta_{i.rator} \cap (\theta_{j.body} - \{([\![j.bv]\!], \_)\})) \implies (x, x) \in \theta_{i.rator}$
[invariance relation constraints]

(47) $(x, y) \in (\theta_{i.rator} \cap (\theta_{j.body} - \{([\![j.bv]\!], \_)\})) \implies \psi(x) = \psi'(x)$
[by (4), (46)]

(48) $$(x, y) \in (\theta_{i.rator} \cap (\theta_{j.body} - \{([\![j.bv]\!], \_)\})) \implies \begin{array}{l} x \in Dom(\psi),\ y \in Dom(\psi'''), \\ \psi(x) = \psi'''(y) \end{array}$$
[by (45), (47)]

(49) $$(x, y) \in \theta_i \implies \begin{array}{l} y \in Dom(\psi) \cap Dom(\psi'''), \\ \psi(x) = \psi'''(y) \end{array}$$
[by (19), (20), (21), (48)]

There are three immediate subproofs. By step (1), $\psi \overset{\Pi}{\models}_{env} A_{i.rator}, A_{i.rand}$. By step (2), $\psi \models_{alias} \mathbf{A}^-_{i.rator}$, and by step (8), $\psi \models_{alias} \mathbf{A}^-_{i.rand}$. Therefore, the consequents hold for 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. By step (32), $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \overset{\Pi}{\models}_{env} A_{j.body}$, and by step (40), $\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \models_{alias} \mathbf{A}^-_{j.body}$. Hence, the consequents hold for all subproofs of the evaluation of the procedure body.

case *Cond*(i)

By the local consistency of Γ, $\mathbf{A}^-_{i.then} \subseteq \mathbf{A}^-_i$ and $\mathbf{A}^-_{i.else} \subseteq \mathbf{A}^-_i$, so by Lemma 10, $\psi \models_{alias} \mathbf{A}^-_{i.then}, \mathbf{A}^-_{i.else}$. Also by local consistency, $A_{i.then} = A_{i.else} = A_i$, so $\psi \overset{\Pi}{\models}_{env} A_{i.then}, A_{i.else}$. Depending on the result of the test, either the then-part or the else-part will be evaluated. Let (j, ψ') be the result of evaluating the then-part or the else-part. The induction hypothesis says that $\psi' \overset{\Pi}{\models}_{val} \mathcal{P}_{i.then}$, in one case, or $\psi' \overset{\Pi}{\models}_{val} \mathcal{P}_{i.then}$ in the other. (j, ψ') is also the result of evaluating the conditional itself. By the constraints $\mathcal{P}_{i.then} \leq \mathcal{P}_i$ and $\mathcal{P}_{i.else} \leq \mathcal{P}_i$, so by Lemma 13, $(j, \psi') \overset{\Pi}{\models}_{val} \mathcal{P}_i$. By the induction hypothesis at $i.then$ (or $i.else$), $\forall (x, y) \in \theta_{i.then}$ (or $\theta_{i.else}$), $x \in Dom(\psi)$, $y \in Dom(\psi')$, and $\psi(x) = \psi'(y)$. By the local consistency of Γ, $\mathcal{P}_{i.then} \leq \mathcal{P}_i$ and $\mathcal{P}_{i.else} \leq \mathcal{P}_i$, so $\theta_i \subseteq \theta_{i.then}$ and $\theta_i \subseteq \theta_{i.else}$. Therefore, $\forall (x, y) \in \theta_i$, $x \in Dom(\psi)$, $y \in Dom(\psi')$, and $\psi(x) = \psi'(y)$.

There are two immediate subproofs, one for the evaluation of the test, the other for the evaluation of either the then-part or the else-part. We showed above that $\psi \overset{\Pi}{\models}_{env} A_{i.then}, A_{i.else}$ and $\psi \models_{alias} \mathbf{A}^-_{i.then}, \mathbf{A}^-_{i.else}$. Therefore, the consequents hold for all subproofs of the evaluation of the then-part or the else-part. Since Γ is locally-consistent, $A_{i.test} = A_i$, and $\mathbf{A}^-_{i.test} \subseteq \mathbf{A}^- i$, so $\psi \overset{\Pi}{\models}_{env} A_{i.test}$ and $\psi \models_{alias} \mathbf{A}^-_{i.test}$. Hence, the consequents hold for all subproofs of the evaluation of the test. ∎

5.8 The Ultra-β transformation

We define a transformation

$$\mathcal{B}: O \to \Lambda_{thunk}$$

in Figure 18. Recall, O is the set of all occurrence indices. As usual, we extend the transformation to occurrence closures:

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

$$Var(i) \Longrightarrow \mathcal{B}(i) = [\![i]\!]$$

$$Const(i) \Longrightarrow \mathcal{B}(i) = [\![i]\!]$$

$$Abs(i) \wedge \pi_{i.body} = id \Longrightarrow \mathcal{B}(i) = \lambda[\![i.bv]\!].\mathcal{B}(i.body)$$

$$Abs(i) \wedge \pi_{i.body} = th \Longrightarrow \mathcal{B}(i) = \mathbf{thunk}\ \mathcal{B}(i.body)[x/[\![i.bv]\!]], \quad \text{where } (x, [\![i.bv]\!]) \in \mathbf{STClos}\,(\mathbf{A}^{-}_{i.body})$$

$$PrimApp(i) \Longrightarrow \mathcal{B}(i) = [\![i.rator]\!]\ \mathcal{B}(i.rand)$$

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

$$App(i) \wedge \pi_{i.rator} = th \Longrightarrow \mathcal{B}(i) = \mathbf{run}\ \mathcal{B}(i.rator)$$

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

Figure 18: The Ultra-β transformation

Because occurrence environments are finitely deep, the extended transformation is well-founded.

5.9 Correctness

Before we prove the Correctness Theorem, we show:

Lemma 14 *Let Γ be a locally-consistent annotation map. If i is the occurrence index of a value in Λ_{in}, then for any occurrence environment ψ, $\hat{\mathcal{B}}(i,\psi) \underset{t}{\Longrightarrow} \hat{\mathcal{B}}(i,\psi)$.*

Proof. Induction on the depth of ψ.

Base case $\psi = \emptyset$

Then

$$\begin{aligned} \hat{\mathcal{B}}(i,\psi) &= \mathcal{B}(i)\{\hat{\mathcal{B}} \circ \psi\} \\ &= \mathcal{B}(i) \\ &\underset{t}{\Longrightarrow} \mathcal{B}(i) \\ &= \hat{\mathcal{B}}(i,\psi) \end{aligned}$$

Why does $\mathcal{B}(i)$ self-evaluate? If $[\![i]\!]$ is a variable or constant, then the transform is the same variable or constant, which self-evaluates. If $[\![i]\!]$ is a procedure, then the transform is either a procedure or a thunk, which self-evaluates.

Induction $\psi \neq \emptyset$

case $\mathit{Var}(i) \wedge [\![i]\!] \notin \mathit{Dom}(\psi)$

Then $(i, \psi) \underset{oc}{\Longrightarrow} (i, \emptyset)$, and

$$\begin{aligned} \hat{\mathcal{B}}(i, \psi) &= \mathcal{B}(i)\{\hat{\mathcal{B}} \circ \psi\} \\ &= [\![i]\!] \\ &\underset{t}{\Longrightarrow} [\![i]\!] \\ &= \mathcal{B}(i)\{\hat{\mathcal{B}} \circ \emptyset\} \\ &= \hat{\mathcal{B}}(i, \emptyset) \end{aligned}$$

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

Then $(i, \psi) \underset{oc}{\Longrightarrow} \psi([\![i]\!])$, so

$$\begin{aligned} \hat{\mathcal{B}}(i, \psi) &= \mathcal{B}(i)\{\hat{\mathcal{B}} \circ \psi\} \\ &= [\![i]\!]\{\hat{\mathcal{B}} \circ \psi\} \\ &= \hat{\mathcal{B}}(\psi([\![i]\!])) \\ &\underset{t}{\Longrightarrow} \hat{\mathcal{B}}(\psi([\![i]\!])) \end{aligned}$$

where the evaluation step holds by the induction hypothesis.

case $\mathit{Const}(i)$

Then

$$\begin{aligned} \hat{\mathcal{B}}(i, \psi) &= \mathcal{B}(i)\{\hat{\mathcal{B}} \circ \psi\} \\ &= [\![i]\!]\{\hat{\mathcal{B}} \circ \psi\} \\ &= [\![i]\!] \\ &\underset{t}{\Longrightarrow} [\![i]\!] \end{aligned}$$

case $Abs(i)$

In this case, the transform depends upon the binder tag for the procedure body. If $\pi_{i.body} = id$, the transform is a procedure, which self-evaluates:

$$\begin{aligned}
\hat{\mathcal{B}}(i,\psi) &= \mathcal{B}(i)\{\hat{\mathcal{B}} \circ \psi\} \\
&= (\lambda[\![i.bv]\!].\mathcal{B}i.body)\{\hat{\mathcal{B}} \circ \psi\} \\
&\underset{t}{\Longrightarrow} (\lambda[\![i.bv]\!].\mathcal{B}i.body)\{\hat{\mathcal{B}} \circ \psi\}
\end{aligned}$$

If $\pi_{i.body} = th$, the transform is a substitution instance of a thunk, which also self-evaluates:

$$\begin{aligned}
\hat{\mathcal{B}}(i,\psi) &= \mathcal{B}(i)\{\hat{\mathcal{B}} \circ \psi\} \\
&= (\textbf{thunk } \mathcal{B}(i.body)[x/[\![i.bv]\!]])\{\hat{\mathcal{B}} \circ \psi\} \\
&= \textbf{thunk } \mathcal{B}(i.body)[x/[\![i.bv]\!]]\{\hat{\mathcal{B}} \circ \psi\} \\
&\underset{t}{\Longrightarrow} \textbf{thunk } \mathcal{B}(i.body)[x/[\![i.bv]\!]]\{\hat{\mathcal{B}} \circ \psi\}
\end{aligned}$$

where $(x, [\![i.bv]\!]) \in \mathbf{STClos}\,(\mathbf{A}^-_{i.body})$. We know such a pair of variables exists by the local consistency of Γ. ∎

Now we are ready for the main event.

The Correctness Theorem for Ultra-β is much like those for the closure conversion transformations. So that we can rely on the Soundness Theorem in the proof, the Correctness Theorem has the premise that the starting environment, ψ, satisfies the starting alias relation, $\mathbf{A}^-_i$.

Theorem 11 (Correctness) *Let Γ be a monovariant and locally-consistent annotation map, and let Π be the binder tag assignment defined by $\forall i, \Pi(i) = \pi_i$. Let i be an occurrence index and ψ an occurrence environment such that $\psi \overset{\Pi}{\models}_{env} A_i$, and $\psi \models_{alias} \mathbf{A}^-_i$. If $(i,\psi) \underset{oc}{\Longrightarrow} (j,\psi')$, then $\hat{\mathcal{B}}(i,\psi) \underset{t}{\Longrightarrow} \hat{\mathcal{B}}(j,\psi')$.*

The diagram is:

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

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

Base cases

case *Var*(i)

By the local consistency of Γ, $A_i([\![i]\!]) = (\phi_i, \pi_i, \theta_i)$, so $[\![i]\!] \in Dom(A_i)$. Since $\psi \models^{\Pi}_{env} A_i$, $[\![i]\!] \in Dom(\psi)$, hence $(i,\psi) \underset{oc}{\Longrightarrow} \psi([\![i]\!])$.

We have

$$\begin{array}{rcl} \hat{\mathcal{B}}(i,\psi) & = & \mathcal{B}(i)\{\hat{\mathcal{B}} \circ \psi\} \\ & = & \hat{\mathcal{B}}(\psi([\![i]\!])) \\ & \underset{t}{\Longrightarrow} & \hat{\mathcal{B}}(\psi([\![i]\!])) \end{array}$$

Since the index-part of the occurrence closure $\psi([\![i]\!])$ is the index of a value, the evaluation step holds by Lemma 14.

case *Const*(i)

So $(i,\psi) \underset{oc}{\Longrightarrow} (i,\emptyset)$, and

$$\begin{array}{rcl} \hat{\mathcal{B}}(i,\psi) & = & \mathcal{B}(i)\{\hat{\mathcal{B}} \circ \psi\} \\ & = & [\![i]\!]\{\hat{\mathcal{B}} \circ \psi\} \\ & = & [\![i]\!] \\ & \underset{t}{\Longrightarrow} & [\![i]\!] \\ & = & \mathcal{B}(i)\{\hat{\mathcal{B}} \circ \emptyset\} \\ & = & \hat{\mathcal{B}}(i,\emptyset) \end{array}$$

case *Abs*(i)

We have $(i,\psi) \underset{oc}{\Longrightarrow} (i,\psi)$. The transformation depends on the binder tag for the procedure body. We can use the same argument here as in the proof of Lemma 14, as follows.

If $\pi_{i.body} = id$, we have a procedure, which self-evaluates:

$$\begin{array}{rcl} \hat{\mathcal{B}}(i,\psi) & = & \mathcal{B}(i)\{\hat{\mathcal{B}} \circ \psi\} \\ & = & (\lambda[\![i.bv]\!].\mathcal{B}(i.body))\{\hat{\mathcal{B}} \circ \psi\} \end{array}$$

$$\underset{t}{\Longrightarrow} \quad (\lambda[\![i.bv]\!].\mathcal{B}(i.body))\{\hat{\mathcal{B}} \circ \psi\}$$

Similarly, if $\pi_{i.body} = th$, the transform is a a thunk, which self-evaluates:

$$\begin{aligned}
\hat{\mathcal{B}}(i,\psi) &= \mathcal{B}(i)\{\hat{\mathcal{B}} \circ \psi\} \\
&= (\textbf{thunk } \mathcal{B}(i.body)[x/[\![i.bv]\!]])\{\hat{\mathcal{B}} \circ \psi\} \\
&= \textbf{thunk } \mathcal{B}(i.body)[x/[\![i.bv]\!]]\{\hat{\mathcal{B}} \circ \psi\} \\
&\underset{t}{\Longrightarrow} \textbf{thunk } \mathcal{B}(i.body)[x/[\![i.bv]\!]]\{\hat{\mathcal{B}} \circ \psi\}
\end{aligned}$$

Induction

case *PrimApp*(i)

Then $(i,\psi) \underset{t}{\Longrightarrow} (c,\emptyset)$, for some constant c. There must be a subproof $(i.rand,\psi) \underset{oc}{\Longrightarrow} (j,\psi')$, where $Const(j)$, where $[\![j]\!]$ is related to c in a way that depends on the operator.

The transform is:

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

By the induction hypothesis

$$\begin{aligned}
\hat{\mathcal{B}}(i.rand,\psi) &\underset{t}{\Longrightarrow} \hat{\mathcal{B}}(j,\psi') \\
&= [\![j]\!]\{\hat{\mathcal{B}} \circ \psi'\} \\
&= [\![j]\!]
\end{aligned}$$

So by the relevant term evaluation rule

$$\begin{aligned}
\hat{\mathcal{B}}(i,\psi) &\underset{t}{\Longrightarrow} c \\
&= [\![c]\!] \\
&= [\![c]\!]\{\hat{\mathcal{B}} \circ \emptyset\} \\
&= \hat{\mathcal{B}}(c,\emptyset)
\end{aligned}$$

case $App(i)$

Suppose we have the derivation:

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

The effect of the transformation depends on the binder tag $\pi_{i.rator}$.

Consider the subcase where $\pi_{i.rator} = id$. We have

$$\begin{aligned} \hat{\mathcal{B}}(i, \psi) &= \mathcal{B}(i)\{\hat{\mathcal{B}} \circ \psi\} \\ &= (\mathcal{B}(i.rator)\ \mathcal{B}(i.rand))\{\hat{\mathcal{B}} \circ \psi\} \\ &= \hat{\mathcal{B}}(i.rator, \psi)\ \hat{\mathcal{B}}(i.rand, \psi) \end{aligned}$$

By assumption, $\psi \overset{\Pi}{\models}_{env} A_i$, and since Γ is locally-consistent, $A_{i.rator} = A_{i.rand} = A_i$, so $\psi \overset{\Pi}{\models}_{env} A_{i.rator}, A_{i.rand}$. Also by assumption, $\psi \models_{alias} \mathbf{A}_i^-$, and by the local consistency of Γ, $\mathbf{A}_{i.rator}^-, \mathbf{A}_{i.rator}^- \subseteq \mathbf{A}_i^-$. Therefore, by Lemma 10, $\psi \models_{alias} \mathbf{A}_{i.rator}^-, \mathbf{A}_{i.rator}^-$. So by the induction hypothesis at $i.rator$:

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

and by the induction hypothesis at $i.rand$:

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

To use the evaluation rule for applications, we need to know more about the transform of the operator result.

Now, by Theorem 10, $(j, \psi') \overset{\Pi}{\models}_{val} (\phi_{i.rator}, \pi_{i.rator}, \theta_{i.rator}, \mathbf{A}_{i.rator}^+)$. By the definition of $\overset{\Pi}{\models}_{val}$, $\Pi(j) = \pi_{i.rator}$, and by the construction of Π, $\pi_j = \pi_{i.rator} = id$. So

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

where x' is fresh.

We now have a procedure as the result of evaluating the operator, and another term, a value, as the result of evaluating the operand. To use the application rule in the term evaluator, we need to evaluate the substituted body of the procedure:

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

To see that the first equality holds, consider a free variable y in $\mathcal{B}(j.body)$. If $y = [\![j.bv]\!]$, then on both sides, we substitute $\hat{\mathcal{B}}(k, \psi'')$. If $y \neq [\![j.bv]\!]$, then we substitute $\hat{\mathcal{B}}(\psi'(y))$ on both sides of the equation.

By Theorem 10, we have

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

and

$$\psi'[[\![j.bv]\!] \mapsto (k, \psi'')] \models_{alias} \mathbf{A}^{-}_{j.body}$$

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

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

and by the term evaluator application rule

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

Now consider the subcase where $\pi_{i.rator} = th$. The transform is

$$\begin{array}{rcl} \hat{\mathcal{B}}(i, \psi) & = & \mathcal{B}(i)\{\hat{\mathcal{B}} \circ \psi\} \\ & = & (\mathbf{run}\ \mathcal{B}(i.rator))\{\hat{\mathcal{B}} \circ \psi\} \\ & = & \mathbf{run}\ \hat{\mathcal{B}}(i.rator, \psi) \end{array}$$

As we showed in the other subcase, $\psi \overset{\Pi}{\models}_{env} A_{i.rator}$ and $\psi \models_{alias} \mathbf{A}^{-}_{i.rator}$. Therefore, we can invoke the induction hypothesis at $i.rator$:

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

By Theorem 10, $\pi_j = \pi_{i.rator} = th$. So

$$\begin{aligned}
\hat{\mathcal{B}}(j,\psi') &= \mathcal{B}(j)\{\hat{\mathcal{B}} \circ \psi'\} \\
&= (\textbf{thunk } \mathcal{B}(j.body)[x/[\![j.bv]\!]])\{\hat{\mathcal{B}} \circ \psi'\} \\
&= \textbf{thunk } \mathcal{B}(j.body)[x/[\![j.bv]\!]]\{\hat{\mathcal{B}} \circ \psi'\}
\end{aligned}$$

where $(x, [\![j.bv]\!]) \in \mathbf{STClos}(\mathbf{A}^-_{j.body})$, and $x \neq [\![j.bv]\!]$. This term may appear unuseful in proving the theorem; but let us investigate further.

By the local consistency of Γ, since $(x, [\![j.bv]\!]) \in \mathbf{STClos}(\mathbf{A}^-_{j.body})$, it must be that $Var(i.rand)$. Also by local consistency, $A_{i.rand}([\![i.rand]\!]) = (\phi_{i.rand}, \pi_{i.rand}, \theta_{i.rand})$, so $[\![i.rand]\!] \in Dom(A_{i.rand})$. Again by local consistency, $A_{i.rand} = A_i$, and by assumption, $\psi \overset{\Pi}{\models}_{env} A_i$, so $\psi \overset{\Pi}{\models}_{env} A_{i.rand}$. Therefore, $[\![i.rand]\!] \in Dom(\psi)$, so $\psi([\![i.rand]\!]) = (k, \psi'')$.

By Theorem 10,

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

and

$$\psi'[[\![j.bv]\!] \mapsto (k,\psi'')] \models_{alias} \mathbf{A}^-_{j.body}$$

Now, by Lemma 12

$$\psi'[[\![j.bv]\!] \mapsto (k,\psi'')] \models_{alias} \mathbf{STClos}(\mathbf{A}^-_{j.body})$$

Since $(x, [\![j.bv]\!]) \in \mathbf{STClos}(\mathbf{A}^-_{j.body})$, by the definition of $\models_{alias}$,

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

So

$$\begin{aligned}
&\mathcal{B}(j.body)[x/[\![j.bv]\!]]\{\hat{\mathcal{B}} \circ \psi'\} \\
=\ &\mathcal{B}(j.body)\{\hat{\mathcal{B}} \circ \psi'[[\![j.bv]\!] \mapsto (k,\psi'')]\} \\
=\ &\hat{\mathcal{B}}(j.body, \psi'[[\![j.bv]\!] \mapsto (k,\psi'')]) \\
\underset{t}{\Longrightarrow}\ &\hat{\mathcal{B}}(m,\psi''')
\end{aligned}$$

To see that the first equality holds, consider a free variable y in $\mathcal{B}(j.body)$. If y is $[\![j.bv]\!]$, then on both sides of the equation, we substitute $\hat{\mathcal{B}}(k, \psi'')$. If y is any other variable, we substitute $\hat{\mathcal{B}}(\psi'(y))$ on both sides. The evaluation step holds by the induction hypothesis at $j.body$.

So by the term evaluation rule for running thunks

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

case $Cond(i)$

As usual, we show only one of the subcases.

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

Transforming the conditional, we get

$$\begin{aligned}
& \hat{\mathcal{B}}(i, \psi) \\
= \ & \mathcal{B}(i)\{\hat{\mathcal{B}} \circ \psi\} \\
= \ & (\textbf{if } \mathcal{B}(i.test) \textbf{ then } \mathcal{B}(i.then) \textbf{ else } \mathcal{B}(i.else))\{\hat{\mathcal{B}} \circ \psi\} \\
= \ & \textbf{if } \hat{\mathcal{B}}(i.test, \psi) \textbf{ then } \hat{\mathcal{B}}(i.then, \psi) \textbf{ else } \hat{\mathcal{B}}(i.else, \psi)
\end{aligned}$$

By assumption, $\psi \overset{\Pi}{\models}_{env} A_i$. By the local consistency of Γ, $A_{i.test} = A_{i.then} = A_i$, so $\psi \overset{\Pi}{\models}_{env} A_{i.test}, A_{i.then}$. Also by assumption, $\psi \models_{alias} \mathbf{A}_i^-$. Again by local consistency, $\mathbf{A}_{i.test}^- \subseteq \mathbf{A}_i^-$ and $\mathbf{A}_{i.then}^- \subseteq \mathbf{A}_i^-$, so $\psi \models_{alias} \mathbf{A}_{i.test}^-, \mathbf{A}_{i.then}^-$.

So by the induction hypothesis at $i.test$:

$$\begin{aligned}
\hat{\mathcal{B}}(i.test, \psi) \quad & \underset{t}{\Longrightarrow} \quad \hat{\mathcal{B}}(j, \psi') \\
& = \quad \mathcal{B}(j)\{\hat{\mathcal{B}} \circ \psi'\} \\
& = \quad \textbf{true}\{\hat{\mathcal{B}} \circ \psi'\} \\
& = \quad \textbf{true}
\end{aligned}$$

By the induction hypothesis at $i.then$:

$$\hat{\mathcal{B}}(i.then, \psi) \underset{t}{\Longrightarrow} \hat{\mathcal{B}}(k, \psi'')$$

So by the term evaluation rule for conditionals where the test is true,

$$\hat{\mathcal{B}}(i, \psi) \underset{t}{\Longrightarrow} \hat{\mathcal{B}}(k, \psi'')$$

$\blacksquare$

5.10 Conclusion

In this chapter, we have shown that the method developed for our closure conversion transformations can be applied to a much different problem, replacing copies of variables by their originals. Some components of our earlier analysis, such as flows and invariance relations, were reused in the analysis supporting the Ultra-β transformation. Other parts of the analysis are new, such as the alias relations. Neatly, the statements of the Ultra-β soundness and correctness theorems are quite similar to their counterparts in the two closure conversion analyses and transformations.

Chapter 6

Selective thunkification

In this chapter, we present a transformation that takes call-by-name programs to call-by-value equivalents. Although some changes are required to accomodate the different evaluation strategy, our flow analysis method is flexible enough to support the transformation. As the chapter title suggests, the transformation incorporates elements of the closure conversion and Ultra-β transformations.

6.1 Examples and discussion

In a call-by-name evaluation strategy, an argument to a procedure is not evaluated when the procedure is applied. The formal parameter of the procedure becomes bound to the actual argument rather than its value. When a free occurrence of the formal parameter in the procedure's body is referenced, the actual argument is evaluated.

A call-by-name program can be transformed into a call-by-value program by turning all operands at all call sites into thunks. Let **loop** be a term in Λ_{in} that cannot be evaluated under call-by-name because, as its name suggests, it just loops. We say that such a term *diverges*. Under call-by-name, the program

$$(\lambda x.c)\ \mathbf{loop}$$

evaluates to c. The operand is never evaluated, so it poses no problem. Under call-by-value, the same program will not terminate, because the attempt to evaluate the operand results in divergence.

If we transform the program to

$$(\lambda x.c)\ (\textbf{thunk loop})$$

then the transform will evaluate to c under call-by-value.

In this example, evaluation of the application did not require evaluation of the operand. In technical jargon, the application was not *strict* in the operand. Therefore, when we transform the program, turning the operand into a thunk prevents evaluation of the operand under call-by-value.

If a particular operand in a program is certain to be evaluated and divergence does not occur, thunkification may slow evaluation considerably. Consider:

$$(\lambda x.\ \ldots\ x\ \ldots\ x\ \ldots\ x\ \ldots)\ M$$

Assume the program is strict in the operand M. Therefore, if M diverges, the entire program diverges. If we enclose the operand in a thunk, the transform is:

$$(\lambda x.\ \ldots\ (\textbf{run}\ x)\ \ldots\ (\textbf{run}\ x)\ \ldots\ (\textbf{run}\ x)\ \ldots)\ (\textbf{thunk}\ M)$$

We may have to run the thunk three times. It is better to evaluate M just once, so that x becomes bound to M's value. Therefore, it is desirable to avoid thunkification when possible.

In simple examples, we may readily see whether an argument is certain to be evaluated or not. In general, this is an undecidable question. Therefore, our analysis will include an approximate strictness analysis that will determine that some arguments in a given program are certain to be evaluated; all other arguments may or may not be evaluated. In the transformation, all arguments in the latter classification are enclosed in thunks. Therefore, if the call-by-name program does not diverge, its call-by-value transform will not diverge. Since only some operands are thunkified, the transformation is *selective*.

As in the other transformations, the current transformation raises issues of protocol agreement. When a variable bound to a thunkified term is evaluated, the thunked term is evaluated. If instead the variable is bound to an ordinary value, the value must not be treated as a thunk. Since we do not want to make that decision at run-time, the dataflow analysis assures that a given variable is bound only to thunks or only to values.

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

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

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

$$\frac{M \underset{tn}{\Longrightarrow} \lambda x.Q \quad Q[N/x] \underset{tn}{\Longrightarrow} V}{MN \underset{tn}{\Longrightarrow} V}$$

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

Figure 19: Rules for the call-by-name term evaluator

6.2 Languages and evaluators

Our input language will be Λ_{in}, even though we will be evaluating it call-by-name. Our output language is Λ_{thunk}, which we introduced in Chapter 5.

The call-by-name term evaluator is shown in Figure 19. We use the symbol $\underset{tn}{\Longrightarrow}$ for the relation on terms and their values.

In order to create an occurrence evaluator that simulates the operations of the call-by-name term evaluator, we need to modify some definitions:

Definition 8 *The following definitions are mutually referential.*

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

2. *An* occurrence environment ψ *is a finite map from variables to occurrence closures.*

Unlike in our previous definition in Chapter 2, if (j, ψ') is in the range of an occurrence environment ψ, then j may be the occurrence index of any term in Λ_{in}, not necessarily a value.

As before, we can retrieve the term represented by an occurrence closure by applying an unwinding function:

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

We write $(i,\psi) \underset{ocn}{\Longrightarrow} (j,\psi')$ to indicate that (i,ψ) evaluates to (j,ψ'). We present the occurrence evaluator in Figure 20. As we proved for the call-by-value evaluators, the call-by-name occurrence closure evaluator simulates the operation of the call-by-name term evaluator on unwound occurrence closures.

Theorem 12 (Simulation) *If* $(i,\psi) \underset{ocn}{\Longrightarrow} (j,\psi')$, *then* $\mathcal{U}[\![(i,\psi)]\!] \underset{tn}{\Longrightarrow} \mathcal{U}[\![(j,\psi')]\!]$

The diagram is:

$$\begin{tikzcd}
(i,\psi) \arrow[r, Rightarrow, "ocn"'] \arrow[d, "\mathcal{U}[\![\cdot]\!]"'] & (j,\psi') \arrow[d, "\mathcal{U}[\![\cdot]\!]"] \\
\mathcal{U}[\![(i,\psi)]\!] \arrow[r, Rightarrow, "tn"'] & \mathcal{U}[\![(j,\psi')]\!]
\end{tikzcd}$$

Proof. Induction on the definition of $\underset{ocn}{\Longrightarrow}$.

Each of the evaluation rules in the call-by-name occurrence evaluator has a corresponding rule in the call-by-value occurrence evaluator. The same can be said of the rules in the call-by-name and call-by-value term evaluators. If we substitute $\underset{ocn}{\Longrightarrow}$ for $\underset{oc}{\Longrightarrow}$, and $\underset{tn}{\Longrightarrow}$ for $\underset{t}{\Longrightarrow}$ in the proof of Theorem 1, p. 24, the Simulation Theorem for call-by-value, we almost have a proof of the current theorem. We have to prove anew the case for ordinary applications, exactly the case that distinguishes a call-by-name from a call-by-value evaluation strategy.

case $App(i)$

Suppose:

$$\frac{(i.rator,\psi) \underset{ocn}{\Longrightarrow} (j,\psi'), \quad Abs(j) \qquad (j.body, \psi'[[\![j.bv]\!] \mapsto (i.rand,\psi)]) \underset{ocn}{\Longrightarrow} (k,\psi'')}{(i,\psi) \underset{ocn}{\Longrightarrow} (k,\psi'')}$$

By the induction hypothesis at $i.rator$

$$\mathcal{U}[\![(i.rator,\psi)]\!] \underset{tn}{\Longrightarrow} \mathcal{U}[\![(j,\psi')]\!]$$

$$\frac{Var(i) \wedge [\![i]\!] \notin Dom(\psi)}{(i,\psi) \underset{ocn}{\Longrightarrow} (i,\emptyset)} \qquad \frac{\begin{array}{l} Var(i) \wedge [\![i]\!] \in Dom(\psi) \\ \psi([\![i]\!]) \underset{ocn}{\Longrightarrow} (j,\psi') \end{array}}{(i,\psi) \underset{ocn}{\Longrightarrow} (j,\psi')}$$

$$\frac{Const(i)}{(i,\psi) \underset{ocn}{\Longrightarrow} (i,\emptyset)} \qquad \frac{Abs(i)}{(i,\psi) \underset{ocn}{\Longrightarrow} (i,\psi)}$$

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

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

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

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

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

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

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

Figure 20: Rules for the call-by-name occurrence evaluator

Now

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

where x' is fresh.

We claim:

$$\begin{aligned}
& [\![j.body]\!][x'/[\![j.bv]\!]]\{\mathcal{U}[\![\cdot]\!]\circ\psi'\}[\mathcal{U}[\![(i.rand,\psi)]\!]/x']\\
= \;& [\![j.body]\!]\{\mathcal{U}[\![\cdot]\!]\circ\psi'[[\![j.bv]\!]\mapsto(i.rand,\psi)]\}\\
= \;& \mathcal{U}[\![(j.body,\psi'[[\![j.bv]\!]\mapsto(i.rand,\psi)])]\!]\\
\underset{tn}{\Longrightarrow}\;& \mathcal{U}[\![(k,\psi'')]\!]
\end{aligned}$$

For the first equality, we use a now-familiar argument. For $[\![j.bv]\!]$ free in $[\![j.body]\!]$, we substitute $\mathcal{U}[\![(i.rand,\psi)]\!]$ on both sides of the equation. For any other variable y, we substitute $\mathcal{U}[\![(\psi(y))]\!]$ on both sides. The next equality holds by the definition of unwinding. The evaluation holds by the induction hypothesis at $j.body$.

So by the call-by-name term application rule, $\mathcal{U}[\![(i,\psi)]\!] \underset{tn}{\Longrightarrow} \mathcal{U}[\![(k,\psi'')]\!]$. ∎

For the same reasons as in the call-by-value case, the converse of the Simulation Theorem fails. We can prove a comparable partial converse, but the following lemma allows us to drop the restriction to chain-free environments:

Lemma 15 *If* $(i,\psi) \underset{ocn}{\Longrightarrow} (j,\psi')$ *and* $Var(j)$*, then* $\psi' = \emptyset$.

Proof. Easy induction on the definition of $\underset{ocn}{\Longrightarrow}$. ∎

For call-by-name, we have:

Theorem 13 (Adequacy) *If* $\mathcal{U}[\![(i,\psi)]\!] \underset{tn}{\Longrightarrow} M$ *then* $\exists j,\psi'$ *such that* $(i,\psi) \underset{ocn}{\Longrightarrow} (j,\psi')$ *and* $M = \mathcal{U}[\![(j,\psi')]\!]$.

Proof. Induction on the size of the derivation that $\mathcal{U}[\![(i,\psi)]\!] \underset{tn}{\Longrightarrow} M$ and the depth of ψ.

The halting measure for the induction is given by $\langle s, d\rangle$, where s is the size of a derivation, and d is the depth of an occurrence environment. We define $\langle s, d\rangle < \langle s', d'\rangle$ iff $s < s'$ or $(s = s' \wedge d < d')$.

Again, we can bootstrap our proof for the call-by-value case (Theorem 2, p. 30). Substitute $\underset{ocn}{\Longrightarrow}$ for $\underset{oc}{\Longrightarrow}$, and $\underset{tn}{\Longrightarrow}$ for $\underset{t}{\Longrightarrow}$.

Use the following induction-step cases as replacements:

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

We have:

$$\begin{array}{rcl} \mathcal{U}[\![(i, \psi)]\!] & = & [\![i]\!]\{\mathcal{U}[\![\cdot]\!] \circ \psi\} \\ & = & \mathcal{U}[\![\psi([\![i]\!])]\!] \\ & \underset{tn}{\Longrightarrow} & M \end{array}$$

where the evaluation holds by assumption. Since $\mathcal{U}[\![(i, \psi)]\!] = \mathcal{U}[\![\psi([\![i]\!])]\!]$, the sizes of their evaluations are identical. But $\psi([\![i]\!])$ is contained in ψ, so by the induction hypothesis, there are j, ψ' such that $\psi([\![i]\!]) \underset{ocn}{\Longrightarrow} (j, \psi')$, where $\mathcal{U}[\![(j, \psi')]\!] = M$. Then by the occurrence evaluator rule for variables in the domain of their closing environment, $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$.

case $PrimApp(i)$

Then

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

and

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

where N is a constant, and the relation between M and N depends on the operator.

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

We know N is a constant and $[\![j]\!]$ is a value. $[\![j]\!]$ cannot be a procedure, else $\mathcal{U}[\![(j, \psi')]\!]$ would also be a procedure. If $Var(j)$, then by Lemma 15, $\psi' = \emptyset$, so $\mathcal{U}[\![(j, \psi')]\!] = [\![j]\!]$, a variable, which is a contradiction. Therefore, $[\![j]\!]$ must be the constant N. By the appropriate occurrence evaluator rule, $(i, \psi) \underset{ocn}{\Longrightarrow} (c, \emptyset)$, for some constant c. As desired,

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

because the same primitive operator in the occurrence and term evaluators has the same effect.

case $App(i)$

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 derivation must be

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

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

We know $[\![j]\!]$ is a value. It cannot be a constant, else the unwind $\mathcal{U}[\![(j,\psi')]\!]$ is the same constant. If we assume $Var(j)$ is a variable, then by Lemma 15, $\psi' = \emptyset$, and the unwind would be the variable $[\![j]\!]$, a contradiction. Therefore, $[\![j]\!]$ must a procedure.

So

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

where x' is fresh.

Observe that:

$$Q[\mathcal{U}[\![(i.rand,\psi)]\!]/x] \quad = \quad [\![j.body]\!][x'/[\![j.bv]\!]]\{\mathcal{U}[\![\cdot]\!] \circ \psi\}[\mathcal{U}[\![(i.rand,\psi)]\!]/x']$$

$$= [\![j.body]\!]\{\mathcal{U}[\![\cdot]\!] \circ \psi'[[\![j.bv]\!] \mapsto (i.rand, \psi)]\}$$
$$= \mathcal{U}[\![(j.body, \psi'[[\![j.bv]\!] \mapsto (i.rand, \psi)])]\!]$$

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 $\mathcal{U}[\![(i.rand, \psi)]\!]$ on both sides. The last line holds by the definition of unwinding.

Now we can invoke the induction hypothesis and claim there are k, ψ'' such that

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

where $\mathcal{U}[\![(k, \psi'')]\!] = V$. Further, by the occurrence evaluator rule for applications, $(i, \psi) \underset{ocn}{\Longrightarrow} (k, \psi'')$. ∎

6.3 The semantics of annotations

We need to add some definitions describing the language of annotations, and revise other definitions.

Definition 9 *A* thunk tag π *is an element of* $\{id, th\}$.

A thunk tag of th means that an associated occurrence of in operand position should become a thunk under the transformation.

As expected, we have:

Definition 10 *A* thunk tag assignment Π *is a map from occurrences to thunk tags.*

A value proposition $\mathcal{P}$ is now a pair (ϕ, π), where, as usual, ϕ is a flow. Value propositions are ordered as follows: $(\phi, \pi) \leq (\phi', \pi')$ iff $\phi \subseteq \phi'$ and $\pi = \pi'$.

As before, an environment proposition A is a finite map from variables to value propositions.

Occurrences now have another associated proposition:

Definition 11 *A* strictness proposition $\mathcal{S}$ *is a set of variables.*

Informally, a strictness proposition is the set of free variables of an occurrence certain to be evaluated when the occurrence is evaluated.

The semantics of annotations for the selective thunkification analysis differs from previous semantics in two significant ways. First, occurrence environments may contain arbitrary occurrence closures, not just values, yet we want to be able to say when such occurrence closures satisfy value propositions. Second, we have an entirely new satisfaction relation on occurrence evaluations and strictness propositions. We define:

1. An occurrence environment ψ satisfies an environment proposition A under a thunk tag assignment Π, as follows:

$$\psi \models^{\Pi}_{env} A$$

   iff $\forall x \in Dom(A)$

   (a) $x \in Dom(\psi)$ and $\psi(x) \models^{\Pi}_{thunk} A(x)$, and

   (b) if $A(x) = (\phi, \pi)$ and $\psi(x) = (i, \psi')$, then $\Pi(i) = \pi$

2. An occurrence closure (i, ψ) satisfies a value proposition (ϕ, π) under a thunk tag assignment Π, as follows:

$$(i, \psi) \models^{\Pi}_{thunk} (\phi, \pi)$$

   iff

   (a) $(i, \psi) \models^{\Pi}_{val} (\phi, \pi)$ or

   (b) $(i, \psi) \models^{\Pi}_{ocn} (\phi, \pi)$

3. Define a relation:

$$(i, \psi) \models^{\Pi}_{val} (\phi, \pi)$$

   iff

   (a) $Var(i) \wedge [\![i]\!] \notin Dom(\psi) \wedge \Pi(i) = \pi$, or

   (b) $Const(i)$, or

   (c) $Abs(i)$, and $\exists A$ such that $(i, A) \in \phi$ and $\psi \models^{\Pi}_{env} A$

4. Define a relation:

$$(i, \psi) \overset{\Pi}{\models}_{ocn} (\phi, \pi)$$

iff

(a) $Var(i) \wedge [\![i]\!] \in Dom(\psi)$, or $PrimApp(i)$, or $App(i)$, or $Cond(i)$, and

(b) if $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$ then $(j, \psi') \overset{\Pi}{\models}_{val} (\phi, \pi)$

5. An evaluation $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$ satisfies a strictness proposition $\mathcal{S}$, as follows:

$$(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi') \models_{strict} \mathcal{S}$$

iff $\forall x \in \mathcal{S}$, $\exists k, m, \psi'', \psi'''$ such that

(a) k is a free occurrence of x in $[\![i]\!]$,

(b) $(k, \psi'') \underset{ocn}{\Longrightarrow} (m, \psi''')$ is a subproof of $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$, and

(c) if $x \in Dom(\psi)$, then $x \in Dom(\psi'')$ and $\psi(x) = \psi''(x)$

These definitions are well-founded, since occurrence closures are finite.

For the $\overset{\Pi}{\models}_{env}$ relation on occurrence environments and environment propositions, we require pointwise $\overset{\Pi}{\models}_{thunk}$-satisfaction, and that the environment proposition gives the right thunk tags for environment lookups.

For the $\overset{\Pi}{\models}_{thunk}$ relation on occurrence closures and value propositions, we have to concern ourselves whether the occurrence closure is a value or not.[1] For values, we delegate to the $\overset{\Pi}{\models}_{val}$ relation; for other occurrence closures, we delegate to the $\overset{\Pi}{\models}_{ocn}$ relation, which itself depends on $\overset{\Pi}{\models}_{val}$.

The definition of $\models_{strict}$ makes precise the notion that a strictness proposition is a set of variables certain to be referenced during the evaluation of an occurrence.

The current ordering on value propositions gives us the following results:

Lemma 16 *If* $(i, \psi) \overset{\Pi}{\models}_{val} \mathcal{P}$ *and* $\mathcal{P} \leq \mathcal{P}'$ *then* $(i, \psi) \overset{\Pi}{\models}_{val} \mathcal{P}'$.

Proof. Immediate from the definition of $\overset{\Pi}{\models}_{val}$. ∎

[1] More accurately, the concern is whether the occurrence closure may appear on the right-hand side of $\underset{ocn}{\Longrightarrow}$. For instance, if $Var(i)$, and $[\![i]\!] \in Dom(\psi)$, then by Lemma 15, (i, ψ) cannot appear on the right-hand side of $\underset{ocn}{\Longrightarrow}$. So even though $[\![i]\!]$ is a value, it cannot be that (i, ψ) satisfies a value proposition via $\overset{\Pi}{\models}_{val}$. Our informal use of "values" for occurrence closures whose index-part is a value, which was accurate under call-by-value, has become slightly misleading.

Lemma 17 *If* $(i, \psi) \models^{\Pi}_{thunk} \mathcal{P}$ *and* $\mathcal{P} \leq \mathcal{P}'$ *then* $(i, \psi) \models^{\Pi}_{thunk} \mathcal{P}'$.

Proof. By the definition of $\models^{\Pi}_{thunk}$.

If $(i, \psi) \models^{\Pi}_{thunk} \mathcal{P}$, then either

1. $(i, \psi) \models^{\Pi}_{val} \mathcal{P}$, and the result holds by the preceding Lemma 16, or

2. $(i, \psi) \models^{\Pi}_{ocn} \mathcal{P}$, and one of the following holds:

   - $\exists j, \psi'$ such that $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi') \models^{\Pi}_{val} \mathcal{P}$, so by Lemma 16, $(j, \psi') \models^{\Pi}_{val} \mathcal{P}'$, so $(i, \psi') \models^{\Pi}_{ocn} \mathcal{P}'$, hence $(i, \psi) \models^{\Pi}_{thunk} \mathcal{P}'$, or

   - (i, ψ) cannot be evaluated, so trivially $(i, \psi) \models^{\Pi}_{ocn} \mathcal{P}'$, hence $(i, \psi) \models^{\Pi}_{thunk} \mathcal{P}'$.

∎

Lemma 18 *If* $\psi \models^{\Pi}_{env} A$ *and* $\forall x \in Dom(A')$, $A(x) \leq A'(x)$, *then* $\psi \models^{\Pi}_{env} A'$.

Proof. By the preceding Lemma 17 and the definition of $\models^{\Pi}_{env}$. ∎

From the definition of $\models_{strict}$, we have:

Lemma 19 *If* $x \in \mathcal{S}$, $x \in Dom(\psi)$, *and* $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi') \models_{strict} \mathcal{S}$, *then* $\exists k, m, \psi'', \psi'''$ *such that*

1. *k is a free occurrence of x in $[\![i]\!]$,*

2. *$(k, \psi'') \underset{ocn}{\Longrightarrow} (m, \psi''')$ is a subproof of $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$, and*

3. *that subproof has an immediate subproof $\psi(x) \underset{ocn}{\Longrightarrow} (m, \psi''')$*

Proof. By the definitions of $\models_{strict}$ and $\underset{ocn}{\Longrightarrow}$.

Consequents (1) and (2) are stated in the definition of $\models_{strict}$. Since $x \in Dom(\psi)$, by that definition, $x \in Dom(\psi'')$ and $\psi(x) = \psi''(x)$. Since $Var(k)$, consequent (3) follows from the appropriate variable evaluation rule in Figure 20. ∎

$$Var(i) \Longrightarrow \begin{cases} A_i(\llbracket i \rrbracket) = \mathcal{P}_i, \text{ and} \\ \mathcal{S}_i \subseteq \{\llbracket i \rrbracket\} \end{cases}$$

$$Const(i) \Longrightarrow \mathcal{S}_i = \emptyset$$

$$Abs(i) \Longrightarrow \begin{cases} A_{i.body} = A_i[\llbracket i.bv \rrbracket \mapsto \mathcal{P}_{i.bv}], \\ \{(i, A_i)\} \subseteq \phi_i, \\ \pi_i = th \Longrightarrow \forall \text{ free variable occurrences } j \text{ in } \llbracket i \rrbracket,\ \pi_j = th, \text{ and} \\ \mathcal{S}_i = \emptyset \end{cases}$$

$$PrimApp(i) \Longrightarrow \begin{cases} A_{i.rand} = A_i, \\ \pi_i = th \Longrightarrow \forall \text{ free variable occurrences } j \text{ in } \llbracket i \rrbracket,\ \pi_j = th, \text{ and} \\ \mathcal{S}_i \subseteq \mathcal{S}_{i.rand} \end{cases}$$

$$App(i) \Longrightarrow \begin{cases} A_{i.rator} = A_{i.rand} = A_i, \\ \pi_i = th \Longrightarrow \forall \text{ free variable occurrences } j \text{ in } \llbracket i \rrbracket,\ \pi_j = th, \text{ and} \\ \forall (j, B) \in \phi_{i.rator} \\ \quad \begin{cases} Abs(j), \\ \mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}, \\ \mathcal{P}_{j.body} \leq \mathcal{P}_i, \text{ and} \end{cases} \\ \text{if } \forall (j, B) \in \phi_{i.rator}, \llbracket j.bv \rrbracket \in \mathcal{S}_{j.body} \\ \text{then } \mathcal{S}_i \subseteq \mathcal{S}_{i.rator} \cup \mathcal{S}_{i.rand} \\ \text{else } \begin{cases} \mathcal{S}_i \subseteq \mathcal{S}_{i.rator} \text{ and} \\ \pi_{i.rand} = th \end{cases} \end{cases}$$

$$Cond(i) \Longrightarrow \begin{cases} A_{i.test} = A_{i.then} = A_{i.else} = A_i, \\ \mathcal{P}_{i.then} \leq \mathcal{P}_i, \\ \mathcal{P}_{i.else} \leq \mathcal{P}_i, \\ \pi_i = th \Longrightarrow \forall \text{ free variable occurrences } j \text{ in } \llbracket i \rrbracket,\ \pi_j = th, \text{ and} \\ \mathcal{S}_i \subseteq \mathcal{S}_{i.test} \cup (\mathcal{S}_{i.then} \cap \mathcal{S}_{i.else}) \end{cases}$$

Figure 21: Local consistency conditions for selective thunkification

6.4 Local consistency conditions

In Figure 21, we give the local consistency conditions that support the selective thunkification transformation. An annotation map Γ is now a map from occurrence indices to triples consisting of an environment proposition, a strictness proposition, and a value proposition. We write $\Gamma(i) = (A_i, \mathcal{S}_i, \mathcal{P}_i)$ for the value of Γ at i. We retain the notions of local consistency and monovariance for annotation maps.

To solve the constraints, begin by setting the strictness proposition for each occurrence to the set of free variables in that occurrence. Next, solve the flows as we have done for all the earlier analyses, applying the constraints on strictness propositions as new elements are added to the flows. Observe that we can disregard strictness information and set all thunk tags to th. Otherwise, partition occurrences into equivalence classes, as usual, and tentatively set all thunk tags to id. For all occurrences i such that $App(i)$, and for all occurrences j such that j is in the flow for $i.rator$, test the condition

$$[\![j.bv]\!] \in \mathcal{S}_{j.body}$$

If the test fails for any occurrence in a given operator flow, then for $i.rand$ and all members of its equivalence class, set their thunk tags to th. Finally, for each occurrence i tagged th, merge the equivalence classes for i and the free variable occurrences in $[\![i]\!]$. All members of the resulting equivalence class receive a tag of th. Since additional occurrences may now have a thunk tag of th, this merger and retagging step may need to be repeated. This step is repeated until no more occurrences are retagged.

6.5 Soundness

The Soundness Theorem requires another definition:

Definition 12 *Let Γ be an annotation map, and let Π be a thunk tag assignment. An occurrence closure (i, ψ) is* well-thunked *with respect to Γ and Π iff*

1. *$Var(i) \wedge [\![i]\!] \notin Dom(\psi)$, or $Const(i)$, or*
2. *$\psi \models^{\Pi}_{env} A_i$, and $\forall x \in FV([\![i]\!]) \cap Dom(\psi)$, $\psi(x)$ is well-thunked with respect to Γ and Π*

Since occurrence environments are finitely deep, this definition is well-founded. Any occurrence closure containing the empty occurrence environment is well-thunked, and so serves as the base case. Where the context assumes a particular annotation map and thunk tag assignment, we will just say that an occurrence closure is well-thunked.

Lemma 20 *Let Γ be a locally-consistent annotation map, and let Π be the thunk tag assignment defined by $\forall i,\ \Pi(i) = \pi_i$. Suppose (i, ψ) is well-thunked with respect to Γ and Π. Then:*

1. *If $PrimApp(i)$, then $(i.rand, \psi)$ is well-thunked*

2. *If $App(i)$, then $(i.rator, \psi)$ and $(i.rand, \psi)$ are well-thunked*

3. *If $Cond(i)$, then $(i.test, \psi)$, $(i.then, \psi)$, and $(i.else, \psi)$ are well-thunked*

Proof. By the definition of well-thunkedness, and the local consistency of Γ.

If $PrimApp(i)$, then by the local consistency of Γ, $A_{i.rand} = A_i$. If $App(i)$, then by local consistency $A_{i.rator} = A_{i.rand} = A_i$. Similarly, if $Cond(i)$, then $A_{i.test} = A_{i.then} = A_{i.else} = A_i$. Also, the free variables of the immediate subterms of a term $[\![i]\!]$, where $PrimApp(i)$, or $App(i)$, or $Cond(i)$, are a subset of the free variables of $[\![i]\!]$. The result holds by the definition of well-thunkedness. ∎

Recall the definition of a scalar environment, given in Chapter 2, p. 18. Observe:

Lemma 21 *Let Γ be an annotation map, let Π be a thunk tag assignment, and let i be an occurrence index. If ψ is a scalar environment, and $\psi \overset{\Pi}{\models}_{env} A_i$, then (i, ψ) is well-thunked with respect to Γ.*

Proof. Immediate from the definition of well-thunkedness. ∎

In the Soundness Theorem, well-thunkedness is a premise in place of $\psi \overset{\Pi}{\models}_{env} A_i$, and the satisfaction of a strictness proposition appears in a consequent. Therefore, the statement and proof of the Soundness Theorem are somewhat different from its previous counterparts.

Theorem 14 (Soundness) *Let Γ be a monovariant and locally-consistent annotation map, let Π be the thunk tag assignment defined by $\forall i,\ \Pi(i) = \pi_i$, and let (i, ψ) be an occurrence closure that is well-thunked with respect to Γ and Π. Suppose $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$. Then for any subproof $(k, \psi'') \underset{ocn}{\Longrightarrow} (m, \psi''')$ of this derivation:*

1. *(k, ψ'') is well-thunked,*

2. *(m, ψ''') is well-thunked,*

3. *$(m, \psi''') \overset{\Pi}{\models}_{val} (\phi_k, \pi_k)$, and*

4. *$(k, \psi'') \underset{ocn}{\Longrightarrow} (m, \psi''') \models_{strict} \mathcal{S}_k$*

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

Base cases

For the base cases, there are no proper subproofs, so we show that the consequents hold only at the root of the derivation tree. To assist the reader, in each case below, a bracketed number indicates which consequent is supported by a statement.

For each base case, we have (i, ψ) is well-thunked by assumption (consequent [*1*]).

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

Then $(i, \psi) \underset{ocn}{\Longrightarrow} (i, \emptyset)$. Trivially, $\pi_i = \pi_i$, so by the definition of $\overset{\Pi}{\models}_{val}$, $(i, \emptyset) \overset{\Pi}{\models}_{val} (\phi_i, \pi_i)$ [*3*]. By Definition 12, $(i, \emptyset)$ is well-thunked [*2*].

$\mathcal{S}_i$ may be the empty set, in which case it is satisfied trivially by the evaluation. Otherwise, $\mathcal{S}_i = \{i\}$, i is a free occurrence of $[\![i]\!]$ in $[\![i]\!]$, and $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$ is a subproof of itself. By assumption, $[\![i]\!] \notin Dom(\psi)$ [*4*].

case $Const(i)$

Then $(i, \psi) \underset{ocn}{\Longrightarrow} (i, \emptyset)$.

Since i is the index of a constant, $(i, \emptyset) \overset{\Pi}{\models}_{val} (\phi_i, \pi_i)$ [*3*]. By Definition 12, $(i, \emptyset)$ is well-thunked [*2*].

By the local consistency conditions, $\mathcal{S}_i = \emptyset$, so any evaluation satisfies the strictness proposition [*4*].

case $Abs(i)$

We have $(i, \psi) \underset{ocn}{\Longrightarrow} (i, \psi)$.

By the local consistency of Γ, $(i, A_i) \in \phi_i$. Since $Abs(i)$ and (i, ψ) is well-thunked, by the definition of well-thunkedness, $\psi \overset{\Pi}{\models}_{env} A_i$, so by the definition of $\overset{\Pi}{\models}_{val}$, $(i, \psi) \overset{\Pi}{\models}_{val} (\phi_i, \pi_i)$ [*3*]. By assumption, (i, ψ) is well-thunked [*2*].

$\mathcal{S}_i = \emptyset$, so any evaluation satisfies $\mathcal{S}_i$ [*4*].

Induction

For each case in the induction step, we show that all consequents hold at the root of the derivation tree, and that the premise about well-thunkedness holds for all immediate subproofs, allowing us to use the induction hypothesis.

For each case, (i, ψ) is well-thunked by assumption (consequent [*1*]).

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

Suppose we have the derivation $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$. By the evaluation rules, there is an immediate subproof $\psi([\![i]\!]) \underset{ocn}{\Longrightarrow} (j, \psi')$. Let $\psi([\![i]\!]) = (k, \psi'')$. Since $Var(i)$ and $[\![i]\!] \in Dom(\psi)$, by Definition 12, (k, ψ'') is well-thunked. Therefore, by the induction hypothesis, (j, ψ') is well-thunked [*2*], and $(j, \psi') \overset{\Pi}{\models}_{val} (\phi_k, \pi_k)$.

We want to show that $(j, \psi') \overset{\Pi}{\models}_{val} (\phi_i, \pi_i)$. Since $Var(i) \wedge [\![i]\!] \in Dom(\psi)$ and (i, ψ) is well-thunked, by the definition of well-thunkedness, $\psi \overset{\Pi}{\models}_{env} A_i$. By the local consistency of Γ, $A_i([\![i]\!]) = (\phi_i, \pi_i)$. Therefore, $(k, \psi'') \overset{\Pi}{\models}_{thunk} (\phi_i, \pi_i)$ and $\pi_k = \pi_i$.

We enumerate the possibilities for (k, ψ''), showing that its evaluation result satisfies (ϕ_i, π_i) via $\overset{\Pi}{\models}_{val}$:

1. If $Var(k) \wedge [\![k]\!] \notin Dom(\psi'')$, then $(k, \psi'') \underset{ocn}{\Longrightarrow} (k, \emptyset)$. Since $\pi_k = \pi_i$, $(k, \emptyset) \overset{\Pi}{\models}_{val} (\phi_i, \pi_i)$.

2. If $Const(k)$, then $(k, \psi'') \underset{ocn}{\Longrightarrow} (k, \emptyset)$, so $(k, \emptyset) \overset{\Pi}{\models}_{val} (\phi_i, \pi_i)$.

3. If $Abs(k)$, then (k, ψ'') self-evaluates, and we already showed $(k, \psi'') \overset{\Pi}{\models}_{thunk} (\phi_i, \pi_i)$. So either $(k, \psi'') \overset{\Pi}{\models}_{val} (\phi_i, \pi_i)$ or $(k, \psi'') \overset{\Pi}{\models}_{ocn} (\phi_i, \pi_i)$. Since k is the index of a procedure, only the first alternative is possible.

4. Otherwise, $Var(k) \wedge [\![k]\!] \in Dom(\psi'')$, or $PrimApp(i)$, or $App(k)$, or $Cond(k)$. We know that $(k, \psi'') \overset{\Pi}{\models}_{thunk} (\phi_i, \pi_i)$. So either $(k, \psi'') \overset{\Pi}{\models}_{val} (\phi_i, \pi_i)$ or $(k, \psi'') \overset{\Pi}{\models}_{ocn} (\phi_i, \pi_i)$. But only the second of these is possible. Since $(k, \psi'') \underset{ocn}{\Longrightarrow} (j, \psi')$, it must be that $(j, \psi') \overset{\Pi}{\models}_{val} (\phi_i, \pi_i)$.

These are all the possibilities for (k, ψ''), establishing consequent [*3*].

We can use the same argument here as in the other variable case to show that the evaluation satisfies the strictness proposition $\mathcal{S}_i$ [*4*]. In this case, by assumption, $[\![i]\!] \in Dom(\psi)$.

There is one immediate subproof, as we have shown above. As we showed, $\psi([\![i]\!]) = (k, \psi'')$ is well-thunked. Therefore, by the induction hypothesis, the consequents hold for all proper subproofs.

case *PrimApp*(i)

By the evaluation rules, for any operator, $(i, \psi) \underset{ocn}{\Longrightarrow} (c, \emptyset)$, for some constant c. So by the definition of $\overset{\Pi}{\models}_{val}$, $(c, \emptyset) \overset{\Pi}{\models}_{val} (\phi_i, \pi_i)$ [*3*]. By Definition 12, $(c, \emptyset)$ is well-thunked [*2*].

By assumption, (i, ψ) is well-thunked, so by Lemma 20, $(i.rand, \psi)$ is well-thunked. The evaluation proof has a subproof $(i.rand, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$ that, by the induction hypothesis at $i.rand$, satisfies the strictness proposition $\mathcal{S}_{i.rand}$. By the local consistency conditions, $\mathcal{S}_i \subseteq \mathcal{S}_{i.rand}$. Any free occurrence of a variable in the operand is also a free occurrence in the application, and by the evaluation rules, any subproof of the evaluation of the operand is a subproof of the evaluation of the application itself [*4*].

By the evaluation rules, there is one immediate subproof for the evaluation of $(i.rand, \psi)$. As we mentioned, $(i.rand, \psi)$ is well-thunked. Therefore, by the induction hypothesis at $i.rand$, the consequents hold for all proper subproofs.

case *App*(i)

Suppose we have the derivation:

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

Then:

(1) $(i.rator, \psi)$ is well-thunked
[(i, ψ) is well-thunked, Lemma 20]

(2) $(i.rand, \psi)$ is well-thunked
[(i, ψ) is well-thunked, Lemma 20]

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

(4) $(j, A_j) \in \phi_{i.rator}$
[(3), definition of $\overset{\Pi}{\models}_{val}$, monovariance]

(5) (j, ψ') is well-thunked
[IH re well-thunkedness at $i.rator$]

(6) $\psi' \overset{\Pi}{\models}_{env} A_j$
[(3), (4), definition of $\overset{\Pi}{\models}_{val}$]

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

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

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

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

(11) $(i.rand, \psi) \overset{\Pi}{\models}_{thunk} (\phi_{i.rand}, \pi_{i.rand})$
[see below]

(12) $(i.rand, \psi) \overset{\Pi}{\models}_{thunk} (\phi_{j.bv}, \pi_{j.bv})$
[by (7), (8), (11), Lemma 17]

(13) $\psi'[\llbracket j.bv \rrbracket \mapsto (i.rand, \psi)] \overset{\Pi}{\models}_{env} A_j[\llbracket j.bv \rrbracket \mapsto (\phi_{j.bv}, \pi_{j.bv})]$
[by (6), (8), (12)]

(14) $A_{j.body} = A_j[\llbracket j.bv \rrbracket \mapsto (\phi_{j.bv}, \pi_{j.bv})]$
[definition of locally-consistent annotation]

(15) $\psi'[\llbracket j.bv \rrbracket \mapsto (i.rand, \psi)] \overset{\Pi}{\models}_{env} A_{j.body}$
[by (13), (14)]

(16) $(j.body, \psi'[\llbracket j.bv \rrbracket \mapsto (i.rand, \psi)])$ is well-thunked
[(2), $Abs(j)$, (5), (15)]

(17) $(k, \psi'') \overset{\Pi}{\models}_{val} (\phi_{j.body}, \pi_{j.body})$
[IH re $\overset{\Pi}{\models}_{val}$ at $j.body$]

(18) (k, ψ'') is well-thunked
[by IH re well-thunkedness at $j.body$]

(19) $(k, \psi'') \overset{\Pi}{\models}_{val} (\phi_i, \pi_i)$
[by (9), (10), (17)]

Step 18 establishes consequent [*2*]; step 19 gives consequent [*3*].

Let us justify step (11) in detail. We enumerate the possibilities for $i.rand$:

1. Suppose $Var(i.rand) \wedge [\![i.rand]\!] \notin Dom(\psi)$. Trivially, $\pi_{i.rand} = \pi_{i.rand}$, so by the definition of $\overset{\Pi}{\models}_{val}$, $(i.rand, \psi) \overset{\Pi}{\models}_{val} (\phi_{i.rand}, \pi_{i.rand})$, therefore $(i.rand, \psi) \overset{\Pi}{\models}_{thunk} (\phi_{i.rand}, \pi_{i.rand})$.
2. Suppose $Const(i.rand)$. Immediately by the definition of $\overset{\Pi}{\models}_{val}$, $(i.rand, \psi) \overset{\Pi}{\models}_{val} (\phi_{i.rand}, \pi_{i.rand})$, hence $(i.rand, \psi) \overset{\Pi}{\models}_{thunk} (\phi_{i.rand}, \pi_{i.rand})$.
3. If $Abs(i.rand)$, then $(i.rand, \psi) \underset{ocn}{\Longrightarrow} (i.rand, \psi)$. By step (2), $(i.rand, \psi)$ is well-thunked. Therefore, by the induction hypothesis, $(i.rand, \psi) \overset{\Pi}{\models}_{val} (\phi_{i.rand}, \pi_{i.rand})$, hence $(i.rand, \psi) \overset{\Pi}{\models}_{thunk} (\phi_{i.rand}, \pi_{i.rand})$.
4. Suppose $Var(i.rand) \wedge [\![i.rand]\!] \in Dom(\psi)$, or $PrimApp(i.rand)$, or $App(i.rand)$, or $Cond(i.rand)$. It is not certain that we can evaluate $(i.rand, \psi)$. First, suppose the evaluation fails. By the definition of $\overset{\Pi}{\models}_{ocn}$, $(i.rand, \psi) \overset{\Pi}{\models}_{ocn} (\phi_{i.rand}, \pi_{i.rand})$, so $(i.rand, \psi) \overset{\Pi}{\models}_{thunk} (\phi_{i.rand}, \pi_{i.rand})$. Next, suppose $(i.rand, \psi) \underset{ocn}{\Longrightarrow} (m, \psi''')$ for some m and ψ'''. By the induction hypothesis, $(m, \psi''') \overset{\Pi}{\models}_{val} (\phi_{i.rand}, \pi_{i.rand})$, so by the definition of $\overset{\Pi}{\models}_{ocn}$, $(i.rand, \psi) \overset{\Pi}{\models}_{ocn} (\phi_{i.rand}, \pi_{i.rand})$, hence $(i.rand, \psi) \overset{\Pi}{\models}_{thunk} (\phi_{i.rand}, \pi_{i.rand})$.

These exhaust the possibilities for $i.rand$.

It remains to show that the consequent involving strictness propositions holds. Any variable $x \in \mathcal{S}_i$ must also be an element of the strictness propositions for the operator or operand, according to the local consistency conditions.

Suppose $x \in \mathcal{S}_i \cap \mathcal{S}_{i.rator}$. By the induction hypothesis regarding strictness propositions, $(i.rator, \psi) \underset{ocn}{\Longrightarrow} (j, \psi') \models_{strict} \mathcal{S}_{i.rator}$. A free occurrence of x in the operator is also free in the

$$
\begin{array}{l}
(i, \psi) \\
\quad (i.rator, \psi) \\
\quad \Downarrow \quad \cdots \\
\quad (j, \psi') \\
\\
\quad (j.body, \psi'[\llbracket j.bv \rrbracket \mapsto (i.rand, \psi)]) \\
\qquad\qquad\qquad (p, \psi_3),\ \llbracket p \rrbracket = \llbracket j.bv \rrbracket \\
\qquad\qquad\qquad\quad (i.rand, \psi) = \psi_3(\llbracket p \rrbracket) = \psi'[\llbracket j.bv \rrbracket \mapsto (i.rand, \psi)](\llbracket j.bv \rrbracket) \\
\qquad \cdots \qquad\qquad\quad \Downarrow \quad \cdots \\
\qquad\qquad\qquad\quad (q, \psi_4) \\
\qquad\qquad\qquad (q, \psi_4) \\
\quad (k, \psi'') \\
(k, \psi'')
\end{array}
$$

Figure 22: Applying a procedure strict in its argument

application, and any subproof of $(i.rator, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$ is also a subproof of the evaluation of the application.

Now suppose $x \in \mathcal{S}_i \cap \mathcal{S}_{i.rand}$ but $x \notin \mathcal{S}_{i.rator}$. Hence, we must be dealing with the case in the local consistency conditions where $\forall (j, B) \in \phi_{i.rator}$, $\llbracket j.bv \rrbracket \in \mathcal{S}_{j.body}$. First, we note that any free occurrence of x in the operand is also free in the application. Next, we show that evaluating the application requires that the operand be evaluated. Figure 22 illustrates the situation.

By step (4) above, (j, A_j) in $\phi_{i.rator}$. Therefore, by the local consistency conditions, $\llbracket j.bv \rrbracket \in \mathcal{S}_{j.body}$. We showed above in step (16) that $(j.body, \psi'[\llbracket j.bv \rrbracket \mapsto (i.rand, \psi)])$ is well-thunked. So by the induction hypothesis at $j.body$, we have $(j.body, \psi[\llbracket j.bv \rrbracket \mapsto (i.rand, \psi)]) \underset{ocn}{\Longrightarrow} (k, \psi'') \models_{strict} \mathcal{S}_{j.body}$. By the definition of $\models_{strict}$, there are p, q, ψ_3, ψ_4, where p is a free occurrence of $\llbracket j.bv \rrbracket$ in $\llbracket j.body \rrbracket$, such that in the evaluation of $(j.body, \psi[\llbracket j.bv \rrbracket \mapsto (i.rand, \psi)])$, there is a subproof $(p, \psi_3) \underset{ocn}{\Longrightarrow} (q, \psi_4)$, and $\psi_3(\llbracket j.bv \rrbracket) = \psi[\llbracket j.bv \rrbracket \mapsto (i.rand, \psi)](\llbracket j.bv \rrbracket) = (i.rand, \psi)$. By Lemma 19, that subproof has an immediate subproof $(i.rand, \psi) \underset{ocn}{\Longrightarrow} (q, \psi_4)$. By step (2), $(i.rand, \psi)$ is well-thunked, so by the induction hypothesis at $i.rand$ regarding strictness propositions, the evaluation of $(i.rand, \psi)$ satisfies $\mathcal{S}_{i.rand}$. Recall, we supposed $x \in \mathcal{S}_{i.rand}$. So by the definition of $\models_{strict}$, there are s, t, ψ_5, ψ_6 such that the evaluation of $(i.rand, \psi)$ has a subproof $(s, \psi_5) \underset{ocn}{\Longrightarrow} (t, \psi_6)$, where s is a free occurrence of x, and if $x \in Dom(\psi)$, then $x \in Dom(\psi_5)$, and $\psi(x) = \psi_5(x)$. But since that subproof is also a subproof of the evaluation of the procedure body, it is a subproof of the evaluation of the application. This establishes consequent [*4*].

There are two immediate subproofs of the root of the derivation tree. By step (1), $(i.rator, \psi)$ is well-thunked, and by step (16), $(j.body, \psi'[\llbracket j.bv \rrbracket \mapsto (i.rand, \psi)])$ is well-thunked. Therefore, by the induction hypothesis, the consequents hold for all proper subproofs.

case $Cond(i)$

Suppose:

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

Since (i, ψ) is well-thunked, by Lemma 20, $(i.test, \psi)$ and $(i.then, \psi)$ are well-thunked. By the induction hypothesis, $(k, \psi'') \overset{\Pi}{\models}_{val} (\phi_{i.then}, \pi_{i.then})$, and (k, ψ'') is well-thunked [*2*]. By the local consistency of Γ, $\mathcal{P}_{i.then} \leq \mathcal{P}_i$. so by Lemma 16, $(k, \psi'') \overset{\Pi}{\models}_{val} (\phi_i, \pi_i)$ [*3*].

By the induction hypothesis

$$(i.test, \psi) \underset{ocn}{\Longrightarrow} (j, \psi') \models_{strict} \mathcal{S}_{i.test}$$

and

$$(i.then, \psi) \underset{ocn}{\Longrightarrow} (k, \psi'') \models_{strict} \mathcal{S}_{i.then}$$

By the local consistency of Γ, $\mathcal{S}_i \subseteq \mathcal{S}_{i.test} \cup (\mathcal{S}_{i.then} \cap \mathcal{S}_{i.else})$. Therefore, any variable in $\mathcal{S}_i$ is in either $\mathcal{S}_{i.test}$ or $\mathcal{S}_{i.then}$. Now, any free variable occurrence in the test or the then-part is also a free variable occurrence in the conditional itself. Any subproof of the evaluation of the test is a subproof of the evaluation of the conditional. Similarly, any subproof of the evaluation of the then-part is a subproof of the evaluation of the conditional [*4*].

There are two immediate subproofs. As we showed, $(i.test, \psi)$ and $(i.then, \psi)$ are well-thunked. By the induction hypothesis, the consequents hold for all proper subproofs.

We just presented one of the two subcases for the case that $Cond(i)$. The other is similar. ∎

6.6 The selective thunkification transformation

Let $\mathcal{T}$ be the transformation

$$\mathcal{T} : O \rightarrow \Lambda_{thunk}$$

defined in Figure 23. For occurrence closures, define the transformation

$$\hat{\mathcal{T}}(i, \psi) = \mathcal{T}(i)\{\mathcal{T}' \circ \psi\}$$

$$Var(i) \wedge \pi_i = id \Longrightarrow \mathcal{T}(i) = [\![i]\!]$$

$$Var(i) \wedge \pi_i = th \Longrightarrow \mathcal{T}(i) = \mathbf{run}\,[\![i]\!]$$

$$Const(i) \Longrightarrow \mathcal{T}(i) = [\![i]\!]$$

$$Abs(i) \Longrightarrow \mathcal{T}(i) = \lambda[\![i.bv]\!].\mathcal{T}(i.body)$$

$$PrimApp(i) \Longrightarrow \mathcal{T}(i) = [\![i.rator]\!]\ \mathcal{T}(i.rand)$$

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

$$App(i) \wedge \pi_{i.rand} = th \Longrightarrow \mathcal{T}(i) = \mathcal{T}(i.rator)\ (\mathbf{thunk}\ \mathcal{T}(i.rand))$$

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

Figure 23: The selective thunkification transformation

where

$$\mathcal{T}'(i,\psi) = \begin{cases} \hat{\mathcal{T}}(j,\psi') & \text{if } \pi_i = id \text{ and } (i,\psi) \underset{ocn}{\Longrightarrow} (j,\psi') \\ \mathbf{thunk}\ \hat{\mathcal{T}}(i,\psi) & \text{if } \pi_i = th \end{cases}$$

Note that the $\mathcal{T}'$ transformation invokes the occurrence evaluator on occurrence closures that are tagged id. It is possible that the evaluation will fail because of divergence. Even if the evaluation succeeds, the $\hat{\mathcal{T}}$ transformation of the result may fail. For these reasons, we need to impose some restrictions on occurrence closures and their annotations.

6.7 Thunk-safety and thunk-acceptability

The restrictions we impose assure that $\hat{\mathcal{T}}$ transformation does not introduce divergence. Nonetheless, even with those restrictions, if an occurrence closure (i,ψ) diverges, $\hat{\mathcal{T}}(i,\psi)$ may not be well-defined.

To obtain these results, we introduce the notions of *thunk-safety* and *thunk-acceptability*.

Definition 13 *An occurrence closure* (i,ψ) *is* thunk-safe *with respect to an annotation map* Γ *iff* $\forall x \in FV([\![i]\!]) \cap Dom(\psi)$, $\psi(x) = (j,\psi')$ *implies either*

1. $Var(j) \wedge [\![j]\!] \notin Dom(\psi')$, *or* $Const(j)$, *or*

2. $\pi_j = th$, *and* (j, ψ') *is thunk-safe with respect to* Γ

This definition is well-founded because of the finite depth of occurrence environments. When a particular annotation map is understood from the context, we will simply say that an occurrence closure is thunk-safe.

Thunk-safety may seem a severe restriction on occurrence closures and their annotations, but it is easy to produce examples:

Lemma 22 *For any annotation map* Γ *and occurrence index* i*, if* ψ *is a scalar environment, then* (i, ψ) *is thunk-safe with respect to* Γ.

Proof. Immediate from Definition 13. ∎

We also need:

Definition 14 *An annotation map* Γ *is* thunk-consistent *iff* $\forall i \in Dom(\Gamma)$, $\pi_i = th$ *implies that for all free variable occurrences* j *in* $[\![i]\!]$, $\pi_j = th$.

Observe:

Lemma 23 *A locally-consistent annotation map is thunk-consistent.*

Proof. By the local consistency conditions in Figure 21.

If $Var(i)$, then i is the only free variable occurrence in $[\![i]\!]$, so the result holds. If $Const(i)$, there are no free variable occurrences in $[\![i]\!]$, and the result holds trivially. For all other cases for i, the condition

$$\pi_i = th \Longrightarrow \forall \text{ free variable occurrences } j \text{ in } [\![i]\!],\ \pi_j = th$$

is stated explicitly in Figure 21. ∎

We can now show that thunk-safety also holds under certain other conditions.

Lemma 24 *Let* Γ *be a locally-consistent annotation map, and let* Π *be the thunk tag assignment defined by* $\forall i,\ \Pi(i) = \pi_i$*. If* (i, ψ) *is well-thunked with respect to* Γ *and* Π*, and* $\pi_i = th$*, then* (i, ψ) *is thunk-safe with respect to* Γ.

Proof. Induction on the depth of ψ.

Base case $\psi = \emptyset$

Then $FV(\llbracket i \rrbracket) \cap Dom(\psi) = \emptyset$, so trivially, (i, ψ) is thunk-safe.

Induction $\psi \neq \emptyset$

Since (i, ψ) is well-thunked, one of the clauses of Definition 12 holds. If clause (1) holds, then $Var(i) \wedge \llbracket i \rrbracket \notin Dom(\psi)$, or $Const(i)$. In either case, $FV(\llbracket i \rrbracket) \cap Dom(\psi) = \emptyset$, so trivially, (i, ψ) is thunk-safe.

Otherwise, clause (2) holds. If $FV(\llbracket i \rrbracket) \cap Dom(\psi) = \emptyset$, then trivially, (i, ψ) is thunk-safe.

Now suppose $x \in FV(\llbracket i \rrbracket) \cap Dom(\psi)$, and let j be a free occurrence of x in $\llbracket i \rrbracket$. Since Γ is locally-consistent, by Lemma 23, Γ is thunk-consistent, so $\pi_j = th$. Let $\psi(\llbracket j \rrbracket) = (k, \psi')$. By clause (2) of Definition 12, $\psi \overset{\Pi}{\models}_{env} A_i$, and (k, ψ') is well-thunked.

We will show that clause (2) of Definition 13 holds. By the local consistency of Γ, $A_j(\llbracket j \rrbracket) = (\phi_j, \pi_j)$. Since $\llbracket j \rrbracket$ is a subterm of $\llbracket i \rrbracket$, and $\llbracket j \rrbracket$ is free in $\llbracket i \rrbracket$, by the local consistency conditions on environment propositions, and an easy induction on the size of terms, $A_i(\llbracket j \rrbracket) = A_j(\llbracket j \rrbracket) = (\phi_j, \pi_j)$. Since $\psi \overset{\Pi}{\models}_{env} A_i$, $\Pi(k) = \pi_k = \pi_j = th$. Since $\pi_k = th$ and (k, ψ') is well-thunked, by the induction hypothesis, (k, ψ') is thunk-safe. By Definition 13, (i, ψ) is thunk-safe. ∎

Now we can put thunk-safety to use:

Lemma 25 *If an occurrence closure (i, ψ) is thunk-safe with respect to an annotation map, then $\hat{\mathcal{T}}(i, \psi)$ is well-defined.*

Proof. Induction on the depth of ψ.

Base case $\psi = \emptyset$

Then

$$\begin{aligned} \hat{\mathcal{T}}(i, \psi) &= \mathcal{T}(i)\{\mathcal{T}' \circ \psi\} \\ &= \mathcal{T}(i)\{\mathcal{T}' \circ \emptyset\} \\ &= \mathcal{T}(i) \end{aligned}$$

which is well-defined by the definition of the transformation in Figure 23.

Induction $\psi \neq \emptyset$

Then

$$\begin{aligned}\hat{\mathcal{T}}(i,\psi) &= \mathcal{T}(i)\{\mathcal{T}' \circ \psi\} \\ &= \mathcal{T}(i)[\mathcal{T}'(\psi(x_1))/x_1, \ldots, \mathcal{T}'(\psi(x_n))/x_n]\end{aligned}$$

where $\{x_1, \ldots, x_n\} = FV(\mathcal{T}(i)) \cap Dom(\psi)$. Since $\mathcal{T}(i)$ is well-defined, the only possible sources of divergence are the invocations of $\mathcal{T}'$. By the definition of thunk-safety, there are two cases to consider. Let x_k be one of the $x_1, \ldots, x_n$, and let $\psi(x_k) = (j, \psi')$.

Suppose clause (1) of Definition 13 holds. Then either $Var(j) \wedge [\![j]\!] \notin Dom(\psi')$, or $Const(j)$. Now, if $\pi_j = th$, then

$$\begin{aligned}\mathcal{T}'(j,\psi') &= \textbf{thunk}\ \hat{\mathcal{T}}(j,\psi') \\ &= \textbf{thunk}\ \mathcal{T}(j)\{\mathcal{T}' \circ \psi'\} \\ &= \textbf{thunk}\ \mathcal{T}(j)\end{aligned}$$

which is well-defined. $(j, \psi') \underset{ocn}{\Longrightarrow} (j, \emptyset)$, so if $\pi_j = id$, then

$$\begin{aligned}\mathcal{T}'(j,\psi') &= \hat{\mathcal{T}}(j,\emptyset) \\ &= \mathcal{T}(j)\{\mathcal{T}' \circ \emptyset\} \\ &= \mathcal{T}(j)\end{aligned}$$

which is also well-defined.

If clause (2) of Definition 13 holds, $\pi_j = th$, and (j, ψ') is thunk-safe. So

$$\mathcal{T}'(j,\psi') = \textbf{thunk}\ \hat{\mathcal{T}}(j,\psi')$$

Since (j, ψ') is contained in ψ, this last term is well-defined by the induction hypothesis. ∎

Thunk-acceptability is slightly less restrictive than thunk-safety:

Definition 15 *An occurrence closure* (i, ψ) *is* thunk-acceptable *with respect to an annotation map* Γ *iff* $\forall x \in FV([\![i]\!]) \cap Dom(\psi)$, $\psi(x) = (j, \psi')$ *implies either*

1. $\pi_j = id$ *and either*

- *$x \in \mathcal{S}_i$, and if $\exists k, \psi''$ such that $(j, \psi') \underset{ocn}{\Longrightarrow} (k, \psi'')$, then (k, ψ'') is thunk-safe with respect to Γ, or*
- *$Var(j) \wedge [\![j]\!] \notin Dom(\psi')$, or $Const(j)$, or*

2. $\pi_j = th$ and (j, ψ') is thunk-safe with respect to Γ

Because this definition just relies on Definition 13, it is well-founded. When a particular annotation map is understood from the context, we will just say that an occurrence closure is thunk-acceptable.

Thunk-safety implies thunk-acceptability:

Lemma 26 *If an occurrence closure (i, ψ) is thunk-safe with respect to an annotation map Γ, then (i, ψ) is thunk-acceptable with respect to Γ.*

Proof. By Definitions 13 and 15.

If $FV([\![i]\!]) \cap Dom(\psi) = \emptyset$, then trivially, (i, ψ) is thunk-acceptable.

Otherwise, suppose $x \in FV([\![i]\!]) \cap Dom(\psi)$, and let $\psi(x) = (j, \psi')$. By the definition of thunk-safety, there are two cases to consider. We will show that in either case, one of the clauses of the definition of thunk-acceptability is satisfied.

If clause (1) of Definition 13 holds, then either $Var(j) \wedge [\![j]\!] \notin Dom(\psi')$, or $Const(j)$. If $\pi_j = id$, then the second part of clause (1) of Definition 15 is satisfied. Otherwise, $\pi_j = th$, so we have to show that clause (2) of Definition 15 is satisfied. Since $FV([\![j]\!]) \cap Dom(\psi') = \emptyset$, (j, ψ') is trivially thunk-safe, so the result holds.

If clause (2) of Definition 13 holds, then $\pi_j = th$, and (j, ψ') is thunk-safe. But those are exactly the requirements of clause (2) of Definition 15. ∎

Given a locally-consistent annotation map, the reverse implication holds for procedures:

Lemma 27 *Let Γ be a locally-consistent annotation map. If $Abs(i)$, then (i, ψ) is thunk-safe with respect to Γ iff (i, ψ) is thunk-acceptable with respect to Γ.*

Proof. By the local consistency of Γ.

$\Longrightarrow$

By Lemma 26.

$\Leftarrow$

We need to show that $\forall x \in FV(\llbracket i \rrbracket) \cap Dom(\psi)$, one of the clauses of Definition 13 is satisfied. If $FV(\llbracket i \rrbracket) \cap Dom(\psi) = \emptyset$, then trivially, (i, ψ) is thunk-safe. Otherwise, suppose $x \in FV(\llbracket i \rrbracket) \cap Dom(\psi)$, and let $\psi(x) = (j, \psi')$. By the local consistency of Γ, $\mathcal{S}_i = \emptyset$, eliminating use of the first part of clause (1) of Definition 15. Therefore, either

1. $\pi_j = id$, and $Var(j) \wedge \llbracket j \rrbracket \notin Dom(\psi')$ or $Const(j)$, or
2. $\pi_j = th$, and (j, ψ') is thunk-safe

If (1) holds, clause (1) of Definition 13 is satisfied; if (2) holds, clause (2) is satisfied. Therefore, (i, ψ) is thunk-safe. ∎

We now show how the notion of thunk-acceptability is used.

Lemma 28 *Let Γ be a monovariant and locally-consistent annotation map, let Π be the thunk tag assignment defined by $\forall i,\ \Pi(i) = \pi_i$, and let (i, ψ) be an occurrence closure that is well-thunked with respect to Γ and Π, and thunk-acceptable with respect to Γ. If $\exists j, \psi'$ such that $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$, then $\hat{\mathcal{T}}(i, \psi)$ is well-defined.*

Proof. By Theorem 14 and the definitions of $\hat{\mathcal{T}}$ and thunk-acceptability.

By definition, $\hat{\mathcal{T}}(i, \psi) = \mathcal{T}(i)\{\mathcal{T}' \circ \psi\}$. From Figure 23, $\mathcal{T}(i)$ is well-defined, so any divergence is due to the substitution. That is, whether the substitution diverges is equivalent to whether there is a variable $x \in FV(\mathcal{T}(i)) \cap Dom(\psi)$ such that $\mathcal{T}'(\psi(x))$ diverges. Suppose $\psi(x) = (k, \psi'')$.

Suppose $\pi_k = id$. By Definition 15, there are two subcases to consider.

In the first subcase, $x \in \mathcal{S}_i$, and if $\exists m, \psi'''$ such that $(k, \psi'') \underset{ocn}{\Longrightarrow} (m, \psi''')$, then (m, ψ''') is thunk-safe. By Theorem 14, $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi') \models_{strict} \mathcal{S}_i$. Therefore, by the definition of $\models_{strict}$, there are p, q, ψ_4, ψ_5 such that p is a free occurrence of x in $\llbracket i \rrbracket$, there is a subproof $(p, \psi_4) \underset{ocn}{\Longrightarrow} (q, \psi_5)$, and $\psi(x) = \psi_4(x) = (k, \psi'')$. By Lemma 19, that subproof has an immediate subproof $(k, \psi'') \underset{ocn}{\Longrightarrow} (q, \psi_5)$. Let $(m, \psi''') = (q, \psi_5)$. By modus ponens, (m, ψ''') is thunk-safe. Therefore, $\hat{\mathcal{T}}(m, \psi''')$ is well-defined by Lemma 25. But $\mathcal{T}'(\psi(x)) = \mathcal{T}'(k, \psi'') = \hat{\mathcal{T}}(m, \psi''')$, so $\mathcal{T}'(\psi(x))$ is well-defined.

In the second subcase, if $Var(k) \wedge \llbracket k \rrbracket \notin Dom(\psi'')$, or $Const(k)$, then $(k, \psi'') \underset{ocn}{\Longrightarrow} (k, \emptyset)$, so $\mathcal{T}'(k, \psi'') = \hat{\mathcal{T}}(k, \emptyset) = \mathcal{T}(k)$, which is well-defined.

Now suppose $\pi_k = th$. Now, $\mathcal{T}'(k, \psi'') = \textbf{thunk}\ \hat{\mathcal{T}}(k, \psi'')$. By Definition 15, (k, ψ'') is thunk-safe, so by Lemma 25, $\hat{\mathcal{T}}(k, \psi'')$ is well-defined, hence the thunk is well-defined. ∎

We have shown that under certain conditions, if an occurrence closure evaluates, we can apply $\hat{\mathcal{T}}$ to it and get a well-defined term. For the Correctness Theorem, we also want to be sure that the $\hat{\mathcal{T}}$ transform of the evaluation result is a well-defined term. We can show:

Lemma 29 *Let Γ be a monovariant and locally-consistent annotation map, and let Π be the thunk tag assignment defined by $\forall i,\ \Pi(i) = \pi_i$. Let (i, ψ) be an occurrence closure that is well-thunked with respect to Γ and Π, and thunk-acceptable with respect to Γ. If $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$, then (j, ψ') is thunk-safe with respect to Γ.*

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

Base cases

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

Then $(i, \psi) \underset{ocn}{\Longrightarrow} (i, \emptyset)$. By Lemma 1, p. 18, the empty environment is a scalar environment, so the result holds by Lemma 22.

case $Const(i)$

Same as for the preceding case.

case $Abs(i)$

Then $(i, \psi) \underset{ocn}{\Longrightarrow} (i, \psi)$, and, by assumption, (i, ψ) is thunk-acceptable. By Lemma 27, (i, ψ) is thunk-safe.

Induction

By the Soundness Theorem, the premise about well-thunkedness holds for all occurrence closures on the left-hand side of $\underset{ocn}{\Longrightarrow}$ for all subproofs of the main derivation.

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

Suppose we have the derivation:

$$\frac{\psi([\![i]\!]) \underset{ocn}{\Longrightarrow} (k, \psi'')}{(i, \psi) \underset{ocn}{\Longrightarrow} (k, \psi'')}$$

Let $\psi([\![i]\!]) = (j, \psi')$. By Definition 15, there are three possibilities to consider.

One possibility is that $\pi_j = id$, $[\![i]\!] \in \mathcal{S}_i$ and if $\exists m, \psi'''$ such that $(j, \psi') \underset{ocn}{\Longrightarrow} (m, \psi''')$, then (m, ψ''') is thunk-safe. Since in fact $(j, \psi') \underset{ocn}{\Longrightarrow} (k, \psi'')$, by modus ponens, (k, ψ'') is thunk-safe.

A second possibility is that $\pi_j = id$, and either $Var(j) \wedge [\![j]\!] \notin Dom(\psi')$, or $Const(j)$. Then $(j, \psi') \underset{ocn}{\Longrightarrow} (j, \emptyset)$. By Lemma 22, $(j, \emptyset)$ is thunk-safe.

Third, it might be that $\pi_j = th$, and (j, ψ') is thunk-safe. By Lemma 26, (j, ψ') is thunk-acceptable, so by the induction hypothesis, (k, ψ'') is thunk-safe.

case $PrimApp(i)$

The evaluation result is $(c, \emptyset)$ for some constant c, so the result holds by Lemma 22.

case $App(i)$

Suppose we have the derivation:

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

Since $FV([\![i.rator]\!]) \subseteq FV([\![i]\!])$, $(i.rator, \psi)$ is thunk-acceptable. Therefore, by the induction hypothesis, (j, ψ') is thunk-safe. By Lemma 26, (j, ψ') is also thunk-acceptable.

Now, either $FV([\![j.body]\!]) = FV([\![j]\!])$, or $FV([\![j.body]\!]) = FV([\![j]\!]) \cup \{[\![j.bv]\!]\}$. If the first equation holds, then immediately $(j.body, \psi'[[\![j.bv]\!] \mapsto (i.rand, \psi)])$ is thunk-acceptable, so by the induction hypothesis, (k, ψ'') is thunk-safe. If the second equation holds, to show that $(j.body, \psi'[[\![j.bv]\!] \mapsto (i.rand, \psi)])$ is thunk-acceptable, we must show that one of the two clauses of Definition 15 holds for $(i.rand, \psi)$.

Suppose $\pi_{i.rand} = id$. We will show that $[\![j.bv]\!] \in \mathcal{S}_{j.body}$, that there are m and ψ''' such that $(i.rand, \psi) \underset{ocn}{\Longrightarrow} (m, \psi''')$, and that (m, ψ''') is thunk-safe. By Theorem 14, $(j, \psi') \models^{\Pi}_{val} (\phi_{i.rator}, \pi_{i.rator})$. By the monovariance assumption, and the definition of $\models^{\Pi}_{val}$, $(j, A_j) \in \phi_{i.rator}$. Since $\pi_{i.rand} = id$, by the local consistency of Γ, it must be that $[\![j.bv]\!] \in \mathcal{S}_{j.body}$. Also by Theorem 14, $(j.body, \psi'[[\![j.bv]\!] \mapsto (i.rand, \psi)]) \underset{ocn}{\Longrightarrow} (k, \psi'') \models_{strict} \mathcal{S}_{j.body}$. Therefore, there are p, q, ψ_4, ψ_5 such that p is a free occurrence of $[\![j.bv]\!]$ in $[\![j.body]\!]$, the evaluation of the procedure body has a subproof $(p, \psi_4) \underset{ocn}{\Longrightarrow} (q, \psi_5)$, and $\psi([\![j.bv]\!]) = \psi_4([\![p]\!]) = \psi'[[\![j.bv]\!] \mapsto (i.rand, \psi)]([\![j.bv]\!]) = (i.rand, \psi)$. By Lemma 19, the preceding subproof has an immediate subproof $(i.rand, \psi) \underset{ocn}{\Longrightarrow} (q, \psi_5)$. Let $(m, \psi''') = (q, \psi_5)$. Since $FV([\![i.rand]\!]) \subseteq FV([\![i]\!])$, $(i.rand, \psi)$ is thunk-acceptable. Therefore, by the induction hypothesis, (m, ψ''') is thunk-safe.

Suppose $\pi_{i.rand} = th$. Since (i, ψ) is well-thunked, by Lemma 20, $(i.rand, \psi)$ is well-thunked. Since Γ is locally-consistent, by Lemma 23, Γ is thunk-consistent. Therefore, by Lemma 24, $(i.rand, \psi)$ is thunk-safe.

Therefore, by Definition 15, $(j.body, \psi[[\![j.bv]\!] \mapsto (i.rand, \psi)])$ is thunk-acceptable, so by the induction hypothesis, (k, ψ'') is thunk-safe.

case $Cond(i)$

As usual, we show only one subcase. Suppose we have the derivation:

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

Since (i, ψ) is thunk-acceptable, and $FV([\![i.else]\!]) \subseteq FV([\![i]\!])$, $(i.else, \psi)$ is thunk-acceptable. By the induction hypothesis, (k, ψ'') is thunk-safe. ∎

Now we can show that, under certain conditions, the $\hat{\mathcal{T}}$ transform of the result of an occurrence closure evaluation is well-defined.

Lemma 30 *Let Γ be a monovariant and locally-consistent annotation map, and let Π be the thunk tag assignment defined by $\forall i,\ \Pi(i) = \pi_i$. Let (i, ψ) be an occurrence closure that is well-thunked with respect to Γ and Π, and thunk-acceptable with respect to Γ. If $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$, then $\hat{\mathcal{T}}(j, \psi')$ is well-defined.*

Proof. By Lemmas 25 and 29. ∎

We need one more new concept before going on to the Correctness Theorem.

6.8 Open-safety

Consider the occurrence closure $(i, \emptyset)$, which self-evaluates. Now suppose $\pi_i = th$. Then $\hat{\mathcal{T}}(i, \emptyset) =$ **run** $[\![i]\!]$, which cannot be evaluated. For the Correctness Theorem, we want such occurrences of variables not in the domain of their closing environments to be tagged id.

Definition 16 *An occurrence closure (i, ψ) is* open-safe *with respect to an annotation map Γ iff for all free variable occurrences j in $[\![i]\!]$:*

1. *if $[\![j]\!] \notin Dom(\psi)$, then $\pi_j = id$*
2. *if $[\![j]\!] \in Dom(\psi)$, then $\psi([\![j]\!])$ is open-safe with respect to Γ*

As usual, the well-foundedness of this definition proceeds from the finiteness of occurrence closures. When a particular annotation map is apparent from the context, we will say that an occurrence closure is open-safe.

For example, we have:

Lemma 31 *Let i be an occurrence index, and let ψ be a scalar environment such that $FV([\![i]\!]) \subseteq Dom(\psi)$. Then for any annotation map Γ, (i, ψ) is open-safe with respect to Γ.*

Proof. Immediate from the definition of open-safe. ∎

We can show:

Lemma 32 *If (i, ψ) is open-safe with respect to an annotation map Γ, and $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$, then for all subproofs $(k, \psi'') \underset{ocn}{\Longrightarrow} (m, \psi''')$ of this derivation, (k, ψ'') and (m, ψ''') are open-safe with respect to Γ.*

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

Base cases

For the base cases, there are no proper subproofs.

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

Then $(i, \psi) \underset{ocn}{\Longrightarrow} (i, \emptyset)$. Since (i, ψ) is open-safe, $\pi_i = id$, so $(i, \emptyset)$ is also open-safe.

case $Const(i)$

Then $(i, \psi) \underset{ocn}{\Longrightarrow} (i, \emptyset)$. Since $[\![i]\!]$ has no free variables, trivially, $(i, \emptyset)$ is open-safe.

case $Abs(i)$

Then (i, ψ) self-evaluates, and by assumption, is open-safe.

Induction

For each case in the induction step, we show the result holds at the root of the derivation tree. We show that the open-safe premise holds for all immediate subproofs, allowing us to use the induction hypothesis for those subproofs.

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

The derivation is:

$$\frac{\psi([\![i]\!]) \underset{ocn}{\Longrightarrow} (j, \psi')}{(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')}$$

By the definition of open-safe, $\psi([\![i]\!])$ is open-safe, so (j, ψ') is open-safe by the induction hypothesis. Since $\psi([\![i]\!])$ is open-safe, the result holds for all proper subproofs.

case $PrimApp(i)$

For any operator, by the evaluation rules $(i, \psi) \underset{ocn}{\Longrightarrow} (c, \emptyset)$, for some constant c. Trivially, $(c, \emptyset)$ is open-safe.

Also, since $FV([\![i.rand]\!]) \subseteq FV([\![i]\!])$, $(i.rand, \psi)$ is open-safe, so the result holds for all proper subproofs.

case $App(i)$

Suppose we have a derivation:

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

Since (i, ψ) is open-safe and $FV([\![i.rator]\!]) \subseteq FV([\![i]\!])$, $(i.rator, \psi)$ is also open-safe. By the induction hypothesis, (j, ψ') is open-safe.

If $FV(\llbracket j.body \rrbracket) = FV(\llbracket j \rrbracket)$, then immediately $(j.body, \psi'[\llbracket j.bv \rrbracket \mapsto (i.rand)])$ is open-safe.

Otherwise, $FV(\llbracket j.body \rrbracket) = FV(\llbracket j \rrbracket) \cup \{\llbracket j.bv \rrbracket\}$, so there is at least one free occurrence of $\llbracket j.bv \rrbracket$ in $\llbracket j.body \rrbracket$. Now, $\llbracket j.bv \rrbracket \in Dom(\psi'[\llbracket j.bv \rrbracket \mapsto (i.rand)])$. But $FV(\llbracket i.rand \rrbracket) \subseteq FV(\llbracket i \rrbracket)$, so $(i.rator, \psi)$ is open-safe. Therefore, $(j.body, \psi'[\llbracket j.bv \rrbracket \mapsto (i.rand)])$ is open-safe.

Since both $(i.rator, \psi)$ and $(j.body, \psi'[\llbracket j.bv \rrbracket \mapsto (i.rand)])$ are open-safe, the result holds for all proper subproofs.

case $Cond(i)$

We show only one subcase, as usual.

Suppose we have the derivation:

$$\frac{(i.test, \psi) \underset{ocn}{\Longrightarrow} (j, \psi'), \quad \llbracket j \rrbracket = \mathbf{true} \qquad (i.then, \psi) \underset{ocn}{\Longrightarrow} (k, \psi'')}{(i, \psi) \underset{ocn}{\Longrightarrow} (k, \psi'')}$$

Since (i, ψ) is open-safe, and $FV(\llbracket i.then \rrbracket) \subseteq FV(\llbracket i \rrbracket)$, $(i.then, \psi)$ is open-safe. By the induction hypothesis, (k, ψ'') is open-safe.

There are two immediate subproofs of the derivation. We just showed $(i.then, \psi)$ is open-safe. By similar reasoning, $(i.test, \psi)$ is open-safe. Therefore, the result holds for all proper subproofs. ∎

With the extensive groundwork just laid, we can proceed to the Correctness Theorem.

6.9 Correctness

Comparing the current Correctness Theorem with those for previous transformations, the most important difference is the appearance of $\underset{ocn}{\Longrightarrow}$ in the evaluation premise. Note that we still have $\underset{t}{\Longrightarrow}$ in the consequent.

Theorem 15 (Correctness) *Let Γ be a monovariant and locally-consistent annotation map, let Π be the thunk tag assignment defined by $\forall i, \Pi(i) = \pi_i$, and let (i, ψ) be an occurrence closure that is well-thunked with respect to Γ and Π, and thunk-acceptable and open-safe with respect to Γ. If $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$, then $\hat{\mathcal{T}}(i, \psi) \underset{t}{\Longrightarrow} \hat{\mathcal{T}}(j, \psi')$.*

The diagram for the theorem is:

$$\begin{tikzcd}
(i,\psi) \arrow[rr, Rightarrow, "ocn"'] \arrow[d, "\hat{\mathcal{T}}"'] & & (j,\psi') \arrow[d, "\hat{\mathcal{T}}"] \\
\hat{\mathcal{T}}(i,\psi) \arrow[rr, Rightarrow, "t"'] & & \hat{\mathcal{T}}(j,\psi')
\end{tikzcd}$$

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

Observe that by Lemma 25, $\hat{\mathcal{T}}(i,\psi)$ is well-defined. Similarly, by Lemma 30, $\hat{\mathcal{T}}(j,\psi')$ is well-defined.

Base cases

case $\mathit{Var}(i) \wedge [\![i]\!] \notin \mathit{Dom}(\psi)$

Then $(i,\psi) \underset{ocn}{\Longrightarrow} (i,\emptyset)$.

Since (i,ψ) is open-safe, $\pi_i = id$. So we have:

$$\begin{array}{rcl}
\hat{\mathcal{T}}(i,\psi) & = & \mathcal{T}(i)\{\mathcal{T}' \circ \psi\} \\
& = & [\![i]\!]\{\mathcal{T}' \circ \psi\} \\
& = & [\![i]\!] \\
& \underset{t}{\Longrightarrow} & [\![i]\!] \\
& = & [\![i]\!]\{\mathcal{T}' \circ \emptyset\} \\
& = & \mathcal{T}(i)\{\mathcal{T}' \circ \emptyset\} \\
& = & \hat{\mathcal{T}}(i,\emptyset)
\end{array}$$

case $\mathit{Const}(i)$

Then $(i,\psi) \underset{ocn}{\Longrightarrow} (i,\emptyset)$, and we have:

$$\begin{array}{rcl}
\hat{\mathcal{T}}(i,\psi) & = & \mathcal{T}(i)\{\mathcal{T}' \circ \psi\} \\
& = & [\![i]\!]\{\mathcal{T}' \circ \psi\} \\
& = & [\![i]\!] \\
& \underset{t}{\Longrightarrow} & [\![i]\!]
\end{array}$$

Transforming the $\underset{ocn}{\Longrightarrow}$ result, we get

$$\begin{aligned}\hat{\mathcal{T}}(i,\emptyset) &= \mathcal{T}(i)\{\mathcal{T}'\circ\emptyset\}\\ &= \mathcal{T}(i)\\ &= [\![i]\!]\end{aligned}$$

case $Abs(i)$

Then $(i,\psi)\underset{ocn}{\Longrightarrow}(i,\psi)$, and $\hat{\mathcal{T}}(i,\psi)$ is a procedure, which self-evaluates.

Induction

By Lemma 32, the premise about open-safety holds for all subproofs of the main derivation. Similarly, by the Soundness Theorem, the premise about well-thunkedness holds for all subproofs. There is no comparable result for the premise about thunk-acceptability, so in each case below we need to show that property holds to use the induction hypothesis.

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

So we have the derivation:

$$\frac{\psi([\![i]\!])\underset{ocn}{\Longrightarrow}(j,\psi')}{(i,\psi)\underset{ocn}{\Longrightarrow}(j,\psi')}$$

In the case that $\pi_i = id$, we cannot use the induction hypothesis. But:

$$\begin{aligned}\hat{\mathcal{T}}(i,\psi) &= \mathcal{T}(i)\{\mathcal{T}'\circ\psi\}\\ &= [\![i]\!]\{\mathcal{T}'\circ\psi\}\\ &= \mathcal{T}'(\psi([\![i]\!]))\\ &= \hat{\mathcal{T}}(j,\psi')\end{aligned}$$

Since $Var(i)\wedge[\![i]\!]\in Dom(\psi)$ and (i,ψ) is well-thunked, by the definition of well-thunkedness, $\psi\overset{\Pi}{\models}_{env} A_i$. By the definition of $\overset{\Pi}{\models}_{env}$, since $\pi_i = id$, the thunk tag for the occurrence index in the occurrence closure $\psi([\![i]\!])$ must also be id. Using the tag information and the fact that $\psi([\![i]\!])\underset{ocn}{\Longrightarrow}(j,\psi')$, the definition of $\mathcal{T}'$ gives the result.

We want to show that $\hat{\mathcal{T}}(i,\psi)\underset{t}{\Longrightarrow}\hat{\mathcal{T}}(j,\psi')$. Since (j,ψ') appears on the right-hand side of $\underset{ocn}{\Longrightarrow}$, by an easy induction on the definition of $\underset{ocn}{\Longrightarrow}$, one of the following holds:

1. $Var(j) \wedge \psi' = \emptyset$, or
2. $Const(j) \wedge \psi' = \emptyset$, or
3. $Abs(j)$

In case (1), by Theorem 14, $(j, \psi') \overset{\Pi}{\models}_{val} (\phi_i, \pi_i)$, so $\Pi(j) = \pi_j = \pi_i = id$, hence $\hat{\mathcal{T}}(j, \psi') = \mathcal{T}(j) = [\![j]\!]$. In case (2), $\hat{\mathcal{T}}(j, \psi') = \mathcal{T}(j) = [\![j]\!]$. In case (3), $\hat{\mathcal{T}}(j, \psi')$ is a procedure. In all three cases, the transform is a value, which self-evaluates. Therefore:

$$\hat{\mathcal{T}}(i, \psi) = \hat{\mathcal{T}}(j, \psi') \underset{t}{\Longrightarrow} \hat{\mathcal{T}}(j, \psi')$$

In the case that $\pi_i = th$:

$$\begin{aligned} \hat{\mathcal{T}}(i, \psi) &= \mathcal{T}(i)\{\mathcal{T}' \circ \psi\} \\ &= (\mathbf{run}\,[\![i]\!])\{\mathcal{T}' \circ \psi\} \\ &= \mathbf{run}\,[\![i]\!]\{\mathcal{T}' \circ \psi\} \\ &= \mathbf{run}\ \mathcal{T}'(\psi([\![i]\!])) \\ &= \mathbf{run}\,(\mathbf{thunk}\ \hat{\mathcal{T}}(\psi([\![i]\!]))) \end{aligned}$$

We can justify the last step by a similar argument as in the previous subcase, except that the tags involved are th, so the effect of $\mathcal{T}'$ is different. Now, $\psi([\![i]\!])$ is well-thunked and open-safe; to use the induction hypothesis we must also show it is thunk-acceptable. By assumption, (i, ψ) is thunk-acceptable. Since the thunk tag for the index-part of $\psi([\![i]\!])$ is th, by Definition 15, $\psi([\![i]\!])$ is thunk-safe. Therefore, by Lemma 26, $\psi([\![i]\!])$ is also thunk-acceptable. So by the induction hypothesis, $\hat{\mathcal{T}}(\psi([\![i]\!])) \underset{t}{\Longrightarrow} \hat{\mathcal{T}}(j, \psi')$, and by the evaluation rule for running thunks,

$$\hat{\mathcal{T}}(i, \psi) = \mathbf{run}\,(\mathbf{thunk}\ \hat{\mathcal{T}}(\psi([\![i]\!]))) \underset{t}{\Longrightarrow} \hat{\mathcal{T}}(j, \psi')$$

case $PrimApp(i)$

Suppose we have the derivation:

$$\frac{(i.rand, \psi) \underset{ocn}{\Longrightarrow} (j, \psi') \quad Const(j)}{(i, \psi) \underset{ocn}{\Longrightarrow} (c, \emptyset)}$$

where the relation between the constants $[\![j]\!]$ and c depends on the operator.

For the transformation, we have:

$$\begin{aligned}
\hat{\mathcal{T}}(i,\psi) &= \mathcal{T}(i)\{\mathcal{T}'\circ\psi\} \\
&= ([\![i.rator]\!]\ \mathcal{T}(i.rand))\{\mathcal{T}'\circ\psi\} \\
&= [\![i.rator]\!]\ \mathcal{T}(i.rand)\{\mathcal{T}'\circ\psi\} \\
&= [\![i.rator]\!]\ \hat{\mathcal{T}}(i.rand,\psi)
\end{aligned}$$

Since (i,ψ) is thunk-acceptable and $FV([\![i.rand]\!]) \subseteq FV([\![i]\!])$, $(i.rand,\psi)$ is thunk-acceptable. Therefore, by the induction hypothesis at $i.rand$, we have:

$$\begin{aligned}
\hat{\mathcal{T}}(i.rand,\psi) &\underset{t}{\Longrightarrow} \hat{\mathcal{T}}(j,\psi') \\
&= \mathcal{T}(j)\{\mathcal{T}'\circ\psi\} \\
&= [\![j]\!]\{\mathcal{T}'\circ\psi\} \\
&= [\![j]\!]
\end{aligned}$$

By the relevant term evaluation rule, which depends on the operator,

$$\begin{aligned}
\hat{\mathcal{T}}(i,\psi) &\underset{t}{\Longrightarrow} c \\
&= [\![c]\!] \\
&= [\![c]\!]\{\hat{\mathcal{T}}\circ\emptyset\} \\
&= \hat{\mathcal{T}}(c,\emptyset)
\end{aligned}$$

case $App(i)$

Suppose we have the derivation:

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

We have:

$$\hat{\mathcal{T}}(i,\psi) = \hat{\mathcal{T}}(i.rator,\psi)\ \hat{\mathcal{T}}(i.rand,\psi)$$

Suppose $\pi_{i.rand} = id$. Since (i,ψ) is thunk-acceptable and $FV([\![i.rator]\!]) \subseteq FV([\![i]\!])$, $(i.rator,\psi)$ is thunk-acceptable. So by the induction hypothesis at $i.rator$:

$$\hat{\mathcal{T}}(i.rator,\psi)$$

$$
\begin{aligned}
&\underset{t}{\Longrightarrow} && \hat{\mathcal{T}}(j, \psi') \\
&= && \mathcal{T}(j)\{\mathcal{T}' \circ \psi'\} \\
&= && (\lambda [\![j.bv]\!] . \mathcal{T}(j.body))\{\mathcal{T}' \circ \psi'\} \\
&= && (\lambda x' . \mathcal{T}(j.body)[x' / [\![j.bv]\!]])\{\mathcal{T}' \circ \psi'\} \\
&= && \lambda x' . \mathcal{T}(j.body)[x' / [\![j.bv]\!]]\{\mathcal{T}' \circ \psi'\}
\end{aligned}
$$

where x' is a fresh variable.

We cannot use the induction hypothesis for the operand, since the call-by-name occurrence closure evaluator does not evaluate the operand when evaluating the application. On the other hand, the Soundness Theorem (Theorem 14) and the local consistency conditions assure that

$$(i.rand, \psi) \underset{ocn}{\Longrightarrow} (m, \psi''')$$

for some m and ψ'''. To see this, we use the following reasoning (similar to that used in the proof of Lemma 28):

(1) $(j, \psi') \overset{\Pi}{\models}_{val} (\phi_{i.rator}, \pi_{i.rator})$
[proof that $(i, \psi) \underset{ocn}{\Longrightarrow} (k, \psi'')$, Theorem 14]

(2) $(j, A_j) \in \phi_{i.rator}$
[by (1), definition of $\overset{\Pi}{\models}_{val}$, monovariance]

(3) $[\![j.bv]\!] \in \mathcal{S}_{j.body}$
[(2), $\pi_{i.rand} = id$, local consistency of Γ]

(4) $(j.body, \psi'[[\![j.bv]\!] \mapsto (i.rand, \psi)]) \underset{ocn}{\Longrightarrow} (k, \psi'') \models_{strict} \mathcal{S}_{j.body}$
[proof that $(i, \psi) \underset{ocn}{\Longrightarrow} (k, \psi'')$, Theorem 14]

(5) $\exists p, q, \psi_4, \psi_5$ such that
  (a) p is a free occurrence of $[\![j.bv]\!]$ in $[\![j.body]\!]$, and
  (b) $(p, \psi_4) \underset{ocn}{\Longrightarrow} (q, \psi_5)$ is a subproof of
    $(j.body, \psi'[[\![j.bv]\!] \mapsto (i.rand, \psi)]) \underset{ocn}{\Longrightarrow} (k, \psi'')$
  (c) $\psi_4([\![j.bv]\!]) = (i.rand, \psi)$
[by (3), (4), definition of $\models_{strict}$]

(6) $(i.rand, \psi) \underset{ocn}{\Longrightarrow} (q, \psi_5)$

[by (5), Lemma 19]

Let $(m, \psi''') = (q, \psi_5)$. Since $FV([\![i.rand]\!]) \subseteq FV([\![i]\!])$, $(i.rand, \psi)$ is thunk-acceptable. So now we can call upon the induction hypothesis at $i.rand$:

$$\hat{\mathcal{T}}(i.rand, \psi) \underset{t}{\Longrightarrow} \hat{\mathcal{T}}(m, \psi''')$$

To use the rule for applications, we need to evaluate

$$\begin{aligned}
& \mathcal{T}(j.body)[x'/[\![j.bv]\!]]\{\mathcal{T}' \circ \psi'\}[\hat{\mathcal{T}}(m, \psi''')/x'] \\
= \quad & \mathcal{T}(j.body)\{\mathcal{T}' \circ \psi'[[\![j.bv]\!] \mapsto (i.rand, \psi)]\} \\
= \quad & \hat{\mathcal{T}}(j.body, \psi'[[\![j.bv]\!] \mapsto (i.rand, \psi)]) \\
\underset{t}{\Longrightarrow} \quad & \hat{\mathcal{T}}(k, \psi'')
\end{aligned}$$

By the same argument we used in the proof of Lemma 29, p. 156, $(j.body, \psi'[[\![j.bv]\!] \mapsto (i.rand, \psi)])$ is thunk-acceptable, so the evaluation step proceeds from the induction hypothesis at $j.body$.

To see that the first equality holds, we use a variation of our familiar argument for such cases. Consider a variable y free in $\mathcal{T}([\![j.body]\!])$, where $y \neq [\![j.bv]\!]$. On both sides of the equality, we substitute $\mathcal{T}'(\psi'(y))$ for y. Now consider a free occurrence of $[\![j.bv]\!]$. On the left-hand side, we substitute $\hat{\mathcal{T}}(m, \psi''')$. By assumption, we have $\pi_{i.rand} = id$. Above, we showed that $(i.rand, \psi) \underset{ocn}{\Longrightarrow} (m, \psi''')$. So on the right-hand side, we substitute

$$\begin{aligned}
& \mathcal{T}'((\psi'[[\![j.bv]\!] \mapsto (i.rand, \psi)])([\![j.bv]\!])) \\
= \quad & \mathcal{T}'(i.rand, \psi) \\
= \quad & \hat{\mathcal{T}}(m, \psi'')
\end{aligned}$$

By the evaluation rule for applications

$$\hat{\mathcal{T}}(i, \psi) \underset{t}{\Longrightarrow} \hat{\mathcal{T}}(k, \psi'')$$

Now suppose that $\pi_{i.rand} = th$. All arguments about the evaluation of the operator in the previous subcase still apply. But now the operand of the transformed application self-evaluates:

$$\textbf{thunk}\ \hat{\mathcal{T}}(i.rand, \psi) \underset{t}{\Longrightarrow} \textbf{thunk}\ \hat{\mathcal{T}}(i.rand, \psi)$$

This time, to use the application rule, we need to evaluate

$$\begin{aligned}
& \mathcal{T}(j.body)[x'/[\![j.bv]\!]]\{\mathcal{T}' \circ \psi'\}[(\mathbf{thunk}\ \hat{\mathcal{T}}(i.rand, \psi))/x'] \\
= \;& \mathcal{T}(j.body)\{\mathcal{T}' \circ (\psi'[[\![j.bv]\!] \mapsto (i.rand, \psi)])\} \\
= \;& \hat{\mathcal{T}}(j.body, \psi'[[\![j.bv]\!] \mapsto (i.rand, \psi)])
\end{aligned}$$

To see that the first equality holds, consider a variable y free in $\mathcal{T}([\![j.body]\!])$, where $y \neq [\![j.bv]\!]$. Again, the substitutions agree on both sides of the equality. For free occurrences of $[\![j.bv]\!]$, we substitute **thunk** $\hat{\mathcal{T}}(i.rand, \psi)$ on the left-hand side, and $\mathcal{T}'(i.rand, \psi)$ on the right-hand side. But $\pi_{i.rand} = th$, so $\mathcal{T}'(i.rand, \psi) = \mathbf{thunk}\ \hat{\mathcal{T}}(i.rand, \psi)$.

By the induction hypothesis at $j.body$:

$$\hat{\mathcal{T}}(j.body, \psi'[[\![j.bv]\!] \mapsto (i.rand, \psi)]) \underset{t}{\Longrightarrow} \hat{\mathcal{T}}(k, \psi'')$$

so by the application rule

$$\hat{\mathcal{T}}(i, \psi) \underset{t}{\Longrightarrow} \hat{\mathcal{T}}(k, \psi'')$$

case $Cond(i)$

We show only one subcase; the other is similar.

Suppose we have the derivation:

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

Transforming the conditional, we get

$$\begin{aligned}
& \hat{\mathcal{T}}(i, \psi) \\
= \;& \mathcal{T}(i)\{\mathcal{T}' \circ \psi\} \\
= \;& (\mathbf{if}\ \mathcal{T}(i.test)\ \mathbf{then}\ \mathcal{T}(i.then)\ \mathbf{else}\ \mathcal{T}(i.else))\{\mathcal{T}' \circ \psi\} \\
= \;& \mathbf{if}\ \hat{\mathcal{T}}(i.test, \psi)\ \mathbf{then}\ \hat{\mathcal{T}}(i.then, \psi)\ \mathbf{else}\ \hat{\mathcal{T}}(i.else, \psi)
\end{aligned}$$

Since $FV([\![i.test]\!]) \subseteq FV([\![i]\!])$, and $FV([\![i.then]\!]) \subseteq FV([\![i]\!])$, $(i.test, \psi)$ and $(i.then, \psi)$ are thunk-acceptable. By the induction hypothesis at $i.test$:

$$\hat{\mathcal{T}}(i.test, \psi) \quad \underset{t}{\Longrightarrow} \quad \hat{\mathcal{T}}(j, \psi')$$

$$
\begin{aligned}
&= \mathcal{T}(j)\{\mathcal{T}' \circ \psi'\} \\
&= \mathbf{true}\,\{\mathcal{T}' \circ \psi'\} \\
&= \mathbf{true}
\end{aligned}
$$

The induction hypothesis at $i.then$ gives us:

$$\hat{\mathcal{T}}(i.then, \psi) \underset{t}{\Longrightarrow} \hat{\mathcal{T}}(k, \psi'')$$

So by the relevant evaluation rule for conditionals

$$\hat{\mathcal{T}}(i, \psi) \underset{t}{\Longrightarrow} \hat{\mathcal{T}}(k, \psi'')$$

∎

6.10 Related work

The problem of transforming programs with delayed evaluation into call-by-value equivalents was first studied by Mycroft using a first-order language [41]. Instead of considering full call-by-name programs, in which arguments may be evaluated an arbitrary number of times, Mycroft looked at call-by-need programs, in which arguments are evaluated at most once. His strictness analysis was based on an abstract interpretation.

Many authors since have studied strictness analysis. Burn, Hankin, and Abramsky proved the correctness of an analysis using an abstract interpretation [11]. Jensen presented a lattice-theoretic framework for comparing strictness analyses based on abstract interpretation and type inference [28]. Kuo and Mishra used a type system with strictness annotations on base types [35]. Wright developed a type system in which all types are decorated with strictness information [61].

Danvy and Hatcliff presented a continuation-passing style (CPS) conversion algorithm using thunkification and relying on strictness information [18]. Burn and Le Métayer claimed the correctness of a CPS transformation for a typed language that allows the evaluation of arguments to strict procedures, although they did not give their own strictness analysis [10]. Hudak and Young proved the correctness of a strictness analysis for an untyped λ-calculus [25]; unlike ours, their analysis was not guaranteed to terminate.

Closest to the current work is that of Amtoft [4], who used a type system derived from Wright to prove the correctness of a transformation from call-by-name to call-by-value. Our strictness propositions are borrowed from Amtoft, whose inference system computed a set of “needed variables” for a typed expression.

6.11 Conclusion

Of the four transformations in this dissertation, the selective thunkification transformation was the most difficult to develop, and provided the greatest challenge for our method. Adding a strictness analysis and adding a call-by-name to call-by-value transformation were nonincremental changes from the earlier analyses and transformations. The fact that we were able to stretch our techniques to handle these new features suggests our method is robust.

Chapter 7

Conclusions and Future Research

7.1 Goals and achievements

The research in this dissertation is an attempt to develop an approach for performing correct transformations for higher-order programming languages. Our analyses are grounded in the familiar concepts of sets, relations, and operational semantics. The implementation of any of the analyses is straightforward. The transformations themselves are simple. We believe the goal we set out to achieve has been accomplished; the method works.

With that claim in mind, we wish to confirm (especially for the sake of those readers brave enough to have followed the proofs) that developing an analysis and proving it correct is not so easy. We began the research by developing proofs for a closure conversion algorithm involving only selectivity. That proof was more or less routine. For several weeks, we thought hard about how to build lightweightness into the analysis. Once it became clear that we needed something like the invariance sets of Chapter 3, things began to fall into place. The idea of the invariance relations of Chapter 4 was a logical next step. With that experience, it was fairly clear how to do the Ultra-β analysis. Using call-by-name in the selective thunkification analysis Chapter 6 required much additional insight to get the right semantics of annotations; the notions of well-thunkedness, thunk-safety, and thunk-acceptability were other leaps along the path to getting the analysis and transformation right. Developing an analysis and transformation in this style is far from automatic, but it is tractable. Given the variety of transformations to which we have successfully applied our techniques, we believe the method presented here is a good one.

7.2 Future research

The current research raises issues for future investigation.

- *Composing code modules.* Our analyses are *global*; we assume that the analyzed code is all the code to be considered. However, a generally-accepted principle of good software engineering practice is the *modularity* of a software system design. At the source language level, modularity is supported by the separate compilation of code.

  The notion of *observational equivalence* provides a theoretical basis for reasoning about the composition of separately compiled code. A context $C[\,]$ is a term with a "hole" in it. We write $C[M]$ to indicate that M fills the hole in $C[\,]$. Two terms M and N are said to be observationally equivalent iff for all contexts $C[\,]$

  $$C[M] \underset{t}{\Longrightarrow} c \text{ iff } C[N] \underset{t}{\Longrightarrow} c$$

  If we know the conditions under which a program and its transform are observationally equivalent, we have the choice of using either a module or its transform to compose with another, separately-analyzed chunk of code.

  For instance, consider a procedure $\lambda x.x$. Suppose we transform the procedure using one of our closure-conversion algorithms. We can always produce a locally-consistent annotation in which all occurrences in the procedure have a protocol tag of id. In that case, the closure-conversion transformation acts as an identity function, so the procedure and its transform are operationally equivalent. On the other hand, suppose all occurrences have a protocol tag of $cl_{\langle\rangle}$, which is also always possible. In that case, the transform of the procedure is a closure record. Now suppose we substitute the original procedure into the context $(\lambda f.f\ c)\ \_$. The substituted context evaluates to c. If instead we substitute the closure record into the same context, we cannot evaluate the resulting term. Therefore, we wish to identify what restrictions on annotations are necessary to make a term and its transform observationally equivalent. Clearly, those restrictions will depend on the annotations and transformation in question.

- *Polyvariant analyses.* All the analyses in the dissertation rely on a monovariant annotation map. In a monovariant analysis, each procedure is represented by one abstract closure. In a polyvariant analysis, there are possibly several abstract closures to represent each procedure, corresponding to several possible sets of closing environments for the procedure. Such a finer analysis might reveal additional opportunities for optimization.

  To perform a polyvariant analysis, each a state of a finite-state machine M is identified with a set of possible environments in a program. The sets of environments for each state are pairwise disjoint, so the states represent a partition of the set of all possible environments. Suppose we evaluate an application in an environment corresponding to a state q of M. Evaluating the application requires that we evaluate the body of some procedure in a new environment that corresponds to a state q' of M.

  For each state, or variant, we perform a separate analysis and transformation. When a procedure is called, the code generated for one variant may branch to the code for another variant. The benefit of this finer analysis is possibly better code; the cost is extra time for the analysis and larger code. With suitable modifications, the method presented in this dissertation should be able to handle polyvariant analyses and transformations.

- *Lifetime analyses.* If a data structure's lifetime is bounded by the lifetime of the stack frame active when the structure is created, then it can be stack-allocated. Such a data structure might be a closure for a procedure or consist of data as such. In this dissertation, we used occurrence closures to track the textual origin of answers produced by programs. Occurrence closures could be extended to track the creation of occurrence environment bindings, and so serve as the basis of lifetime analyses.

  For example, assume we have a machine model in which a stack contains bindings for the formal parameters of procedures. Now suppose we add another component to occurrence closures that represents the stack locations where the bindings in the environment component were created. If a procedure is called only at sites where the bindings for its free variables are in pending stack frames, the closure for the procedure can itself be stack-allocated. We may be able to make that decision by comparing the extra component in the occurrence closure for a procedure with the comparable component in occurrence closures that are evaluated at the procedure's possible call sites.

- *Mixed type and flow analyses.* Our source language Λ_{in} is untyped; all the analyses are based on dataflow information. Using a typed language instead might lead to simpler proofs or simpler analyses.

  For example, the analyses for our closure-conversion transformations assure that all possible procedures at a call site agree on protocol. By making a protocol a component of a type, a typing discipline could enforce such agreement where a dataflow analysis does so now.

- *Automating analyses.* In the same way that parser generators produce parser code from grammars, or type checkers from type annotations, it should be possible to generate program analyzers from dataflow constraint specifications. To do so, we would define some formal language for specifying constraints corresponding to the local consistency conditions we gave for each analysis. The output would be a program that would solve the annotations for any valid input program.

- *Other programming paradigms.* The method developed in the dissertation could be applied to languages from other paradigms, such as object-oriented, logic, or parallel languages. As an example, suppose we have an object-oriented language that uses a lazy evaluation strategy for arguments accompanying message sends. Just as we used a strictness analysis to determine when arguments to procedures were certain to be evaluated for the selective thunkification transformation, we could perform a comparable analysis for message sends. If it can be determined that an argument is certain to be evaluated in all cases, the argument can be evaluated at the time of the message send. An essential component of such an analysis would be determining the classes of possible receivers at message send sites, much like a closure analysis.

Glossary

Abstract interpretation: The execution of a program using abstract values, rather than actual values, in order to provide summary information about the actual execution of the program. "Execution" should be broadly construed here, since an abstract interpretation may be provided by a non-standard semantics.

Call-by-name: A program evaluation strategy in which the arguments to procedures are passed directly to the procedure. An argument bound to a formal parameter is evaluated when an occurrence of the formal parameter in the procedure body is referenced.

Call-by-value: A program evaluation strategy in which the arguments to procedures are evaluated when the procedure is called, so that the values of the arguments are passed to the procedure.

Closure: A data structure that represents a procedure in a higher-order language. Typically, a closure is a pair or record consisting of the procedure code and bindings for the free variables of the procedure.

Closure conversion: A compiler transformation that builds source-level closure representations for procedures.

Copy propagation: A compiler transformation in which variables, known to be copies of other, original variables, are replaced by their originals.

Denotational semantics: A formal method for specifying the meaning of programs, in which each syntactic construct in a program is mapped to a value in some domain.

Domain: A partially-ordered set with sufficient structure to allow the interpretation of recursive definitions, such as $D = D \rightarrow D$.

Dynamic variable: In selective and lightweight closure conversion, a variable that is supplied as an extra argument at a call site, so that some variable may be omitted from the closures of procedures flowing to that call site. Using the available value analysis in Chapter 4, the presence of a dynamic variable may allow the the same or a different variable to be omitted from the closures.

Lightweight closure: A closure that omits one or more of the free variables of the procedure it represents. The code part of a lightweight closure has extra formal parameters, which become bound to the values of dynamic variables at the closure's call sites.

Non-standard semantics: A denotational semantics in which the syntactic constructs of a programming language are mapped to values in an abstract domain representing abstract properties of programs.

Source-to-source transformation: A program transformation in which the input and output languages are similar or identical.

Thunk: A procedure without parameters, which is used to represent a suspended computation.

Bibliography

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

[2] Samson Abramsky and Chris Hankin. *Abstract Interpretation of Declarative Languages*, chapter An Introduction to Abstract Interpretation. Ellis Horwood Ltd., Chichester, 1987.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[4] Torben Amtoft. Minimal Thunkification. In Patrick Cousot et al., editor, *Proc. 3rd International Workshop on Static Analysis*, Lecture Notes in Computer Science 724, pages 218–29, Padova, Italy, 1993.

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

[6] Lennart Augustsson. A Compiler for Lazy ML. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–27, August 1984.

[7] Andrew E. Ayers. *Abstract Analysis and Optimization of Scheme*. PhD thesis, MIT, September 1993.

[8] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, 1981.

[9] Anders Bondorf. Automatic Autoprojection of Higher-order Recursive Equations. *Science of Computer Programming*, 17(1-3):3–34, December 1991.

[10] Geoffrey Burn and Daniel Le Métayer. Proving the Correctness of Compiler Optimisations Based on a Global Program Analysis: A Study of Strictness Analysis. Technical Report TR93-42, Department of Computing, Imperial College of Science, Technology and Medicine, August 1993.

[11] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness Analysis for Higher-order Functions. *Science of Computer Programming*, 7:249–78, 1986.

[12] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In Andreas Paepcke, editor, *Proc. 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15, 1993.

[13] Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, Princeton, NJ, 1941. Reprinted 1963 by University Microfilms, Ann Arbor, MI.

[14] William Clinger and Jonathan Rees, eds. Revised^4 Report on the Algorithmic Language Scheme, November 1991. Available as MIT Artificial Intelligence Laboratory Memo 848b.

[15] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, Cambridge, MA/New York, 1990.

[16] Bruno Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:96–169, 1983.

[17] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoints. In *Conference Record of 4th ACM Symposium on Principles of Programming Languages*, pages 238–52, 1977.

[18] Olivier Danvy and John Hatcliff. CPS Transformation After Strictness Analysis. *ACM Letters on Programming Languages*, 1(3):195–212, 1993.

[19] R. W. Floyd. Assigning Meanings to Programs. In *Proc. Symp. on Appl. Math.* American Mathematical Society, 1967.

[20] Pascal Fradet and Daniel Le Métayer. Compilation of Functional Languages by Program Transformation. *ACM Transactions on Programming Languages and Systems*, 13(1):21–51, January 1991.

[21] Michael J.C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, Berlin, Heidelberg, and New York, 1979.

[22] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, Cambridge, MA, 1992.

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

[24] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12, 1969.

[25] Paul Hudak and Jonathan Young. Higher-Order Strictness Analysis in Untyped Lambda Calculus. In *Conference Record of 13th ACM Symposium on Principles of Programming Languages*, pages 97–109, 1986.

[26] Peter Z. Ingerman. Thunks, a Way of Compiling Procedure Statements with Some Comments on Procedure Declarations. *Comm. ACM*, 4(1):55–58, 1961.

[27] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report.* Springer-Verlag, Berlin, Heidelberg, and New York, 4th edition, 1991. Revised by Andrew B. Mickel and James F. Miner.

[28] Thomas P. Jensen. Strictness Analysis in Logical Form. In John Hughes, editor, *International Conf. on Functional Programming Languages and Computer Architecture*, pages 352–66, 1991.

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

[30] Neil D. Jones and Steven S. Muchnick. 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*, pages 66–74, 1982.

[31] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1988.

[32] Gary Kildall. A Unified Approach to Global Program Optimization. In *Conference Record of ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[33] J.W. Klop and R.C. De Vrijer. Unique Normal Forms for Lambda Calculus with Surjective Pairing. *Information and Computation*, 80(2):97–113, February 1989.

[34] David Andrew Kranz. *ORBIT: An Optimizing Compiler for Scheme.* PhD thesis, Yale University, 1988.

[35] Tsung-Min Kuo and Prateek Mishra. Strictness Analysis: A New Perspective. In *International Conf. on Functional Programming and Computer Architecture*, pages 260–72, 1989.

[36] P.J. Landin. The Mechanical Evaluation of Expressions. *Computer J.*, 6(4):308–20, 1964.

[37] John Launchbury. A Natural Semantics for Lazy Evaluation. In *Conference Record of 20th ACM Symposium on Principles of Programming Languages*, pages 144–54, 1993.

[38] Pierre Lescanne. From $\lambda\sigma$ to λv, a Journey through Calculi of Explicit Substitutions. In *Conference Record of 21st ACM Symposium on Principles of Programming Languages*, pages 60–69, 1994.

[39] John McCarthy. A Basis for a Mathematical Theory of Computation. In Braffort and Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[40] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1989.

[41] Alan Mycroft. The Theory and Practice of Transforming Call-by-Need Into Call-by-Value. In B. Robinet, editor, *International Symposium on Programming*, pages 269–81, 1980.

[42] Flemming Nielson. Program Transformations in a Denotational Setting. *ACM Transactions on Programming Languages*, 7(3):359–79, July 1985.

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

[44] Gordon D. Plotkin. Call-by-Name, Call-by-Value and the λ-calculus. *Theoretical Computer Science*, 1:125–59, 1975.

[45] Amr Sabry and Matthias Felleisen. Is Continuation-Passing Useful for Data Flow Analysis? In *Proc. ACM SIGPLAN '94 Conf. on Programming Language Design and Implementation*, pages 1–12, 1994.

[46] Dana Scott. Data Types as Lattices. *SIAM Journal of Computing*, 5(3):522–87, 1976.

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

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

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

[50] Paul Steckler and Mitchell Wand. Tracking Available Values for Lightweight Closures (Summary). In Neil Jones and Carolyn Talcott, editors, *Proc. Atlantique Workshop on Semantics Based Program Manipulation*, pages 63–70, 1994. Available as DIKU Report No. 94/12, University of Copenhagen.

[51] Guy Lewis Steele. Rabbit: A Compiler for Scheme. Master's thesis, MIT, May 1978. MIT Artificial Intelligence Laboratory Technical Report 474.

[52] Guy L. Steele, Jr. *Common Lisp: The Language.* Digital Press, Burlington, MA, 2nd edition, 1990.

[53] Dan Stefanescu and Yuli Zhou. An Equational Framework for the Flow Analysis of Higher-Order Functions. In *Proc. 1994 ACM Symposium on Lisp and Functional Programming*, pages 318–27, 1994.

[54] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[55] Robert E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, 1983.

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

[57] Mitchell Wand. Specifying the Correctness of Binding-Time Analysis. In *Conference Record of 20th ACM Symposium on Principles of Programming Languages*, pages 137–43, 1993.

[58] Mitchell Wand. Specifying the Correctness of Binding-Time Analysis. *Journal of Functional Programming*, 3(3):365–87, 1993.

[59] Mitchell Wand and Paul Steckler. Selective and Lightweight Closure Conversion. In *Conference Record of 21st ACM Symposium on Principles of Programming Languages*, pages 435–45, 1994.

[60] Glynn Winskel. *The Formal Semantics of Programming Languages.* Foundations of Computing Series. MIT Press, Cambridge, MA, 1993.

[61] David A. Wright. A New Technique for Strictness Analysis. In *Proc. TAPSOFT '91*, pages 235–58, 1991.