Proving the Correctness of Storage Representations

Mitchell Wand and Dino P. Oliva*

College of Computer Science
Northeastern University
360 Huntington Avenue, 161CN
Boston, MA 02115, USA
wand@corwin.ccs.northeastern.edu
oliva@corwin.ccs.northeastern.edu

Abstract

Conventional techniques for semantics-directed compiler derivation yield abstract machines that manipulate trees. However, in order to produce a real compiler, one has to represent these trees in memory. In this paper we show how the technique of *storage-layout relations* (introduced by Hannan [7]) can be applied to verify the correctness of storage representations in a very general way. This technique allows us to separate denotational from operational reasoning, so that each can be used when needed. As an example, we show the correctness of a stack implementation of a language including dynamic catch and throw. The representation uses static and dynamic links to thread the environment and continuation through the stack. We discuss other uses of these techniques.

1 Introduction

Typical semantically-derived compiler systems [2, 3, 7, 18, 19] translate the parse trees of the source language into the language of an abstract byte-code machine. This abstract byte code machine typically manipulates trees that represent the environment, continuation, local stack, etc. However, in order to turn this into a real compiler, one has to represent these trees in machine storage. There are a variety of strategies for doing this [8], but this representation step remains a critical and untrustworthy part of compiler-development technology. Indeed, in two recent compiler-correctness projects [9, 10], it turned out to be the most difficult part of the task.

Hannan [7] has shown how, under certain restrictions, one can automatically transform an abstract machine into a storage machine. In this paper we show how similar techniques can be used to verify the correctness of storage representations in a far more general way. As in [18, 19, 20], we do not attempt to derive these representations automatically. Instead, our goal is to build a framework in which compiler writers can prove the correctness of arbitrarily clever human-invented representations.

The representations are formalized by the use of representations that we call *storage-layout relations*. Storage-layout relations allow a compiler writer to specify the representation decisions he or she has made. Storage-layout relations were introduced in [7]. This work extends that of [7] by liberating it from the particular architecture considered there, and by considering more complex representation strategies, such as multiple terms threaded through a single stack.

As an application, we consider a simple (not necessarily functional) language with continuation semantics and **catch** and **throw** operators. The language is designed to obey *stack discipline*: that is, the language can be implemented using only a stack. Our goal is to prove the correctness of this assertion. In fact, we will show that an ordinary stack representation, with static and dynamic chain pointers, constitutes a representation of the three trees (display, local stack, and continuation) of the abstract bytecode machine.

We first give a semantics and a simple byte-code compiler using the techniques of [3]. This compiler produces code for an abstract machine that manipulates combinator trees representing the environment, continuation, and a local stack. We then consider two successively more concrete implementations of this abstract machine. In the first version, we represent the display, local stack, and continuation in a single stack kept in memory, while the program remains as a tree. In the second version, we represent programs linearly as well.

This example shows how we can separate denotational from operational reasoning in a compiler correctness proof, using each to best advantage: the translation from source code to combinator trees is proved correct using structural induction and denotational reasoning, the translation from the abstract machine to the the concrete machines is proved correct using induction on length of execution and operational reasoning. The link between the two regimes is obtained using general results about the completeness of reduction strategies in the λ-calculus. We believe that this strategy will be of general use.

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

2 Comparison with Previous Work

Early proofs of compiler correctness, such as [11], did the translation from source to target code in a single step. In modern terminology, such proofs used a source semantics in a direct style and a target semantics in a continuation style or an operational style.

The restriction to direct-style semantics made it infeasible to use these techniques to prove the correctness of compilers for languages with non-local jumps. Therefore, in the 1970's, attention turned to correctness proofs in which the source language was given using continuation semantics. Such proofs typically involved complex arguments about inclusive predicates that related the domains in the source and target semantics [12, 17]. These proofs were quite complex because they sought to model reasoning that was essentially operational (induction on reduction sequences) by methods that were essentially denotational (induction on approximations to inverse limit domains).

At the beginning of the 1980's, Wand [18] showed how to avoid the use of inclusive predicates by modelling the target machine in the same semantic domains as the source semantics. In this paradigm, one showed that the meanings of the source and target programs were the same. Because the semantics was compositional, this could be done by a simple structural induction. This approach was refined by Clinger [3] to prove the correctness of a compiler for Scheme.

In this approach, the target language was a sublanguage of the λ-calculus, designed to be closed under reduction. The target machine simply had to simulate the reduction of the target program. One could use any desired mechanism to do this. The simulation proof could use operational reasoning. The reduction strategy was known to be complete, so the machine was guaranteed to reduce the target program to a normal form if one existed.

The Wand-Clinger approach simplified the inclusive-predicate approach in two ways. first, it allowed the denotational reasoning to be done in a single λ-model that gave denotations to both source and target terms, and second, it separated operational from denotational reasoning, so each could be used where appropriate. This paper extends the operational part of the proof by showing how the tree-rewriting systems can be implemented in a conventional storage model, and how that implementation can be proved correct using operational reasoning.

Raoult and Sethi [14] and Schmidt [16] considered the issue of single-threading: when a particular term (or class of terms) could be represented by a single destructively-updated quantity, but they did not consider the representation of such quantities in a linear store. Hannan [7] showed how certain classes of tree-rewriting systems could be transformed into a stored-program form, including the linear representation of programs, much like our stored-program machine in Section 8, but he represented all mutable data either in the heap or in multiple stacks, rather than considering the single-stack representation considered here.

3 Source Language

Our source language is a simple language with continuation semantics. Its values are integers, recursive procedures, and continuations. Operations on integers are addition and conditional branch (on zero). Operations on continuations are catch and throw. Catch captures the current continuation and binds it to an identifier. Throw invokes a bound continuation on a value, discarding the current continuation.

Procedures are recursive, using a local identifier for self-reference. Any kind of value may be passed to a procedure, but procedures may return only integers, so that the language can be implemented using only a stack. The semantics is given relative to an unspecified domain of command continuations, so that non-functional primitives may be added.

The language is given by

$$\begin{aligned} E \;::=\;& Id \mid (E + E) \mid (\texttt{if}\; E\; E\; E) \mid (\texttt{catch}\; Id\; E) \\ &\mid (\texttt{throw}\; Id\; E) \mid (E\; E) \mid (\texttt{rec}\; Id(Id)\; E) \end{aligned}$$

The semantics of the language is given by a valuation $\mathcal{E}[\![-]\!]$. We typically think of $\mathcal{E}[\![-]\!]$ as a syntactic transformation, whose output is a λ-term of the appropriate type. That is, if we write $\underline{T}$ for the set of λ-terms of type T, then we think of $\mathcal{E}$ as having type $E \to \underline{U \to K \to C}$, where U, K, and C are types given by

$$\begin{aligned} C &= \text{Command Continuations} \\ V &= \mathit{Int} + [V \to K \to C] + [V \to C] \\ K &= V \to C \\ U &= \mathit{Id} \to V \end{aligned}$$

The lattice-theoretical details are not relevant, since we are dealing with λ-terms rather than denotations; we assume only the existence of a non-trivial model with this type theory. We write the summands of V as $\langle \mathbf{int}, n\rangle$, $\langle \mathbf{proc}, p\rangle$ or $\langle \mathbf{cont}, \kappa\rangle$. We assume the existence of combinators that can distinguish these pairwise; this is easy to do by coding

The semantics is given by

$$\begin{aligned} \mathcal{E}[\![I]\!] &= \lambda\rho\kappa.\kappa(\rho[\![I]\!]) \\ \mathcal{E}[\![(e_1 + e_2)]\!] &= \lambda\rho\kappa.\mathcal{E}[\![e_1]\!]\rho(\lambda v_1.\mathcal{E}[\![e_2]\!]\rho(\lambda v_2\; \mathit{sum}\, v_1 v_2 \kappa)) \\ \mathcal{E}[\![(\texttt{if}\, e_1\; e_2\; e_3)]\!] &= \lambda\rho\kappa.\mathcal{E}[\![e_1]\!]\rho(\lambda v_1.v_1 = \langle \mathbf{int}, 0\rangle \to \\ &\qquad\qquad \mathcal{E}[\![e_3]\!]\rho\kappa, \\ &\qquad\qquad \mathcal{E}[\![e_2]\!]\rho\kappa) \\ \mathcal{E}[\![(\mathrm{rec}\, F(I)\; e)]\!] &= \lambda\rho\kappa.\kappa\langle \mathbf{proc}, \\ &\qquad (\mathit{fix}(\lambda p\lambda v\kappa.\mathcal{E}[\![e]\!](\rho[\langle \mathbf{proc}, p\rangle/F, v/I]) \\ &\qquad\qquad (\lambda v.\, \mathit{return}\,\kappa v))))\rangle \\ \mathcal{E}[\![(e_1\; e_2)]\!] &= \lambda\rho\kappa.\mathcal{E}[\![e_1]\!]\rho(\lambda v_1.\mathcal{E}[\![e_2]\!]\rho \\ &\qquad\qquad (\lambda v_2.\, \mathit{apply}_p\, v_1 v_2 \kappa)) \\ \mathcal{E}[\![(\texttt{catch}\, I\; e)]\!] &= \lambda\rho\kappa.\mathcal{E}[\![e]\!]\rho[\langle \mathbf{cont}, \kappa\rangle/I]k \\ \mathcal{E}[\![(\texttt{throw}\, I\; e)]\!] &= \lambda\rho\kappa.\mathcal{E}[\![e]\!]\rho(\lambda v_1.\mathcal{E}[\![I]\!]\rho(\lambda v_2\; \mathit{apply}_k\, v_1 v_2)) \end{aligned}$$

The operators *sum*, apply_p, apply_k, and *return* all check their arguments for membership in the appropriate summand. In particular, *return* only allows procedures to return integers; this prevents upward funargs and allows a stack-discipline implementation.

$$\begin{aligned} \mathit{sum} = \lambda v_1 v_2 \kappa.(v_1 = \langle \mathbf{int}, n_1\rangle) &\to \\ (v_2 = \langle \mathbf{int}, n_2\rangle) &\to \\ \kappa(\langle \mathbf{int}, n_1 + n_2\rangle)&, \\ (\mathit{error}\ \text{“Non-integer second arg”})&, \\ (\mathit{error}\ \text{“Non-integer first arg”})& \end{aligned}$$

$$\begin{aligned} \mathit{apply}_k = \lambda v_1 v_2.(v_2 = \langle \mathbf{cont}, \kappa\rangle) &\to \\ (v_1 = \langle \mathbf{int}, n\rangle) \to \kappa(v_1)&, \\ (\mathit{error}\ \text{“Non-integer second arg”})&, \\ (\mathit{error}\ \text{“Non-continuation first arg”})& \end{aligned}$$

$$\begin{aligned}return &= \lambda\kappa v.(v = \langle\mathbf{int}, n\rangle) \rightarrow \kappa(v),\\ &\qquad (error \text{ “Non-integer return value”})\end{aligned}$$

$$\begin{aligned}apply_p &= \lambda v_1 v_2 \kappa.(v_1 = \langle\mathbf{proc}, p\rangle) \rightarrow p v_2 \kappa,\\ &\qquad (error \text{ “Non-procedure to apply”})\end{aligned}$$

4 The Byte-Code Compiler

Our compiler is a straightforward byte-code compiler based on [3, 18]. Our abstract machine has three registers: a display u, a local stack ζ (essentially a local register file, which is manipulated as a stack), and a continuation. A program acts on these to produce a command continuation. The machine uses the following domains:

$l :$	L	Lexical Addresses ($Int \times Int$)
$\Gamma :$	$U_C = Id \rightarrow L$	Symbol Tables
$u :$	$U_R = L \rightarrow V$	Run-Time Environments (displays)
$\pi :$	$P \;= U_R \rightarrow V^* \rightarrow K \rightarrow C$	Programs
$\zeta :$	V^*	Local Stacks

A program is a term of type P. The compiler takes an expression and a symbol table, and returns a basic block (that is, a function $\underline{P} \rightarrow \underline{P}$) for that expression. The goal of the basic block is to execute its sequel in the same run-time environment and continuation, but with the value of the expression pushed on the local stack This specification will be the induction hypothesis for the compiler correctness proof. We write this specification as follows:

$$(\mathcal{C}[\![e]\!]\Gamma\pi)u\zeta\kappa = \mathcal{E}[\![e]\!](u \circ \Gamma)(\lambda z.\pi u(z :: \zeta)\kappa) \qquad (1)$$

Here we see that the expression is to be evaluated in the environment which is the composition of the symbol table and the display. Given the binding-time annotations suggested for $\mathcal{E}$ and $\mathcal{C}$, each side of this equality can be interpreted as a λ-term of type C, so the equality is to be interpreted as equivalence in some suitable λ-theory.

The compiler is defined as follows:

$$\begin{aligned}
&\mathcal{C}[\![I]\!]\Gamma\pi = fetch(\Gamma[\![I]\!])\pi\\
&\mathcal{C}[\![(e_1 + e_2)]\!]\Gamma\pi = \mathcal{C}[\![e_1]\!]\Gamma(\mathcal{C}[\![e_2]\!]\Gamma(add\,\pi))\\
&\mathcal{C}[\![(\mathtt{if}\; e_1\; e_2\; e_3)]\!]\Gamma\pi = \mathcal{C}[\![e_1]\!]\Gamma(brz(\mathcal{C}[\![e_2]\!]\Gamma\pi)(\mathcal{C}[\![e_3]\!]\Gamma\pi))\\
&\mathcal{C}[\![(\mathtt{rec}\; F(I)\; e)]\!]\Gamma\pi = close(\mathcal{C}[\![e]\!](ext_C\langle I, F\rangle\Gamma)\; rts)\pi\\
&\mathcal{C}[\![(e_1\; e_2)]\!]\Gamma\pi = \mathcal{C}[\![e_1]\!]\Gamma(\mathcal{C}[\![e_2]\!]\Gamma(jsr\,\pi))\\
&\mathcal{C}[\![(\mathtt{catch}\, I\; e)]\!]\Gamma\pi = save\text{-}cont(\mathcal{C}[\![e]\!](ext_C\langle I, d\rangle\Gamma)\; rts)\pi\\
&\qquad (d \text{ a fresh identifier })\\
&\mathcal{C}[\![(\mathtt{throw}\, I\; e)]\!]\Gamma\pi = \mathcal{C}[\![e]\!]\Gamma(\mathcal{C}[\![I]\!]\Gamma\; throw)
\end{aligned}$$

Thus the code for $e_1 + e_2$ consists of the code for e_1 followed by the code for e_2, followed by an *add* instruction, followed by the program π that is to follow this basic block. (We leave it to the reader to verify that this definition is consistent with the binding-time annotation given above).

The instructions *fetch*, *add*, etc., are combinators that satisfy the following equations:

$$\begin{aligned}
&(fetch\; l\, \pi)u\zeta\kappa = \pi u(u(l) :: \zeta)\kappa\\
&(add\,\pi)u(\langle\mathbf{int}, n_2\rangle :: \langle\mathbf{int}, n_1\rangle :: \zeta)\kappa\\
&\quad = \pi u(\langle\mathbf{int}, n_1 + n_2\rangle :: \zeta)\kappa\\
&(brz\,\pi_1\pi_2)u(\langle\mathbf{int}, n_1\rangle :: \zeta)\kappa\\
&\quad = ((n_1 = 0) \rightarrow (\pi_2 u\zeta\kappa), (\pi_1 u\zeta\kappa))\\
&(save\text{-}cont\,\pi_1\pi_2)u\zeta\kappa\\
&\quad = \pi_2(ext_R\langle\langle\mathbf{cont}, (cont\,\pi_1 u\zeta\kappa)\rangle, \langle\rangle\rangle u)\langle\rangle(cont\,\pi_1 u\zeta\kappa)\\
&(throw)u(\langle\mathbf{int}, n\rangle :: \langle\mathbf{cont}, (cont\,\pi' u'\zeta'\kappa')\rangle :: \zeta)\kappa\\
&\quad = \pi' u'(\langle\mathbf{int}, n\rangle :: \zeta')\kappa'\\
&(close\,\pi'\pi)u\zeta\kappa = \pi u(\langle\mathbf{proc}, closure\, u\pi'\rangle :: \zeta)\kappa\\
&(rts)u(\langle\mathbf{int}, n\rangle :: \zeta)(cont\,\pi' u'\zeta'\kappa') = \pi' u'(\langle\mathbf{int}, n\rangle :: \zeta')\kappa'\\
&(jsr\,\pi)u(v :: \langle\mathbf{proc}, closure\, u'\pi'\rangle :: \zeta)k\\
&\quad = \pi'(ext_R\langle v, \langle\mathbf{proc}, closure\, u'\pi'\rangle\rangle u')\langle\rangle(cont\,\pi u\zeta\kappa)
\end{aligned}$$

In addition, the instructions must also satisfy a set of equations for error conditions in the semantics, such as

$$(add\,\pi)u(v :: \langle\mathbf{proc}, p\rangle :: \zeta)\kappa = error \text{ “Non-integer first arg”}$$

Here *cont* and *closure* are combinators defined by:

$$\begin{aligned}
&(cont\,\pi u\zeta\kappa) = \lambda z.\pi u(z :: \zeta)\kappa\\
&(closure\, u\pi) = fix(\lambda p.\lambda v\kappa.\pi(ext_R\langle v, \langle\mathbf{proc}, p\rangle\rangle u)\langle\rangle\kappa)
\end{aligned}$$

and ext_C and ext_R are combinators such that

$$(ext_R\langle v, v'\rangle u) \circ (ext_C\langle I, I'\rangle\Gamma) = (u \circ \Gamma)[v/I, v'/I']$$

The equations for the instructions are not quite combinators as presented here, but it is straightforward to write combinators that obey these equations.

Now we can state the correctness of the compiler:

Theorem 1 *The compiler $\mathcal{C}$ is correct, that is, for all source programs e, symbol tables Γ, run-time environments u, stacks ζ, and continuations κ,*

$$(\mathcal{C}[\![e]\!]\Gamma\pi)u\zeta\kappa = \mathcal{E}[\![e]\!](u \circ \Gamma)(\lambda z.\pi u(z :: \zeta)\kappa)$$

Proof: By a straightforward structural induction on the expressions of the source language. □

The bytecode machine definition is denotational. However, except for the error rules, each instruction definition is of the form

$$\pi u\zeta\kappa = \pi' u'\zeta'\kappa'$$

where π, u, ζ, κ, π' u' ζ' κ' are given by the following grammar:

$$\begin{aligned}
\pi \quad &::= \quad (fetch\; l\,\pi) \mid (add\,\pi) \mid (brz\,\pi\pi)\\
&\qquad \mid (save\text{-}cont\,\pi\pi) \mid (throw) \mid (close\,\pi\pi)\\
&\qquad \mid (rts) \mid (jsr\,\pi)\\
u \quad &::= \quad emptydisplay \mid (ext_R\langle v, \langle\mathbf{proc}, closure\, u'\pi\rangle\rangle u')\\
&\qquad \mid (ext_R\langle\langle\mathbf{cont}, (cont\,\pi u'\zeta\kappa)\rangle, \langle\rangle\rangle u')\\
\zeta \quad &::= \quad \langle\rangle \mid (v :: \zeta)\\
v \quad &::= \quad \langle\mathbf{int}, n\rangle \mid \langle\mathbf{proc}, closure\, u\pi\rangle\\
&\qquad \mid \langle\mathbf{cont}, (cont\,\pi u\zeta\kappa)\rangle\\
k \quad &::= \quad initcont \mid (cont\,\pi u\zeta\kappa)
\end{aligned}$$

(where in the productions for u, the occurrences of u' are the same; this could be avoided by introducing two new combinators).

Each step in the reduction sequence of $\pi u\zeta\kappa$ will have the form $\pi' u'\zeta'\kappa'$ for some program π', display u', stack ζ', and continuation κ' as specified by this grammar. Therefore we can give the bytecode machine an operational semantics by thinking of it as a 4-register machine and regarding each of the equations for the instruction combinators as rewriting rules. The resulting abstract machine is shown in Figure 1. We can state the soundness of this machine as follows:

Theorem 2 *If* $\langle \pi, u, \zeta, \kappa \rangle \Longrightarrow \langle \pi', u', \zeta', \kappa' \rangle$, *then* $\pi u \zeta \kappa \rightarrow \pi' u' \zeta' \kappa'$.

It can also be shown that the machine is complete with respect to either call-by-value or call-by-name reduction [13], since these reduction strategies coincide on the terms manipulated by the machine, and the machine emulates these reduction strategies.

So far this is just an example of the method set out in [3, 18] (see also [5] for some more extended examples). We next turn to our main topic: the representation of these trees in a linear store.

5 Stack Representation

This abstract machine simulates the reduction of the terms produced by the compiler. However, it is still a tree manipulation system. In order to implement this abstract machine on a real computer, one must represent these trees in memory. It is the correctness of this representation that is our main concern in this paper.

In particular, our language was carefully designed to obey *stack discipline*. that is, the language can be implemented using only a stack. Our goal is to prove the correctness of this assertion. In fact, we will show that an ordinary stack representation, with static and dynamic chain pointers, constitutes a representation of the three trees (display, local stack, and continuation) of the abstract bytecode machine.

We will do this in two stages. In the first stage, we will represent the display, local stack, and continuation in a single stack kept in memory, whereas the program π will remain as a tree. In the second stage, we will represent programs linearly as well.

Our concrete machine will be a 5-tuple

$$\langle \pi, up, sp, kp, \sigma \rangle$$

consisting of a program π, a display pointer up, a stack pointer sp, a continuation pointer kp, and a store σ. The pointers are addresses in the store, satisfying $sp \geq kp > up$. We assume that each cell of the store can hold a tag (**proc** or **int**), a pointer, and occasionally a program. (The program will be condensed to a pointer in the second step).

We now sketch the representation of the abstract machine's quantities in the concrete machine.

1. Values (integers or closures) are represented as single words in memory, including their tags. Continuations bound by `catch` are represented as single words containing a tag and a pointer to the continuation structure.

2. A local stack $(v_n :: v_{n-1} :: \ldots :: v_1 :: \langle\rangle)$ is specified by giving upper and lower bounds as indices, and is represented as a series of words in the stack

ub:	v_n
	v_{n-1}
	$\ldots$
	v_1
lb:	

This notation is intended to mean that ub and lb are pointers into the stack, that ub points higher in the stack than lb, and that the values from position ub, moving downward, are $v_n, v_{n-1}, \ldots, v_1$.

3. A run-time environment (display) extension of the form $ext_R(v, \langle \mathbf{proc}, closure\, u' \pi' \rangle) u'$ is specified by a pointer p into the stack:

p :	v
	$\langle \mathbf{proc}, closure\, p' \pi' \rangle$

where p' is a pointer to the display u' (ie, a static chain pointer). Here the second slot is used both for the closure that is bound to the local name for the procedure and for the static chain that is in the closure.

4. A run-time environment (display) extension of the form $ext_R(\langle \mathbf{cont}, (cont\, \pi' u \zeta \kappa) \rangle, \langle\rangle) u$ is specified by a pointer p into the stack.

p :	$\langle \mathbf{cont}, p' \rangle$
	p''

where $\langle \mathbf{cont}, p' \rangle$ represents $(cont\, \pi' u \zeta \kappa)$ (according to item 1 above) and p'' is a pointer to the display u (ie, a static chain pointer).

5. The continuation $(cont\, \pi u \zeta \kappa)$ is represented by a pointer p into the stack as follows:

p :	π	the saved program (counter)
	up	pointer to the saved environment
	kp	dynamic chain (base) pointer
		left open for current environment
		left open for current environment
	v_n	top value in ζ
	$\ldots$	
	v_1	bottom value in ζ
kp:		top of frame for next continuation κ

6 Storage Layout Relations

This pictorial specification is adequate for compiler writers, but it is not quite formal enough to allow us to do proofs. We formalize these pictures as a family of relations, called *storage layout relations*, that tell precisely when a pointer into a stack corresponds to a given abstract structure. In general, a storage layout relation will be a ternary relation $\sigma \models p \simeq m$, which means that pointer p corresponds to abstract structure m in store σ. We will consistently put the concrete object on the left of the equivalence, and read $\simeq$ as "corresponds to."

We will have four storage layout relations, one each for values, stacks, environments, and continuations. These will be defined by simultaneous induction on the terms for values, stacks, environments, and continuations. Since these are relations on terms (not their denotations), the existence of the relations is assured.

The definition of the relations is now a straightforward transcription of the data in the diagrams above.

Abstract machine:

$$\begin{array}{rcl}
\langle (fetch\, l\, \pi), u, \zeta, \kappa\rangle & \Longrightarrow & \langle \pi, u, (u(l) :: \zeta), \kappa\rangle \\
\langle (add\, \pi), u, (\langle \mathbf{int}, n_2\rangle :: \langle \mathbf{int}, n_1\rangle :: \zeta), \kappa\rangle & \Longrightarrow & \langle \pi, u, (\langle \mathbf{int}, n_1 + n_2\rangle :: \zeta), \kappa\rangle \\
\langle (brz\, \pi'\pi), u, v :: \zeta), \kappa\rangle & \Longrightarrow & \langle (choose\ v\, \pi'\pi), u, \zeta, \kappa\rangle \\
\langle (save\text{-}cont\, \pi'\pi), u, \zeta, \kappa\rangle & \Longrightarrow & \langle \pi, (ext_R\langle\langle \mathbf{cont}, (cont\, \pi' u\zeta\kappa)\rangle, \langle\rangle\rangle u), \\
 & & \quad \langle\rangle, (cont\, \pi' u\zeta\kappa)\rangle \\
\langle (throw), u, \langle \mathbf{cont}, (cont\, \pi' u'\zeta'\kappa')\rangle :: \langle \mathbf{int}, n\rangle :: \zeta), \kappa\rangle & \Longrightarrow & \langle \pi', u', (\langle \mathbf{int}, n\rangle :: \zeta'), \kappa'\rangle \\
\langle (close\, \pi'\pi), u, \zeta, \kappa\rangle & \Longrightarrow & \langle \pi, u, (\langle \mathbf{proc}, closure\, u\pi'\rangle :: \zeta), \kappa\rangle \\
\langle (rts), u, (\langle \mathbf{int}, n\rangle :: \zeta), (cont\, \pi' u'\zeta'\kappa')\rangle & \Longrightarrow & \langle \pi', u', (\langle \mathbf{int}, n\rangle :: \zeta'), \kappa'\rangle \\
\langle (jsr\, \pi), u, (v :: \langle \mathbf{proc}, closure\, u'\pi'\rangle :: \zeta), \kappa\rangle & \Longrightarrow & \langle \pi', ext_R\langle v, \langle \mathbf{proc}, closure\, u'\pi'\rangle\rangle u', \\
 & & \quad \langle\rangle, (cont\, \pi u\zeta\kappa)\rangle
\end{array}$$

Auxiliaries:

$$(choose\ v\, \pi'\pi) = (v = \langle \mathbf{int}, 0\rangle) \rightarrow \pi, \pi'$$

Figure 1: Actions of the abstract machine.

- Relating Pointers and Tagged Values

$$\begin{array}{lll}
\sigma \models_V p \simeq v & \Longleftrightarrow & \\
 & \text{either} & v = \langle \mathbf{int}, n\rangle = \sigma(p) \\
 & \text{or} & v = \langle \mathbf{proc}, (closure\, u\pi)\rangle \\
 & & \text{and } \sigma(p) = \langle \mathbf{proc}, (closure\, up\, \pi)\rangle \\
 & & \text{and } up < p \text{ and } \sigma \models_U up \simeq u \\
 & \text{or} & v = \langle \mathbf{cont}, (cont\, \pi u\zeta\kappa)\rangle \\
 & & \text{and } \sigma(p) = \langle \mathbf{cont}, p'\rangle \\
 & & \text{and } p' < p \\
 & & \text{and } \sigma \models_K p' \simeq (cont\, \pi u\zeta\kappa)
\end{array}$$

- Relating Pointers and Stacks

$$\begin{array}{lll}
\sigma \models_Z \langle p, p'\rangle \simeq \zeta & \Longleftrightarrow & \\
 & \text{either} & \zeta = \langle\rangle \text{ and } p = p' \\
 & \text{or} & \zeta = (z :: \zeta') \text{ and } p > p' \text{ and } \sigma \models_V p \simeq z \\
 & & \text{and } \sigma \models_Z \langle p-1, p'\rangle \simeq \zeta'
\end{array}$$

- Relating Pointers and Environments

$$\begin{array}{lll}
\sigma \models_U p \simeq u & \Longleftrightarrow & \\
 & \text{either} & u = emptydisplay \text{ and } p = -1 \\
 & \text{or} & u = ext_R\langle v, \langle \mathbf{proc}, closure\, u'\pi'\rangle\rangle u' \\
 & & \text{and } \sigma \models_V p \simeq v \\
 & & \text{and } \sigma \models_V p - 1 \simeq \langle \mathbf{proc}, closure\, u'\pi'\rangle \\
 & & \text{and } \sigma(p-1).u < p \\
 & & \text{and } \sigma \models_U \sigma(p-1).u \simeq u' \\
 & \text{or} & u = (ext_R\langle\langle \mathbf{cont}, (cont\, \pi' u'\zeta\kappa)\rangle, \langle\rangle\rangle u') \\
 & & \text{and } \sigma(p) = \langle \mathbf{cont}, p'\rangle \\
 & & \text{and } p' = p + 3 \\
 & & \text{and } \sigma \models_K p' \simeq (cont\, \pi u\zeta\kappa) \\
 & & \text{and } \sigma(p-1).u < p \\
 & & \text{and } \sigma \models_U \sigma(p-1).u \simeq u'
\end{array}$$

Here we use the notation $\sigma(p).u$ to denote the contents of environment-pointer field of location p of σ. We will use similar notation for field selectors throughout. Note that this relation specifies that that the static chain is kept in the same field of the second word of the environment representation, whichever variation (**proc** or **cont**) is used.

The $p+3$ case arises when a *save-cont* instruction is executed. this instruction pushes a new continuation on the stack, using the format in item 5 above, and binds the continuation in the vacant environment frame.

- Relating Pointers and Continuations

$$\begin{array}{lll}
\sigma \models_K p \simeq \kappa & \Longleftrightarrow & \\
 & \text{either} & \kappa = initcont \text{ and } p = 0 \\
 & \text{or} & \kappa = (cont\, \pi u\zeta\kappa') \\
 & & \text{and } \sigma(p) = \pi \\
 & & \text{and } \sigma(p-1) < p \text{ and } \sigma \models_U \sigma(p-1) \simeq u \\
 & & \text{and } \sigma(p-2) < p \text{ and } \sigma \models_K \sigma(p-2) \simeq \kappa' \\
 & & \text{and } \sigma \models_Z \langle p-5, \sigma(p-2)\rangle \simeq \zeta
\end{array}$$

7 The Proof

We begin by stating some useful lemmas about these relations. The most important of these lemmas state that the changing locations in the stack above a pointer does not change the quantity that the pointer represents. Let $\sigma =_p \sigma'$ denote the proposition $\forall x, 0 \leq x \leq p \Longrightarrow \sigma(x) = \sigma'(x)$. Then we have the following:

Lemma 1 *1. If* $\sigma =_p \sigma'$ & $\sigma \models_V p \simeq v$ *then* $\sigma' \models_V p \simeq v$.

2. If $\sigma =_p \sigma'$ & $\sigma \models_Z \langle p, p'\rangle \simeq \zeta$ *then* $\sigma' \models_Z \langle p, p'\rangle \simeq \zeta$.

3. If $\sigma =_{up+3} \sigma'$ & $\sigma \models_U up \simeq u$ *then* $\sigma' \models_U up \simeq u$.

4. If $\sigma =_{kp} \sigma'$ & $\sigma \models_K kp \simeq \kappa$ *then* $\sigma' \models_K kp \simeq \kappa$.

Proof: By induction on the definition of $\simeq$. For stacks, the condition $\sigma \models_Z \langle p, p' \rangle$ implies $p' \leq p$. For environments, we need to write $up+3$ because the first slot might contain a continuation, we potentially can point 3 slots up in the stack. □

We can now define the correspondence between abstract and concrete machine states

Definition 1 *We say a concrete machine state* $\langle \pi, up, sp, kp, \sigma \rangle$ corresponds *to an abstract machine state* $\langle \pi', u, \zeta, \kappa \rangle$ *(written* $\langle \pi, up, sp, kp, \sigma \rangle \simeq \langle \pi', u, \zeta, \kappa \rangle$*) if and only if the following conditions are satisfied:*

1. $\pi = \pi'$,
2. $\sigma \models_U up \simeq u$,
3. $\sigma \models_Z (sp, kp) \simeq \zeta$, *and*
4. $\sigma \models_K kp \simeq \kappa$.

This says (among other things) that the local stack ζ is kept at the top of the stack, and is immediately followed by the representation of κ; the display u is threaded through the stack but need not be near the top.

With this definition of correspondence between machine states, we can work out the action of our instructions on the concrete machine. Figure 2 shows the most complex example, the *jsr* instruction. Here sp', up', and kp' denote the new values of the concrete machine registers. Since $sp' = kp'$, the new value of ζ is the empty stack $\langle\rangle$

A diagram such as that in Figure 2 can be translated into a formal representation more suitable for formal reasoning, similar to the definition of the abstract machine in Section 4. This is done in Figure 3. Here we use $v.kp$ to extract the continuation field from a stored continuation, and $v.\pi$ to extract the program field from a stored closure.

We can show that the following invariant is preserved by the machine:

Lemma 2 *In all machine states,* $sp \geq kp \geq up+3$.

The correctness of the auxiliary *find* is also shown:

Lemma 3 *If* $\sigma \models_U up \simeq u$ *then*

$$\sigma \models_V (\mathit{find}\ l\ up\, \sigma) \simeq u(l)$$

Now we can show the main theorem, which asserts that the concrete machine correctly simulates the abstract machine:

Theorem 3 *Let* A_1 *and* A_2 *be states of the abstract machine and* C_1 *and* C_2 *be states of the concrete machine. If* $C_1 \simeq A_1$, $A_1 \Longrightarrow A_2$, *and* $C_1 \Longrightarrow C_2$, *then* $C_2 \simeq A_2$

Proof By analysis of each instruction Here we show two cases, *fetch* and *jsr*.

- Assume we have $\langle (\mathit{fetch}\ l\ \pi), up, sp, kp, \sigma \rangle \simeq \langle (\mathit{fetch}\ l\ \pi), u, \zeta, \kappa \rangle$ We must show that

  $$\langle \pi, up, sp+1, kp, \sigma' \rangle \simeq \langle \pi, u, (u(l) :: \zeta), \kappa \rangle$$

  where $\sigma' = \sigma[(\mathit{lookup}\ l\ up\, \sigma)/(sp+1)]$. We consider each of the conditions for $\simeq$ in turn:

  1. $\pi = \pi$
  2. We have $sp+1 > up+3$ and $\sigma \models_U up \simeq u$. Therefore $\sigma' \models_U up \simeq u$ by Lemma 1.
  3. We have $\sigma \models_V (\mathit{find}\ l\ up\, \sigma) \simeq u(l)$ and $\sigma \models_Z \langle sp, kp \rangle \simeq \zeta$. Hence $\sigma' \models_Z \langle sp+1, kp \rangle \simeq (u(l) :: \zeta)$
  4. $sp+1 > kp$ and $\sigma \models_K kp \simeq \kappa$. Therefore $\sigma' \models_K kp \simeq \kappa$

- Assume we have $\langle (jsr\, \pi), up, sp, kp, \sigma \rangle \simeq \langle (jsr\, \pi), u, (v :: \langle \mathbf{proc}, \mathit{closure}\, u' \pi' \rangle :. \zeta), \kappa \rangle$. We need

  $$\begin{aligned}&\langle \sigma(sp-1).\pi, sp, sp+3, sp+3, (\mathit{push\text{-}cont}\, up\, sp\, kp\, \sigma) \rangle \\ &\simeq \langle \pi', (\mathit{ext}_R \langle v, \langle \mathbf{proc}, \mathit{closure}\, u' \pi' \rangle \rangle u'), \langle\rangle, (\mathit{cont}\, \pi u \zeta \kappa) \rangle\end{aligned}$$

  where $\zeta' = (v :: \langle \mathbf{proc}, \mathit{closure} u' \pi' \rangle :: \zeta)$ and $\sigma' = (\mathit{push\text{-}cont}\, up\, sp\, kp\, \sigma)$ By the definition of *push-cont*, σ and σ' agree on all locations less than or equal to sp.

  1. We have $\sigma \models_Z \langle sp, kp \rangle \simeq \zeta'$ and $|\zeta'| \geq 2$. Hence $\sigma(sp-1).\pi = \pi'$.
  2. $\sigma \models_Z \langle sp, kp \rangle \simeq \zeta'$ and $|\zeta'| \geq 2$ Hence $\sigma' \models_U sp \simeq (\mathit{ext}_R \langle v, \langle \mathbf{proc}, \mathit{closure} u' \pi' \rangle \rangle u')$.
  3. $\sigma' \models_Z \langle sp+3, sp+3 \rangle \simeq \langle\rangle$
  4. $\sigma'(sp+3) = \pi$
     $\sigma'(sp+2) = up+3 < sp+1$ and $\sigma \models_U up \simeq u$
     $\Longrightarrow \sigma' \models_U \sigma'(sp+2) \simeq u$
     $\sigma'(sp+1) = kp < sp+1$ and $\sigma \models_K kp \simeq \kappa$
     $\Longrightarrow \sigma' \models_K \sigma'(sp+1) \simeq \kappa$
     $\sigma \models_Z \langle sp, kp \rangle \simeq \zeta'$ and $|\zeta'| \geq 2 \Longrightarrow \sigma' \models_Z \langle sp-2, \sigma'(sp+1) \rangle \simeq \zeta$
     $\Longrightarrow \sigma' \models_K sp+3 \simeq (\mathit{cont}\, \pi u \zeta \kappa)$

□

This theorem shows that the concrete machine and the abstract machine compute in the same way. started in corresponding states, they compute by passing through a sequence of corresponding states. Thus the concrete machine will give the same answer as the abstract machine

This theorem is not surprising, of course. What is new here is a way of formalizing the representation of environments, continuations, etc., so that we can see how even very low-level structures directly represent the quantities in the language definition. Furthermore, this organization allows us to separate denotational from operational reasoning, so that each can be used to best advantage.

8 Code Representation

We can similarly represent the program trees π in a linear program store M. The details here are similar, but easier. Instead of storing a program π in a single word of memory, as we did in the previous machine, we represent π explicitly in a linear program store M. For example, the program $(\mathit{fetch}\ l\ \pi')$ would be represented as a location p in M.

p :	*fetch*
	l

where $p+2$ represented π'. We assume that M has a domain that is disjoint from the data store σ and that its range is

Abstract machine:

$$\langle (jsr\,\pi), u, (v :: \langle \mathbf{proc}, closure\,u'\pi' \rangle :: \zeta), \kappa \rangle \Longrightarrow$$
$$\langle \pi', (ext_R \langle v, \langle \mathbf{proc}, closure\,u'\pi' \rangle \rangle u'), \langle \rangle, (cont\,\pi u \zeta \kappa) \rangle$$

Concrete machine:

	Before		After	Comments
		sp', kp' :	π	return address
			up	saved environment pointer
			kp	dynamic chain
sp :	v	up' :	v	old top of ζ, now in environment
	$\langle \mathbf{proc}, closure\,u'\pi' \rangle$		$\langle \mathbf{proc}, closure\,u'\pi' \rangle$	closure, including static chain
	v_n		v_n	rest of ζ, now saved
	...		...	
kp :	...		...	representation of κ

Figure 2: Executing a *jsr* instruction on the abstract and concrete machines.

Concrete machine:

$$\begin{array}{rcl}
\langle (fetch\ l\ \pi), up, sp, kp, \sigma \rangle & \Longrightarrow & \langle \pi, up, sp+1, kp, \sigma[(lookup\ l\ up\,\sigma)/\ sp+1] \rangle \\
\langle (add\,\pi), up, sp, kp, \sigma \rangle & \Longrightarrow & \langle \pi, up, sp-1, kp, \sigma[(int\text{-}add\,sp(sp-1)\sigma)/\ sp-1] \rangle \\
\langle (brz\,\pi'\pi), up, sp, kp, \sigma \rangle & \Longrightarrow & \langle (choose\ \sigma(sp)\,\pi'\,\pi), up, sp-1, kp, \sigma \rangle \\
\langle (save\text{-}cont\,\pi'\pi), up, sp, kp, \sigma \rangle & \Longrightarrow & \langle \pi', sp+1, sp+5, sp+5, (ext\text{-}cp\,\pi\ up\,sp\,kp\,\sigma) \rangle \\
\langle (throw), up, sp, kp, \sigma \rangle & \Longrightarrow & \langle \sigma(\sigma(sp).\,kp), \sigma((\sigma(sp).kp)-1), \sigma(sp).\,kp-4, \\
 & & \quad \sigma((\sigma(sp).kp)-2), \sigma[\sigma(sp-1)/\sigma(sp).kp-4] \rangle \\
\langle (close\,\pi'\pi), up, sp, kp, \sigma \rangle & \Longrightarrow & \langle \pi, up, sp+1, kp, \sigma[\langle \mathbf{proc}, closure\,up\,\pi' \rangle/\ sp+1] \rangle \\
\langle (rts), up, sp, kp, \sigma \rangle & \Longrightarrow & \langle \sigma(kp), \sigma(kp-1), kp-4, \sigma(kp-2), \sigma[\sigma(sp)/kp-4] \rangle \\
\langle (jsr\,\pi), up, sp, kp, \sigma \rangle & \Longrightarrow & \langle \sigma(sp-1).\pi, sp, sp+3, sp+3, (push\text{-}cont\,\pi\ up\,sp\,kp\,\sigma) \rangle
\end{array}$$

Auxiliaries:

$$\begin{gathered}
(lookup\langle n, m\rangle\ up\,\sigma) = \sigma(find\langle n, m\rangle\ up\,\sigma) \\
(find\langle n, m\rangle\ up\,\sigma) = ()(n = 0) - (up-m), (find\langle n-1, m\rangle(\sigma(up).up)\sigma) \\
(int\text{-}add\,p\,p'\,\sigma) = \langle \mathbf{int}, (\sigma(p)\ int) + (\sigma(p').int) \rangle \\
(choose\ v\,\pi'\,\pi) = ((v = \langle \mathbf{int}, 0 \rangle) \to \pi, \pi') \\
(push\text{-}cont\,\pi\ up\,sp\,kp\,\sigma) = \sigma[kp/\ sp+1, up/\ sp+2, \pi/\ sp+3] \\
(ext\text{-}cp\,\pi\ up\,sp\,kp\,\sigma) = (push\text{-}cont\,\pi\ up\,sp+2\ kp\,\sigma)[up/\ sp+1, \langle \mathbf{cont}, sp+5 \rangle/\ sp+2]
\end{gathered}$$

Figure 3: Actions of the concrete machine.

large enough to hold a single opcode or a pointer (either to data or program).

Again, we define a storage layout relation $M \models_P p \simeq \pi$ as the least fixpoint of a monotonic operator. (Note the presence of *goto*, which prevents the use of structural induction. The *goto* instruction is not necessary, but illustrates the power of the technique).

- Relating pointers and programs

$$
\begin{array}{lll}
M \models_P p \simeq \pi \iff & & \\
& \text{either} & \pi = (\mathit{fetch}\ l\ \pi') \\
& & \text{and } M(p) = \mathit{fetch} \text{ and } M(p+1) = l \\
& & \text{and } M \models_P p+2 \simeq \pi' \\
& \text{or} & \pi = (\mathit{add}\,\pi') \\
& & \text{and } M(p) = \mathit{add} \text{ and } M \models_P p+1 \simeq \pi' \\
& \text{or} & \pi = (\mathit{brz}\,\pi'\pi'') \\
& & \text{and } M(p) = \mathit{brz} \\
& & \text{and } M \models_P M(p+1) \simeq \pi' \\
& & \text{and } M \models_P p+2 \simeq \pi'' \\
& \text{or} & \pi = (\mathit{save\text{-}cont}\,\pi'\pi'') \\
& & \text{and } M(p) = \mathit{save\text{-}cont} \\
& & \text{and } M \models_P M(p+1) \simeq \pi' \\
& & \text{and } M \models_P p+2 \simeq \pi'' \\
& \text{or} & \pi = \mathit{rts} \text{ and } M(p) = \mathit{rts} \\
& \text{or} & \pi = \mathit{throw} \text{ and } M(p) = \mathit{throw} \\
& \text{or} & \pi = (\mathit{close}\,\pi'\pi'') \\
& & \text{and } M(p) = \mathit{close} \\
& & \text{and } M \models_P M(p+1) \simeq \pi' \\
& & \text{and } M \models_P p+2 \simeq \pi'' \\
& \text{or} & \pi = (\mathit{jsr}\,\pi') \\
& & \text{and } M(p) = \mathit{jsr} \text{ and } M \models_P p+1 \simeq \pi' \\
& \text{or} & M(p) = \mathit{goto} \text{ and } M \models_P M(p+1) \simeq \pi
\end{array}
$$

- Relating stored values

  Stored values correspond if they are alike except for the representation of programs:

$$
\begin{array}{lll}
M \models_{SV} sv \simeq sv' \iff & & \\
& \text{either} & sv = \langle \mathbf{proc}, (\mathit{closure}\ up\ p)\rangle \\
& & \text{and } sv' = \langle \mathbf{proc}, (\mathit{closure}\ up\ \pi)\rangle \\
& & \text{and } M \models_P p \simeq \pi \\
& \text{or} & sv = p \text{ and } sv' = \pi \text{ and } M \models_P p \simeq \pi \\
& \text{or} & sv = sv'
\end{array}
$$

- Relating data stores

  Data stores correspond up to pointer p if they correspond location by location for all smaller locations:

$$
\begin{array}{l}
M \models_p \sigma \simeq \sigma' \iff \\
\quad 0 \leq p' \leq p, M \models_{SV} \sigma(p') \simeq \sigma'(p')
\end{array}
$$

We call the machine using this linear program store the *stored-program machine*. In order to describe the machine using simple pattern-matching rules, we specify the stored-program machine using a fetch-execute cycle. The controller of the machine has two states, fetch and execute. In the fetch state, the machine uses 4 registers: an instruction pointer ip, which is a pointer into the program store, pointers up, sp, and kp, as in the preceding machine. In the execute state, the machine uses in addition an instruction register ir.

In the fetch state, the machine retrieves the opcode to which the instruction pointer points, places it in the instruction register, and goes to the execute state:

$$F(ip, up, sp, kp, \sigma) \Longrightarrow E(ip, M(ip), up, sp, kp, \sigma)$$

The execute state then performs a rewrite, dispatching on the contents of the instruction register, and returns to the fetch state. The rewriting system for the stored-program machine is shown in Figure 4. This is similar to Figure 3 except for the treatment of pointers to programs.

Now we can define the correspondence between the concrete and stored-program machines. We define the correspondence at the start of the fetch-execute cycle, that is, at the fetch state of the stored-program machine. Since the pointers into the data store are exactly the same, all that is required is that the programs correspond, and that the stores correspond up to sp.

Definition 2 *We say a stored-program machine state $F(ip, up, sp, kp, \sigma)$ corresponds to a concrete machine state $\langle \pi', up', sp', kp', \sigma' \rangle$ (written $M \models F(ip, up, sp, kp, \sigma) \simeq \langle \pi, up, sp, kp, \sigma' \rangle$) if and only if:*

1. $M \models_P ip \simeq \pi'$
2. $up = up'$
3. $sp = sp'$
4. $kp = kp'$
5. $M \models_{sp} \sigma \simeq \sigma'$

Now we can state the main theorem:

Theorem 4 *Let C_1 and C_2 be states of the concrete machine and L_1 and L_2 be fetch states of the stored-program machine. If $M \models L_1 \simeq C_1$, $C_1 \Longrightarrow C_2$, and $L_1 \Longrightarrow L_2$, then $M \models L_2 \simeq C_2$.*

Proof By analysis of each instruction. This is tedious but straightforward; since M is immutable, we do not need free-location lemmas like Lemma 1. □

If we call the algorithm that translates from program trees π to stored programs the "assembler," then the correctness of the assembler factors into two parts:

1. Showing that the assembler is correct: that it emits a stored program corresponding to the program tree π.
2. Showing that if a stored program corresponds to a program tree π, then the machines will deliver corresponding outputs.

These two steps are somewhat analogous to the decomposition of the compiler: showing that the compiler is correct by showing it emits code that meets its specification (Theorem 1), and then showing that the bytecode machine simulates reduction of the bytecode term (Theorem 2).

As in the compiler case, the second part of the proof is independent of the assembler algorithm, and that is the part we have shown here; what remains is the first, assembler-dependent part. We hope to report on that elsewhere.

Stored-Program machine:

$$\begin{array}{rcl}
F(ip, up, sp, kp, \sigma) & \Longrightarrow & E(ip, M(ip), up, sp, kp, \sigma) \\
E(ip, fetch, up, sp, kp, \sigma) & \Longrightarrow & F(ip{+}2, up, sp{+}1, kp, \sigma[(lookup\ M(ip+1)\ up\, \sigma)/\ sp{+}1]) \\
E(ip, add, up, sp, kp, \sigma) & \Longrightarrow & F(ip{+}1, up, sp{-}1, kp, \sigma[(int\text{-}addsp(sp{-}1)\sigma)/\ sp{-}1]) \\
E(ip, brz, up, sp, kp, \sigma) & \Longrightarrow & F((choose(\sigma(sp))(ip{+}2)(M(ip{+}1))), up, sp{-}1, kp, \sigma) \\
E(ip, save\text{-}cont, up, sp, kp, \sigma) & \Longrightarrow & F(M(ip{+}1), sp{+}1, sp{+}5, sp{+}5, (ext\text{-}cp(ip{+}2)\ up\, sp\, kp\, \sigma)) \\
E(ip, throw, up, sp, kp, \sigma) & \Longrightarrow & F((\sigma(sp).\,kp), \sigma(\sigma(sp).kp - 1), \sigma(sp).\,kp{-}4, \\
 & & \quad \sigma(\sigma(sp).kp - 2), \sigma[\sigma(sp{-}1)/\sigma(sp).kp - 4]) \\
E(ip, close, up, sp, kp, \sigma) & \Longrightarrow & F(ip{+}2, up, sp{+}1, kp, \sigma[\langle \mathbf{proc}, closure\ up\, M(ip{+}1)\rangle/\ sp{+}1]) \\
E(ip, rts, up, sp, kp, \sigma) & \Longrightarrow & F(\sigma(kp), \sigma(kp{-}1), kp{-}4, \sigma(kp{-}2), \sigma[\sigma(sp)/kp - 4]) \\
E(ip, jsr, up, sp, kp, \sigma) & \Longrightarrow & F(\sigma(sp - 1).\,ip, sp, sp{+}3, sp{+}3, (push\text{-}cont(ip{+}1)\ up\, sp\, kp\, \sigma))
\end{array}$$

Auxiliaries:

$$\begin{gathered}
(lookup\langle n, m\rangle\ up\, \sigma) = \sigma(find\langle n, m\rangle\ up\, \sigma) \\
(find\langle n, m\rangle\ up\, \sigma) = (n = 0) \rightarrow (up{-}m), (find\langle n-1, m\rangle(\sigma(up).up)\sigma) \\
(int\text{-}add\, p\, p'\sigma) = \langle \mathbf{int}, (\sigma(p).int) + (\sigma(p').int)\rangle \\
(choose\ v\ ip'\ ip) = (v = \langle \mathbf{int}, 0\rangle) \rightarrow ip, ip' \\
(push\text{-}cont\ ip\, up\, sp\, kp\, \sigma) = \sigma[kp/\ sp{+}1, up/\ sp{+}2, ip/\ sp{+}3] \\
(ext\text{-}cp\, ip\, up\, sp\, kp\, \sigma) = (push\text{-}cont\ ip\, up\, sp{+}2\, kp\, \sigma)[up/\ sp{+}1, \langle \mathbf{cont}, sp{+}5\rangle/\ sp{+}2]
\end{gathered}$$

Figure 4: The fetch/execute cycle of the stored-program machine.

9 Partial Correctness and Untagged Representations

The stack machine derivation that we have presented is only one application of these proof strategies. In this section we will sketch another such application.

Our implementation still differs from an "ordinary" implementation of a stack-based language such as C or Pascal in that data values in C or Pascal do not include tags like our **int** or **proc**. Tags like these would impose a considerable run-time penalty in both time and space.[1]

In languages like C or Pascal, the compiler typically uses some form of type-checking to ascertain that primitive operations like procedure call or arithmetic are invoked only on legal quantities. For languages that are not type-safe, such as C, these compile-time checks are of course less than foolproof, and may lead to unpredictable consequences (e.g. core dumps).

Here we have a semantic mismatch: the semantics is given using disjoint sums, with specified error behavior (continuations of the form $(error\ \ldots)$), but the implementation uses non-disjoint sums and has unspecified error behavior. In what sense can we say that an implementation of such a language is correct?

We can model this situation by using untagged storage layout relations and weakening the requirements on the implementation. To get a storage layout relation for an untagged implementation, we replace pattern-matching in the definition of the storage relation by field selection. Thus we might replace

$$\begin{array}{lll}
\sigma \models_V p \simeq v \iff & & \\
\quad \text{either} & v = \langle \mathbf{int}, n\rangle = \sigma(p) & \\
\quad \text{or} & v = \langle \mathbf{proc}, (closure\, u\, \pi_1)\rangle & \\
 & \quad \text{and } \sigma(p) = \langle \mathbf{proc}, (closure\, up\, \pi_1)\rangle & \\
 & \quad \text{and } up < p \text{ and } \sigma \models_U up \simeq u & \\
\quad \ldots & &
\end{array}$$

by

$$\begin{array}{lll}
\sigma \models_V p \simeq v \iff & & \\
\quad \text{either} & v = \langle \mathbf{int}, n\rangle \text{ and } \sigma(p) = n & \\
\quad \text{or} & v = \langle \mathbf{proc}, (closure\, u\, \pi_1)\rangle & \\
 & \quad \text{and } M \models_P \sigma(p).\pi \simeq \pi_1 & \\
 & \quad \text{and } \sigma(p).u < p \text{ and } \sigma \models_U \sigma(p)\ u \simeq u & \\
\quad \ldots & &
\end{array}$$

We can weaken the requirement on the implementation by saying that if the abstract machine proceeds without invoking an error, then the concrete machine must do so also. If the abstract machine invokes the error continuation, then the behavior of the concrete machine is unspecified. This contract corresponds to the Scheme notion [4] of "is an error." With this contract, the correctness theorem becomes:

Theorem 5 *Let A_1 and A_2 be states of the abstract machine and C_1 and C_2 be states of the concrete machine. If $C_1 \simeq A_1$, $A_1 \Longrightarrow A_2$, A_2 is not an error state, and $C_1 \Longrightarrow C_2$, then $C_2 \simeq A_2$.*

The only change from Theorem 3 is the addition of the phrase "A_2 is not an error state." The proof goes as before.

10 Conclusions and Further Work

We have shown the correctness of a stack implementation of a programming language. The stack was shown to be a

[1]The benefits of tag elimination are less clear for languages that require garbage collection, though even in this case it may be possible to remove all tags [1, 6]

data structure that represented three different terms in the bytecode machine. The techniques generalize those introduced by Hannan [7]. We believe that these techniques will generalize to other crucial representation strategies, such as caching portions of the local stack ζ in registers, representing mutable entities with dynamic extent on the stack, and modelling program loops by circular storage structures. These techniques, especially the variant in Section 9, should also be adaptable to model the finite word size of real machines.

Acknowledgements

Paul Hudak originally suggested the correctness of a stack-discipline language as a useful exercise. Joshua Guttman, John Ramsdell, Vipin Swarup, Larry Monk, and Bill Farmer of the MITRE Corporation provided useful feedback on our Pure PreScheme compiler, which led us to the questions considered in this paper.

References

[1] Appel, A.W., "Runtime Tags Aren't Necessary," *Lisp and Symbolic Computation 2* (1989), 153–162.

[2] Clément, D., Despeyroux, J., Despeyroux, T., and Kahn, G. "A Simple Applicative Language: Mini-ML" *Proc. 1986 ACM Symp. on Lisp and Functional Programming*, 13–27.

[3] Clinger, W. "The Scheme 311 Compiler An Exercise in Denotational Semantics," *Conf. Rec. 1984 ACM Symposium on Lisp and Functional Programming* (August, 1984), 356–364

[4] Clinger, W., and Rees, J., eds. "Revised^4 Report on the Algorithmic Language Scheme", Indiana University Computer Science Department Technical Report No. 341, November, 1991, to appear in *LISP Pointers*, 1992 See also "Revised^3 Report on the Algorithmic Language Scheme", *SIGPLAN Notices 21*,12 (December, 1986), 37–79.

[5] Friedman, D.P., Wand, M., and Haynes, C.T., *Essentials of Programming Languages*, MIT Press, Cambridge, MA, and McGraw-Hill, Chicago, 1992.

[6] Goldberg, B. "Tag-Free Garbage Collection for Strongly Typed Programming Languages," *Proc. ACM SIGPLAN '91 Symp. on Programming Language Design and Implementation*, June, 1991, 165–176.

[7] Hannan, J. "Making Abstract Machines Less Abstract," *Functional Programming Languages and Computer Architecture, 5th ACM Conference* (J. Hughes, ed.) Springer Lecture Notes in Computer Science, Vol. 523 (1991), 618–635.

[8] Lee, P. *Topics in Advanced Language Implementation*, MIT Press, Cambridge, MA, 1991

[9] Farmer, W.M., Guttman, J.D., Monk, L.G., Ramsdell, J.D., and Swarup, V. "VLISP. A Verified Language Implementation: FY 91 Year-End Report," The MITRE Corporation Technical Report MTR11232, October, 1991.

[10] Oliva, D.P., and Wand, M., "A Verified Compiler for Pure PreScheme," Final Report for MITRE Contract Number F19628-89-C-001. Included in [9].

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

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

[13] Plotkin, G.D. "Call-by-Name, Call-by-Value and the λ-Calculus," *Theoret. Comp Sci. 1* (1975) 125–159.

[14] Raoult, J.-C. and Sethi, R. "The Global Storage Needs of a Subcomputation," *Conf. Rec. 11th ACM Symp. on Principles of Programming Languages* (1984), 148–157.

[15] Reynolds, J.C. *The Craft of Programming*, Prentice-Hall International, 1981.

[16] Schmidt, D A. *Denotational Semantics: A Methodology for Language Development* Boston: Allyn and Bacon, 1986.

[17] Stoy, J.E. *Denotational Semantics. The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA., 1977.

[18] Wand, M. "Semantics-Directed Machine Architecture" *Conf. Rec. 9th ACM Symp. on Principles of Prog. Lang.* (1982), 234–241

[19] Wand, M. "Loops in Combinator-Based Compilers," *Info. and Control 57*, 2–3 (May/June, 1983), 148–164.

[20] Wand, M. "On the Correctness of Procedure Representations in Higher-Order Assembly Language" *Proc. MFPS '91* (S. Brookes, ed) Springer Lecture Notes in Computer Science, to appear