State-Transition Machines for Lambda-Calculus Expressions

David A. Schmidt* (schmidt@cis.ksu.edu)
Computer Science Department, Aarhus University, Aarhus, Denmark

Abstract. The process of compiler generation from lambda-calculus definitions is studied. The compiling schemes developed utilize as their object language the set of *state transition machines (STMs)*: automata-like transition sets using first-order arguments. An intermediate definition form, the *STM-interpreter*, is treated as central to the formulation of STMs. Three compiling schemes are presented: one derived directly from an STM-interpreter for the lambda-calculus; one formulated from an STM-interpreter variant of Landin's SECD-machine; and one defined through meaning-preserving transformations upon a denotational definition of the lambda-calculus. The results are compared and some tentative conclusions are made regarding the utility of compiler generation with the STM forms.

1. Introduction

The work in this paper stems from the conjecture that once one has defined a programming language via formal means and has selected a target (object) language, then a class of compilers for the language is implicitly described. The diverse levels of formal definitions and object languages make it difficult to formalize the actions taken to develop these compilers. Consequently, we explore compiler development from formal definitions transformable to a primitive operational form, the *STM-interpreter*. An STM-interpreter generates transitions that belong to an object language of *state-transition machines (STMs)*.

For our studies, the lambda-calculus is used as the source language interpreted by the STM-interpreter because it is a well-known universal language. Three compiling schemes are developed: one derived directly from an STM-interpreter; one formulated from an STM-interpreter variant of Landin's SECD-machine; and one defined through meaning-preserving transformations upon a denotational definition of the lambda-calculus itself [11]. The different starting points provide insight into the techniques of compiler generation via the STM format. Finally, conclusions are drawn as to the utility of a compiler-generation methodology based on use of the STM forms.

* Permanent address: Computing and Information Science Department, Kansas State University, Manhattan, KS 66506 USA

2. STMs and STM-interpreters

The automata-like language of state-transition machines (STMs) is used as the object language for the compiling schemes. Informally stated, an STM is a finite-state automaton, where each state possesses a finite number of first-order (non-functional) arguments. The actions upon the arguments are limited to a set of "machine primitive" operations (e.g., addition, concatenation) and are performed when a transition from one machine state to another occurs. The STM is specified by a set of transition rules, each rule stating the possible state, argument pairs reachable from the current control state. The STM format provides a structure which is low level but not yet tied to any machine architecture. A transformation analogous to assembly could be applied to obtain concrete object code.

Definition 1. An STM is a $\langle \mathcal{S}, \mathcal{E} \rangle$ pair, where

- $\mathcal{S} = \{s_0, \cdots, s_m\}$, a finite set of *state names*, including the *entry state*, s_0;
- $\mathcal{E} = \{s_i, x \Rightarrow t_i\}_{1 \leq i \leq m}$, a finite set of rewriting rules, where $s_i \in \mathcal{S}$, x is a variable name, and t_i has the form,
  1. s_j, e, a jump transition, where $1 \leq j \leq m$ and e is a first-order expression, defined below;
  2. e_1, e_2, a computed state transition, whose next state is computed by evaluating the first-order expression, e_1;
  3. $e \rightarrow t_1 \, [\!] \, t_2$, a conditional state transition, where e is a boolean-valued expression which selects from t_1 and t_2, which themselves have the form just stated in Items 1 and 2;

  where e is an expression composed of first-order constants (0, 1, *nil*, ...), primitive operations defined on first-order values ($+$, $*$, $\langle \cdots \rangle$, $\downarrow i$, ...), state names, and variable x.

Typically, a formal presentation of an STM can be deduced from an informal one; we use the informal version in most cases. In particular, when a left-hand-side variable, x, represents a tuple, $\langle e_1, \cdots, e_n \rangle$, we use the tupled-pattern form instead of x.

A *state configuration* is a pair, s, d, where $s \in \mathcal{S}$ and d is a first-order value (constant or tuple).

An example of an STM that computes the factorial function (using pairs and the numerical operations equality, subtraction, and multipli-

cation) is

$$\mathcal{S} = \{s_0, s_1, s_2\}$$

$$\begin{aligned}\mathcal{E} = \{ & s_0, x \;\Rightarrow\; s_1, \langle x, 1\rangle \\ & s_1, \langle x, y\rangle \;\Rightarrow\; (x = 0) \rightarrow s_2, y \,[\!]\, s_1, \langle x - 1, x * y\rangle \;\}\end{aligned}$$

An *STM-computation history* is a sequence of state configurations generated from an STM and an initial state configuration whose state name is the entry state. Here is an STM- computation history for the previous STM and the initial configuration, $s_0, 2$:

$$s_0, 2 \Rightarrow s_1, \langle 2, 1\rangle \Rightarrow s_1, \langle 1, 2\rangle \Rightarrow s_1, \langle 0, 2\rangle \Rightarrow s_2, 2$$

If a programming language's semantics is defined by an interpreter, we say that the interpreter is an *STM-interpreter* if, for every input program and its data, the interpreter generates an STM-computation history.

Our objective is to transform STM-interpreters into compilers that translate source programs into STMs. We will study how this is done for three different STM-interpreters for the lambda-calculus.

3. The lambda-calculus

Following convention [1], the set of lambda-calculus expressions, Exp, is the smallest set formed from a set of variables, $Var = \{x, y, z, \cdots\}$, and symbols, λ, (,), such that

(i) a variable, x, is a member of Exp, that is, $x \in Var \subseteq Exp$;

(ii) if $x \in Var$ and $B \in Exp$, then $(\lambda x\, B)$ is a member of Exp, that is, $(\lambda x\, B) \in Abs \subseteq Exp$;

(iii) if $M, N \in Exp$, then $(M\, N)$ is a member of Exp, that is, $(M\, N) \in Comb \subseteq Exp$.

Abs is the set of *abstractions* and *Comb* is the set of *combinations*. We abbreviate expressions of the form $(\lambda x(B))$ to $(\lambda x\, B)$ and $((M\, N)P)$ to $(M\, N\, P)$. Outermost parentheses will be dropped in most cases.

Using the standard meanings of the terms, *free variable*, *bound variable*, and *closed term* [1], $[M/x]B$ denotes the syntactic substitution of expression M for all free occurrences of x in B (with the renaming of bound variables in B to avoid name clashes with free variables in M). The conversion rules are

$$\begin{aligned}&\alpha\text{: } \lambda x\, B \;>\; \lambda y [y/x] B \\ &\beta\text{: } (\lambda x\, B)\, M \;>\; [M/x] B\end{aligned}$$

The α-rule renames bound variables, and the β-rule binds an argument to an abstraction. The utility of the two rules is augmented by these contextual rules:

$$M > M \quad \frac{M > M'}{\lambda x\, M > \lambda x\, M'} \quad \frac{M > M' \;\; N > N'}{M\, N > M'\, N'}$$

The expression, $M >^* N$, denotes the application of zero or more of the above rules to M to obtain N.

A lambda-expression has *normal form* if it contains no subexpression (*redex*) of the form $(\lambda x\, B)\, M$. An expression has a *head redex* [2] if it has the form, $(\lambda x\, B)\, N_1 \cdots N_m$, for $m \geq 1$. (Note that the term, "head redex," is used differently from that in [13].) A notion we find useful is weak-normal form: *a lambda-expression has weak-normal form when it has no head redex.*

For expressions, M and N, the assertion $M \equiv N$ states that M and N are syntactically identical with the exception of bound-variable names, that is, a finite number of applications of the α-rule to M yields N.

The following result is well known:

Theorem 1. [2]: If $M >^* N_1$, $M >^* N_2$, and both N_1 and N_2 have normal form, then $N_1 \equiv N_2$.

The analogue does not hold for weak-normal form. For example,

$$(\lambda x\, x)\, y((\lambda x\, x)\, y) >^* y((\lambda x\, x)\, y)$$

and also

$$(\lambda x\, x)\, y((\lambda x\, x)\, y) >^* y\, y.$$

A given expression may reduce to many distinct weak-normal forms. But if we restrict all uses of the β-rule to *head redexes only*, the resulting determinism produces a unique weak-normal form (if one exists).

The expression, $M \xrightarrow{*} N$, denotes zero or more uses of β-reduction restricted to head redexes. We study $\xrightarrow{*}$ because it describes the "sequential" evaluation of a lambda-calculus "program" — instructions are executed from first to last, and procedure bodies are evaluated only when actual parameters are bound to formal ones.

If a lambda-calculus expression reduces to a *ground value* (here, ground values would be some subset of *Var*), then $\xrightarrow{*}$ is exactly leftmost reduction, and the standardization theorem (that leftmost reduction is adequate for discovering normal forms) [2] guarantees that computation using $\xrightarrow{*}$ produces the expected result.

4. A lambda-calculus machine

We now develop an STM-interpreter for the lambda-calculus, based on the tenets that:

(i) a lambda-calculus expression represents a computation state;

(ii) application of the β-rule causes a state transition.

The interpreter will use as its state names the subexpressions of the input lambda-calculus expression. (The subexpressions can be represented by labels, if desired.) The binding of variables in β-reduction will be accounted for by an environment argument. An operand stack for handling nested applications is also needed, because we treat a lambda-calculus expression as a nested application, $(M_0\, M_1 \cdots M_n)$, which we represent as the state configuration,

$$M_0,\ \langle e_0,\ \langle M_1, e_1\rangle : \cdots : \langle M_n, e_n\rangle\rangle$$

where M_0 is the state name, e_0 holds the bindings for free variables in M_0, and $\langle M_1, e_1\rangle : \cdots : \langle M_n, e_n\rangle$ is the operand stack holding the representations of $M_1 \cdots M_n$.

For ease of reading, we write the state configuration as the triple,

$$M_0,\ e_0,\ \langle M_1, e_1\rangle : \cdots : \langle M_n, e_n\rangle$$

which uses the following notation: $\langle a_1, a_2, \cdots, a_n\rangle$ denotes an n-tuple, and $\langle a_1, a_2, \cdots, a_n\rangle \downarrow i = a_i$, for all $1 \leq i \leq n$, denotes the indexing operations. We simulate stacks with nested pairs: Given value a and tuple L, let $a : L$ denote the pair, $\langle a, L\rangle$; let the empty tuple be $\langle\,\rangle$; and abbreviate $a_1 : \cdots : a_n : \langle\,\rangle$ to $a_1 : \cdots : a_n$.

To perform lookup in a stack, m, let $m[x] = \langle y_1, \cdots, y_n\rangle$, if $\langle x, y_1, \cdots, y_n\rangle$ is the leftmost tuple in m whose first component is x. That is,

$$\begin{array}{l}(\langle x, y_1, \cdots, y_n\rangle : L)[x] = \langle y_1, \cdots, y_n\rangle\\ (\langle z, y_1, \cdots, y_n\rangle : L)[x] = L[x],\ \text{if } x \neq z.\end{array}$$

Finally, for lambda-calculus expression, M, let $\mathcal{L}(M)$ define the set of subexpressions (or their labels) of M.

We now define the interpreter. Its state is a triple, $s,\ e,\ c$, where

$$\begin{array}{l}s \in \mathcal{L}(M),\ \text{the state name}\\ e \in Env = (Var \times \mathcal{L}(M) \times Env)^*,\ \text{the environment}\\ c \in Cont = (\mathcal{L}(M) \times Env)^*,\ \text{the operand stack.}\end{array}$$

The transition function for the STM-interpreter is defined in Figure 1. Rule (1.1) is a computed state transition, and (1.2) and (1.3) are jump

$$
\begin{array}{lll}
x,\ e,\ c & \Rightarrow\ e[x]\downarrow 1,\ e[x]\downarrow 2,\ c & \text{if } x \in \mathit{Var} \qquad (1.1)\\
\lambda x B,\ e,\ \langle M, e'\rangle : c & \Rightarrow\ B,\ \langle x, M, e'\rangle : e,\ c & \text{if } \lambda x B \in \mathit{Abs} \qquad (1.2)\\
M N,\ e,\ c & \Rightarrow\ M,\ e,\ \langle N, e\rangle : c & \text{if } M N \in \mathit{Comb} \qquad (1.3)
\end{array}
$$

Figure 1. STM-interpreter for lambda-calculus

For $M = (\lambda y(\lambda y(yy))y)(\lambda x x)$, let the following numeric labels denote M's subexpressions:

$$((\lambda y((\lambda y(y^5 y^6)^4)^3 y^7)^2)^1 (\lambda x x^9)^8)^0$$

$$
\begin{array}{lll}
\mathit{init}(M) & = 0,\ \langle\,\rangle,\ \langle\,\rangle & \\
& \Rightarrow 1,\ \langle\,\rangle,\ \langle 8, \langle\,\rangle\rangle & \\
& \Rightarrow 2,\ \langle y, 8, \langle\,\rangle\rangle,\ \langle\,\rangle & \text{let } e_1 = \langle y, 8, \langle\,\rangle\rangle \\
& \Rightarrow 3,\ e_1,\ \langle 7, e_1\rangle & \\
& \Rightarrow 4,\ \langle y, 7, e_1\rangle : e_1,\ \langle\,\rangle & \text{let } e_2 = \langle y, 7, e_1\rangle : e_1 \\
& \Rightarrow 5,\ e_2,\ \langle 6, e_2\rangle & \\
& \Rightarrow 7,\ e_1,\ \langle 6, e_2\rangle & \\
& \Rightarrow 8,\ \langle\,\rangle,\ \langle 6, e_2\rangle & \\
& \Rightarrow 9,\ \langle x, 6, e_2\rangle,\ \langle\,\rangle & \\
& \Rightarrow 6,\ e_2,\ \langle\,\rangle & \\
& \Rightarrow 7,\ e_2,\ \langle\,\rangle & \\
& \Rightarrow 8,\ \langle\,\rangle,\ \langle\,\rangle &
\end{array}
$$

$$\mathit{Unload}(8,\ \langle\,\rangle,\ \langle\,\rangle) = \mathit{Real}(8, \langle\,\rangle) = \lambda x x$$

Figure 2. Interpretation of an expression

transitions. The initial state of the machine is $\mathit{init}(M) = M,\ \langle\,\rangle,\ \langle\,\rangle$. The machine reaches a *final state* when none of rules (1.1)-(1.3) apply. The lambda-calculus expression denoted by a state configuration, $s_0,\ e_0,\ c_1 : \cdots : c_n$, is defined by the function, Unload:

$$\mathit{Unload}(s_0,\ e_0,\ c_1 : \cdots : c_n) = \mathit{Real}(s_0, e_0)\mathit{Real}(c_1)\cdots\mathit{Real}(c_n)$$

where

$$\mathit{Real}(s,\ \langle x_1, \cdots\rangle : \cdots : \langle x_m, \cdots\rangle) = [\mathit{Real}(e[x_m])/x_m]\cdots[\mathit{Real}(e[x_1])/x_1]s$$

An example of the interpreter's operation appears in Figure 2.

We find the following notation useful. Given state configurations a and b, let $a \Rightarrow b$ denote an application of a transition rule to state a, yielding b. Similarly, $a \stackrel{m}{\Rightarrow} b$ denotes m transitions from a to obtain b. Let $\mathit{Eval}_m(M)$ describe $\mathit{Unload}(s_m,\ e_m,\ c_m)$, where $M,\ \langle\,\rangle,\ \langle\,\rangle \stackrel{m}{\Rightarrow} s_m,\ e_m,\ c_m$. An execution of $M,\ \langle\,\rangle,\ \langle\,\rangle$ to an Unloaded final state is denoted by $\mathit{Eval}(M)$.

Since the interpreter models leftmost β-reduction to weak-normal form, its operation must be consistent with the rule

$$\beta_\ell : (\lambda x B) M N_1 \cdots N_m \ >_\ell \ ([M/x]B) N_1 \cdots N_m, m \geq 0$$

In the results that follow, $M \ >_\ell^n \ N$ denotes n applications of rule β_ℓ to M, obtaining N. The consistency of the machine is guaranteed by the following lemma:

Lemma 1. For all $n \geq 0$, if $M \ >_\ell^n \ N$, then there exists $m \geq 0$ such that $Eval_m(M) = N$.

The converse also holds:

Lemma 2. For all $m \geq 0$, if $Eval_m(M) = N$, then there exists $n \geq 0$ such that $M \ >_\ell^n \ N$.

Together the two lemmas yield the result,

Theorem 2. $Eval(M) \equiv N$ iff $M \ >_\ell^* \ N$ and N is in weak-normal form.

The proofs of these properties are somewhat long and tedious, and they will not be presented; they can be found in [9].

In the sections to follow, we refer to the interpreter developed here as the *WNF (weak-normal form)-machine.*

5. A compiling scheme

The translation rules specified for the WNF-machine can be used to produce a syntax-directed translation scheme (SDTS) for the lambda-calculus. The scheme is defined by an attribute grammar [14], where a nonterminal symbol,

$$\langle Exp \Downarrow label \Uparrow code \rangle$$

accepts the inherited attribute, *label* (the next free instruction label for code generation), and generates the synthesized attribute, *code* (the target code, a set of STM-transition rules). The label expression, $label \cdot i$, generates a new label by appending i to *label*.

The compiling scheme is defined in Figure 3. The SDTS generates an expression's code by unioning the code for the subexpressions with the code corresponding to the expression itself. Unique labels are generated for subexpressions. Finally, the state name, $e[x] \downarrow 1$, denotes a run-time computed state name, as allowed by Definition 1.

$$\langle Exp \Downarrow label \Uparrow code \rangle ::= x$$
$$\quad \text{where } code = \{label,\ e,\ c \Rightarrow e[x] \downarrow 1,\ e[x] \downarrow 2,\ c\}$$

$$\langle Exp \Downarrow label \Uparrow code \cup code_0 \rangle ::= \lambda x \langle Exp \Downarrow label \cdot 0 \Uparrow code_0 \rangle$$
$$\quad \text{where } code = \{label,\ e,\ \langle a, e' \rangle : c \Rightarrow label \cdot 0,\ \langle x, a, e' \rangle : e,\ c\}$$

$$\langle Exp \Downarrow label \Uparrow code \cup code_0 \cup code_1 \rangle$$
$$\quad ::= \langle Exp \Downarrow label \cdot 0 \Uparrow code_0 \rangle \langle Exp \Downarrow label \cdot 1 \Uparrow code_1 \rangle$$
$$\quad\quad \text{where } code = \{label,\ e,\ c \Rightarrow label \cdot 0,\ e,\ \langle label \cdot 1, e \rangle : c\}$$

Figure 3. Compiling scheme for lambda-calculus

$$\begin{array}{ll}
0,\ e,\ c & \Rightarrow 1,\ e,\ \langle 8, e \rangle : c \\
1,\ e,\ \langle a, e' \rangle : c & \Rightarrow 2,\ \langle y, a, e' \rangle : e,\ c \\
2,\ e,\ c & \Rightarrow 3,\ e,\ \langle 7, e \rangle : c \\
3,\ e,\ \langle a, e' \rangle : c & \Rightarrow 4,\ \langle y, a, e' \rangle : e,\ c \\
4,\ e,\ c & \Rightarrow 5,\ e,\ \langle 6, e \rangle : c \\
5,\ e,\ c & \Rightarrow e[y] \downarrow 1,\ e[y] \downarrow 2,\ c \\
6,\ e,\ c & \Rightarrow e[y] \downarrow 1,\ e[y] \downarrow 2,\ c \\
7,\ e,\ c & \Rightarrow e[y] \downarrow 1,\ e[y] \downarrow 2,\ c \\
8,\ e,\ \langle a, e' \rangle : c & \Rightarrow 9,\ \langle x, a, e' \rangle : e,\ c \\
9,\ e,\ c & \Rightarrow e[x] \downarrow 1,\ e[x] \downarrow 2,\ c
\end{array}$$

Note: rather than generate binary-numbered labels, we reuse the label numbers from Figure 2.

Figure 4. Compiled lambda-expression

The compilation of the expression in Figure 2 is shown in Figure 4. The execution of the transition set is exactly that of Figure 2. We give the name, *Next*, to the anonymous function that controls the traversal of the STM. For an expression, M, *Next* receives the initial configuration, $m_0,\ \langle\,\rangle,\ \langle\,\rangle$, and the STM generated from M by the SDTS with initial label, m_0 — call this $SDTS(M, m_0)$:

Theorem 3. $Eval(M) \equiv Next((m_0,\ \langle\,\rangle,\ \langle\,\rangle), SDTS(M, m_0))$.

The STM is sufficiently low level for easy translation to object code, and the structure of the transition rules encourages substantial optimization upon the STM before its execution. This optimization may take the form of traversal of run-time invariant transitions (*mixed computation* [3]) or replacement of complex argument structures by simpler descriptions.

$$c\ell : S,\ E,\ \langle\,\rangle,\ \langle S', E', C', D'\rangle \Rightarrow c\ell : S',\ E',\ C',\ D' \quad (5.1)$$

$$S,\ E,\ x : C,\ D \Rightarrow E[x] : S,\ E,\ C,\ D \quad \text{if } x \in \mathit{Var} \quad (5.2)$$

$$S,\ E,\ a : C,\ D \Rightarrow \langle a, E\rangle : S,\ E,\ C,\ D \quad \text{if } a \in \mathit{Const} \quad (5.3)$$

$$S,\ E,\ (\lambda x\, B) : C,\ D \Rightarrow \langle(\lambda x\, B), E\rangle : S,\ E,\ C,\ D \quad \text{if } \lambda x\, B \in \mathit{Abs} \quad (5.4)$$

$$\langle(\lambda x\, B), E'\rangle : c\ell : S,\ E,\ \mathit{apply} : C,\ D \Rightarrow\ \langle\,\rangle,\ \langle x, c\ell\rangle : E',\ B,\ \langle S, E, C, D\rangle \quad (5.5)$$

$$S,\ E,\ (M\, N) : C,\ D \Rightarrow S,\ E,\ N : M : \mathit{apply} : C,\ D \quad \text{if } M\, N \in \mathit{Comb} \quad (5.6)$$

Figure 5. SECD machine

6. The SECD-machine

The archetypical lambda-calculus machine is Landin's SECD machine [5]. We present a brief explanation of its operation and constrast it with the WNF-machine. The definition is derived from one presented by Plotkin [6].

First, we retain $\mathit{Exp} = \mathit{Var} \cup \mathit{Abs} \cup \mathit{Comb}$, and designate a set $\mathit{Const} \subseteq \mathit{Var}$ of *constant values*. We next define the sets of environments and closures as

$$EN = (\mathit{Var} \times CL)^*$$
$$CL = \mathit{Exp} \times EN$$

The state of the SECD machine is a four-tuple, $S,\ E,\ C,\ D$, where

$S \in CL^*$, a stack of closures
$E \in EN$, the current active environment
$C \in (\mathit{Exp} \cup \{\mathit{apply}\})^*$, the control string
$D \in (S \times E \times C \times D)^*$, the stack of activation records (*dump*)

The transition function, $\Rightarrow$, takes states to states and is given in Figure 5. The start state for a closed lambda-calculus expression, M, is $(\langle\,\rangle,\ \langle\,\rangle,\ M,\ \langle\,\rangle)$, and a final state has the form, $(c\ell,\ \langle\,\rangle,\ \langle\,\rangle,\ \langle\,\rangle)$, where $c\ell \in CL$. A function analogous to *Unload* can be defined to extract the expression denoted by a final state.

We note three major differences between the WNF-machine and the SECD-machine:

1. the SECD-machine processes expressions in a rightmost innermost, call-by-value style, whereas the WNF-machine processes in a leftmost outermost, call-by-name fashion;

2. the control in the SECD-machine is embodied in the stack argument, C, whereas the WNF-machine is driven by automata-state names;

3. the SECD-machine uses a dump, D, to maintain scopes of enclosing expressions and their environments, whereas the WNF-machine has no enclosing expressions to maintain and keeps environments locally with the expressions that use them.

7. Constructing an SDTS from SECD

We use our earlier experiences to transform the SECD-machine into an STM-interpreter from which we extract an SDTS. Our idea is to encode as the state names all sequences of control-strings, C, that may arise during an input expression M's evaluation. This idea is feasible because the size of the control string, C, is bounded, depending only on the structure of M.

The derivation of the STM-SECD-interpreter based on this idea is left as an exercise, and we move directly to the compiling scheme. The scheme's attribute grammar uses nonterminals of the form,

$$\langle Exp \Downarrow control \Downarrow label \Uparrow code \rangle$$

where *control* is an inherited attribute that holds the current contents of the control string, and *label* and *code* are as before.

The SDTS is given in Figure 6. Note that an application token, ap, is specialized to its point of creation, s, giving ap_s, and an abstraction generates a pop_b token, which recovers values from the dump after the processing of abstraction body, b. The SDTS generates a pop_0 transition to obtain the final value. Constants, a, are compiled to environment-free "closures," $\langle val, a \rangle$.

The initial configuration is $0 \cdot pop_0$, $\langle\rangle$, $\langle\rangle$, $\langle\rangle$. An example of compiled code and its evaluation are seen in Figure 7.

8. Compilation from denotational definitions

We next develop a compiling scheme by applying meaning-preserving transformations to a Scott-Strachey denotational semantics definition of the lambda calculus [11]. Production of a compiling scheme from a denotational definition presents difficulties not encountered with the low-level machines we have dealt with so far. In particular, arguments

$$\langle M \Uparrow code \cup code_0 \rangle ::= \langle Exp \Downarrow pop_0 \Downarrow 0 \Uparrow code_0 \rangle$$
where $code = \{pop_0,\ c\ell : S,\ E,\ D \Rightarrow Unload(c\ell)\}$

$$\langle Exp \Downarrow control \Downarrow label \Uparrow code \rangle ::= x$$
where $code = \{label \cdot control,\ S,\ E,\ D \Rightarrow control,\ E[x] : S,\ E,\ D\}$

$$\langle Exp \Downarrow control \Downarrow label \Uparrow code \rangle ::= a$$
where $code = \{label \cdot control,\ S,\ E,\ D \Rightarrow control,\ \langle val, a\rangle : S,\ E,\ D\}$

$$\langle Exp \Downarrow control \Downarrow label \Uparrow code \cup code_0 \rangle$$
$$::= \lambda x \langle Exp \Downarrow pop_{label} \Downarrow label \cdot 0 \Uparrow code_0 \rangle$$
where $code =$
$$\{\, label \cdot control,\ S,\ E,\ D \Rightarrow control,\ \langle label \cdot 0 \cdot pop_{label}, x, E\rangle : S,\ E,\ D \,\}$$
$$\cup\{\, pop_{label},\ c\ell : S,\ E,\ \langle t, S', E', D'\rangle \Rightarrow t,\ c\ell : S',\ E',\ D' \,\}$$

$$\langle Exp \Downarrow control \Downarrow label \Uparrow code \cup code_0 \cup code_1 \rangle$$
$$::= \langle Exp \Downarrow ap_{label} \cdot control \Downarrow label \cdot 0 \Uparrow code_0 \rangle$$
$$\langle Exp \Downarrow label \cdot 0 \cdot ap_{label} \cdot control \Downarrow label \cdot 1 \Uparrow code_1 \rangle$$
where $code =$
$$\{label \cdot control,\ S,\ E,\ D \Rightarrow label \cdot 1 \cdot label \cdot 0 \cdot ap_{label} \cdot control,\ S,\ E,\ D\}$$
$$\cup\{ap_{label} \cdot control,\ c\ell_1 : c\ell_2 : S,\ E,\ D \Rightarrow c\ell_1 \downarrow 1,\ \langle\,\rangle,\ \langle c\ell_1 \downarrow 2, c\ell_2\rangle : cl_1 \downarrow 3,\ \langle control, S, E, D\rangle\}$$

Figure 6. SDTS corresponding to the SECD machine

to the semantic definitions may be functions, and no order of evaluation is indicated.

Figure 8 shows the standard semantics of the lambda-calculus as given in Stoy [11]; the reader is advised to refer there for notational conventions. The semantic definition uses an evaluation function, $\mathcal{E}$, analogous to $Eval$, and an environment argument, e, which is a function. Unlike the machines examined earlier, the definition maps the input lambda-expression into a mathematical value, a *denotation*. The domain of denotations is named D; this domain is realizable in Scott's models of denotational semantics [10]. In the definitions, we use the syntactic domain, $Exp = Var \cup Abs \cup Comb$.

The right-hand sides of (8.1)-(8.3) are expressions in Scott's LAMBDA meta-language [10]. Thus, λ and $e[a/x]$ in (8.2) are not syntactic lambda-

Use the following numeric labels for M's subexpressions:

$$((\lambda z\, z^2)^1\, k^3)^0$$

Compilation:

$$\begin{array}{l}
pop_0,\ c\ell : S,\ E,\ D \Rightarrow Unload(c\ell) \\
0 \cdot pop_0,\ S,\ E,\ D \Rightarrow 3 \cdot 1 \cdot ap_0 \cdot pop_0,\ S,\ E,\ D \\
ap_0 \cdot pop_0,\ c\ell_1 : c\ell_2 : S,\ E,\ D \\
\qquad \Rightarrow c\ell_1 \downarrow 1,\ \langle\,\rangle,\ \langle c\ell_1 \downarrow 2, c\ell_2\rangle : c\ell_1 \downarrow 3,\ \langle pop_0, S, E, D\rangle \\
1 \cdot ap_0 \cdot pop_0,\ S,\ E,\ D \Rightarrow ap_0 \cdot pop_0,\ \langle 2 : pop_1, z, E\rangle : S,\ E,\ D \\
2 \cdot pop_1,\ S,\ E,\ D \Rightarrow pop_1,\ E[z] : S,\ E,\ D \\
pop_1,\ c\ell : S,\ E,\ \langle t, S', E', D'\rangle \Rightarrow t,\ c\ell : S',\ E',\ D' \\
3 \cdot 1 \cdot ap_0 \cdot pop_0,\ S,\ E,\ D \Rightarrow 1 \cdot ap_0 \cdot pop_0,\ \langle val, k\rangle : S,\ E,\ D
\end{array}$$

Evaluation:

$$\begin{array}{ll}
0 \cdot pop_0,\ \langle\,\rangle,\ \langle\,\rangle,\ \langle\,\rangle & \Rightarrow 3 \cdot 1 \cdot ap_0 \cdot pop_0,\ \langle\,\rangle,\ \langle\,\rangle,\ \langle\,\rangle \\
& \Rightarrow 1 \cdot ap_0 \cdot pop_0,\ \langle val, k\rangle,\ \langle\,\rangle,\ \langle\,\rangle \\
& \Rightarrow ap_0 \cdot pop_0,\ \langle 2 \cdot pop_1, z, \langle\,\rangle\rangle : \langle val, k\rangle,\ \langle\,\rangle,\ \langle\,\rangle \\
& \Rightarrow 2 \cdot pop_1,\ \langle\,\rangle,\ \langle z, \langle val, k\rangle\rangle,\ \langle pop_0, \langle\,\rangle, \langle\,\rangle, \langle\,\rangle\rangle \\
& \Rightarrow pop_1,\ \langle val, k\rangle,\ \langle z, \langle val, k\rangle\rangle,\ \langle pop_0, \langle\,\rangle, \langle\,\rangle, \langle\,\rangle\rangle \\
& \Rightarrow pop_0,\ \langle val, k\rangle,\ \langle\,\rangle,\ \langle\,\rangle \\
& \Rightarrow Unload(\langle val, k\rangle) \Rightarrow k
\end{array}$$

Figure 7. A compiled expression and its evaluation

calculus expressions, but LAMBDA notation that define denotations. Whatever STM-interpreter and compiling scheme we develop from Figure 8 will compile to object code that computes LAMBDA representations of denotations (and not syntactic lambda-calculus terms).

To convert the above definition to an STM-interpreter, we might use $\mathcal{E}[\![x]\!]$, $\mathcal{E}[\![\lambda x\, B]\!]$, and $\mathcal{E}[\![M\, N]\!]$ to generate the state names. Unfortunately, the right-hand-side definitions do not conform to the STM format: (8.3) contains two state names, so the next state is uncertain, and all three equations use an environment that is a function. To handle the first problem, we reorganize the sequencing in (8.3) by converting the definition to a continuation-passing format [12]. The equivalent (*congruent* [8]) definition of Figure 8 is given in Figure 9. It is taken from Reynolds [8]. An extra argument, $c \in C$, a *continuation*, is added. The continuation acts as a sequencing device in that the right-hand-side of each equation has but one function with all its arguments available — this function will serve as the state name in an STM state configuration. (In (9.3), it is $\mathcal{N}[\![M]\!]$). The introduction of continuations has

Domains:

$\mathcal{E} \in Exp \to E \to D$ the evaluation function
$e \in E = Var \to D$ environments
$a \in D = D \to D$ denotations: every denotation is a function

Evaluation function:

$$\begin{array}{lll} \mathcal{E}[\![x]\!]e = e[\![x]\!] & \text{if } x \in Var & (8.1) \\ \mathcal{E}[\![\lambda x\, B]\!]e = \lambda a.\, \mathcal{E}[\![B]\!]\, e[a/x] & \text{if } \lambda x\, B \in Abs & (8.2) \\ \mathcal{E}[\![M\, N]\!]e = \mathcal{E}[\![M]\!]e\, (\mathcal{E}[\![N]\!]e) & \text{if } M\, N \in Comb & (8.3) \end{array}$$

Figure 8. Denotational semantics of the lambda-calculus

Domains:

$\mathcal{N} \in Exp \to E \to C \to D'$
$e \in E = Var \to D'$
$c \in C = D'' \to D'$
$f \in D'' = D' \to D'$
$a \in D' = C \to D'$

Evaluation function:

$$\begin{array}{ll} \mathcal{N}[\![x]\!]e\, c = e[\![x]\!]c & (9.1) \\ \mathcal{N}[\![\lambda x\, B]\!]e\, c = c(\lambda a.\, \mathcal{N}[\![B]\!]\, e[a/x]) & (9.2) \\ \mathcal{N}[\![M\, N]\!]e\, c = \mathcal{N}[\![M]\!]e\, (\lambda f.\, f(\mathcal{N}[\![N]\!]e)\, c) & (9.3) \end{array}$$

Figure 9. Continuation-passing semantics

increased the complexity of the semantic domains: the D domain has been fractured into two forms: D'', representing elements of D treated as functions, and D', representing elements of D treated as arguments. Further explanation is found in [8].

Now, each equation has the desired STM-state-argument format, but the environment and continuation arguments are higher order and still not acceptable. A well-known technique for converting functions to first-order structures is the introduction of closures [5]. A closure names a function. Once all the function's arguments are supplied, the name-plus-arguments form is converted to the function-plus-arguments form.

Figure 10 shows the *defunctionalization* of the environment and continuation arguments via closures; it is based upon the construction in [7].

All the domains have become nonfunctional — all explicitly constructed E, C, D', and D'' denotations are represented as closures of the form, $mk_V\langle a_1, \cdots a_m\rangle$, and each domain requires an auxiliary function,

Domains:

$$\begin{aligned}
&\mathcal{N} \in Exp \times E \times C \to D' \\
&e \in E = \{mk_{E_1}\} \times Var \times D' \times E \\
&c \in C = \{mk_{C_1}\} \times Exp \times E \times C \\
&f \in D'' = \{mk_{V_1}\} \times Var \times Exp \times E \\
&a \in D' = \{mk_{V_2}\} \times Exp \times E
\end{aligned}$$

Evaluation function:

$$\begin{aligned}
\mathcal{N}[\![x]\!](e,c) &= apE(e,[\![x]\!],c) && (10.1)\\
\mathcal{N}[\![\lambda x\, B]\!](e,c) &= apC(c, mk_{V_1}\langle [\![x]\!],[\![B]\!],e\rangle) && (10.2)\\
\mathcal{N}[\![M\, N]\!](e,c) &= \mathcal{N}[\![M]\!](e, mk_{C_1}\langle [\![N]\!],e,c\rangle) && (10.3)
\end{aligned}$$

Auxiliary functions:

$$\begin{aligned}
&apC \in C \times D'' \to D' \\
&apD'' \in D'' \times D' \to D' \\
&apE \in E \times Var \times C \to D' \\
&apD' \in D' \times C \to D'
\end{aligned}$$

$$\begin{aligned}
apC(mk_{C1}\langle [\![N]\!],e,c\rangle, f) &= apD''(f, mk_{V_2}\langle [\![N]\!],e\rangle, c) && (10.4)\\
apD''(mk_{V1}\langle [\![x]\!],[\![B]\!],e\rangle, a, c) &= \mathcal{N}[\![B]\!](mk_{E_1}\langle [\![x]\!],a,e\rangle, c) && (10.5)\\
apE(mk_{E1}\langle [\![x]\!],a,e\rangle,[\![y]\!],c) &= \begin{array}{l}([\![x]\!]=[\![y]\!]) \to apD'(a,c)\\ \quad [\!] \; apE(e,[\![y]\!],c)\end{array} && (10.6)\\
apD'(mk_{V2}\langle [\![N]\!],e\rangle, c) &= \mathcal{N}[\![N]\!](e,c) && (10.7)
\end{aligned}$$

Figure 10. Defunctionalized semantics

which converts the closure-plus-arguments form to the original function applied to the arguments.

The new equations are (10.4)-(10.7). The simple structure of Figure 9 has produced only one new closure constructor per defunctionalized domain, letting us state the auxiliary functions in four equations, one per closure constructor. See [7] for further explanation.

The equations of Figure 10 are in STM-form and use state names from the set, $\{\mathcal{N}[\![M]\!] \mid M \in Exp\} \cup \{apC, apE, apD', apD''\}$. All arguments are nonfunctional.

The seven equations contain much redundancy — for example, the closures' names are unnecessary because each domain has but one closure. We eliminate the names. Also, since its c argument must have the form, $\langle [\![N]\!], e', c\rangle$, the right-hand side of (10.2) can be stated as $\mathcal{N}[\![\lambda x\, B]\!]$ $(e, \langle [\![N]\!], e', c\rangle)$, which allows the reduction of the sequence, (10.2) $\Rightarrow$ (10.4) $\Rightarrow$ (10.5), to one equation. This gives us

$$
\begin{array}{lll}
\mathcal{N}[\![x]\!](e,c) & = apE(e,[\![x]\!],c) & (11.1)\\
\mathcal{N}[\![\lambda x\, B]\!](e,\langle[\![N]\!],e'\rangle : c) & = \mathcal{N}[\![B]\!](\langle[\![x]\!],[\![N]\!],e'\rangle : e, c) & (11.2)\\
\mathcal{N}[\![M\, N]\!](e,c) & = \mathcal{N}[\![M]\!](e,\langle[\![N]\!],e\rangle : c) & (11.3)\\
apE(\langle[\![x]\!],[\![N]\!],e'\rangle : e,[\![y]\!],c) = & \begin{array}{ll}([\![x]\!] = [\![y]\!]) & \rightarrow \mathcal{N}[\![N]\!](e',c)\\ & [\!] \; apE(e,[\![y]\!],c)\end{array} & (11.4)
\end{array}
$$

Figure 11. STM-interpreter derived from denotational-semantics definition

$$
\begin{array}{lll}
\mathcal{N}[\![x]\!](e,c) & = apE(e,[\![x]\!],c) & (10.1')\\
\mathcal{N}[\![\lambda x\, B]\!](e,\langle[\![N]\!],e',c\rangle) & = \mathcal{N}[\![B]\!](\langle[\![x]\!],\langle[\![N]\!],e'\rangle,e\rangle,c) & (10.2')\\
\mathcal{N}[\![M\, N]\!](e,c) & = \mathcal{N}[\![M]\!](e,\langle[\![N]\!],e,c\rangle) & (10.3')\\
apE(\langle[\![x]\!],a,e\rangle,[\![y]\!],c) & = \begin{array}{ll}([\![x]\!] = [\![y]\!]) & \rightarrow apD'(a,c)\\ & [\!] \; apE(e,[\![y]\!],c)\end{array} & (10.6')\\
apD'(\langle[\![N]\!],e\rangle,c) & = \mathcal{N}[\![N]\!](e,c) & (10.7')
\end{array}
$$

Since (10.6') $\Rightarrow$ (10.7'), we can collapse the two into one equation. Converting the nested tuples into stacks produces Figure 11. The results look like Figure 1, but here the environment lookup function is explicitly provided. It is easy to extract an SDTS from the Figure.

Some explanation is required as to what the equations in Figure 11 truly denote. Figure 1 interprets lambda expressions by leftmost reduction, this activity made clear by the *Unload* function. In contrast, Figure 11 interprets lambda expressions into LAMBDA denotations. In the conversion to defunctionalized form, however, the LAMBDA denotations in domains D' and D'' are not obtained unless *all* arguments to the semantic functions are present. But since the results are *themselves* functions, the LAMBDA-coded denotations never appear! The disadvantage of requiring first-order arguments to STMs is that no higher-order value can be computed as a result. Another side effect of the conversion is that the definition becomes non-compositional. (The meaning of an expression is no longer completely determined by the meanings of its subexpressions; see (11.1) $\Rightarrow$ (11.4).) Such a consequence is inevitable when converting to STM-format.

9. Conclusion

We have constructed compiling schemes from STM-interpreters for three semantic definitions: an explicitly contrived interpreter; an existing machine whose control structure was altered; and a higher-order definition upon which two significant transformations were performed. Our intention was to utilize the STM-interpreter format to expose the structure of each definition and to promote an easy conversion to an SDTS.

We saw that the arguments of the STM-interpreter display the data organization of the original definitions. This was most obvious in the SECD-machine — its rigid operational structure was preserved in STM form. In contrast, the WNF-machine contained a simple structure explicitly oriented toward β-reduction. The format produced by the essentially "structureless" denotational definition could have been overtly reorganized into many different argument structures.

Of equal interest are the transformations applied to the definitions to obtain the STM-interpreter format, for they elicit the complementary information as to which features of the definition are counterproductive to easy compilation for primitive sequential machines — conversion of the control of the SECD-machine comes immediately to mind. The imposition of operational constraints on the denotational-semantics definition suggests that great latitude is available to its implementors for both optimization and error.

It is not surprising that operational-semantics definitions lead to compiling schemes, but the STM-interpreter format defines a class of definitions that are especially useful. We might ask whether the STM restriction is too strong — in particular, can the requirement of finite-state control be replaced by a weaker notion? On the other hand, the STM format is itself quite general; a straightforward implementation of the examples in this paper would use heap-storage management, and some of the "primitive" operations may require many lines of assembly code to perform.

We make a final remark in regard to developing an automated compiling methodology. We might use the lambda-calculus as a universal meta-language for programming-language semantics. Then, we can compose a programming language's semantics definition with an SDTS for the lambda-calculus, thus producing an SDTS from the programming language to object code. An application of this technique is presented in [4].

Acknowledgements

Neil Jones contributed substantially by his critical reading of an earlier draft of this paper. Thanks also go to Karen Møller for her assistance in organizing the material.

References

1. Church, A.: 1951, *The Calculi of Lambda-Conversion. Annals of Mathematical Studies 6*. Princeton Univ. Press, Princeton, NJ.
2. Curry, H. and R. Feys: 1958, *Combinatory Logic, Vol. 1.* North-Holland, Amsterdam.
3. Ershov, A.: 1978, 'On the essence of compilation'. In: E. Neuhold (ed.): *Formal Description of Programming Concepts*. North-Holland, Amsterdam, pp. 391–420.
4. Jones, N. and D. Schmidt: 1980, 'Compiler Generation from denotational semantics'. In: N. Jones (ed.): *Semantics-Directed Compiler Generation*. Lecture Notes in Computer Science 94, Springer-Verlag, pp. 70–93.
5. Landin, P.: 1964, 'The mechanical evaluation of expressions'. *Computer Journal* **6(4)**, 308–320.
6. Plotkin, G.: 1975, 'Call-by-name, Call-by-value and the λ-calculus'. *Theoretical Comp. Sci.* **1**, 125–159.
7. Reynolds, J.: 1972, 'Definitional interpreters for higher-order programming languages'. In: *Proc. 25th ACM National Conference, Boston*. pp. 717–740.
8. Reynolds, J.: 1974, 'On the relation between direct and continuation semantics'. In: *Proc. 2d Colloquium on Automata, Languages, and Programming*. Lecture Notes in Computer Science, Springer-Verlag, pp. 141–156.
9. Schmidt, D.: 1981, 'Compiler generation from lambda-calculus definitions of programming languages'. Ph.D. thesis, Kansas State University, Manhattan, KS.
10. Scott, D.: 1976, 'Data types as lattices'. *SIAM J. of Computing* **5**, 522–587.
11. Stoy, J.: 1977, *Denotational Semantics*. MIT Press, Cambridge, MA.
12. Strachey, C. and C. Wadsworth: 1974, 'Continuations — a mathematical semantics for handling full jumps'. Technical Report PRG-11, Programming Research Group, Oxford University.
13. Wadsworth, C.: 1976, 'The relation between computational and denotational properties for Scott's models of the lambda-calculus'. *SIAM J. of Computing* **5**, 488–521.
14. Watt, D. and O. Madsen: 1977, 'Extended attribute grammars'. Technical Report 10, University of Glasgow. Revised version: On defining semantics by means of extended attribute grammars. In N.D. Jones, ed., *Semantics-Directed Compiler Generation*, Lecture Notes in Computer Science 94, Springer-Verlag, 1980.