Deriving Target Code as a Representation of Continuation Semantics

by

Mitchell Wand
Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT NO. 94

DERIVING TARGET CODE AS A REPRESENTATION OF CONTINUATION SEMANTICS

MITCHELL WAND

JUNE, 1980

(REVISED: SEPTEMBER, 1981)

To appear: ACM Trans. on Prog. Lang. and Sys.

This material is based upon work supported by the National Science Foundation under Grant MCS79-04183.

Deriving Target Code As a Representation
Of Continuation Semantics

ABSTRACT: We extend Reynolds' technique for deriving interpreters to derive compilers from continuation semantics. The technique starts by eliminating λ-variables from the semantic equations through the introduction of special-purpose combinators. The semantics of a program phrase may be represented by a term built from these combinators. We then use associative and distributive laws to simplify the terms. Last, we build a machine to interpret the simplified terms as the functions they represent. The combinators reappear as the instructions of this machine. We illustrate the technique with three examples.

Key Words and Phrases: Compilers, Semantics, Continuations, Combinators, λ-calculus, code generation.

CR Categories (old scheme) 4.22, 4.12, 5.24

CR Categories (new scheme)
C.3.1 - Semantics
C.3.4 - Compilers
E.3.3 - Denotational Semantics
E.3.1 - Lambda calculus and related systems

1. Introduction

In this paper, we shall attack the question of how a denotational semantics for a language is related to an implementation of that language. Typically, one constructs the semantics of a target machine and a (suitably abstract) compiler, and proves a congruence between the two different semantics [12].

Our approach is quite different. Starting with a continuation semantics for the source language, we construct, via a series of transformations and representation decisions, a target machine and a compiler. A typical semantics has functionality

$$\mathcal{P} : \text{Pgms} \to [\text{Inputs} \to \text{Outputs}]$$

A compiler/target machine, on the other hand, uses the pair of functions:

$$\mathit{Compile} : \text{Pgms} \to \text{Reps}$$

$$\mathit{Machine} : \text{Reps} \to [\text{Inputs} \to \text{Outputs}]$$

where Reps is some domain of representations of functions Inputs → Outputs. The purpose of the machine is thus to interpret these function representations.

One may then consider a spectrum of tradeoffs between compilers and machines. Reynolds [20] showed how one could analyze a semantics and arrive at a tree-structured representation; the resulting machine looks like an interpreter. Wand [34] considered how one could use cleverer representations of functions. We use these ideas to arrive at representations which look like typical assembly code, with machines that look like typical "abstract language processors " [18]. Such an abstract language processor may be implemented either by emitting appropriate code sequences for the "real" machine or by interpreting the language-machine code; either may be done at either the conventional or microcode levels [29].

This work also owes a great deal to Mosses [16], which contributed the key idea that one should work with not the language but the semantic algebra -- that is, the abstract data type which is the target of the original semantic function. If one correctly implements that algebra, then one has correctly implemented the language.

Our primary concern in this paper is the development of some heuristics for analyzing the compilation process. Hence we are not concerned with the automatic analysis of semantic descriptions [7, 23], nor are we concerned with the software engineering problem of verifying actual compilers (as opposed to compilation algorithms). Likewise, we shall only hint at methods of proof; we plan to report on this topic elsewhere.

We shall present three examples in this paper. The first, addition expressions, gives a simple introduction to these techniques. For the second, a language with procedures, we derive a simple stack processor similar to the SECD machine [9]. For our third example, we derive another processor for this language, in which lexical scoping is used to optimize the machine structure.

Section 2 gives an introduction to our methodology. Section 3 shows how addition expressions are treated, and compares the techniques of [20] to ours. Sections 4 and 5 give the two processors for the procedure example. Section 6 discusses correctness proofs, and in Section 7, we compare our techniques to previous work.

2. Methodology

We consider the semantics of a programming language to be given as an enrichment of some semantic algebra. That is, one starts with a collection of semantic domains and some operations for manipulating those elements, and then one adds appropriate syntactic domains and semantic valuations as new sorts and operators in the algebra. Thus, the original algebra might consist of sorts including

In		Inputs
Out		Outputs
C =	[In → Out]	Transformations

and an operator

$$\mathit{apply} : \mathrm{C} \times \mathrm{In} \to \mathrm{Out}$$

To extend this to a language semantics, one would add a new sort

Pgm	Programs

and a new operator

$$\mathcal{P} : \mathrm{Pgm} \to \mathrm{C}$$

The output of program p on input x is then given by the term $\mathit{apply}(\mathcal{P}(p),x)$. This is somewhat different from the conventional view, in which the syntactic domains belong to a different algebra from the semantic domain, and the valuation $\mathcal{P}$ appears as a homomorphism. We follow the conventional practice, however, of using particular algebras rather than equationally-specified ones (cf. [16, 33]). The reasons for this choice are discussed in Section 7.

An implementation of this semantics is a representation of the semantic algebra in the sense of Hoare [6]: an algebra of concrete values (the "implementation algebra") with a homomorphism (Hoare's "abstraction function") from the concrete values to the abstract ones (See Figure 2.1). We shall develop representations of C which look like machine code; the functions corresponding to *apply* will look like machines.

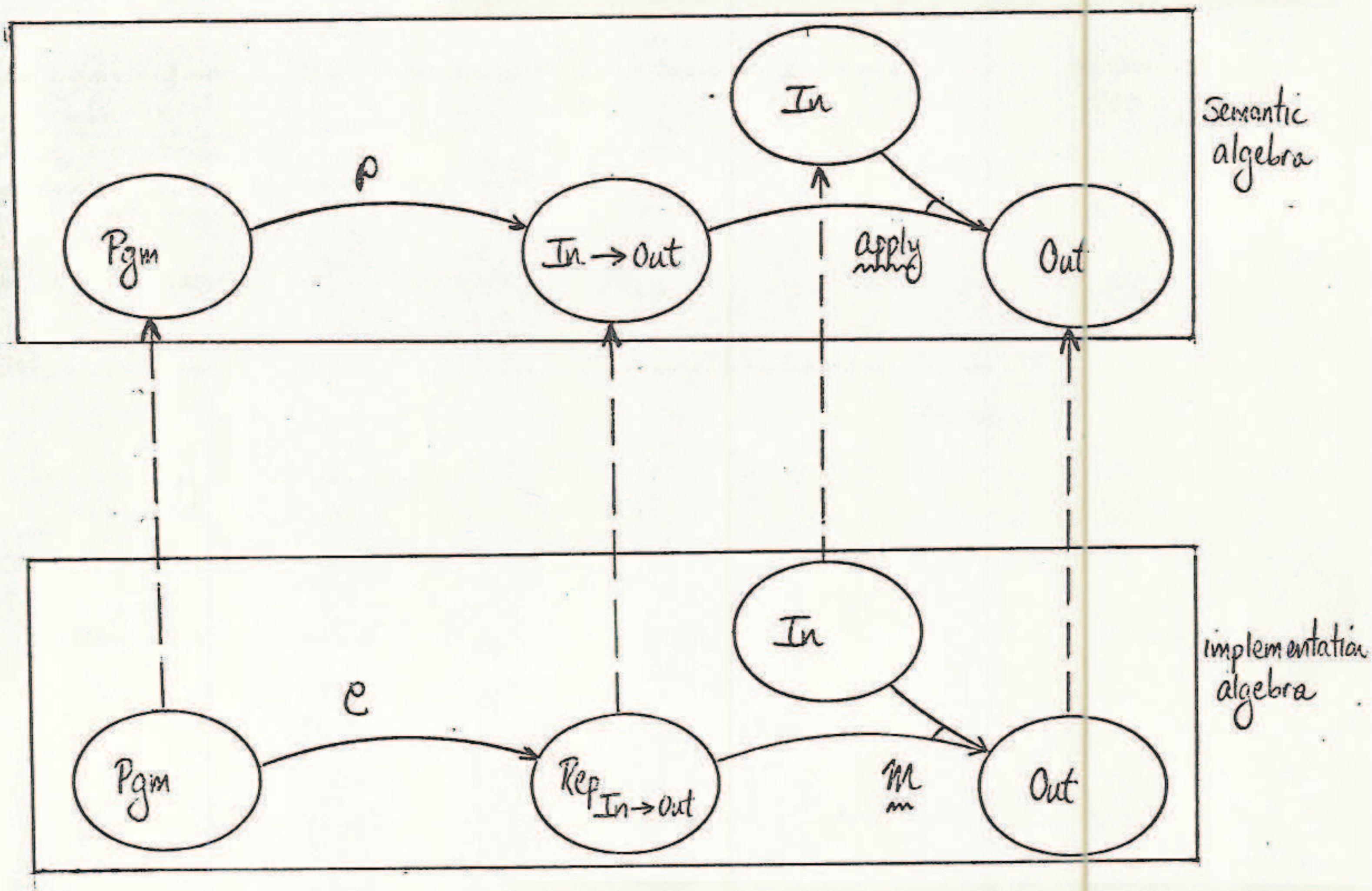
Semantic algebra
implementation algebra
Pgm
ρ
In → Out
In
apply
Out
Pgm
C
Rep In→Out
In
M
Out

Figure 2.1 The Relationship between the semantic and implementation algebras. The homomorphism goes up from lower-level concepts to the higher-level concepts they represent.

3. Addition Expressions

Let us consider addition expressions. Let V be some unspecified domain of values, and let "+" denote an arbitrary binary operator on V. Table 3.1 gives a continuation semantics for addition expressions over V. This familiar example illustrates our notation, which is largely standard [12, 26, 30]. We do not use open-face brackets ⟦...⟧ around syntactic arguments, but we do use conventional brackets in a manner similar to Quine's quasi-quotes [19, 26]. We also incorporate syntax into the domain definitions, as in the definition of Exp.

The issue is how to represent the domain C. The "obvious" representation would consist of lambda-expressions, which could then be interpreted by a lambda-calculus machine, e.g. [25]. We would like, however, to consider other representations. To do this, we replace the lambda terms by appropriate combinators.

For example, we might introduce auxiliary functions:

$$\mathit{halt} = \lambda v\sigma . v$$

$$\mathit{fetch} = \lambda I\kappa\sigma . \kappa(\sigma(I))\sigma$$

$$\mathit{add1} = \lambda\alpha\beta\kappa . \alpha(\lambda v_1 . \beta(\lambda v_2 . \kappa(v_1 + v_2)))$$

and rewrite our three equations as

$$\mathcal{P}e = \mathcal{E}e\,\mathit{halt}$$

$$\mathcal{E}I = \mathit{fetch}\,I$$

$$\mathcal{E}[e_1 + e_2] = \mathit{add1}(\mathcal{E}e_1)(\mathcal{E}e_2)$$

This suggests that we represent the result of $\mathcal{E}$ (an element of $K \to C$) as a tree structure:

Domains

V	Values
Id	Identifiers
$\mathrm{Exp} = \mathrm{Id} \mid [\mathrm{Exp} + \mathrm{Exp}]$	Expressions
$S = \mathrm{Id} \to V$	States
$C = S \to V$	Command continuations
$K = V \to C$	Expression continuations

Valuations

$\mathcal{P} : \mathrm{Exp} \to C$	Evaluate program
$\mathcal{E} : \mathrm{Exp} \to K \to C$	Evaluate expression

Equations

$$\mathcal{P}e = \mathcal{E}e(\lambda v\sigma.v)$$
$$\mathcal{E}I = \lambda\kappa\sigma.\kappa(\sigma(I))\sigma$$
$$\mathcal{E}[e_1 + e_2] = \lambda\kappa.\mathcal{E}e_1(\lambda v_1.\mathcal{E}e_2(\lambda v_2.\kappa(v_1+v_2)))$$

Table 3.1. Continuation Semantics for Addition Expressions

$$\mathcal{R}I = [\underline{\underline{fetch}}\ I]$$

$$\mathcal{R}[e_1 + e_2] = [\underline{\underline{add1}}\ \mathcal{R}e_1\ \mathcal{R}e_2]$$

This is the same analysis as [20], and, when carried through, yields an interpreter with a stack that represents C. All $\mathcal{R}$ does is give an internal representation of the original expression. Given a representation of the form $[\underline{\underline{add1}}\ \alpha\ \beta]$, the interpreter stacks β with an appropriate tag and proceeds to consider α. This behavior is quite different from that of the standard compiler for this kind of expression, a postfix representation with a stack machine, which stacks only values, never code.

Let us try a different route for variable elimination. We observe that in the equation for $\mathcal{E}[e_1 + e_2]$, the lambda-variables need to be routed to the operand parts of applications. This suggests that we may use a generalization of the combinator $\mathcal{B} = \lambda\alpha\beta x.\alpha(\beta x)$. For $k \geq 0$, let $\mathcal{B}_k$ be a combinator defined by:

$$\mathcal{B}_k(\alpha,\beta)x_1 \ldots x_k = \alpha(\beta x_1 \ldots x_k)$$

Then

$$\begin{aligned}\mathcal{E}[e_1 + e_2] &= \lambda\kappa.\mathcal{E}e_1(\lambda v_1.\mathcal{E}e_2(\lambda v_2.\kappa(v_1 + v_2)))\\ &= \mathcal{B}_1(\mathcal{E}e_1, \lambda\kappa v_1.\mathcal{E}e_2(\lambda v_2.\kappa(v_1+v_2)))\\ &= \mathcal{B}_1(\mathcal{E}e_1, \mathcal{B}_2(\mathcal{E}e_2, \lambda\kappa v_1 v_2.\kappa(v_1+v_2)))\end{aligned}$$

If we introduce $add = \lambda\kappa v_1 v_2.\kappa(v_1+v_2)$, we derive the following equations:

$$\mathcal{P}e = \mathcal{B}_0(\mathcal{E}e, \mathit{halt})$$

$$\mathcal{E}I = \mathit{fetch}\,I$$

$$\mathcal{E}[e_1 + e_2] = \mathcal{B}_1(\mathcal{E}e_1, \mathcal{B}_2(\mathcal{E}e_2, \mathit{add}))$$

This suggests a representation as a tree in which the internal nodes are labelled with **B**'s and the leaves are labelled with **halt**, **add**, and [**fetch** I]. This would not look promising, except for the following:

Proposition ($\mathcal{B}$ is right-associative): If $p \geq 1$, then

$$\mathcal{B}_k(\mathcal{B}_p(\alpha,\beta),\gamma) = \mathcal{B}_{k+p-1}(\alpha,\mathcal{B}_k(\beta,\gamma))$$

Proof:

$$\begin{aligned}
&\mathcal{B}_k(\mathcal{B}_p(\alpha,\beta),\gamma)x_1\ldots x_{k+p-1} \\
&= \mathcal{B}_p(\alpha,\beta)(\gamma x_1\ldots x_k)x_{k+1}\ldots x_{k+p-1} \\
&= \alpha(\beta(\gamma x_1\ldots x_k)x_{k+1}\ldots x_{k+p-1}) && \text{(since } p \geq 1\text{)} \\
&= \alpha(\mathcal{B}_k(\beta,\gamma)x_1\ldots x_k x_{k+1}\ldots x_{k+p-1}) \\
&= \mathcal{B}_{k+p-1}(\alpha,\mathcal{B}_k(\beta,\gamma))x_1\ldots x_{k+p-1} && \text{Q.E.D.}
\end{aligned}$$

One might have wished for full associativity in this proposition, and we shall show how to achieve it in Section 7. The present version, however, allows us to convert any such tree into a linear form, in which the left son of any **B**-node is either a [**fetch** I] or an **add**; the **halt** will be the rightmost leaf of the binary tree. This may be done by a function rot:

$$\mathit{rot}[\mathbf{B}\ [\mathbf{B}\ \alpha\ \beta]\ \gamma] = \mathit{rot}[\mathbf{B}\ \alpha\ [\mathbf{B}\ \beta\ \gamma]]$$

$$\mathit{rot}[\mathbf{B}\ [\mathbf{fetch}\ I]\ \beta] = [\mathbf{B}\ [\mathbf{fetch}\ I]\ \mathit{rot}\,\beta]$$

$$\mathit{rot}[\mathbf{B}\ \mathbf{add}\ \beta] = [\mathbf{B}\ \mathbf{add}\ \mathit{rot}\,\beta]$$

$$\mathit{rot}\,\mathbf{halt} = \mathbf{halt}$$

$$\mathit{rot}\,\mathbf{add} = \mathbf{add}$$

We may now write down the representation functions:

$$CP : \mathrm{Exp} \rightarrow \mathrm{Rep}_C$$

$$CE : \mathrm{Exp} \rightarrow \mathrm{Rep}_{K\rightarrow C}$$

$$CPe = \mathit{rot}[\underline{B}\ CEe\ \underline{\underline{halt}}]$$

$$CEI = [\underline{\underline{fetch}}\ I]$$

$$CE[e_1 + e_2] = \mathit{rot}[\underline{B}\ CEe_1\ [\underline{B}\ CEe_2\ \underline{\underline{add}}]]$$

We have suppressed the subscripts on the B's; we shall see that they may be reconstructed from context. Figure 3.1 shows a typical expression and its representation. Note that this is precisely the standard postfix code!

It will be convenient to have a notation for the function denoted by a tree. We use the symbol "$^{\vee}$" (the haček) for this. We use the haček either as a prefix or superfix operator; thus $^{\vee}[\underline{B}\ \underline{\underline{add}}\ \beta] = B(\mathit{add},\check{\beta})$. The "v" operation is the abstraction function in Figure 2.1, and may be constructed using the initial algebra property of trees [5].

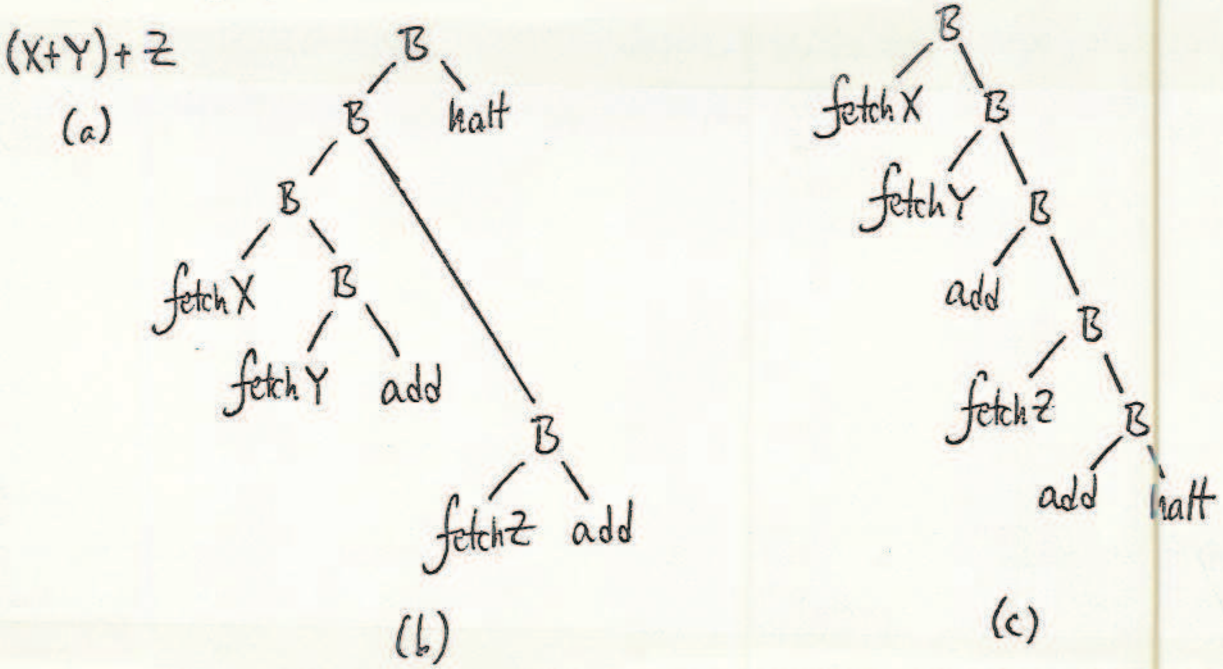

Figure 3.1 (a) an addition expression; (b) internal representation without *rot*; (c) internal representation using *rot*.

Now all we must do is write a function $\mathcal{M}\text{:Rep}_C \to S \to V$ which decodes these representations. Unfortunately, the lambda expressions associated with the fragments of code turn out to be not only of types C and $K = V \to C$, but of type $V^n \to C$ for $n \geq 0$. Intuitively, n corresponds to the number of operands to be absorbed from the stack, i.e. the length of the stack. We must write a family of functions

$$\mathcal{M}_n\text{: Rep}_n \to V^n \to C$$

where Rep_n refers to the domain of representations of $V^n \to C$. Rep_n consists of trees whose left sons are [$\underline{\underline{\text{fetch}}}$ I] or $\underline{\underline{\text{add}}}$, and whose right sons are similar trees or the leaf $\underline{\underline{\text{halt}}}$. We first consider what $\mathcal{M}_n$ should do with a tree whose left son is [$\underline{\underline{\text{fetch}}}$ I]:

$$\mathcal{M}_n[\underline{\underline{\text{B}}}\ [\underline{\underline{\text{fetch}}}\ \text{I}]\ \beta]x_1\ldots x_n\sigma \tag{1}$$

$$= \mathcal{B}_n(\mathit{fetch}\text{I},\check{\beta})x_1\ldots x_n\sigma \tag{2}$$

$$= \mathit{fetch}\text{I}(\check{\beta}x_1\ldots x_n)\sigma \tag{3}$$

$$= (\check{\beta}x_1\ldots x_n)(\sigma(\text{I}))\sigma \tag{4}$$

$$= \mathcal{M}_{n+1}\beta x_1\ldots x_n(\sigma(\text{I}))\sigma \tag{5}$$

Here we have expanded $C = S \to V$, so $\mathcal{M}_n\text{: Rep}_n \to V^n \to S \to V$. $\mathcal{M}_n$'s job is to take a representation of an element of $V^n \to C$, a tuple in V^n, a state σ, and apply the function to get an answer. Since we know what each of the tags in the tree denotes, we know that the desired answer is (2). Since $\mathit{fetch}\text{I}$ is of type $K \to C$, and $\mathcal{B}(\mathit{fetch}\text{I},\beta)$ is of type $V^n \to C$, we conclude that β must be of type $V^n \to K$ and the subscript on $\mathcal{B}$ must be n. (Note that $\mathcal{B}$ is just a flavor of composition). This gets us to (3). The definition of *fetch* gets us to (4), which we recognize as equivalent to (5).

We may analyze the behavior of $\underline{\underline{add}}$ in the same way:

$$
\begin{aligned}
&\mathcal{M}_n[\underline{\underline{B}}\ \underline{\underline{add}}\ \beta]x_1\ldots x_n\sigma \\
&= \mathcal{B}(add,\check{\beta})x_1\ldots x_n\sigma \\
&= add(\check{\beta}x_1\ldots x_{n-2})x_{n-1}x_n\sigma && (6) \\
&= \check{\beta}x_1\ldots x_{n-2}(x_{n-1}+x_n)\sigma && (7) \\
&= \mathcal{M}_{n-1}\beta x_1\ldots x_{n-2}(x_{n-1}+x_n)\sigma
\end{aligned}
$$

We need to show that the subscript on $\mathcal{B}$ was n-2. If $\mathcal{B}$'s subscript was p, then β must be of type $V^p \to K$ so that $(\beta x_1\ldots x_p)$ can be used as an argument to *add* in line 6. Since *add* is of type $K \to V^2 \to C$, $\mathcal{B}_p(add,\beta)x_1\ldots x_p$ must be of type $V^2 \to C$. Therefore $n = p + 2$. To get to line (7), we use the definition of *add*. Last,

$$
\begin{aligned}
&\mathcal{M}_1\underline{\underline{halt}}x_1\sigma \\
&= halt\,x_1\sigma \\
&= x_1 \qquad \text{(Definition of } halt\text{).}
\end{aligned}
$$

The result is summarized in Table 3.2.

What have we accomplished? The function in Table 3.2 is in iterative form [10] and is therefore easily realizable using a finite state control. The machine looks at the left son of the tree. If it is a $\underline{\underline{fetch}}$, the appropriate value is added to the right-hand end of the x's; if it is an $\underline{\underline{add}}$, then the two rightmost values are removed and replaced with their sum. In either case, the machine goes on to the right son. The sequence of values $x_1\ldots x_n$ behaves as a stack with the top at the right.

Our M is thus unmasked as the familiar stack machine for postfix expressions, with three instructions: **fetch**, **add**, and **halt**, and a familiar fetch-execute cycle of operation. The combinators B_k, which we originally conceived simply as argument-steerers, have become sequencers (similar to Mosses' $\vec{K}$ [16]). What is new is the fact that we derived the structure of the machine directly from the semantics, without explicit use of operational considerations. Consequently, the correctness of the machine and compiler may be obtained much more easily. This is in contrast to the conventional approach, in which the language and machine are given independently, and in which their correspondence must be painstakingly established.

$$M_n[\mathbf{B}\ [\mathbf{fetch}\ I]\ \beta]x_1\ldots x_n\sigma = M_{n+1}\beta x_1\ldots x_n(\sigma I)\sigma$$
$$M_{n+2}[\mathbf{B}\ \mathbf{add}\ \beta]x_1\ldots x_{n+2}\sigma = M_{n+1}\beta x_1\ldots x_n(x_{n+1}+x_{n+2})\sigma$$
$$M_1\mathbf{halt}\,x_1\sigma = x_1$$

Table 3.2 Stack machine for addition expressions

4. Procedures

We now apply this technique to a less trivial example. Table 4.1 shows the semantics of a simple expression language with procedures and input-output. Expressions are either identifiers, λ-expressions (procedures), input-output statements, or applications. Values are either basic values (which are left unspecified) or function values $(V \to K \to C)$. States, an unspecified domain, abstracts the state of the input-output streams. Also left unspecified are:

initenv : Env	an initial environment
halt : $K = V \to S \to A$	an initial expression continuation, which extracts an "answer" from a value and a state
doio : $K \to V \to C$	an input-output operation which takes a value, and a state, and returns a value and a possibly altered state

A procedure expects a parameter and an expression continuation when it is invoked. In an application, the operator is evaluated first, the operand is evaluated second, and then the function is applied. The *doio* operation provides a way to alter the state. *doio*$\kappa v\sigma$ is generally equal to $\kappa v'\sigma'$, where v' is a suitable value to be returned to DOIO's caller, and σ' is a possibly altered state. The functionality of *doio*, however, allows it to initiate an error and ignore the continuation κ. The ability to model such behavior is an important advantage of continuation semantics.

Domains

Id	Identifiers
$\mathrm{Exp} ::= \mathrm{Id} \mid [\lambda\ \mathrm{Id}\ \mathrm{Exp}]$	Expressions
$\mid [\mathrm{DOIO}\ \mathrm{Exp}]$	
$\mid [\mathrm{Exp}\ \mathrm{Exp}]$	
B	Basic Values
$V = B + [V \to K \to C]$	Values
$\mathrm{Env} = \mathrm{Id} \to V$	Environments
S	States
A	Answers
$C = S \to A$	Command continuations
$K = V \to C$	Expression continuations

Valuations

$\mathcal{P} : \mathrm{Exp} \to C$

$\mathcal{E} : \mathrm{Exp} \to \mathrm{Env} \to K \to C$

Equations

$$\mathcal{P}e = (\mathcal{E}e\,\mathit{initenv})\mathit{halt}$$
$$\mathcal{E}I\rho\kappa = \kappa(\rho I)$$
$$\mathcal{E}[\lambda\ I\ e]\rho\kappa = \kappa(\lambda a\kappa'.\mathcal{E}e(\rho[a/I])\kappa')$$
$$\mathcal{E}[\mathrm{DOIO}\ e]\rho\kappa = \mathcal{E}e\rho(\lambda v.\mathit{doio}\kappa v)$$
$$\mathcal{E}[e_1\ e_2]\rho\kappa = \mathcal{E}e_1\rho(\lambda f.\mathcal{E}e_2\rho(\lambda a.fa\kappa))$$

Table 4.1 A Simple, Almost-Applicative Language

For this example, we introduce the family of combinators $\mathcal{D}_k$:

$$\mathcal{D}_k(\alpha,\beta)\rho\kappa x_1\ldots x_k = \alpha\rho(\beta\rho\kappa x_1\ldots x_k)$$

The $\mathcal{D}_k$'s are right-associative for all $k \geq 0$, under the transformation $\mathcal{D}_k(\mathcal{D}_p(\alpha,\beta),\gamma) = \mathcal{D}_{k+p}(\alpha,\mathcal{D}_k(\beta,\gamma))$. We now eliminate all the λ-terms in the definition, in the by-now familiar way; the operators are designed to take advantage of the $\mathcal{D}$'s:

$$\mathit{fetch}I = \lambda\rho\kappa.\kappa(\rho I)$$
$$\mathit{pushclosure}I\alpha = \lambda\rho\kappa.\kappa(\lambda a\kappa'.\alpha(\rho[a/I]\kappa'))$$
$$\mathit{iot} = \lambda\rho\kappa v\sigma.\mathit{doio}\kappa v\sigma = \lambda\rho.\mathit{doio}$$
$$\mathit{apply} = \lambda\rho\kappa fa.fa\kappa$$
$$\mathit{return} = \lambda\rho\kappa v.\kappa v = \lambda\rho\kappa.\kappa$$

With these operators, we may rewrite the equations as:

$$\mathcal{P}e = \mathcal{D}_0(\mathcal{E}e, \mathit{return})(\mathit{initenv})(\mathit{halt})$$

$$\mathcal{E}I = \mathit{fetch}I$$

$$\mathcal{E}[\lambda\ I\ e] = \mathit{pushclosure}I(\mathcal{D}_0(\mathcal{E}e, \mathit{return}))$$

$$\mathcal{E}[\mathrm{DOIO}\ e] = \mathcal{D}_0(\mathcal{E}e, \mathit{iot})$$

$$\mathcal{E}[e_1\ e_2] = \mathcal{D}_0(\mathcal{E}e_1, \mathcal{D}_2(\mathcal{E}e_2, \mathit{apply}))$$

Again, the $\mathcal{D}$'s are right-associative, so we can associate them to the right, just as we did in the previous example, to get the following compiler:

$$\mathcal{CP}e = \mathit{rot}[\underline{\underline{\mathtt{D}}}\ \mathcal{CE}e\ \underline{\underline{\mathtt{return}}}]$$

$$\mathcal{CE}I = [\underline{\underline{\mathtt{fetch}}}\ I]$$

$$\mathcal{CE}[\lambda\ I\ e] = [\underline{\underline{\mathtt{pushclosure}}}\ I\ \mathit{rot}[\underline{\underline{\mathtt{D}}}\ \mathcal{CE}e\ \underline{\underline{\mathtt{return}}}]]$$

$$\mathcal{CE}[\mathrm{DOIO}\ e] = \mathit{rot}[\underline{\underline{\mathtt{D}}}\ \mathcal{CE}e\ \underline{\underline{\mathtt{iot}}}]$$

$$\mathcal{CE}[e_1\ e_2] = \mathit{rot}[\underline{\underline{\mathtt{D}}}\ \mathcal{CE}e_1\ [\underline{\underline{\mathtt{D}}}\ \mathcal{CE}e_2\ \underline{\underline{\mathtt{apply}}}]]$$

where rot performs the association to the right, as before. The result will be code sequences in the following grammar:

$$\mathrm{seq} ::= \underline{\underline{\mathtt{return}}} \mid [\underline{\underline{\mathtt{D}}}\ \mathrm{instr}\ \mathrm{seq}]$$

$$\mathrm{instr} ::= \underline{\underline{\mathtt{iot}}} \mid \underline{\underline{\mathtt{apply}}} \mid [\underline{\underline{\mathtt{fetch}}}\ I] \mid [\underline{\underline{\mathtt{pushclosure}}}\ I\ \mathrm{seq}]$$

As before, an instruction sequence is a representation of a function $\mathrm{Env} \to V^n \to C$, so we must come up with a family of functions for interpreting these sequences.

The desired relation is:

$$\mathcal{M}_k \beta\rho\kappa x_1 \ldots x_k \sigma = \check{\beta}\rho\check{\kappa} x_1 \ldots x_k \sigma$$

The machine will have a "program counter" (β), an "environment register" (ρ), a continuation (κ), a local stack ($x_1,\ldots x_k$), and a state (σ). We initialize the machine with $M_0\beta(\mathit{initenv})\,\mathbf{halt}$.

We begin the analysis of the machine by considering the fetch cycle, with **iot**:

$$\begin{aligned}
&M_k[\mathbf{D}\ \mathbf{iot}\ \beta]\rho\kappa x_1\ldots x_k\sigma\\
&\quad = \mathcal{D}_{k-1}(\mathit{iot},\check{\beta})\rho\check{\kappa}x_1\ldots x_k\sigma && \text{(intention)}\\
&\quad = \mathit{iot}\rho(\check{\beta}\rho\kappa x_1\ldots x_{k-1})x_k\sigma && \text{(Def'n of } \mathcal{D}\text{)}\\
&\quad = \mathit{doio}(\check{\beta}\rho\check{\kappa}x_1\ldots x_{k-1})x_k\sigma && \text{(Def'n of } \mathit{iot}\text{)}
\end{aligned}$$

Let us assume that this application of *doio* behaves normally. Then for some appropriate v' and σ', this quantity is equal to

$$\begin{aligned}
&\quad = (\check{\beta}\rho\check{\kappa}x_1\ldots x_{k-1})v'\sigma'\\
&\quad = M_k\beta\rho\kappa x_1\ldots x_{k-1}v'\sigma'.
\end{aligned}$$

Thus, execution of an **iot** instruction alters the top of the stack and the state. An argument like the one for **add** guarantees that the subscript on the $\mathcal{D}$ is k-1. We next consider **pushclosure**:

$$\begin{aligned}
&M_k[\mathbf{D}\ [\mathbf{pushclosure}\ I\ \alpha]\ \beta]\rho x_1\ldots x_k\sigma\\
&= \mathcal{D}_k(\mathit{pushclosure}I\alpha,\beta)\rho x_1\ldots x_k\sigma\\
&= \mathit{pushclosure}I\alpha\rho(\beta\rho x_1\ldots x_k)\sigma\\
&= \beta\rho x_1\ldots x_k(\lambda a\kappa'.\alpha\rho[a/I]\kappa')\sigma
\end{aligned}$$

At this point we are temporarily stymied, since this equation requires us to push a function on the stack, and our machines so far have not had the capability to deal with functions except through their representations. We, therefore change our stack to store not items from V, but items from Rep_W, where

$$\mathrm{Rep}_W = V + \mathrm{Rep}_{V\to K\to C}$$

and we assume our machine has some mechanism for distinguishing "real" basic values, "real" function values, and "represented" function values. Since we shall also need to bind identifiers to represented function values, we can no longer use real environments, but must use representations, which we choose as

$$\mathrm{Rep}_{\mathrm{Env}} = \mathrm{Id} \to \mathrm{Rep}_{W}$$

(We shall consider other choices in the next section). The desired relation is thus $M_k \beta\rho\kappa x_1 \ldots x_k \sigma = \check{\beta}\check{\rho}\check{\kappa}\check{x}_1 \ldots \check{x}_k \sigma$.

We now define

$$\mathit{closure}\,\mathrm{I}\alpha\rho = \lambda a\kappa'.\alpha\rho[a/\mathrm{I}]\kappa'$$

and finish the previous calculation to deduce:

$$\begin{aligned} & M_k[\underline{\underline{\mathrm{D}}}\ [\underline{\underline{\mathrm{pushclosure}}}\ \mathrm{I}\ \alpha]\ \beta]\rho\kappa x_1 \ldots x_k \sigma \\ &\quad = \check{\beta}\check{\rho}\check{\kappa}\check{x}_1 \ldots \check{x}_k(\mathit{closure}\,\mathrm{I}\alpha\rho)\sigma \\ &\quad = M_{k+1}\beta\rho\kappa x_1 \ldots x_k[\underline{\underline{\mathrm{closure}}}\ \mathrm{I}\ \alpha\ \rho]\sigma \end{aligned}$$

We next do $\underline{\underline{\mathrm{apply}}}$:

$$\begin{aligned} & M_k[\underline{\underline{\mathrm{D}}}\ \underline{\underline{\mathrm{apply}}}\ \beta]\rho\kappa x_1 \ldots x_k \sigma \\ &\quad = \mathcal{D}_{k-2}(\mathit{apply},\check{\beta})\check{\rho}\check{\kappa}\check{x}_1 \ldots \check{x}_k \sigma \\ &\quad = \mathit{apply}\,\check{\rho}(\check{\beta}\check{\rho}\check{\kappa}\check{x}_1 \ldots \check{x}_{k-2})\check{x}_{k-1}\check{x}_k \sigma \\ &\quad = \check{x}_{k-1}\check{x}_k(\check{\beta}\check{\rho}\check{x}_1 \ldots \check{x}_{k-2})\sigma \end{aligned}$$

We then have three cases, depending on the value of x_{k-1}. If it is a "represented" function value, we continue with:

$$\begin{aligned} & M_k[\underline{\underline{\mathrm{D}}}\ \underline{\underline{\mathrm{apply}}}\ \beta]\rho\kappa x_1 \ldots x_{k-2}[\underline{\underline{\mathrm{closure}}}\ \mathrm{I}\ \alpha\ \rho']x_k \sigma \\ &\quad = \mathit{closure}\,\mathrm{I}\check{\alpha}\check{\rho}'\check{x}_k(\check{\beta}\check{\rho}\check{\kappa}\check{x}_1 \ldots \check{x}_{k-2})\sigma \\ &\quad = \check{\alpha}(\check{\rho}'[\check{x}_k/\mathrm{I}])(\check{\beta}\check{\rho}\check{\kappa}\check{x}_1 \ldots \check{x}_{k-2})\sigma \\ &\quad = M_0\alpha(\rho'[x_k/\mathrm{I}])(\beta\rho\kappa x_1 \ldots x_{k-2})\sigma \end{aligned}$$

Thus, we should start the body α of the procedure with nothing on the stack, and with the expression continuation $\beta\rho\kappa x_1 \ldots x_{k-2}$. Again, we define an appropriate combinator:

$$\mathit{retpt}_p \beta\rho\kappa x_1 \ldots x_p = \beta\rho\kappa x_1 \ldots x_p$$

to yield:

$$M_k[\mathbf{D}\ \mathbf{apply}\ \beta]\rho\kappa x_1 \ldots x_{k-2}[\mathbf{closure}\ I\ \alpha\ \rho']x_k\sigma$$
$$= M_0\alpha(\rho'[x_k/I])[\mathbf{retpt}_{k-2}\ \beta\ \rho\ \kappa\ x_1 \ldots x_{k-2}]\sigma$$

The machine has managed to stack the return address β and the calling environment ρ, and has entered the code α of the procedure with the appropriately extended environment $\rho'[x_k/I]$. Two cases remain. If x_{k-1} is a "real" value in $V \to K \to C$, then the machine must be able to apply it. Such values correspond to primitive operations which were in the initial environment. The development in this case will be analogous to **iot**. Last, any other value causes an error, which can be treated in a similar fashion.

fetch works as before:

$$M_k[\mathbf{D}\ [\mathbf{fetch}\ I]\ \beta]\rho\kappa x_1 \ldots x_k\sigma$$
$$= \mathcal{D}_k(\mathit{fetch}I,\check{\beta})\check{\rho}\check{\kappa}\check{x}_1 \ldots \check{x}_k\sigma$$
$$= \mathit{fetch}I\check{\rho}(\check{\beta}\check{\rho}\check{\kappa}\check{x}_1 \ldots \check{x}_k)\sigma$$
$$= \check{\beta}\check{\rho}\check{x}_1 \ldots \check{x}_k(\check{\rho}I)\sigma$$
$$= M_{k+1}\beta\rho\kappa x_1 \ldots x_k(\rho I)\sigma$$

This leaves **return**:

$$M_2\mathbf{return}\rho\kappa x\sigma$$
$$= \mathit{return}\check{\rho}\check{\kappa}\check{x}\check{\sigma}$$
$$= \check{\kappa}\check{x}\check{\sigma}$$

If κ is a **retpt**, then we can proceed:

$$M_1\mathbf{return}\rho[\mathbf{retpt}_p\ \beta\ \rho'\ \kappa\ x_1\ \ldots\ x_p]v\sigma$$

$$= \check{\beta}\check{\rho}'\check{\kappa}\check{x}_1\ldots\check{x}_p\check{v}\sigma$$

$$= M_{p+1}\beta\rho'\kappa x_1\ldots x_p v\sigma$$

so the value is pushed onto the calling routine's stack. Another possibility is that basic values include continuations (as in SCHEME [28, 35]), in which case the machine would have to handle this as it does other primitive operations. If κ is **halt**, then

$$M_1\mathbf{return}\rho\,\mathbf{halt}\,v\sigma$$

$$= \mathit{halt}\,v\sigma$$

and the machine halts. Because Rep_K is built up solely using **halt** and **retpt**, these are the only possibilities. The machine is summarized in Table 4.2. Note that we could recast our derivation into a proof by subgoal induction [15] or fixpoint induction that if $M_n\beta\rho\kappa x_1\ldots x_n\sigma$ halts, its answer is equal to $\check{\beta}\check{\rho}\check{\kappa}\check{x}_1\ldots\check{x}_n\sigma$, as desired.

We close this section by doing one peephole optimization on this machine. Consider tail-recursive calls [28]. Our machine does not do these iteratively, because the code generated ends with [**D** **apply** **return**], and a **retpt** will be generated on the stack. We can, however, avoid this by noting that the subscript of $\mathcal{D}$ must be 0, and calculating:

$M_2[\underline{\underline{D}}\ \underline{\underline{apply}}\ \underline{\underline{return}}]\rho\kappa fa$

$= \mathcal{D}_0(\mathit{apply},\mathit{return})\rho\kappa fa$

$= \mathit{apply}\rho(\mathit{return}\rho\kappa)fa$

$= \mathit{apply}\rho\kappa fa$ (Definition of *return*)

and continuing as before. Thus the stacked continuation will be used instead of a new one being created. We may patch the machine accordingly. This gives a formal justification for the iterative interpretation of tail recursion. Alternatively, we may eliminate the padding of procedure bodies with $\underline{\underline{return}}$ by writing the semantics of $[\lambda\ I\ e]$ as

$$\mathcal{E}[\lambda\ I\ e] = \mathit{pushclosure}I(\mathcal{E}e)$$

The machine would then have to deal with arbitrary instructions as the right-hand argument to $\underline{\underline{D}}$. This, however, would have substantially complicated the presentation.

$M_k[\underline{\underline{D}}\ [\underline{\underline{fetch}}\ I]\ \beta]\rho\kappa x_1\ldots x_k\sigma = M_{k+1}\beta\rho\kappa x_1\ldots x_k(\rho I)\sigma$

$M_k[\underline{\underline{D}}\ [\underline{\underline{pushclosure}}\ I\ \alpha]\ \beta]\rho\kappa x_1\ldots x_k\sigma$

$= M_{k+1}\beta\rho\kappa x_1\ldots x_k[\underline{\underline{closure}}\ I\ \alpha\ \rho]\sigma$

$M_{k+2}[D\ \underline{\underline{apply}}\ \beta]\rho\kappa x_1\ldots x_k[\underline{\underline{closure}}\ I\ \alpha\ \rho']a\sigma$

$= M_0\alpha(\rho'[a/I])[\underline{\underline{retpt}}_k\ \beta\ \rho\ \kappa\ x_1\ \ldots\ x_k]\sigma$

$M_{k+1}[\underline{\underline{D}}\ \underline{\underline{iot}}\ \beta]\rho\kappa x_1\ldots x_{k+1}\sigma = M_{k+1}\beta\rho\kappa x_1\ldots x_k v'\sigma'$

(v',σ' as described in text)

$M_1\underline{\underline{return}}\rho[\underline{\underline{retpt}}_k\ \beta\ \rho'\ \kappa\ x_1\ \ldots\ x_k]v\sigma$

$= M_{k+1}\beta\rho'\kappa x_1\ldots x_k v\sigma$

$M_1\underline{\underline{return}}\rho\underline{\underline{halt}}v\sigma = \mathit{halt}v\sigma$

Table 4.2 Machine for the almost-applicative language

5. Lexical Scoping

The language we treated in the last section is "lexically scoped." It is well-known that in such a language, identifiers may be bound at compile time, that is, the symbolic identifiers in *fetch* instructions may be replaced by displacements off pointers in a display [18], and these displacements may be computed at compile time. We shall demonstrate this property using our techniques.

In the previous section, we let $Rep_{Env} = Id \to Rep_W$. In this section, we shall see how environments are built using the combinator

$$\mathit{ext} I \rho a = \lambda J.\text{if } J=I \text{ then } a \text{ else } \rho J$$

We will use this combinator to get a better representation of environments.

We begin by changing the formulation of $\mathcal{P}$ from

$$\mathcal{P}e = \mathcal{D}_0(Ee,\mathit{return})(\mathit{initenv})(\mathit{halt})$$

to $$\mathcal{P}e = \mathcal{B}_0(\mathcal{D}_0(Ee,\mathit{return}),\mathit{initenv})\mathit{halt}$$

Our plan for representing $\mathcal{P}e$ is to associate the $\mathcal{D}$'s to the right as before, and then to distribute the $\mathcal{B}_0$ to get the environment information to the individual instructions where it can be used. To formulate a sufficiently general distribution law, however, we must introduce a new family of combinators $\mathcal{S}_{pn}$ defined by

$$\mathcal{S}_{pn}(\alpha,\beta)a_1\ldots a_p\kappa x_1\ldots x_n = \alpha a_1\ldots a_p(\beta a_1\ldots a_p\kappa x_1\ldots x_n)$$

We may then state the distribution law as:

Proposition $\mathcal{B}_p(\mathcal{D}_k(\alpha,\beta),\gamma) = \mathcal{S}_{pk}(\mathcal{B}_p(\alpha,\gamma),\mathcal{B}_p(\beta,\gamma)).$

Using this proposition, we can push the $\mathcal{B}$'s inwards. We next investigate what happens when a $\mathcal{B}$ reaches an instruction. We turn first to the case of closures, since that is where we expect environmental information to be built up. The following combinators will be useful:

$$\mathit{push}_p f = \lambda a_1 \ldots a_p \kappa . \kappa (f a_1 \ldots a_p)$$

$$\mathit{ext} I \rho a = \lambda J . \text{if } J = I \text{ then } a \text{ else } \rho J$$

$$\mathit{get} I \rho = \rho I$$

Now,
$$\begin{aligned} \mathcal{E}[\lambda\ I\ e] &= \lambda \rho \kappa . \kappa (\lambda a \kappa' . \mathcal{E} e (\mathit{ext} I \rho a) \kappa') \\ &= \lambda \rho \kappa . \kappa (\lambda a . \mathcal{E} e (\mathit{ext} I \rho a)) \\ &= \lambda \rho \kappa . \kappa (\mathcal{B}_1 (\mathcal{E} e , \mathit{ext} I \rho)) \\ &= \mathit{push}_1 (\lambda \rho . \mathcal{B}_1 (\mathcal{E} e , \mathit{ext} I \rho)) \\ &= \mathit{push}_1 (\mathcal{B}_2 (\mathcal{E} e , \mathit{ext} I)) \end{aligned}$$

Adopting this version of $\mathcal{E}[\lambda\ I\ e]$, we can then see what happens when $\mathcal{B}_p$ reaches it:

$$\begin{aligned} &\mathcal{B}_p (\mathit{push}_1 (\mathcal{B}_2 (\mathcal{E} e , \mathit{ext} I)) , \tau) \\ &= \lambda a_1 \ldots a_p . \mathit{push}_1 (\mathcal{B}_2 (\mathcal{E} e , \mathit{ext} I)) (\tau a_1 \ldots a_p) \\ &= \lambda a_1 \ldots a_p \kappa . \kappa (\mathcal{B}_2 (\mathcal{E} e , \mathit{ext} I) (\tau a_1 \ldots a_p)) \\ &= \lambda a_1 \ldots a_p \kappa . \kappa (\mathcal{B}_1 (\mathcal{E} e , \mathit{ext} I (\tau a_1 \ldots a_p))) \\ &= \lambda a_1 \ldots a_p \kappa . \kappa (\mathcal{B}_{p+1} (\mathcal{E} e , \mathcal{B}_p (\mathit{ext} I , \tau)) a_1 \ldots a_p) \\ &= \mathit{push}_p (\mathcal{B}_{p+1} (\mathcal{E} e , \mathcal{B}_p (\mathit{ext} I , \tau))) \end{aligned}$$

The new environmental information $\mathcal{B}_p(\mathit{ext} I, \tau)$ can now be rubbed against the code for the procedure e. This also reveals what this "environmental information" is: nothing other than our old friend the symbol table, a function which takes a p-tuple of values and produces an environment. For convenience, define:

$$\mathit{table}_0 = \mathit{initenv}$$
$$\mathit{table}_{p+1} I\tau = \mathcal{B}_p(\mathit{ext}I,\tau)$$

Then the distribution law will be applied only to terms of the form $\mathcal{B}_p(\alpha,\tau)$, where $\tau = \mathit{table}_p I_p(\mathit{table}_{p-1} I_{p-1}(\ldots \mathit{table}_0 \ldots))$ and $I_p,\ldots,I_0$ is a sequence of not-necessarily-distinct identifiers. Letting τ continue to denote this table, it is easy to show that $\tau a_1 \ldots a_p J = a_j$, where $j = \max\{k \mid I_k = J\}$, provided this set is non-empty (i.e., J is found in symbol table τ), and $\tau a_1 \ldots a_p J = \mathit{initenv} J$ if J is not found in τ. Let

$$\mathit{selec}_{pj} a_1 \ldots a_p = a_j$$
$$\mathcal{K}_p v a_1 \ldots a_p = v$$

We may now see how a fetch utilizes the symbol table:

$$\begin{aligned}
&\mathcal{B}_p(\mathcal{E}I,\tau)\\
&\qquad = \mathcal{B}_p(\lambda\rho\kappa.\kappa(\rho I),\tau)\\
&\qquad = \mathcal{B}_p(\mathit{push}_1(\mathit{get}I),\tau)\\
&\qquad = \lambda a_1 \ldots a_p.\mathit{push}_1(\mathit{get}I)(\tau a_1 \ldots a_p)\\
&\qquad = \lambda a_1 \ldots a_p \kappa.\kappa(\tau a_1 \ldots a_p I)\\
&\qquad = \lambda a_1 \ldots a_p \kappa.\kappa(a_j) \qquad\qquad (j = \max\{k \mid I_k = I\})\\
&\qquad = \mathit{push}_p(\mathit{selec}_{pj})
\end{aligned}$$

Thus the identifier may be replaced by a displacement (the selec_{pj}). If I is not found in τ, then we finish the calculation as follows

$$\mathcal{B}_p(\mathcal{E}I,\tau)$$

$$= \lambda a_1 \ldots a_p \kappa . \kappa(\tau a_1 \ldots a_p I)$$

$$= \lambda a_1 \ldots a_p \kappa . \kappa(\mathit{initenv} I)$$

$$= \mathit{push}_p(K_p(\mathit{initenv} I))$$

We also need to deal with $\mathcal{B}_p(\alpha,\tau)$ where α is *apply*, *iot*, or *return*; we define

$$\mathit{apply}_p a_1 \ldots a_p \kappa f a = f a \kappa$$

Then

$$\mathcal{B}_p(\mathit{apply},\tau) = \mathit{apply}_p$$

and similarly for *iot* and *return*.

Now we are prepared to write the compiler. The compiler uses a three-step strategy:

(1) generate the "naive" code

(2) associate the $\mathcal{D}$'s to the right, using *rot*

(3) distribute the symbol table information to the instructions and perform variable binding.

Table 5.1 shows the semantic equations for the language, and the compiler is in Table 5.2. In a traditional compiler, of course, these steps are intertwined; one could obtain such an algorithm from ours by using pipelining techniques such as those in [3, 34]. As before, we have suppressed the subscripts in the representations since that information is readily available at run time. Figures 5.1 - 5.3 show the evolution of code in the compiler.

With the representation developed, we now must design a machine to interpret the representations. In general, an instruction sequence represents a function of the form

$$S_{pn}(\alpha,\beta): \quad V^p \to K \to V^n \to S \to A$$

where α is an instruction. We may deduce the behavior of the machine as before. The length of the display is p and the number of entries in the current stack frame is n. Again, we shall need to put both values and representations of functions and continuations on the stack, so the functionality required is

$$M_{pn}: \quad Rep_{pn} \to Rep^p \to Rep_K \to Rep^n \to S \to A$$

where $\quad Rep_{pn}$ = representations of $V^p \to K \to V^n \to S \to A$

and $\quad Rep^p = [V + Rep_{V\to K\to C}]^p$

We plug in:

$M_{pn}[\underline{\underline{\mathtt{S}}}\ [\underline{\underline{\mathtt{push}}}\ [\underline{\underline{\mathtt{selec}}}_j]]\ \beta]a_1\ldots a_p\kappa x_1\ldots x_n\sigma$ (Push identifier)

$= \mathit{push}_p(\mathit{selec}_j)\check{a}_1\ldots\check{a}_p(\check{\beta}\check{a}_1\ldots\check{a}_p\check{\kappa}\check{x}_1\ldots\check{x}_n)\sigma$

$= \check{\beta}\check{a}_1\ldots\check{a}_p\check{\kappa}\check{x}_1\ldots\check{x}_n\check{a}_j\sigma$

$= M_{p,n+1}\beta a_1\ldots a_p\kappa x_1\ldots x_n a_j\sigma$

$M_{pn}[\underline{\underline{\mathtt{S}}}\ [\underline{\underline{\mathtt{push}}}\ [\underline{\underline{\mathtt{K}}}\ v]]\ \beta]a_1\ldots a_p\kappa x_1\ldots x_n\sigma$ (Push immediate)

$= \mathit{push}_p(K_p\check{v})\check{a}_1\ldots\check{a}_p(\beta\check{a}_1\ldots\check{a}_p\check{\kappa}\check{x}_1\ldots\check{x}_n)\sigma$

$= \check{\beta}\check{a}_1\ldots\check{a}_p\check{\kappa}\check{x}_1\ldots\check{x}_n\check{v}\sigma$

$= M_{p,n+1}\beta a_1\ldots a_p\kappa x_1\ldots x_n v\sigma$

$M_{pn}[\underline{\underline{\mathtt{S}}}\ [\underline{\underline{\mathtt{push}}}\ [\underline{\underline{\mathtt{S}}}\ \alpha\ \beta]]\ \gamma]a_1\ldots a_p\kappa x_1\ldots x_n\sigma$ (Push closure)

$= \mathit{push}_p(S_{p+1,1}(\check{\alpha},\check{\beta}))\check{a}_1\ldots\check{a}_p(\check{\gamma}\check{a}_1\ldots\check{a}_p\check{\kappa}\check{x}_1\ldots\check{x}_n)\sigma$

$= \check{\gamma}\check{a}_1\ldots\check{a}_p\check{\kappa}\check{x}_1\ldots\check{x}_n(S_{p+1,1}(\check{\alpha},\check{\beta})\check{a}_1\ldots\check{a}_p)\sigma$

$= \check{\gamma}\check{a}_1\ldots\check{a}_p\check{\kappa}\check{x}_1\ldots\check{x}_n(\mathit{dclose}_p\check{\alpha}\check{\beta}\check{a}_1\ldots\check{a}_p)\sigma$

$= M_{p,n+1}\gamma a_1\ldots a_p\kappa x_1\ldots x_n[\underline{\underline{\mathtt{dclose}}}_p\ \alpha\ \beta\ a_1\ \ldots\ a_p]\sigma$

where $\mathit{dclose}_p\alpha\beta a_1\ldots a_p = S_{p+1,1}(\alpha,\beta)a_1\ldots a_p$

$M_{pn}[\underline{\underline{\mathtt{S}}}\ \underline{\underline{\mathtt{iot}}}\ \beta]a_1\ldots a_p\kappa x_1\ldots x_n\sigma$ (IO-transfer)

$= \mathit{iot}_p\check{a}_1\ldots\check{a}_p(\check{\beta}\check{a}_1\ldots\check{a}_p\check{\kappa}\check{x}_1\ldots\check{x}_{n-1})\check{x}_n\sigma$

$= \mathit{doio}(\check{\beta}\check{a}_1\ldots\check{a}_p\check{\kappa}\check{x}_1\ldots\check{x}_{n-1})\check{x}_n\sigma$

$= \check{\beta}\check{a}_1\ldots\check{a}_p\check{\kappa}\check{x}_1\ldots\check{x}_{n-1}\check{v}'\sigma'$ (as in Section 4)

$= M_{pn}\beta a_1\ldots a_p\kappa x_1\ldots x_{n-1}v'\sigma'$

$M_{pn}[\underline{\underline{\mathtt{S}}}\ \underline{\underline{\mathtt{apply}}}\ \beta]a_1\ldots a_p\kappa x_1\ldots x_n\sigma$ (Apply)

$= \mathit{apply}_p\check{a}_1\ldots\check{a}_p(\check{\beta}\check{a}_1\ldots\check{a}_n\check{\kappa}\check{x}_1\ldots\check{x}_{n-2})\check{x}_{n-1}\check{x}_n\sigma$

$= \check{x}_{n-1}\check{x}_n(\check{\beta}\check{a}_1\ldots\check{a}_n\check{\kappa}\check{x}_1\ldots\check{x}_{n-2})\sigma$

If $x_{n-1} = [\underline{\underline{\mathtt{dclose}}}_r\ \alpha\ \beta\ b_1 \ldots b_r]$, then we continue:

$$= \mathit{dclose}_r \check{\alpha}\check{\beta}\check{b}_1 \ldots \check{b}_r \check{x}_n (\check{\beta}\check{a}_1 \ldots \check{a}_r \kappa \check{x}_1 \ldots \check{x}_{n-2})\sigma$$

$$= S_{r+1,1}(\check{\alpha},\check{\beta})\check{b}_1 \ldots \check{b}_r \check{x}_n (\mathit{retpt}_{p,n-2} \check{\beta}\check{a}_1 \ldots \check{a}_p \kappa \check{x}_1 \ldots \check{x}_{n-2})\sigma$$

$$= M_{r+1,0}[\underline{\underline{\mathtt{S}}}\ \alpha\ \beta] b_1 \ldots b_r x_n [\underline{\underline{\mathtt{retpt}}}\ \beta\ a_1\ \ldots\ a_p\ \kappa\ x_1\ \ldots\ x_{n-2}]\sigma$$

Observe how the display stored in the closure has been installed as the machine's display and how the actual parameter x_n has been correctly placed at the top of the display. Finishing up:

$$M_{p1}\underline{\underline{\mathtt{return}}} a_1 \ldots a_p [\underline{\underline{\mathtt{retpt}}}\ \beta\ b_1\ \ldots\ b_r\ \kappa\ x_1\ \ldots\ x_n] v\sigma \quad \text{(Return)}$$

$$= \mathit{return}_p \check{a}_1 \ldots \check{a}_p (\check{\beta}\check{b}_1 \ldots \check{b}_r \kappa \check{x}_1 \ldots \check{x}_n)\check{v}\sigma$$

$$= \check{\beta}\check{b}_1 \ldots \check{b}_r \kappa \check{x}_1 \ldots \check{x}_n \check{v}\sigma$$

$$= M_{r,n+1} \beta b_1 \ldots b_r \kappa x_1 \ldots x_n v\sigma$$

$$M_{01}(\underline{\underline{\mathtt{return}}})(\underline{\underline{\mathtt{halt}}})v\sigma = \mathit{halt}\,v\sigma \qquad \text{(Stop)}$$

This machine is summarized in Table 5.3. Iterative interpretation of tail recursion may be achieved by the same modification as before.

Equations

$$\mathcal{P}e = \mathcal{B}_0(\mathcal{D}_0(\mathcal{E}e,\mathit{return}),\mathit{initenv})\mathit{halt}$$

$$\mathcal{E}I = \mathit{push}_1(\mathit{get}I)$$

$$\mathcal{E}[\lambda\ I\ e] = \mathit{push}_1(\mathcal{B}_2(\mathcal{D}_0(\mathcal{E}e,\mathit{return}),\mathit{ext}I))$$

$$\mathcal{E}[\mathrm{DOIO}\ e] = \mathcal{D}_0(\mathcal{E}e,\mathit{iot})$$

$$\mathcal{E}[e_1\ e_2] = \mathcal{D}_0(\mathcal{E}e_1,\mathcal{D}_1(\mathcal{E}e_2,\mathit{apply}))$$

Auxiliaries

$$\mathit{halt}_p = \lambda a_1 \ldots a_p . \mathit{halt}$$

$$\mathit{push}_p f = \lambda a_1 \ldots a_p \kappa . \kappa(f a_1 \ldots a_p)$$

$$\mathit{get}I = \lambda\rho.\rho I$$

$$\mathit{return} = \lambda\rho\kappa v.\kappa v$$

$$\mathit{ext}I = \lambda\rho v J.\mathrm{if}\ \ J = I\ \ \mathrm{then}\ \ v\ \ \mathrm{else}\ \ \rho J$$

$$\mathit{iot} = \lambda\rho.\mathit{doio}$$

$$\mathit{apply} = \lambda\rho\kappa f a.f a\kappa$$

Combinators

$$\mathcal{B}_p(\alpha,\beta) = \lambda a_1 \ldots a_p.\alpha(\beta a_1 \ldots a_p)$$

$$\mathcal{D}_n(\alpha,\beta) = \lambda\rho\kappa x_1 \ldots x_n.\alpha\rho(\beta\rho\kappa x_1 \ldots x_n)$$

Table 5.1 Formulation of the semantic equations for the language.

$$\mathit{CP}\mathtt{e} = \mathit{distr}[\underline{\underline{\mathtt{B}}}\ \mathit{rot}[\underline{\underline{\mathtt{D}}}\ \mathit{CE}\mathtt{e}\ \underline{\underline{\mathtt{return}}}]\ \underline{\underline{\mathtt{table}}}_0]$$

$$\mathit{CE}\mathtt{I} = [\underline{\underline{\mathtt{push}}}\ [\underline{\underline{\mathtt{get}}}\ \mathtt{I}]]$$

$$\mathit{CE}[\lambda\ \mathtt{I}\ \mathtt{e}] = [\underline{\underline{\mathtt{push}}}\ [\underline{\underline{\mathtt{B}}}\ \mathit{rot}[\underline{\underline{\mathtt{D}}}\ \mathit{CE}\mathtt{e}\ \underline{\underline{\mathtt{return}}}]\ [\underline{\underline{\mathtt{ext}}}\ \mathtt{I}]]]$$

$$\mathit{CE}[\mathtt{DOIO}\ \mathtt{e}] = [\underline{\underline{\mathtt{D}}}\ \mathit{CE}\mathtt{e}\ \underline{\underline{\mathtt{iot}}}]$$

$$\mathit{CE}[\mathtt{e}_1\ \mathtt{e}_2] = [\underline{\underline{\mathtt{D}}}\ \mathit{CE}\mathtt{e}_1[\underline{\underline{\mathtt{D}}}\ \mathit{CE}\mathtt{e}_2\ \underline{\underline{\mathtt{apply}}}]]$$

$$\mathit{rot}[\underline{\underline{\mathtt{D}}}\ [\underline{\underline{\mathtt{D}}}\ \alpha\ \beta]\ \gamma] = \mathit{rot}[\underline{\underline{\mathtt{D}}}\ \alpha\ [\underline{\underline{\mathtt{D}}}\ \beta\ \gamma]]$$

$$\mathit{rot}[\underline{\underline{\mathtt{D}}}\ \alpha\ \beta] = [\underline{\underline{\mathtt{D}}}\ \alpha\ \mathit{rot}\beta] \qquad \alpha \neq [\underline{\underline{\mathtt{D}}}\ \mathtt{x}\ \mathtt{y}]$$

$$\mathit{rot}\alpha = \alpha \qquad \alpha \neq [\underline{\underline{\mathtt{D}}}\ \mathtt{x}\ \mathtt{y}]$$

$$\mathit{distr}[\underline{\underline{\mathtt{B}}}\ [\underline{\underline{\mathtt{D}}}\ \alpha\ \beta]\ \tau] = [\underline{\underline{\mathtt{S}}}\ \mathit{distr}[\underline{\underline{\mathtt{B}}}\ \alpha\ \tau]\ \mathit{distr}[\underline{\underline{\mathtt{B}}}\ \beta\ \tau]]$$

$$\mathit{distr}[\underline{\underline{\mathtt{B}}}\ \underline{\underline{\mathtt{apply}}}\ \tau] = \underline{\underline{\mathtt{apply}}}$$

$$\mathit{distr}[\underline{\underline{\mathtt{B}}}\ \underline{\underline{\mathtt{iot}}}\ \tau] = \underline{\underline{\mathtt{iot}}}$$

$$\mathit{distr}[\underline{\underline{\mathtt{B}}}\ \underline{\underline{\mathtt{return}}}\ \tau] = \underline{\underline{\mathtt{return}}}$$

$$\mathit{distr}[\underline{\underline{\mathtt{B}}}\ [\underline{\underline{\mathtt{push}}}\ [\underline{\underline{\mathtt{B}}}\ \alpha\ [\underline{\underline{\mathtt{ext}}}\ \mathtt{I}]]]\ \tau] = [\underline{\underline{\mathtt{push}}}\ \mathit{distr}[\underline{\underline{\mathtt{B}}}\ \alpha\ [\underline{\underline{\mathtt{table}}}\ \mathtt{I}\ \tau]]]$$

$$\mathit{distr}[\underline{\underline{\mathtt{B}}}\ [\underline{\underline{\mathtt{push}}}\ [\underline{\underline{\mathtt{get}}}\ \mathtt{I}]]\ \tau] = [\underline{\underline{\mathtt{push}}}\ [\underline{\underline{\mathtt{selec}}}\ \mathtt{j}]] \quad \text{(j computed as in text)}$$

$$\mathit{distr}[\underline{\underline{\mathtt{B}}}\ [\underline{\underline{\mathtt{push}}}\ [\underline{\underline{\mathtt{get}}}\ \mathtt{I}]]\ \tau] = [\underline{\underline{\mathtt{push}}}\ [\underline{\underline{\mathtt{K}}}\ \mathit{initenv}\mathtt{I}]] \qquad (\text{I not in } \tau)$$

Table 5.2 Compiler using symbol table

```
[[λ X [[PLUS X] ONE]]
 [DOIO ONE]]
```

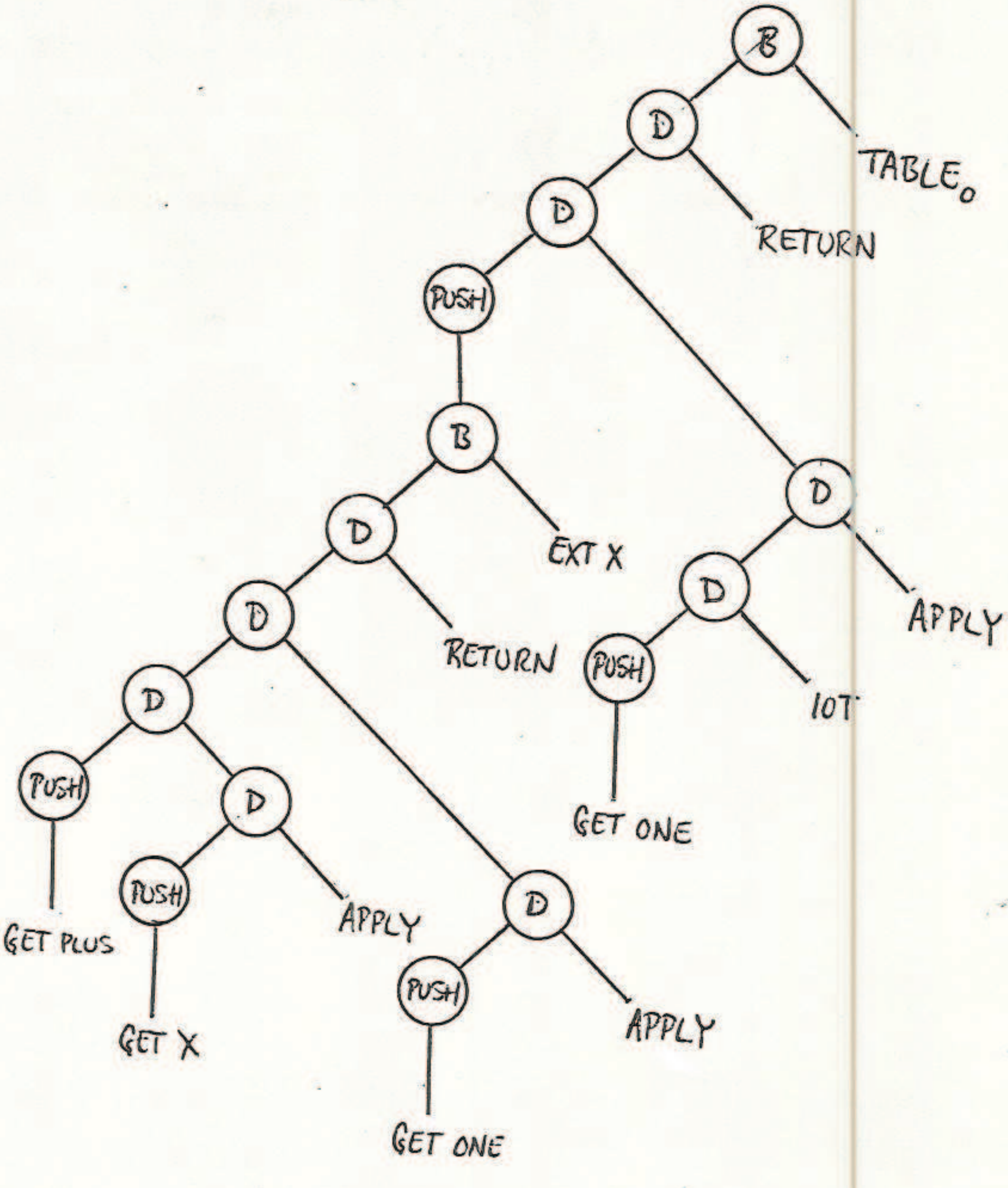
B
D
TABLE$_0$
D
RETURN
PUSH
B
D
EXT X
APPLY
D
D
RETURN
PUSH
IOT
D
PUSH
D
GET ONE
PUSH
APPLY
GET PLUS
D
GET X
PUSH
APPLY
GET ONE

Figure 5.1 Source code and code before rotation. (We imagine [DOIO ONE] is a read operation).

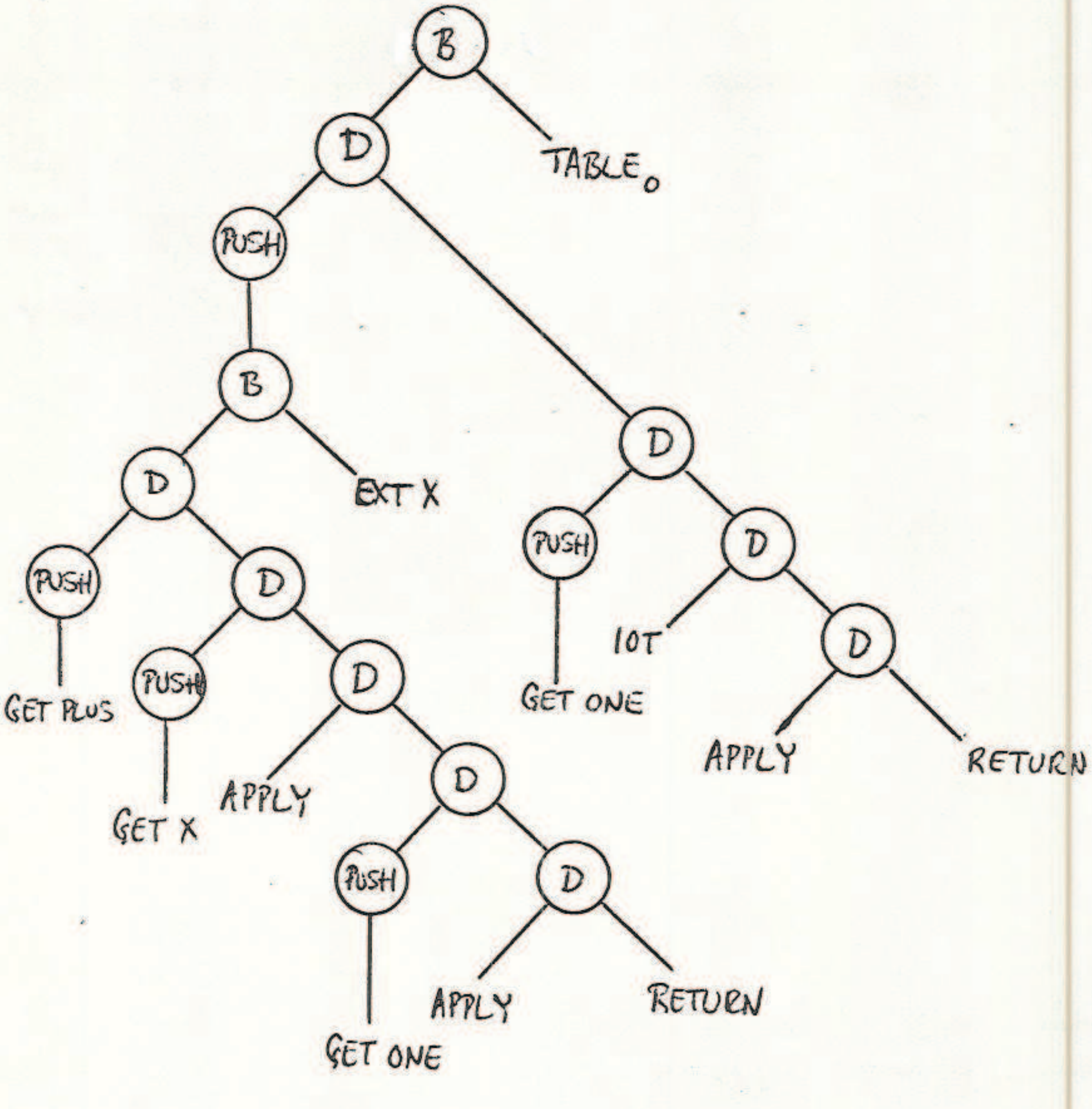
B
D
TABLE$_0$
PUSH
B
EXT X
D
PUSH
GET PLUS
D
PUSH
GET X
D
APPLY
D
PUSH
GET ONE
D
APPLY
RETURN
D
PUSH
GET ONE
D
IOT
D
APPLY
RETURN

Figure 5.2 Code after rotation, but before distribution

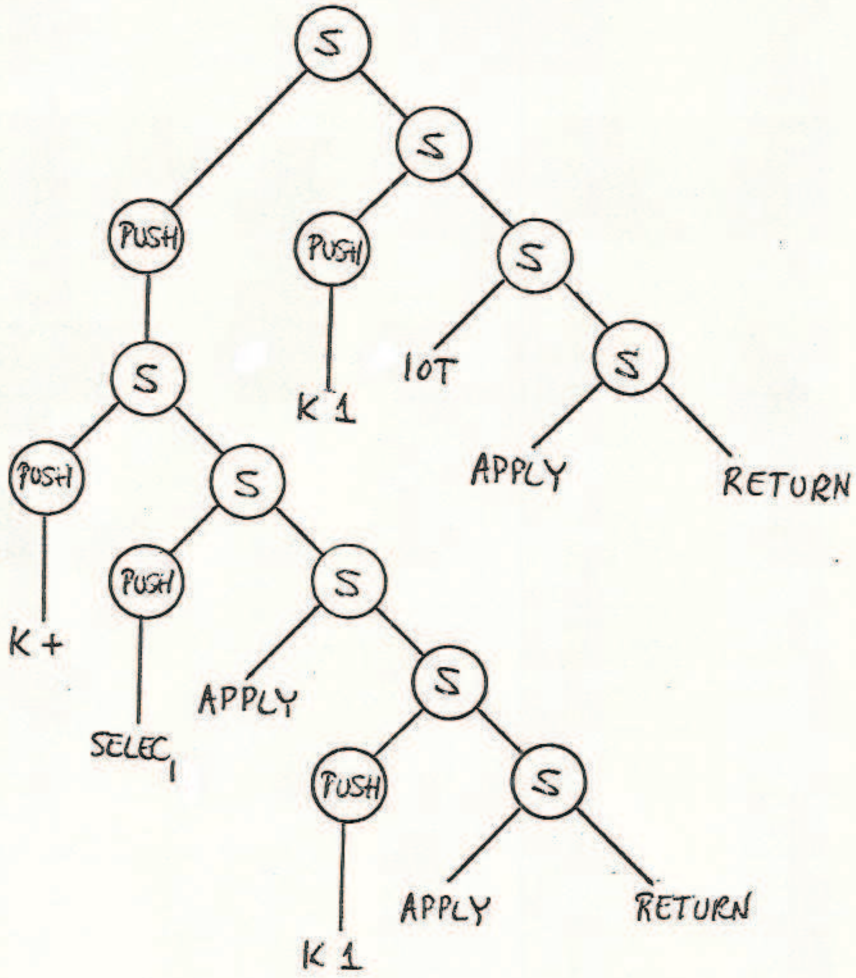
S
S
PUSH
PUSH
S
S
K 1
IOT
S
PUSH
S
APPLY
RETURN
K +
PUSH
S
SELEC$_1$
APPLY
S
PUSH
S
K 1
APPLY
RETURN

Figure 5.3 Code after distribution.

$$M_{pn}[\underline{\underline{S}}\ [\underline{\underline{push}}\ [\underline{\underline{selec}}_j]]\ \beta]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma$$

$$= M_{p,n+1} \beta a_1 \ldots a_p \kappa x_1 \ldots x_n a_j \sigma$$

$$M_{pn}[\underline{\underline{S}}\ [\underline{\underline{push}}\ [\underline{\underline{K}}\ v]]\ \beta]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma$$

$$= M_{p,n+1} \beta a_1 \ldots a_p \kappa x_1 \ldots x_n v \sigma$$

$$M_{pn}[\underline{\underline{S}}\ [\underline{\underline{push}}\ [\underline{\underline{S}}\ \alpha\ \alpha']]\ \beta]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma$$

$$= M_{p,n+1} \beta a_1 \ldots a_p \kappa x_1 \ldots x_n [\underline{\underline{dclose}}_p\ \alpha\ \alpha'\ a_1\ \ldots\ a_p] \sigma$$

$$M_{p,n+2}[\underline{\underline{S}}\ \underline{\underline{apply}}\ \beta]a_1 \ldots a_p \kappa x_1 \ldots x_n [\underline{\underline{dclose}}_r\ \alpha\ \alpha'\ b_1\ \ldots\ b_r] a \sigma$$

$$= M_{r+1,0}[\underline{\underline{S}}\ \alpha\ \alpha']b_1 \ldots b_r a [\underline{\underline{retpt}}\ \beta\ a_1\ \ldots\ a_p\ \kappa\ x_1\ \ldots\ x_n] \sigma$$

$$M_{p1} \underline{\underline{return}} a_1 \ldots a_p [\underline{\underline{retpt}}\ \beta\ b_1\ \ldots\ b_r\ \kappa\ x_1\ \ldots\ x_n] v \sigma$$

$$= M_{r,n+1} \beta\ b_1 \ldots b_r \kappa x_1 \ldots x_n v \sigma$$

$$M_{p1} \underline{\underline{return}} a_1 \ldots a_p \underline{\underline{halt}} v \sigma = \mathit{halt} v \sigma$$

$$M_{pn}[\underline{\underline{S}}\ \underline{\underline{iot}}\ \beta]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma$$

$$= M_{pn} \beta a_1 \ldots a_p \kappa x_1 \ldots x_{n-1} v' \sigma' \qquad \text{(as before)}$$

Table 5.3 Display Machine

6. Correctness of the Method

This paper has been devoted to a heuristic development of the methodology rather than to proofs of its correctness. We can, however, give some hints about how such proofs go. As discussed in Section 2, we need to show that the "abstraction function" in Figure 2.1 is a homomorphism. In that diagram, all the vertical arrows are the identity map, except for the one from $\mathrm{Rep}_{\mathrm{In}\to\mathrm{Out}}$ to In → Out, which is given by $^\vee$. In general, one might deal with $\mathrm{Rep}_{\mathrm{In}}$ and $\mathrm{Rep}_{\mathrm{Out}}$ as well. We need to show that the left-hand square and the right-hand wedge commute. The commutativity of the left-hand square, $^\vee(\mathcal{C}e) = \mathcal{P}e$, states the correctness of the compiler: the compiler produces the right code. This portion is easy, since the rotation and distribution operations in the compiler all preserve $^\vee$. The right-hand wedge, $\mathcal{M}\beta\sigma = \check{\beta}\sigma$, asserts the correctness of the virtual machine: the machine interprets the code properly.

We prove this portion by establishing an appropriate generalization, e.g. $\mathcal{M}_n\beta\rho\kappa x_1\ldots x_n\sigma = \check{\beta}\check{\rho}\check{\kappa}\check{x}_1\ldots\check{x}_n\sigma$. This is proven by considering the inequalities separately. As intimated previously, it is easy to restructure our derivation of the machine's action into a proof by fixpoint induction of the $\sqsubseteq$ direction. For the case of non-reflexive domains, Plotkin's method of induction on types may be used to establish the reverse inequality, as in [17]. We have done one such proof and hope to report on it elsewhere. In the presence of reflexive domains, there seems to be no alternative at present to the use of congruence relations [12, 21, 22, 27] to establish $\sqsupseteq$. This is the same two-part strategy used in the second congruence proof in [27]; Stoy's functions E, A, etc. correspond to our ${}^{\vee}$.

7. Related Work

Previous work on compiler correctness has centered on proving the equivalence of independently given semantics and machines. The proofs have generally proceeded by structural induction on the phrases of the source language, using algebraic techniques in lesser or greater degree to organize the proof. Work on this line includes [4, 11, 13, 14, 31]. Typically, a diagram similar to Figure 7.1 is involved. The most delicate portion of these proofs lies in stating the relation between source and target meanings (in the uncertain bottom arrows) in a way strong enough to support an induction. Typically, such an induction hypothesis says that a source-language expression, when compiled and executed in any suitable run-time context, leaves the source-semantics value of the expression on the top of the stack and leaves the rest of the context unchanged.

$$\begin{tikzcd}
\text{Source Language} \arrow[r, "\text{compile}"] \arrow[d, "\text{Source Semantics}"] & \text{Target Language} \arrow[d, "\text{Target Semantics}"] \\
\text{Source Meanings} \arrow[r, squiggly, shift right, "\text{(or)}"] & \text{Target Meanings} \arrow[l, shift right]
\end{tikzcd}$$

Figure 7.1 Diagram for previous correctness proofs

Unfortunately, this strategy works only if the source semantics is "direct," i.e. if the notion of "the value of a subexpression" is meaningful in the semantics. All of the papers cited above use direct semantics. If, however, one wishes to deal with realistic programming language features, such as input-output or hairy control structure, one must proceed almost immediately to continuation semantics [12], in which the notion of the value of a subexpression is meaningless.

The key which enables us to escape from Figure 7.1 is Mosses' idea of emphasizing the notion of implementation or simulation of data types as a measure of compiler correctness. Our work may be regarded as an extension and simplification of Mosses'. Our combinators $\mathcal{B}_k$ serve a role similar to the $\underset{k}{\rightarrow}$ operation in [16]. The major difference is that in $\underset{k}{\rightarrow}$, the subscript counts the number of items passed from the first action to the second, whereas in $\mathcal{B}_k$, the subscript counts the number of items protected from the first argument. A major conceptual difference is that we use models in preference to Mosses' equationally-specified classes of algebras (adopted from [33]). We have used models in this work because we did not know how to extend abstract sequencers such as $\rightarrow$ to handle escapes, nonlocal jumps, etc. The development of such a theory is a major area for work.

For a smooth theory, it would be convenient to replace right-associativity by full associativity. To do this, let $C_j(\alpha,\beta) = \lambda\kappa.\alpha(B_j(\beta,\kappa))$. Then $\{C_j \mid j \geq 0\} \cup \{B_{k+1} \mid k \geq 0\}$ forms a fully associative family of operations under the following rules:

1. $B_{k+1}(B_{j+1}(\alpha,\beta),\gamma) = B_{k+j+1}(\alpha,B_{k+1}(\beta,\gamma))$
2. $B_{k+1}(C_j(\alpha,\beta),\gamma) = B_{k+1}(\alpha,B_{k+j+1}(\beta,\gamma))$
3. $C_j(B_{k+1}(\alpha,\beta),\gamma) = B_{k+1}(\alpha,C_j(\beta,\gamma))$
4. $C_j(C_k(\alpha,\beta),\gamma) = C_j(\alpha,B_{k+j+1}(\beta,\gamma))$ if $k \geq j$
5. $C_j(C_k(\alpha,\beta),\gamma) = C_k(\alpha,C_{j-k}(\beta,\gamma))$ if $j \geq k$

The $\underset{k}{\rightarrow}$ operators similarly fall into two groups (Mosses, private communication). Combinators like B_k have also been utilized by [1, 2, 32], among others, in a variety of contexts.

A second key idea is the adopting of Hoare's view of implementation and the use of $^\vee$ to give a machine-independent semantics of the target code. This enabled us to factor the problem into compiler correctness and machine correctness. It also allowed the consideration of alternative representations, begun in [34].

The methodology we have used is extendable to other language features. Tests, conditionals, and escapes present no difficulties, nor do other phrase types such as commands. A more interesting problem is dealing with sequential representations of code. While tree-structured code is a viable alternative for custom machines [25, 35], conventional machines deal with instructions stored sequentially in memory. It should be possible to show that such code is a representation of the tree-structured code, using techniques such as those in [8, Sec. 2.3.3]. It should also be possible to treat clever representations of the stack and environment, such as spreading the stack across several registers [24, 35], or the static-chain representation, in which the environment is embedded in the stack [18]. Another possibility is the use of infinite representations, which may in turn be represented by cyclic pointer structures [23].

8. Conclusions

We have presented a method for deriving compilers and abstract machines from the continuation semantics for a language. The technique involves choosing special-purpose combinators to eliminate λ-variables in the semantics, and discovering standard forms for the resulting terms to obtain target code which represents the meaning of the source code. The abstract machine's job is to interpret these representations as functions. The approach seems capable of handling fairly complex languages and deriving interesting machines for them. We hope in the near future to extend and formalize the technique.

Acknowledgements

Peter Mosses pointed out the difference between right- and full associativity. Lee Becker taught me how to pronounce haček. Uwe Pleban provided useful comments on an earlier draft.

REFERENCES

1. J. Backus, "Programming Language Semantics and Closed Applicative Languages," *Proc. 1st ACM Symp. on Principles of Programming Languages*, Boston (1973), 71-86.

2. D. Bjorner & C. B. Jones, eds. *The Vienna Development Method: The Meta-Language*. Springer Lecture Notes in Computer Science, Vol. 61, Springer, Berlin, 1978.

3. R. M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs," *J. ACM 24* (1977), 44-67.

4. R. M. Burstall and P. J. Landin, "Programs and their Proofs; an Algebraic Approach," in *Machine Intelligence 4* (B. Meltzer & D. Michie, eds), pp. 17-44. American Elsevier, New York, 1969.

5. J. L. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright, "Initial Algebra Semantics and Continuous Algebras," *J. ACM 24* (1977), 68-95.

6. C. A. R. Hoare, "Proving Correctness of Data Representations," *Acta Informatica 1* (1972), 271-281.

7. N. D. Jones and S. Schmidt, "Compiler Generation from Denotational Semantics" in *Semantics-Directed Compiler Generation* (N. D. Jones, ed), pp. 70-93. Springer Lecture Notes in Computer Science, Vol. 94 (Berlin, 1980).

8. D. E. Knuth, *The Art of Computer Programming, Vol. I: Fundamental Algorithms*, Addison Wesley, Reading, MA, 1968; 2nd ed., 1973.

9. P. J. Landin, "The Mechanical Evaluation of Expressions," *Computer J. 6* (1964) 308-320.

10. J. McCarthy, "Towards a Mathematical Science of Computation," *Information Processing 62* (Popplewell, ed.) North-Holland, Amsterdam, 1963, pp. 21-28.

11. J. McCarthy, and J. Painter, "Correctness of a Compiler for Arithmetic Expressions," in *Proc. Symp. in Appl. Math., Vol. 19*, Mathematical Aspects of Computer Science (J. T. Schwartz, ed.), pp. 33-41. Amer. Math. Soc., Providence, RI, 1967.

12. R. Milne, and C. Strachey, *A Theory of Programming Language Semantics*, Chapman & Hall, London, and Wiley, New York, 1976.

13. R. Milner and R. Weyrauch, "Proving Compiler Correctness in a Mechanized Logic," in Machine Intelligence 7 (B. Meltzer & D. Michie, eds), pp. 51-72. Edinburgh University Press (1972).

14. L. Morris, "Advice on Structuring Compilers and Proving Them Correct," Proc. ACM Symp. on Principles of Programming Languages (Boston, 1973), 144-152.

15. J. H. Morris and B. Wegbreit, "Subgoal Induction," Comm. ACM 20 (1977), 209-222.

16. P. Mosses, "A Constructive Approach to Compiler Correctness," Automata, Languages, and Programming, Seventh Colloquium (1980), pp. 449-469.

17. G. D. Plotkin, "LCF Considered as a Programming Language" Theoret. Comp. Sci. 5 (1977) 223-255.

18. T. W. Pratt, Programming Languages: Design and Implementation, Prentice-Hall, Englewood Cliffs, N. J., 1975.

19. W. V. O. Quine, Word and Object, MIT Press, Cambridge, MA, 1960.

20. J. C. Reynolds, "Definitional Interpreters for Higher-Order Programming Languages, Proc. ACM Nat'l. Conf. (1972), 717-740.

21. J. C. Reynolds, "On the Relation between Direct and Continuation Semantics," Proc. 2nd Colloq. on Automata, Languages, and Programming (Saarbrucken, 1974) Springer Lecture Notes in Computer Science, Vol. 14 (Berlin: Springer, 1974), pp. 141-156.

22. R. Sethi, and A. Tang, "Constructing Call-by-Value Continuation Semantics" J. ACM 27 (1980), 580-597.

23. R. Sethi, "Circular Expressions: A Progress Report on Semantics-Directed Compiler Generation" to appear in Automata, Languages, Programming: Eighth Colloquium (1981).

24. G. L. Steele, and G. J. Sussman, "The Dream of a Lifetime: A Lazy Scoping Mechanism," Conference Record of the 1980 LISP Conference, pp. 163-172.

25. G. L. Steele and G. J. Sussman, "Design of a LISP-Based Microprocessor" Comm. ACM 23 (1980), 628-645.

26. J. E. Stoy, Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press, Cambridge, MA., 1977.

27. J. E. Stoy, "The Congruence of Two Programming Language Definitions" Theoret. Comp. Sci. 13 (1981), 151-174.

28. G. J. Sussman and G. L. Steele, Jr., "SCHEME: An Interpreter for Extended Lambda Calculus," Mass. Inst. of Tech., AI Memo 349 (December, 1975).

29. A. S. Tanenbaum, Structured Computer Organization, Prentice-Hall, Englewood Cliffs, N. J., 1976.

30. R. D. Tennent, "Denotational Semantics of Programming Languages," Comm. ACM 19 (1976) 437-453.

31. J. W. Thatcher, E. G. Wagner, and J. B. Wright, "More on Advice on Structuring Compilers and Proving Them Correct," in Automata, Languages, and Programming, Sixth Colloquium, Graz, July 1979 (H. A. Maurer, ed.) pp. 596-615. Lecture Notes in Comp. Sci., Vol. 71, Springer, Berlin (1979).

32. D. A. Turner, "A New Implementation Technique for Applicative Languages" Software-Practice and Experience 9 (1979), 31-49.

33. M. Wand, "First-Order Identities as a Defining Language," Acta Informatica 14 (1980), 337-357.

34. M. Wand, "Continuation-Based Program Transformation Strategies," J. ACM 27 (1980), 164-180.

35. M. Wand, SCHEME Version 3.1 Reference Manual, Indiana University, Computer Science Department, Technical Report No. 93, June, 1980.