Chapter 10

Implementation of Denotational Definitions

A language's implementation should be guided by its semantics definition. In this chapter, we survey techniques for deriving a compiler from a language's semantics and examine some of the existing automated tools for compiler generation. We also consider issues regarding the correctness of implementations.

10.1 A GENERAL METHOD OF IMPLEMENTATION

In the previous chapters, we saw many examples of programs that were mapped to their dentotations and simplified to answers. The simplifications resemble the computational steps that occur in a conventional implementation. They suggest a simple, general implementation technique: treat the semantic notation as a ''machine language'' and implement an evaluator for the semantic notation. The denotational equations translate a program to its denotation, and the evaluator applies simplification rules to the denotation until all possible simplifications are performed.

As an example, consider the semantic definition in Figure 5.2. The translation of the program ⟦Z:=A+1⟧ is the expression:

P⟦Z:=A+1⟧ =
λ*n*. let *s* = (*update*⟦A⟧ *n newstore*) in
  let *s*′ = (λ*s*. *update*⟦Z⟧ (λ*s*. (λ*s*. *access*⟦A⟧ *s*)*s plus* (λ*s*. *one*)*s*)*s* *s*)*s*
  in (*access*⟦Z⟧ *s*′)

(We have not bothered to expand the *Store* algebra operators to their underlying function forms, e.g., *access* to (λ*i*.λ*s*. *s*(*i*)). This keeps the overall expression readable. Also, *Store*-level operations are often treated specially.) The expression is applied to its run-time data, say the number *four*, and is given to the evaluator, which applies the simplification rules. The number *five* is the simplified result and is the output of the evaluator.

Let's call this approach the *compile-evaluate* method. There is a simple variation on the method. The example in Section 5.1 suggests that we can simultaneously translate a program into its denotation and evaluate it with its run-time arguments. For example, the expression **P**⟦Z:=A+1⟧ *four* is translated to the intermediate form:

(λ*n*. let *s* = (*update*⟦A⟧ *n newstore*) in
  let *s*′ = **C**⟦Z:=A+1⟧*s* in (*access*⟦Z⟧ *s*′))*four*

which is simplified to the expression:

let s_1 = **C**⟦Z:=A+1⟧([⟦A⟧ ↦ *four*]*newstore*) in (*access*⟦Z⟧ s_1)

which is translated to the expression:

let s_1 = ($\underline{\lambda}s$. *update*⟦Z⟧ **E**⟦A+1⟧s s)([⟦A⟧↦ *four*]*newstore*) in (*access*⟦Z⟧ s_1)

which is simplified to:

let s_1 = *update*⟦Z⟧ (**E**⟦A+1⟧([⟦A⟧↦ *four*]*newstore*)) ([⟦A⟧↦ *four*]*newstore*)
in (*access*⟦Z⟧ s_1)

and so on. The result is again *five.* This is an interpreter approach; the denotational definition and evaluator interact to map a source program directly to its output value. The compile-evaluate method is more commonly used by the existing systems that implement semantic definitions. It is closer in spirit to a conventional compiler-based system and seems to be more amenable to optimizations.

10.1.1 The SIS and SPS Systems

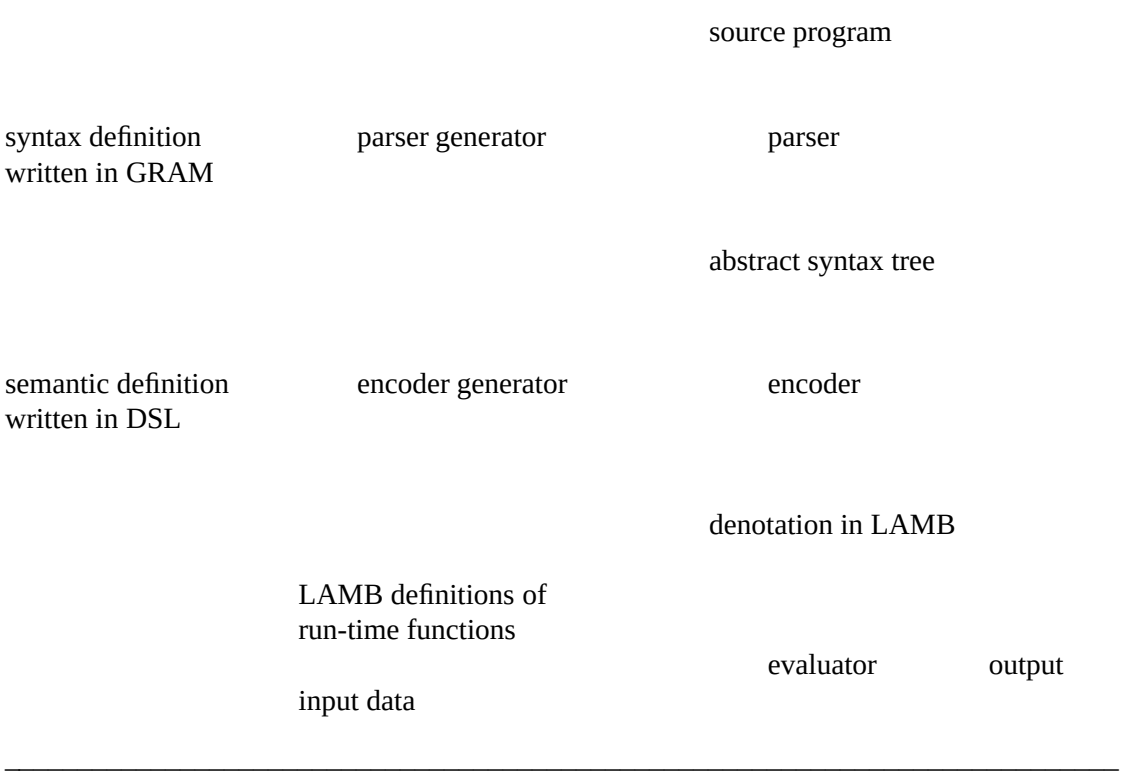
Two compiler generator systems based on the compile-evaluate method are Mosses's Semantics Implementation System (SIS) and Wand's Semantics Prototyping System (SPS). SIS was the first compiler generating system based on denotational semantics. Figure 10.1 shows the components and data flow of the system.

SIS consists of a parser generator and an encoder generator. The parser generator produces an SLR(1) parser from an input BNF definition coded in a notation called GRAM. The semantic definition, coded in DSL, is read by the encoder generator, which produces an *encoder*, that is, a translator from abstract syntax trees to ''LAMB-denotations.'' A source program is parsed by the parser and is translated by the encoder. The source program's denotation plus its input values are passed to the evaluator for simplification. The definitions of run-time operations (such as the operations from the *Store* algebra) are supplied at this time. The evaluator uses a *call-by-need* simplification strategy: an expression $(\lambda x.M)N$ simplifies to M, and the binding (x, N) is retained by the evaluator in an environment table. When an occurrence of x is encountered in M, N is fetched from the table, simplified to its value v, and used for x. The binding (x,v) replaces (x, N) in the table. This strategy handles combinations more efficiently than the usual textual substitution method.

SIS is coded in BCPL. It has been used to implement a number of test languages. Its strengths include its simplicity and generality— virtually any denotational definition can be implemented using SIS. The system's primary weakness is its inefficiency: the generated compilers are large and the compiled programs run slowly. Nonetheless, SIS is an important example of an automated system that produces a *correct* compiler from a language's formal specification. It has inspired many researchers to develop more efficient and specialized systems.

Wand's SPS system is based on existing software tools. The system's parser generator is

Figure 10.1



the YACC parser generator. A language definition is stated as a YACC-coded grammar with the denotational semantics equations appended to the grammar rules. The semantics equations are coded in Scheme, a LISP-like programming language that resembles function notation. The SPS evaluator is just the Scheme interpreter, which evaluates denotations relatively efficiently. SPS also uses a type checker that validates the domain definitions and semantic equations for well-definedness. (SIS does not possess this feature, so a user must carefully hand check the definition.) Like SIS, SPS has been used on a number of test languages. It demonstrates how a useful generator system can be neatly built from software tools.

10.2 STATIC SEMANTICS PROCESSING

The compiler described in the previous section generates denotations that contain a large number of trivial bindings. Here is an example:

C⟦A:=0;B:=A+1⟧ =

$$\underline{\lambda}s.\ (\underline{\lambda}s.\ update[\![\mathrm{B}]\!]\ (\lambda s.\ (\lambda s.\ access[\![\mathrm{A}]\!]\ s)s\ plus\ (\lambda s.\ one)s)s\ s)$$
$$((\underline{\lambda}s.\ update[\![\mathrm{A}]\!]\ (\lambda s.\ zero)s\ s)s)$$

Trivial bindings of the form $(\lambda s.\ E)s$ should be simplified to E prior to run-time. We call these compile-time simplifications *partial evaluation* or even *static semantics processing.* Static semantics processing performs those evaluation steps that are not dependent on run-time values. In traditional compilers, static semantics processing includes declaration processing, type checking, and constant folding.

How do we determine which simplifications to perform? We call an expression *unfrozen* if it can be simplified before run-time. A *frozen* expression may not be simplified. Once we decide which semantic algebras define run-time values, we freeze the operations in those algebras. An example of an algebra that is typically frozen is the *Store* algebra. The algebras of truth values and numbers are frozen if the evaluation of Boolean and arithmetic expressions is left until run-time. (In compilers that do constant folding, *Tr* and *Nat* are unfrozen.)

During static semantics processing, we simplify each subexpression of a denotation as far as possible until we encounter a frozen operation; then we are forced to stop. Say that the *Store* and *Nat* algebras are frozen in the above example. Then the subexpression $(\underline{\lambda}s.\ update\ [\![\mathrm{A}]\!]\ (\lambda s.\ zero)s\ s)s$ simplifies to $(update\ [\![\mathrm{A}]\!]\ zero\ s)$, but no further, because *update* is a frozen operation. The simplified subexpression itself is now ''frozen.'' Frozen subexpressions impact other simplifications; a combination $(\lambda x.\ M)N$, where N is a frozen subexpression, is not simplified. (But M itself may be simplified.) Also, some static semantics simplifiers refuse to simplify $(\lambda x.\ M)N$ if unfrozen N is a nonconstant or nonidentifier and occurs free in M more than once, for the resulting expression would be larger, not smaller, than the original.

For the above example with the *Store* and *Nat* algebras frozen, static semantics produces:

$$\underline{\lambda}s.\ (\underline{\lambda}s.update[\![\mathrm{B}]\!]\ ((access[\![\mathrm{A}]\!]\ s)\ plus\ one)\ s)\ (update[\![\mathrm{A}]\!]\ zero\ s)$$

which is the expected ''machine code'' for the command.

Static semantics processing is most useful for simplifying denotations that contain environment arguments. Recall the block-structured language in Section 7.1. Environments process declarations, reserve storage, and map identifiers to denotable values. Environment-related actions are traditionally performed at compile-time. The example in that section showed that the denotation of a program can be simplified to a point where all references to environment arguments disappear. The simplifications are exactly those that would be performed during static semantics processing, because the *Environment* algebra is unfrozen.

The SIS system does static semantics processing. However, SIS does not freeze any expressions; it simplifies every possible subexpression. The method works because the definitions of the frozen operators (such as the *Store*-based ones) are not supplied until run-time. Thus, any expression using a run-time operation is not simplifiable.

10.3 THE STRUCTURE OF THE EVALUATOR

We use the equalities in Section 3.5 to simplify expressions. From here on, we treat the equalities as *rewriting rules.* An equality $M=N$ induces a rewriting rule $M \Rightarrow N$; an occurrence of M in an expression is *reduced* (simplified) to N by the rule. For example, $(\lambda x.\, M)N \Rightarrow [N/x]M$ is a rewriting rule, and is in fact a rather famous one, called the β-*rule.* An expression whose structure matches the left hand side of a rewriting rule is a *redex,* and the expression that results from reducing a redex is called its *contractum.* We write $E_1 \Rightarrow E_2$ if expression E_1 rewrites to E_2 in one step and write $E_1 \Rightarrow^* E_2$ if zero or more steps are used. An expression that contains no redexes is in *normal form.* We say that an expression *has a normal form* if it can be reduced to an expression in normal form. Not all expressions have normal forms (e.g., $(\lambda x.\, x\,x)(\lambda x.\, x\,x)$, where $x \in G = G \rightarrow G$). An important feature of the rules for function notation is that if an expression does have a normal form then it is unique. This property follows from the *confluence* (Church-Rosser) property: if $E_1 \Rightarrow^* E_2$ and $E_1 \Rightarrow^* E_3$, then some E_4 exists such that $E_2 \Rightarrow^* E_4$ and $E_3 \Rightarrow^* E_4$.

An evaluator applies rewriting rules to its argument until a normal form (if it exists) is reached. The evaluator should apply the rules in a fashion that is sufficient for achieving a normal form. (For example, $(\lambda y.\, zero)\ ((\lambda x.\, x\,x)\ (\lambda x.\, x\,x))$ has the normal form *zero*, but perpetually reducing the argument $(\lambda x.\, x\,x)(\lambda x.\, x\,x)$ will never produce it.) A strategy sufficient for reducing an expression to normal form is the *leftmost-outermost method*: at each reduction step, we reduce the leftmost redex that is not contained within another redex. (We make the statement that the leftmost-outermost method is sufficient with the understanding that a combination $(\underline{\lambda} x.\, M)N$, where N itself is a function, argument combination, should be ''read backwards'' as ''$N(M.x\underline{\lambda})$.'' Recall that a strict abstraction requires a proper argument, hence its argument must be reduced until its proper structure— a pair, injection, abstraction, number, or whatever— appears. Then the β-reduction is made.) Here is a leftmost-outermost reduction:

$(\lambda x.\, (x\,x)zero)((\lambda y.\, (\underline{\lambda} z.\, z))((\lambda x.\, x\,x)(\lambda x.\, x\,x)))$
$\Rightarrow (((\lambda y.\, (\underline{\lambda} z.\, z))((\lambda x.\, x\,x)(\lambda x.\, x\,x)))\ ((\lambda y.\, (\underline{\lambda} z.\, z))((\lambda x.\, x\,x)(\lambda x.\, x\,x))))zero$
$\Rightarrow ((\underline{\lambda} z.\, z)\ ((\lambda y.\, (\underline{\lambda} z.\, z))((\lambda x.\, x\,x)(\lambda x.\, x\,x))))zero$
$\Rightarrow ((\underline{\lambda} z.\, z)\ (\underline{\lambda} z.\, z))zero$
$\Rightarrow (\underline{\lambda} z.\, z)zero$
$\Rightarrow zero$

One way of implementing the leftmost-outermost reduction strategy is to represent the expression to be reduced as a tree. The evaluator does a left-to-right, depth-first traversal of the tree. When a node in the tree is visited, the evaluator determines if the subtree whose root is the visited node is a redex. If it is not, the evaluator visits the next node in its traversal. But if it is, the evaluator removes the tree, does the reduction, and inserts the contractum for the redex. The next node visited is the parent node of the contractum's, for the evaluator must backtrack up the tree to see if the insertion of the contractum created a new outermost redex.

An inefficiency of the tree reduction method lies in its reduction of a redex $(\lambda x.\, M)N$: occurrences of N must be inserted in place of occurrences of x in M in the contractum. A traversal of *M's* tree is required for the insertions. Then, M is traversed a second time to reduce its redexes. These two traversals can be combined into one: the evaluator can insert an N for an x when it encounters x during its traversal of M for reductions. In the meantime, the binding

of x to N can be kept in an environment. An *environment* is a collection of identifier, expression pairs, chained together in the shape of an inverted tree. The environment holds the arguments bound to identifiers as a result of β-reductions. Every node in the expression tree has a pointer into the environment. The pointer points to a linked list (that is, a path in the inverted tree) of the bindings that belong to the node. The inverted tree structure of the environment results because distinct nodes sometimes share bindings in their lists.

When a redex $R = (\lambda x.\, M)N$ reduces to M, the binding (x, N) is chained to the front of the list pointed to by M's environment pointer. M's new pointer points to the binding (x, N) followed by the previous list of bindings. When a free occurrence of x is visited in M, the evaluator follows the environment pointer attached to x to find the first pair (x, N) in the chain of bindings. N (and *its* pointer) replace x in the expression. A clever evaluator evaluates x by leaving N in its place in the environment and simplifying it there. Once N reduces to a proper value, that value is copied over into the expression tree in x's place. Subsequent lookups of x in the environment find the reduced value. This approach is known as *call-by-need* evaluation.

10.3.1 A Stack-Based Evaluator

The tree traversal method is slow and bulky. There is too much copying of contractums in place of redexes into the expression tree, and there is too much backtracking during tree traversal. Further, the representation of the expression as a tree occupies a wasteful amount of space. We can represent the leftmost-outermost reduction of an expression in a more conventional form. We use a stack-based machine as an evaluator; an expression is translated into a sequence of machine instructions that describes a leftmost-outermost reduction of the expression. The traversal and reduction steps can be translated into machine code because function expressions are statically scoped, so environment maintenance is routine, and because the traversal path through the expression can be calculated from the structure of the expression. Figure 10.2 shows the stack machine. We call it the *VEC-machine* because it possesses three components:

1. A temporary value stack, v, which holds subexpressions that have been reduced to proper values and expressions whose evaluation has been postponed.
2. An environment, e, which stacks environment pointers and establishes the scope of the current expression being reduced.
3. A code stack, c, which holds the machine instructions for the reduction. Rather than using an instruction counter, we treat the instructions as a stack. The top instruction on the stack is the one executed, and a stack pop corresponds to an increment of the instruction counter to the next instruction in the code.

We represent a machine configuration as a triple $(v\ e\ c)$. Each of the three components in the configuration is represented in the form $a_1{:}a_2{:}\cdots{:}a_n$, where a_1 is the top value on the component's stack.

Two of the machine's key data structures are the environment pointer and the closure. An *environment pointer* is a pointer value to a linked list of identifier, value bindings. (Read the @ symbol as saying ''a pointer to.'') All the bindings are kept in the environment tree, which

Figure 10.2

VEC-machine components:

$v \in \mathit{Temporary\text{-}value\text{-}stack} = \mathit{Value}^*$

$e \in \mathit{Environment} = \mathit{Environment\text{-}pointer}^*$

$c \in \mathit{Code\text{-}stack} = \mathit{Instruction}^*$

where

$\mathit{Value} = \mathit{Primitive\text{-}value} + \mathit{Closure}$

$a \in \mathit{Primitive\text{-}value} = \mathit{Nat} + \mathit{Tr} + \cdots$

$(\alpha, p) \in \mathit{Closure} = \mathit{Instruction}^* \times \mathit{Environment\text{-}pointer}$

$p \in \mathit{Environment\text{-}pointer} = @((\mathit{Identifier} \times \mathit{Value} \times \mathit{Environment\text{-}pointer}) + \mathit{nil})$

$\mathit{Instruction} = \mathit{pushclosure}(\mathit{Instruction}^*) + \mathit{pushconst}(\mathit{Primitive\text{-}value}) +$
$\mathit{call} + \mathit{return} + \mathit{push}(\mathit{Identifier}) + \mathit{bind}(\mathit{Identifier}) + \mathit{Primitive\text{-}operator} +$
$\mathit{test}(\mathit{Instruction}^* \times \mathit{Instruction}^*)$

Instruction interpretation (note: the operator ‘‘:’’ stands for stack *cons*):

(1) $v \quad p{:}e \quad \mathit{pushclosure}\ \alpha{:}c \quad \Rightarrow \quad (\alpha, p){:}v \quad p{:}e \quad c$

(2) $v \quad e \quad \mathit{pushconst}\ k{:}c \Rightarrow \quad k{:}v \quad e \quad c$

(3) $(\alpha, p){:}v \quad e \quad \mathit{call}{:}c \quad \Rightarrow \quad v \quad p{:}e \quad \alpha{:}c$

(4) $v \quad p{:}e \quad \mathit{return}{:}c \quad \Rightarrow \quad v \quad e \quad c$

(5) $v \quad e \quad \mathit{push}\ x{:}c \quad \Rightarrow \quad a{:}v \quad e \quad c$

where $a \in \mathit{Primitive\text{-}value}, \quad a = \mathit{lookup}\ x\ (\mathit{hd}\ e)$

and $\mathit{lookup}\ x\ p =$ let $(i, r, p') = p@$ in $\mathit{if}\ i{=}x\ \mathit{then}\ r\ \mathit{else}\ \mathit{lookup}\ x\ p'$

(6) $v \quad e \quad \mathit{push}\ x{:}c \quad \Rightarrow \quad v \quad p{:}e \quad \alpha{:}c$

where $(\alpha, p) \in \mathit{Closure}, \quad (\alpha, p) = \mathit{lookup}\ x\ (\mathit{hd}\ e)$

(7) $r{:}v \quad p{:}e \quad \mathit{bind}\ x{:}c \quad \Rightarrow \quad v \quad p'{:}e \quad c$

where $p' = @(x, r, p)$

(8) $a_n{:}\cdots{:}a_1{:}v \quad e \quad f{:}c \quad \Rightarrow \quad a{:}v \quad e \quad c$

where $(f\ a_1 \cdots a_n) = a$

(9) $\mathit{true}{:}v \quad e \quad \mathit{test}(\alpha,\beta){:}c \quad \Rightarrow \quad v \quad e \quad \alpha{:}c$

(10) $\mathit{false}{:}v \quad e \quad \mathit{test}(\alpha,\beta){:}c \quad \Rightarrow \quad v \quad e \quad \beta{:}c$

---

has the structure described in the previous section and is not explicitly depicted in the figure.

A *closure* represents an expression that has not been reduced but must be saved for later use. Both the instructions for the expression and its environment pointer must be kept in the closure. A *call* instruction activates the closure's code; that is, it initiates the expression's evaluation.

The operation of the machine is expressed with rewriting rules. A rule of the form $v \quad e \quad ins{:}c \quad \Rightarrow \quad v' \quad e' \quad c'$ shows the effect of the instruction *ins* on the machine's three components. Here is a brief explanation of the instructions. The *pushclosure* instruction creates a closure out of its code argument. The current environment pointer establishes the scope of the code, so it is included in the closure (see rule 1). A real implementation would not store the code in the closure but would store a pointer to where the code resides in the program store. A *pushconst* instruction pushes its primitive value argument onto the value stack (see rule 2). The *call* instruction activates the closure that resides at the top of the value stack. The closure's code is loaded onto the code stack, and its environment pointer is pushed onto the environment (see rule 3). A hardware implementation would jump to the first instruction in the closure's code rather than copy the code into a stack. The *return* instruction cleans up after a call by popping the top pointer off the environment stack (see rule 4). A hardware implementation would reset the instruction counter as well. The *push* instruction does an environment lookup to find the value bound to its argument. The lookup is done through the linked list of bindings that the active environment pointer marks. In the case that the argument x is bound to a primitive value (rule 5), the value is placed onto the value stack. If x is bound to a closure (rule 6), the closure is invoked so that the argument can be reduced. The *bind* instruction augments the active environment by binding its argument to the top value on the value stack (see rule 7). A primitive operator f takes its arguments from the value stack and places its result there (see rule 8). The *test* instruction is a conditional branch and operates in the expected way (see rules 9 and 10). A hardware implementation would use branches to jump around the clause not selected.

Figure 10.3 defines the code generation map $\mathbf{T} : \mathit{Function\text{-}Expr} \rightarrow \mathit{Instruction}^*$ for mapping a function expression into a sequence of instructions for doing a leftmost-outermost reduction. The leftmost-outermost strategy is easy to discern; consider $\mathbf{T}[\![(E_1\ E_2)]\!]$: the

Figure 10.3

$$\mathbf{T}[\![(E_1\ E_2)]\!] = \mathit{pushclosure}(\mathbf{T}[\![E_2]\!] : \mathit{return}) : \mathbf{T}[\![E_1]\!] : \mathit{call}$$
$$\mathbf{T}[\![\lambda x.\ E]\!] = \mathit{pushclosure}(\mathit{bind}\ [\![x]\!] : \mathbf{T}[\![E]\!] : \mathit{return})$$
$$\mathbf{T}[\![\underline{\lambda} x.\ E]\!] = \mathit{pushclosure}(\mathit{call} : \mathit{bind}\ [\![x]\!] : \mathbf{T}[\![E]\!] : \mathit{return})$$
$$\mathbf{T}[\![x]\!] = \mathit{push}\ [\![x]\!]$$
$$\mathbf{T}[\![k]\!] = \mathit{pushconst}\ k$$
$$\mathbf{T}[\![(f\ E_1 \cdots E_n)]\!] = \mathbf{T}[\![E_1]\!] : \cdots : \mathbf{T}[\![E_n]\!] : f$$
$$\mathbf{T}[\![E_1 \rightarrow E_2\ []\ E_3]\!] = \mathbf{T}[\![E_1]\!] : \mathit{test}(\mathbf{T}[\![E_2]\!], \mathbf{T}[\![E_3]\!])$$

generated code says to postpone the traversal of E_2 by creating a closure and placing it on the value stack. The code for E_1, the left component of the combination, is evaluated first. E_1's code will (ultimately) create a closure that represents an abstraction. This closure will also be pushed onto the value stack. The *call* instruction invokes the closure representing the abstraction. Studying the translation of abstractions, we see that the code in an abstraction's closure binds the top value on the value stack to the abstraction's identifier. In the case of a nonstrict abstraction, a closure is bound. In the case of a strict abstraction, the closure on the value stack is first invoked so that a proper value is calculated and placed onto the value stack, and then the argument is bound. The translations of the other constructs are straightforward.

Figure 10.3 omitted the translations of product and sum elements; these are left as an exercise. A translation of an expression is given in Figure 10.4. The code in the figure can be improved fairly easily: let *popbinding* be a machine instruction with the action:

$$v \quad p{:}e \quad popbinding{:}c \quad \Rightarrow \quad v \quad p_r{:}e \quad c, \quad \text{where } p = @(x, r, p_r)$$

Then a combination's code can be improved to:

$$\mathbf{T}[\![(\lambda x.\, E_1)E_2]\!] = pushclosure(\mathbf{T}[\![E_2]\!]{:}\, return){:}\, bind\, x{:}\, \mathbf{T}[\![E_1]\!]{:}\, popbinding$$
$$\mathbf{T}[\![(\underline{\lambda} x.E_1)E_2]\!] = \mathbf{T}[\![E_2]\!]{:}\, bind\, x{:}\, \mathbf{T}[\![E_1]\!]{:}\, popbinding$$

eliminating many of the *pushclosure, call,* and *return* instructions.

We should prove that the translated code for a function expression does indeed express a leftmost-outermost reduction. We will say that the machine is

faithful to the reduction rules if the computation taken by the machine on a program corresponds to a reduction on the original function expression. Indeed, the VEC-machine is faithful to the rules of function notation. The proof is long, but here is an outline of it. First, we define a mapping $Unload : VEC\text{-}machine \rightarrow Function\text{-}Expr$ that maps a machine configuation back to a function expression. Then we prove: for all $E \in Function\text{-}Expr$, $(nil\ p_0\ \mathbf{T}[\![E]\!]) \Rightarrow^* (v\ e\ c)$ implies $E \Rightarrow^* Unload(v\ e\ c)$, where $p_0 = @nil$. The proof is an

Figure 10.4

$\mathbf{T}[\![(\lambda y.\, zero)((\lambda x.\, x\, x)(\lambda x.\, x\, x))]\!] =$

let Δ be $\mathbf{T}[\![(\lambda x.\, x\, x)]\!] =$
$\quad pushclosure(bind\, x{:}$
$\quad\quad pushclosure(push\, x{:}\, return){:}$
$\quad\quad\quad push\, x{:}\, call{:}\, return)$
in
$pushclosure($
$\quad pushclosure(\Delta{:}\, return){:}\, \Delta{:}\, call){:}$
$\quad\quad pushclosure(bind\, y{:}\, pushconst\, zero{:}\, return){:}\, call$

induction on the number of machine moves. The basis, zero moves, is the proof that $Unload(nil\ p_0\ \mathbf{T}[\![E]\!]) = E$; the inductive step follows from the proof that $(v\ e\ c) \Rightarrow (v', e', c')$ implies $Unload(v\ e\ c) \Rightarrow^* Unload(v', e', c')$. *Unload*'s definition and the proof are left as exercises.

Another aspect of the correctness of the VEC-machine is its termination properties: does the machine produce a completely simplified answer exactly when the reduction rules do? Actually, the VEC-machine is conservative. It ceases evaluation on an abstraction when the abstraction has no argument; the abstraction's body is not simplified. Nonetheless, the machine *does* reduce to final answers those terms that reduce to nonabstraction (hereafter called *first-order*) values. The VEC-machine resembles a real-life machine in this regard.

The VEC-machine evaluates function expressions more efficiently than the reduction rules because it uses its stacks to hold intermediate values. Rather than searching through the expression for a redex, the code deterministically traverses through the expression until (the code for) a redex appears on the top of the *c* stack. Simplified values are moved to the *v* stack— substitutions into the expression are never made.

Here is the current version of the compile-evaluate method that we have developed. The compile step is:

1. Map a program P to its denotation $\mathbf{P}[\![\mathrm{P}]\!]$.
2. Perform static semantics analysis on $\mathbf{P}[\![\mathrm{P}]\!]$, producing a denotation *d.*
3. Map *d* to its machine code $\mathbf{T}[\![d]\!]$.

The evaluate step is: load $\mathbf{T}[\![d]\!]$ into the VEC-machine, creating a configuration $(nil\ @nil\ \mathbf{T}[\![d]\!])$, and run the machine to a final configuration $(r{:}v\ e\ nil)$. The answer is *r*.

10.3.2 PSP and Appel's System

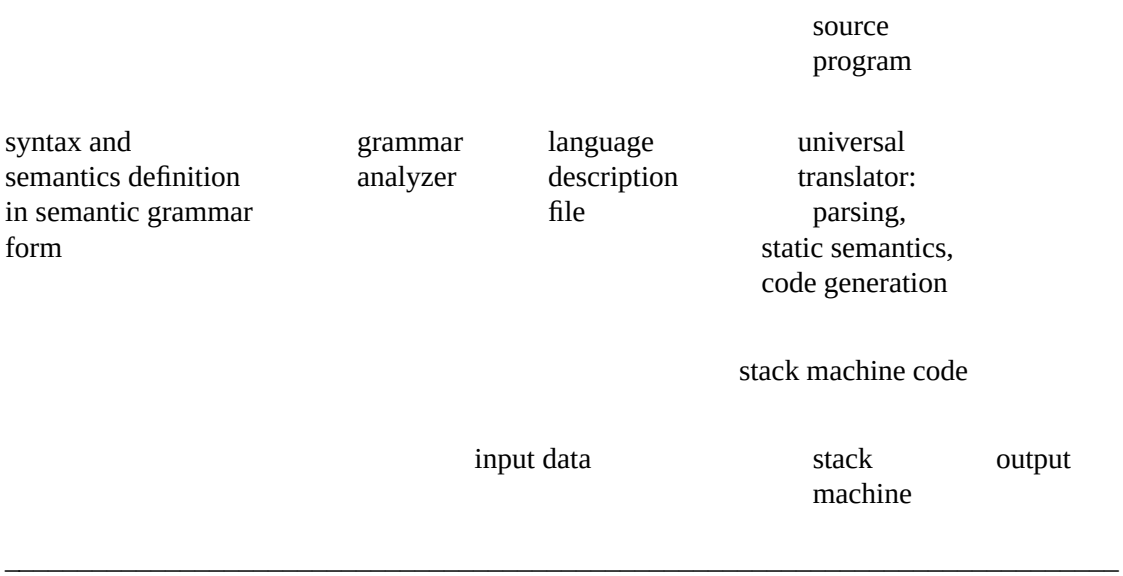
Paulson's Semantic Processor (PSP) system generates compilers that map programs into stack machine code. The PSP evaluator resembles the stack architecture just developed. In PSP, a language is defined with semantic grammars, a hybrid of denotational semantics and attribute grammars. The semantic grammar for a language is input to the *grammar analyzer*, which produces a language description file containing an LALR(1) parse table and the semantic equations for the language. Figure 10.5 shows the components.

The universal translator uses the language description file to compile a source program. A source program is parsed, mapped to its function expression form, partially evaluated, and mapped to stack machine code.

The static semantics stage in PSP does more than just partial evaluation. It also enforces contextual constraints (such as data type compatibility) that are specified in the semantic grammar. Efficient representations of abstractions and data values (like stores) are created. PSP has been used to generate a compiler for a large subset of Pascal; the generated compiler runs roughly 25 times slower than a handwritten one but is *smaller* than the handwritten compiler.

Appel's compiler-generating system produces compilers that translate source programs to register machine code. Static semantics and code generation are simultaneously performed by

Figure 10.5



a *reducer* module, which completely reduces a denotation down to an empty expression. During the process, certain simplifications cause machine code to be emitted as a side effect. For example, the reducer reduces the expression (*update i n s*) to *s*, emitting the code ''*s*[*i*]:=*n*'' as a side effect. Appel's system is intended as a tool for generating quality code for conventional machines.

10.4 COMBINATOR-BASED SEMANTIC NOTATIONS

It is difficult to develop an efficient evaluator for function notation because the notation is so general. In particular, the binding of values to identifiers requires costly time- and space-consuming environment maintenance and lookups. A number of researchers have designed notations for specialized classes of languages. These notations make use of *combinators* that have efficient evaluations.

A *combinator* is a function expression that has no free identifiers. A combinator is normally given a name, and the name is used in place of the expression. As an example, let's use the name ; for the expression $(\lambda f_1.\lambda f_2.\underline{\lambda} s.\, f_2(f_1\, s))$. Hence, $E_1; E_2$ is $(\underline{\lambda} s.\, E_2(E_1\, s))$. The advantage of using the combinator is that a complicated binding and composition structure is hidden within the combinator. The expression $E_1; E_2$ is easier to read than the function expression form. The (derived) rewriting rule $(E_1; E_2)\, s \Rightarrow E_2(E_1 s)$ expresses the binding of argument to abstraction in a fashion that eliminates the binding identifier. If combinators are used

exclusively as the semantic notation, then the binding identifiers (and their maintenance) disappear altogether.

Let's design a combinator set for writing definitions of simple imperative languages. The combinators will manipulate stores and temporary value stacks. Underlying the notation are two semantic algebras: the *Store* algebra and the algebra of lists of expressible values, $EVlist = (Nat + Tr)^*$. An expressible value list, store pair is called a *state*. All expressions written in the combinator notation are mappings from states to states. The combinators are ;, !, *cond*, and *skip*. Here are their rewriting rules:

$(E_1; E_2)(v, s) \Rightarrow E_2(E_1(v, s))$

$f!(v_n : \cdots : v_1 : v, s) \Rightarrow (v' : v, s)$
where $f: Exprval_1 \times \cdots \times Exprval_n \times Store \to Exprval$ is $(f\, v_1 \cdots v_n\, s) = v'$

$f!(v_n : \cdots : v_1 : v, s) \Rightarrow (v, s')$
where $f: Exprval_1 \times \cdots \times Exprval_n \times Store \to Store$ is $(f\, v_1 \cdots v_n\, s) = s'$

$cond(E_1, E_2)(true : v, s) \Rightarrow E_1(v, s)$

$cond(E_1, E_2)(false : v, s) \Rightarrow E_2(v, s)$

$skip(v, s) \Rightarrow (v, s)$

The expression $E_1; E_2$ composes the state-altering actions of E_1 with those of E_2. The expression $f!$ is a primitive state-altering action. If f requires n arguments (plus the store), the top n arguments are taken off the expressible value list and are given to f (along with the store). The answer is pushed onto the list. (If f produces a store for an answer, it replaces the existing one.) The expression $cond(E_1, E_2)$ selects one of its two argument values based on the value at the front of the expressible value list; *skip* is the null expression.

The combinators must be assigned denotations. Then they can be used in semantic definitions and their rewriting rules can be proved sound. The denotation of $E_1; E_2$ is $\lambda(v, s).\ \text{let } (v', s') = E_1(v, s) \text{ in } E_2(v', s')$. The soundness of its rewriting rule easily follows. The denotations and soundness proofs for the other combinators are left as exercises. An important feature of the rewriting rules is that binding identifiers are never used— the state pair (v, s) is passed from combinator to combinator. This suggests that a machine for the combinator language be configured as $(c\ v\ s)$, where c is a combinator expression $c_1; c_2; \cdots; c_n$. The machine language consists of the $f!$, *cond*, and *skip* operators, and the actions of the machine are defined by the rewriting rules. For example, $(cond(E_1, E_2); c\ true : v\ s) \Rightarrow (E_1; c\ v\ s)$. By restricting the semantic notation to simple combinators, we obtain a simple evaluator. An imperative language is defined with the combinators in Figure 10.6.

The example combinator notation is ideally suited to expressing the sequencing and updating concepts in a language, but it is inadequate for expressing many other semantic concepts. There are no combinators for identifier declarations, recursively defined values, and nonstandard control forms. A number of researchers, most notably Mosses, are developing truly general combinator notations.

Figure 10.6

C : Command → *State* → $State_{\perp}$
C⟦C_1;C_2⟧ = **C**⟦C_1⟧; **C**⟦C_2⟧
C⟦I:=E⟧ = **E**⟦E⟧; (*update*⟦I⟧ !)
C⟦**if** B **then** C_1 **else** C_2⟧ = **B**⟦B⟧; *cond*(**C**⟦C_1⟧, **C**⟦C_2⟧)
C⟦**while** B **do** C⟧ = *wh*
where *wh* = **B**⟦B⟧; *cond*(**C**⟦C⟧; *wh*, *skip*)

E : Expression → *State* → $State_{\perp}$
E⟦E_1+E_2⟧ = **E**⟦E_1⟧ ; **E**⟦E_2⟧ ; *add* !
E⟦I⟧ = *access*⟦I⟧ !
E⟦N⟧ = **N**⟦N⟧ !

10.4.1 The Plumb and CERES Systems

Sethi's Plumb and Christiansen and Jones's CERES systems both rely on combinator-based semantic notations. The Plumb system uses a combinator set similar to the one defined in the previous section. Instead of manipulating a store and an expressible value stack, the combinators handle *streams* of values. A typical stream consists of a store as the first value and a sequence of expressible values thereafter, that is, $s, x_1, x_2, \cdots$. The primary combinator | is called a *pipe* and is used in roughly the same way as the ; combinator defined in the previous section. An expression (E_1 | E_2) maps a stream of values to a stream of values as follows: if E_1 requires m_1 values to produce its n_1 answers, the first m_1 values are removed from the stream and given to E_1; its answers are placed on the front of the stream that passes to E_2. E_2 takes the m_2 values it needs and places its n_2 answers onto the front of the stream. A variation on the pipe is $E_1 \mid_k E_2$, which skips over the first k values in the stream when supplying the stream to E_2.

The semantics of command composition, assignment, and addition read:

C⟦C_1;C_2⟧ = **C**⟦C_1⟧ | **C**⟦C_2⟧
C⟦I:=E⟧ = **E**⟦E⟧ | *update*⟦I⟧
E⟦E_1+E_2⟧ = **E**⟦E_1⟧ | **E**⟦E_2⟧ $\mid_1$ *plus*

The Plumb system uses the YACC parser generator to configure a compiler. The generated compiler maps a source program to its combinator denotation, which is represented as a graph. Static semantics is performed on the graph. The code generator linearizes the graph into machine code. The system does sophisticated analysis on recursively defined objects like **while**-loops and circular environments, finding their optimal representation in machine code form. Plumb has been used on a number of test languages.

The CERES system is parameterized on whatever combinator set the user desires. A combinator set is made known to the system by a *compiler generator definition*, which is a mapping from the combinator set to code generation instructions. The system composes a

language's semantic definition with the compiler generator definition to generate the compiler for the language. CERES is designed to be a development tool for a variety of applications, rather than a compiler generator for a narrow class of languages. It also has been used on test languages.

10.5 TRANSFORMATIONS ON THE SEMANTIC DEFINITION

Yet another approach to implementation is to transform the denotational definition into an easily implementable form. The transformations exploit structural properties in the definitions: domains such as *Environment* and *Store* are made into data structures, store arguments are converted into global variables, command denotations become machine code, and so on. The correctness of the transformations is justified with respect to the rewriting rules of the semantic notation; that is, the transformed definition has a reduction strategy that parallels the one used on the original definition. This section presents several useful transformations.

10.5.1 First-Order Data Objects

Those semantic algebras that define data structures should have nonfunctional domains. Then their values are simpler to represent and their associated operations are easier to optimize. We convert a function domain $D = A \to B$ into a first-order (that is, nonfunctional) domain by representing the members of D by tuples (or lists or arrays). The conversion is called *defunctionalization.* Consider the *Store* algebra presented in Figure 5.1. A store is an abstraction value, and a construction operation such as *update* builds an abstraction from its arguments. The defunctionalized version of the algebra is presented in Figure 10.7.

A defunctionalized store value is a now a tuple, tagged with the name of the operation that built it. When a store tuple is used by the *access* operation, (*eval i s*) simulates function application. The definition of *eval* is built directly from the old definitions of the construction operations *newstore* and *update*.

The defunctionalized *Store* domain is *not* isomorphic to the original one (prove this), but every store that was representable using the former versions of the *Store* operations is representable with the new versions. A proof of correctness of the transformation exploits this fact to verify that any reduction using a higher-order store is successfully simulated by a reduction that uses the corresponding first-order store. Further, any reduction using a first-order store parallels a reduction that uses a higher-order store. The proof is left as an exercise.

Figure 10.8 shows a reduction with first-order stores.

It is usually straightforward to convert a defunctionalized algebra into a more efficient form. For example, store tuples are just lists of identifier, number pairs, and an empty store is an empty list. Once an identifier is updated with a new value, its former value is forever inaccessible. This suggests that a store be modelled as an array; that is, $Store = \prod_{i:Identifier} Nat$; the *access* and *update* operations become array indexing and updating.

Figure 10.7

IV.' Store
  Domain $s \in Store = New + Upd$
    where $New = Unit$
      $Upd = Identifier \times Nat \times Store$
  Operations
  $newstore : Store$
  $newstore =$ inNew()
  $access : Identifier \to Store \to Nat$
  $access = \lambda i.\lambda s.(eval\ i\ s)$
  $update : Identifier \to Nat \to Store \to Store$
  $update = \lambda i.\lambda n.\lambda s.$ in$Upd(i,n,s)$
  where
  $eval = \lambda i.\lambda s.$ cases s of
    is$New() \to zero$
    [] is$Upd(i_1,n_1,s_1) \to ((i\ equalid\ i_1) \to n_1$ [] $(eval\ i\ s_1))$
    end

Figure 10.8

C[[X:=Z; Y:=X+X]](*newstore*)
= $F_2(F_1(newstore))$,
  where $F_1 = \underline{\lambda} s.\ update$[[X]] ($access$[[Z]] s) s
    $F_2 = \underline{\lambda} s.\ update$[[Y]] (($access$[[X]] s) $plus$ ($access$[[X]] s)) s
=> $F_2(F_1\ s_0)$, where $s_0 =$ inNew()
=> $F_2(update$[[X]] ($access$[[Z]] s_0) $s_0)$
=> $F_2(update$[[X]] $zero\ s_0)$
=> $F_2\ s_1$, where $s_1 =$ inUpd([[X]], $zero$, s_0)
=> $update$[[Y]] (($access$[[X]] s_1) $plus$ ($access$[[X]] s_1)) s_1
=> $update$[[Y]] (($access$[[X]] s_1) $plus\ zero$) s_1
=> $update$[[Y]] ($zero\ plus\ zero$) s_1
=> $update$[[Y]] $zero\ s_1$
=> s_2, where $s_2 =$ inUpd([[Y]], $zero$, s_1)

The PSP system defunctionalizes data domains. The new values are ordered trees, allowing fast versions of access and update operations.

10.5.2 Global Variables

Intuition tells us that the store argument in a sequential language's definition should be treated as a global value. To make this point, we replace the *Store* domain by a store *variable.* Then the operations of the *Store* algebra no longer require a store argument, because they use the contents of the store variable. Of course, not all semantic definitions can be altered this way. The semantic equations must handle their store arguments in a ''sequential'' fashion: a store transformation function receives a single store argument, makes changes to it, and passes it on to the next function. Sequentiality is an operational notion, and we say that an expression is *single-threaded* (in its store argument) if a reduction strategy can be applied to the expression such that, at each stage of the reduction, there is at most one ''active'' normal form value of store in the stage. (A value is *active* if it does not appear within the body E of an abstraction $(\lambda x.\, E)$.) Raoult and Sethi formalize this concept by using a ''pebbling game''; they call single-threading *single pebbling*.

The reduction strategy that works the best to demonstrate the single-threadedness of an expression is a call-by-value one: treat all abstractions in the expression as if they were strict. (This is acceptable if the expression in question contains no abstractions of form $(\lambda x.\, E) : A_{\perp} \to B$.)

Figure 10.8 shows a call-by-value reduction. The active normal form values of stores are represented by terms s_i. At each stage of the reduction, there is at most one active normal form value of store. For example, the stage *update*⟦Y⟧ ((*access*⟦X⟧ s_1) *plus zero*) s_1 has two occurrences of the one active normal form value s_1. The actions upon the store occur in the same order as they would in a conventional implementation. The multiple copies of the stores could be replaced by a global variable holding a single copy of the store's value. Operations *access* and *update* would use the global variable.

For an expression to be single-threaded, its reduction must never present a stage where a store-updating redex is active at the same time when another active expression is using the current store. This update-access conflict implies that multiple stores are necessary.

Following are syntactic criteria that guarantee that an expression is single-threaded with respect to call-by-value reduction. Say that a *Store*-typed identifier is a *trivial Store*-typed expression; all other *Store*-typed expressions are *nontrivial.* The definition below states that a single-threaded expression is structured so that a store update must not be active in the same subexpression with any other active store expression (the noninterference property), and no store may be bound into an abstraction's body for later use (the immediate evaluation property).

10.1 Definition:

An expression F is single threaded (in its Store argument) if each of its subexpressions E possess the properties:

A (noninterference)

1. *If E is Store-typed, then if E contains multiple, disjoint active occurrences of Store-typed expressions, then they are the same trivial identifier;*
2. *If E is not Store-typed, all occurrences of active Store-typed expressions in E are the same trivial identifier.*

B (immediate evaluation)

1. *If* $E = (\underline{\lambda}x.M)$: $Store \to D$, *then all free Store-typed identifiers in M are x.*
2. *If* $E = (\underline{\lambda}x.M)$: $C \to D$, *and C is not Store-typed, then M contains no active Store-typed expressions.*

The noninterference property directly prevents the update-access conflict from arising in active expressions. The immediate evaluation property guarantees that an abstraction will also avoid conflict when it is reduced and its body is made active.

Let's look at some examples; let $\mathbf{C}$: Command $\to Store_{\perp} \to Store_{\perp}$ and $\mathbf{E}$: Expression $\to Store \to Expressible\text{-}value$. First, the expression $\mathbf{C}[\![C_2]\!](\mathbf{C}[\![C_1]\!]s)$ is single-threaded. The call-by-value reduction strategy lock-steps the reduction so that $\mathbf{C}[\![C_1]\!]$'s reductions must be performed before $\mathbf{C}[\![C_2]\!]$'s. Another example of compliance is $(\lambda s.\ \mathbf{E}[\![E_1]\!]s\ plus\ \mathbf{E}[\![E_2]\!]s)$, for all the *Store*-typed subterms in it are trivial.

In contrast, $(\mathbf{C}[\![C_1]\!]s\ combine\ \mathbf{C}[\![C_2]\!]s)$ violates clause A1, for $\mathbf{C}[\![C_1]\!]s$ and $\mathbf{C}[\![C_2]\!]s$ are nontrivial and disjoint and active in the same subterm. When the abstraction is given a store, which of the two commands should be reduced first? If a single store variable is used, an incorrect final store will likely result. Next, $\mathbf{E}[\![E]\!](\mathbf{C}[\![C]\!]s)$ violates property A2, because $(\mathbf{C}[\![C]\!]s)$ creates a local side effect that must be forgotten after the reduction of $\mathbf{E}[\![E]\!]$. Apparently $\mathbf{E}[\![E]\!]$ needs its own local copy of the store. The expression $(\underline{\lambda}s'.\ \mathbf{C}[\![C]\!]s')$ violates B1, for the store s' is hidden in the abstraction body, and it could be used at some later time in the reduction when a different store is current. Finally, $(\lambda n.\ update\ [\![I]\!]\ n\ s)$ violates B2 and introduces the same problem. All of these constructs would be difficult to implement on a sequential machine.

A denotational definition is single-threaded if all of its semantic equations are. Given a single-threaded denotational definition, we make the *Store* algebra into a *Store* ''module'' and replace all occurrences of stores in the semantic equations with the value $() : Unit$. Figure 10.9 shows the new version of the language in Figure 5.2. (Note: the expressions E_2 and E_3 are treated as *inactive* in $(E_1 \to E_2\ [\!]\ E_3)$.)

The () markers are passed as arguments in place of store values. A () marker is a ''pointer'' to the store variable, awarding access rights. But, more importantly, () is a *control marker,* for a denotation $(\lambda().E)$ can reduce (that is, ''get control'') only when it receives the (). The transformation has exposed the underlying *store-based control* in the programming language.

A program's denotation is no longer a single function expression *E,* but a pair (E, s), where *s* is the current value of the store variable. Rewrite rules operate on the expression, store pairs; for example, the new version of the β-rule is $(\cdots(\lambda x.M)N\cdots,\ s) \Rightarrow (\cdots[N/x]M\cdots,\ s)$. The rewriting rules that manipulate the store are:

$$(\cdots(access\ i\ ())\cdots,\ s) \Rightarrow (\cdots n\cdots,\ s) \qquad \text{where } eval\ i\ s = n$$
$$(\cdots(update\ i\ n\ ())\cdots,\ s) \Rightarrow (\cdots()\cdots,\ \text{in}Upd(i, n, s))$$

The command in Figure 10.8 is reduced with a store variable in Figure 10.10.

A machine implementation of the transformed denotational definition would treat the store variable as a machine component and *access* and *update* as machine instructions. The VEC-machine in Figure 10.2 becomes the *VECS-machine,* and it uses a machine configuration $(v\ e\ c\ s)$. The modification of Figure 10.2 to include the store variable is left as an exercise.

Figure 10.9

VI. Store module
var s : *Store* = *New* + *Upd*, like Figure 10.6
Operations

newstore : *Unit*
newstore = (s := in*New*())

access : *Identifier* → *Nat* → *Unit* → *Unit*
access = λ*i*.λ*n*.λ(). (*eval i s*)

update : *Identifier* → *Nat* → *Unit* → *Unit*
update = λ*i*.λ*n*.λ(). (s := in*Upd*(*i*, *n*, *s*))

Valuation functions:

C: Command → $Unit_\perp$ → $Unit_\perp$
C⟦C_1;C_2⟧ = $\underline{\lambda}$().**C**⟦C_2⟧ (**C**⟦C_1⟧())
C⟦**if** B **then** C_1 **else** C_2⟧ = $\underline{\lambda}$(). **B**⟦B⟧() → **C**⟦C_1⟧() [] **C**⟦C_2⟧()
C⟦**while** B **do** C⟧ = *wh*
where *wh* = $\underline{\lambda}$(). **B**⟦B⟧() → *wh*(**C**⟦C⟧()) [] ()
C⟦I:=E⟧ = $\underline{\lambda}$(). *update*⟦I⟧ (**E**⟦E⟧()) ()

E: Expression → *Unit* → *Nat*
E⟦E_1+E_2⟧ = λ(). **E**⟦E_1⟧() *plus* **E**⟦E_2⟧()
E⟦I⟧ = λ(). *access*⟦I⟧ ()
E⟦N⟧ = λ(). **N**⟦N⟧

Figure 10.10

C⟦X:=Z;Y:=X+X⟧(*newstore*) = $F_2(F_1\ newstore)$,
where F_1 = λ(). *update*⟦X⟧ (*access*⟦Z⟧()) ()
F_2 = λ(). *update*⟦Y⟧ ((*access*⟦X⟧ ()) *plus* (*access*⟦X⟧ ()) ()
=> $F_2(F_1())$, and *s* : *Store* has value in*New*().
=> F_2(*update*⟦X⟧ (*access*⟦Z⟧ ()) ())
=> F_2(*update*⟦X⟧ *zero* ())
=> F_2() and *s* : *Store* has value in*Upd*(⟦X⟧, *zero*, in*New*())
=> *update*⟦Y⟧ ((*access*⟦X⟧ ()) *plus* (*access*⟦X⟧ ())) ()
=> *update*⟦Y⟧((*access*⟦X⟧ ()) *plus zero*) ()
=> *update*⟦Y⟧ (*zero plus zero*) ()
=> *update*⟦Y⟧ *zero* ()
=> () and *s* : *Store* has value in*Upd*(⟦Y⟧, *zero*, in*Upd*(⟦X⟧, *zero*, in*New*())).

10.5.3 Control Structures

The definition in Figure 10.9 can be improved by writing it in a combinator format. We define the following combinators:

$E_1; E_2$ = (λ(). $E_2(E_1())$)
if B then E_1 *else* E_2 = (λ(). $B() \rightarrow E_1() [] E_2()$)
upd i E_1 = (λ(). *update i* E_1 () ())
skip = (λ(). ())
E_1+E_2 = (λ(). $E_1()$ *plus* $E_2()$)
n! = (λ(). *n*)

Figure 10.11 presents Figure 10.9 in combinator form.

Each of the combinators has a useful rewriting rule. A combinator *M* = (λ(). *N*) has the rewriting rule *M*() => *N*. The rewriting rules eliminate the lambda bindings but still distribute the control markers () throughout the subexpressions of an expression. The combinators are rightly called *control structures,* for they distribute control to their arguments. For example, the rewriting rule for $E_1; E_2$ makes it clear that E_1 gets control before E_2. The rule for E_1+E_2 shows that control can be given to the arguments in any order, even in parallel.

A stored program machine can be derived from the rewriting rules. With its instruction counter, the machine mimics the distribution of the () markers. The control markers become redundant— the instruction counter *is* (), and when the counter points to (the code of) an expression *E,* this represents the combination *E*(). You are given the exercise of defining such a machine for the language in Figure 10.11.

We make one remark regarding the **while**-loop in Figure 10.11. The rewriting rules evaluate a recursively defined object like *wh* by unfolding:

Figure 10.11

$\mathbf{C}[\![C_1;C_2]\!] = \mathbf{C}[\![C_1]\!]; \mathbf{C}[\![C_2]\!]$
$\mathbf{C}[\![I{:=}E]\!] = upd\ [\![I]\!]\ \mathbf{E}[\![E]\!]$
$\mathbf{C}[\![\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2]\!] = if\ \mathbf{B}[\![B]\!]\ then\ \mathbf{C}[\![C_1]\!]\ else\ \mathbf{C}[\![C_2]\!]$
$\mathbf{C}[\![\textbf{while } B \textbf{ do } C]\!] = wh$
where $wh = if\ \mathbf{B}[\![B]\!]\ then\ \mathbf{C}[\![C]\!]; wh\ else\ skip$
$\mathbf{E}[\![E_1{+}E_2]\!] = \mathbf{E}[\![E_1]\!] + \mathbf{E}[\![E_2]\!]$
$\mathbf{E}[\![I]\!] = access\ [\![I]\!]$
$\mathbf{E}[\![N]\!] = \mathbf{N}[\![N]\!]$!

$wh \Rightarrow if\ \mathbf{B}[\![B]\!]\ then\ \mathbf{C}[\![C]\!]; wh\ else\ skip$

Unfolding is not an efficient operation on a stored program machine. Steele (1977) points out, however, that certain recursively defined objects *do* have efficient representations: the *tail-recursive* ones. A recursively defined object is tail-recursive if the final value resulting from a reduction sequence of recursive unfoldings is the value produced by the last unfolding in the sequence. (Stated in another way, a return from a recursive call leads immediately to another return; no further computation is done.) An important property of a tail-recursive function is that its code can be abbreviated to a loop. This is justified by applying the ideas in Section 6.6.5; *wh* defines the infinite sequence:

$$if\ \mathbf{B}[\![B]\!]\ \begin{array}{l} then\ \mathbf{C}[\![C]\!]; if\ \mathbf{B}[\![B]\!]\ \begin{array}{l} then\ \mathbf{C}[\![C]\!];\ if\ \mathbf{B}[\![B]\!]\ \begin{array}{l} \ldots \\ else\ skip \end{array} \\ else\ skip \end{array} \\ else\ skip \end{array}$$

It is easy to see that this sequence has a finite abbreviation, just like the example in Section 6.6.5. Using *label* and *jump* operators, we abbreviate the infinite sequence to:

$\mathbf{C}[\![\textbf{while } B \textbf{ do } C]\!] = wh$
where $wh = label\ L{:}\ if\ \mathbf{B}[\![B]\!]\ then\ \mathbf{C}[\![C]\!]; jump\ L\ else\ skip$

The *label* and *jump* instructions are put to good use by the stored program machine. An induction on the number of unfoldings of an evaluation proves that the original and abbreviated definitions of *wh* are operationally equivalent on the stored program machine.

10.6 IMPLEMENTATION OF CONTINUATION-BASED DEFINITIONS

Many workers prefer to implement a language from its continuation semantics definition. An evaluator for a continuation semantics definition is straightforward to derive, because the nesting structure of the continuations suggests a sequential flow of control. The instruction set of

the evaluator is just the collection of operations from the continuation algebras, and the evaluator's components are derived from the semantic domains in the semantic definition.

In this section, we develop a compiler and an evaluator for the language in Figure 9.5. To see the evaluator structure suggested by that definition, consider the denotation of ⟦X:=1; X:=X+2; C⟧; it is:

$$(\textit{return-value one}\ \text{`a'})(\textit{assign l}\ \text{`b'})(\textit{fetch l}\ \text{`c'})(\textit{save-arg}\ \text{`d'})(\textit{return-value two})$$
$$(\text{`e'})(\textit{add}\ \text{`f'})(\textit{assign l}\ \text{`g'})(c_0)))))))$$

The letters in quotes prefix and name the continuations in the expression. For a hypothetical store s_0, a simplification sequence for the denotation is:

$(\textit{return-value one}\ \ \text{`a'})\ s_0$
$= (\textit{assign l}\ \ \text{`b'})\ \textit{one}\ s_0$
$= (\textit{fetch l}\ \ \text{`c'})\ s_1 \quad \text{where } s_1 = [\,l \mapsto \textit{one}\,]s_0$
$= (\textit{save-arg}\ \ \text{`d'}\ \text{`e'})\ \textit{one}\ s_1$
$= (\textit{return-value two}\ \ (\text{`e'}\ \textit{one}))\ s_1 \quad (*)$
$= (\textit{add}\ \ \text{`f'})\ \textit{one two}\ s_1$
$= (\textit{assign l}\ \ \text{`g'})\ \textit{three}\ s_1$
$= c_0\,[\,l \mapsto \textit{three}\,]s_1$

At each stage (except the stage labeled (∗)) the configuration has the form $(c\ n^*\ s)$; that is, a continuation c followed by zero or more expressible values n^*, followed by the store s. These three components correspond to the control, temporary value stack, and store components, respectively, of the VECS-machine. An environment component is not present, because the continuation operations hide binding identifiers.

The configurations suggest that the evaluator for the language has a configuration $(c\,v\,s)$, where $c \in \textit{Control-stack} = \textit{Instruction}^*$, $v \in \textit{Value-stack} = \textit{Exprval}^*$, and $s \in \textit{Store}$. The evaluator's instruction set consists of the operations of the continuation algebras. A nested continuation $c_1(c_2(\cdots c_n \cdots))$ is represented as a control stack $c_1{:}c_2{:}\cdots{:}c_n$.

The denotational definition is not perfectly mated to the machine structure. The problem appears in stage (∗): the *save-arg* operation forces its second continuation argument to hold an expressible value that should be left on the value stack. This is not an isolated occurrence; the functionality of expression continuations forces those expression continuations that require multiple expressible values to acquire them one at a time, creating local ''pockets'' of storage. We would like to eliminate these pockets and remove the *save-arg* operation altogether. One way is to introduce a value stack $\textit{Exprval}^*$ for the expression continuations' use. We define:

$v \in \textit{Value-stack} = \textit{Exprval}^*$
$k \in \textit{Exprcont} = \textit{Value-stack} \to \textit{Cmdcont}$

Regardless of how many expressible values an expression continuation requires, its argument is always the value stack. This form of semantics is called a *stack semantics.* It was designed by Milne, who used it in the derivation of a compiler for an ALGOL68 variant. Figure 10.12 shows the stack semantics corresponding to the definition in Figure 9.5.

Now command continuations pass along both the value stack and the store. The **if** and **while** commands use a different version of the *choose* operation; the new version eliminates

Figure 10.12

XII'. Command continuations
Domain $c \in Cmdcont = Value\text{-}stack \to Store \to Answer$
Operations

$finish : Cmdcont$
$finish = \lambda v.\lambda s.\ \text{in}OK(s)$

$err : String \to Cmdcont$
$err = \lambda t.\lambda v.\lambda s.\ \text{in}Err(t)$

$skip : Cmdcont \to Cmdcont$
$skip = \lambda c.\ c$

XIII'. Expression continuations
Domain $k \in Exprcont = Value\text{-}stack \to Store \to Answer$
Operations

$return\text{-}value : Exprval \to Exprcont \to Exprcont$
$return\text{-}value = \lambda n.\lambda k.\lambda v.\ k(n\ cons\ v)$

$add : Exprcont \to Exprcont$
$add = \lambda k.\lambda v.\ k(((hd(tl\ v))\ plus\ (hd\ v))\ cons\ (tl(tl\ v)))$

$fetch : Location \to Exprcont \to Exprcont$
$fetch = \lambda l.\lambda k.\lambda v.\lambda s.\ k((access\ l\ s)\ cons\ v)\ s$

$assign : Location \to Cmdcont \to Exprcont$
$assign = \lambda l.\lambda c.\lambda v.\lambda s.\ c(tl\ v)\ (update\ l\ (hd\ v)\ s)$

$choose : (Cmdcont \to Cmdcont) \to (Cmdcont \to Cmdcont) \to Cmdcont \to Exprcont$
$choose = \lambda f.\lambda g.\lambda c.\lambda v.\ ((hd\ v)\ greaterthan\ zero \to (f\ c)\ []\ (g\ c))\ (tl\ v)$

Valuation functions:

$\mathbf{P}_{\text{I}}$: Program $\to Location \to Cmdcont$ (like Figure 9.5)

$\mathbf{B}_{\text{I}}$: Block $\to Environment \to Cmdcont \to Cmdcont$ (like Figure 9.5)

$\mathbf{D}_{\text{I}}$: Declaration $\to Environment \to Environment$ (like Figure 9.5)

$\mathbf{C}_{\text{I}}$: Command $\to Environment \to Cmdcont \to Cmdcont$ (like Figure 9.5, except for)
$\mathbf{C}_{\text{I}}[\![\textbf{if}\ \text{B}\ \textbf{then}\ \text{C}_1\ \textbf{else}\ \text{C}_2]\!] = \lambda e.\ \mathbf{E}_{\text{I}}[\![\text{E}]\!] \circ (choose\ (\mathbf{C}_{\text{I}}[\![\text{C}_1]\!]e)\ (\mathbf{C}_{\text{I}}[\![\text{C}_2]\!]e))$
$\mathbf{C}_{\text{I}}[\![\textbf{while}\ \text{B}\ \textbf{do}\ \text{C}]\!] = \lambda e.\ fix(\lambda g.\ \mathbf{E}_{\text{I}}[\![\text{E}]\!]e \circ (choose\ (\mathbf{C}_{\text{I}}[\![\text{C}]\!]e \circ g)\ skip))$

$\mathbf{E}_{\text{I}}$: Expression $\to Environment \to Exprcont \to Exprcont$ (like Figure 9.5, except for)
$\mathbf{E}_{\text{I}}[\![\text{E}_1 + \text{E}_2]\!] = \lambda e.\lambda k.\ \mathbf{E}_{\text{I}}[\![\text{E}_1]\!]e\ (\mathbf{E}_{\text{I}}[\![\text{E}_2]\!]e\ (add\ k))$

---

redundant continuations. The *save-arg* operation disappears from the semantics of addition.

It is nontrivial to prove that the stack semantics denotation of a program is the same as its

continuation semantics denotation (see Milne & Strachey 1976). Since we have altered the expression continuation domain, its correspondence to the original domain is difficult to state. A weaker result, but one that suffices for implementation questions, is that the reduction of a stack semantics denotation of a program parallels the reduction of its continuation semantics denotation. We must show: for every syntactic form ⟦M⟧; for environment e and its corresponding environment e' in the stack semantics; for corresponding continuations k and k'; value stack v; store s; and expressible value n:

$$\mathbf{M}[\![\mathrm{m}]\!]e\,k\,s \Rightarrow^* k(n)\,s' \text{ iff } \mathbf{M}'[\![\mathrm{m}]\!]e'\,k'\,v\,s \Rightarrow^* k'\,(n{:}v)\,s'$$

By ''corresponding'' continuations k and k', we mean that $(k\,n\,s')$ equals $(k'\,(n{:}v)\,s')$. Environments e and e' correspond when they map identifiers to corresponding continuations (or the same values, if the identifier's denotable value is not a continuation). Most of the cases in the proof are easy; the proof for the **while**-loop is by induction on the number of unfoldings of *fix* in a reduction sequence. The block construct is proved similarly to the loop but we must also show that the respective environments created by the block correspond.

The evaluator for the stack semantics appears in Figure 10.13. The evaluator's rules are just the definitions of the continuation operations. A source program is executed by mapping it to its stack semantics denotation, performing static semantics, and evaluating the resulting

Figure 10.13

$c \in \mathit{Control\text{-}stack} = \mathit{Instruction}^*$
$v \in \mathit{Value\text{-}stack} = \mathit{Exprval}^*$
$s \in \mathit{Store}$
where $\mathit{Instruction} = \mathit{return\text{-}value} + \mathit{add} + \mathit{fetch} + \mathit{assign} + \mathit{choose} + \mathit{skip} + \mathit{finish} + \mathit{err}$

Instruction interpretation:

$\mathit{return\text{-}value}\ n{:}c \quad v \quad s \Rightarrow c \quad n{:}v \quad s$
$\mathit{add}{:}c \quad n_2{:}n_1{:}v \quad s \Rightarrow c \quad n_3{:}v\ s$
where n_3 is the sum of n_1 and n_2
$\mathit{fetch}\ l{:}c \quad v \quad s \Rightarrow c \quad n{:}v \quad s$
where $n = \mathit{access}\ l\ s$
$\mathit{assign}\ l{:}c\ n{:}v \quad s \Rightarrow c \quad v \quad s'$
where $s' = \mathit{update}\ l\ n\ s$
$(\mathit{choose}\ f\ g){:}c \quad \mathit{zero}{:}v \quad s \Rightarrow g{:}c \quad v \quad s$
$(\mathit{choose}\ f\ g){:}c \quad n{:}v \quad s \Rightarrow f{:}c \quad v \quad s$
where n is greater than zero
$\mathit{finish} \quad v \quad s \Rightarrow \mathrm{in}\mathit{OK}(s)$
$\mathit{err}\ t \quad v \quad s \Rightarrow \mathrm{in}\mathit{Err}(t)$
$\mathit{skip}{:}c \quad v \quad s \Rightarrow c \quad v \quad s$

---

continuation on the evaluator.

A similar method for inducing the temporary value stack has been developed by Wand. He does not introduce a value stack domain but inserts ''argument steering'' combinators into the semantic equations instead. Run-time configurations take the form ($c\ n_1 \cdots n_m\ s$). The net result is the same, but the proof that the new semantic definition is equal to the original is much simpler than the one needed for our example. Wand's research papers document the method.

Although the instruction set for the interpreter is in place, *fix* operators still appear in those translated programs containing **while**-loops and labels. The fixed point simplification property ($fix\,F = F(fix\,F)$) can be used by the interpreter, but we choose to introduce the operation *while f do g*, representing the expression $fix(\lambda c'.\ f \circ (choose\ (g \circ c')\ skip))$ so that **C**⟦**while** B **do** C⟧ = $\lambda e.\ while$ **E**⟦E⟧*e do* **C**⟦C⟧*e*. The evaluation rule is:

while f do h:*c v* *s* => *f*:(*choose* (*h*:*while f do h*) *skip*):*c* *v* *s*

The **goto**s present a more serious problem. A block with labels does not have a simple, tail-recursive expansion like the **while**-loop. One solution is to carry over the language's environment argument into the evaluator to hold the continuations associated with the labels. A **goto** causes a lookup into the environment and a reloading of the control stack with the continuation associated with the **goto**.

The problem with the **goto**s is better resolved when an instruction counter is incorporated into the evaluator, *jump* and *jumpfalse* instructions are added, and the code for blocks is generated with jumps in it. A continuation ($ctuple{\downarrow}i$) is a tail-recursive call to the *i*th *ctuple* component. If the *i*th component begins at label *L*, we generate *jump L*. (Also, the **while**-loop can be compiled to conventional loop code.) The proof of correctness of these transformations is involved and you are referred to Milne & Strachey (1976) and Sethi (1981).

10.6.1 The CGP and VDM Methods

Raskovsky's Code Generator Process (CGP) is a transformation-oriented compiler generation method. It transforms a denotational definition into a BCPL-coded code generator in a series of transformation steps. Each transformation exploits a property of the semantic definition. Defunctionalization, continuation introduction, and global variable introduction for stores and environments are examples of transformations. The generated compiler maps source programs to BCPL code; the evaluator is the BCPL system. The system has been used to develop compilers for nontrivial languages such as GEDANKEN.

Bjørner and Jones advocate a stepwise transformation method that also uses transformations described in this chapter. Their work is part of a software development methodology known as the Vienna Development Method (VDM). VDM system specifications are denotational definitions; the semantic notation is a continuation-style, combinator-based notation called META-IV. Evaluators for VDM specifications are derived using the ideas in the previous section. Compilers, data bases, and other systems software have been specified and implemented using the method. See Bjørner and Jones, 1978 and 1982, for a comprehensive

presentation.

10.7 CORRECTNESS OF IMPLEMENTATION AND FULL ABSTRACTION

In previous sections, we stated criteria for correctness of implementation. We now consider the general issue of correctness and provide standards for showing that an operational semantics definition is complementary to a denotational one. The study leads to a famous research problem known as the *full abstraction problem.*

Recall that an operational semantics definition of a language is an interpreter. When the interpreter evaluates a program, it generates a sequence of machine configurations that define the program's operational semantics. We can treat the interpreter as an evaluation relation $\Rightarrow$ that is defined by rewriting rules. A program's operational semantics is just its reduction sequence. For a source language L, we might define an interpreter that reduces L-programs directly. However, we have found it convenient to treat L's semantic function $\mathbf{P}: L \rightarrow D$ as a syntax-directed translation scheme and use it to translate L-programs to function expressions. Then we interpret the function expressions. Let $e \in$ *Function-expr* be a function expression representation of a source program after it has been translated by **P**. Since expressions are treated as syntactic entities by the interpreter for function expressions, we write $e_1 \equiv e_2$ to assert that the denotations of the function expressions e_1 and e_2 are the same value.

Let I be the set of interpreter configurations and let ϕ: *Function-expr* $\rightarrow I$ be a mapping that loads a function expression into an initial interpreter configuration. A configuration is *final* if it cannot be reduced further. The set of final interpreter configurations is called **Fin**. We also make use of an ''abstraction'' map $\psi: I \rightarrow$ *Function-expr*, which restores the function expression corresponding to an interpreter configuation. The minimum required to claim that an operational semantics definition is complementary to a denotational one is a form of soundness we call *faithfulness.*

10.2 Definition:

An operational semantics $\Rightarrow$ *is faithful to the semantics of Function-expr if for all* $e \in$ *Function-expr and* $i \in I$, $\phi(e) \Rightarrow^* i$ *implies* $e \equiv \psi(i)$.

Faithfulness by itself is not worth much, for the interpreter can have an empty set of evaluation rules and be faithful. Therefore, we define some subset **Ans** of *Function-expr* to be answer forms. For example, the set of constants in *Nat* can be an answer set, as can the set of all normal forms. A guarantee of forward progress to answers is a form of completeness we call *termination.*

10.3 Definition:

An operational semantics $\Rightarrow$ *is terminating in relation to the semantics of Function-expr if, for all* $e \in$ *Function-expr and* $a \in$ **Ans**, *if* $e \equiv a$, *then there exists some* $i \in$ **Fin** *such that* $\phi(e) \Rightarrow^* i$, $\psi(i) \in$ **Ans**, *and* $\psi(i) \equiv a$.

Termination is the converse of faithfulness, restricted to the i in **Fin**. We use the requirement $\psi(i) \equiv a$ (rather than $\psi(i) = a$) because two elements of **Ans** may share the same value (e.g., normal form answers ($\lambda t.\, t \to a\ []\ b$) and ($\lambda t.\, not(t) \to b\ []\ a$) are distinct answers with the same value). If **Ans** is a set whose elements all have distinct values (e.g., the constants in *Nat*), then $\psi(i)$ must be a. If the operational semantics is faithful, then the requirement that $\psi(i) \equiv a$ is always satisfied.

We apply the faithfulness and termination criteria to the compile-evaluate method described in Sections 10.1 and 10.3. Given a denotational definition $\mathbf{P} : L \to D$, we treat **P** as a syntax-directed translation scheme to function expressions. For program $[\![P]\!]$, the expression $\mathbf{P}[\![P]\!]$ is loaded into the interpreter. In its simplest version, the interpreter requires no extra data structures, hence both ϕ and ψ are identity maps. Recall that the interpreter uses a leftmost-outermost reduction strategy. A final configuration is a normal form. An answer is a (non-$\perp$) normal form expression. By exercise 11 in Chapter 3, the reductions preserve the meaning of the function expression, so the implementation is faithful. In Section 10.3 we remarked that the leftmost-outermost method always locates a normal form for an expression if one exists. Hence the method is terminating.

The definitions of faithfulness and termination are also useful to prove that a low-level operational semantics properly simulates a high-level one: let (the symmetric, transitive closure of) the evaluation relation for the high-level interpreter define $\equiv$ and let the low-level evaluation relation define $\Rightarrow$. **Ans** is the set of final configurations for the high-level interpreter, and **Fin** is the set of final configurations for the low-level one. This method was used in Sections 10.3 and 10.6.

There are other versions of termination properties. We develop these versions using the function expression interpreter. In this case, ϕ and ψ are identity maps, and the **Ans** and **Fin** sets are identical, so we dispense with the two maps and **Fin** and work directly with the function expressions and **Ans**. We define a *context* to be a function expression with zero or more ''holes'' in it. If we view a context as a derivation tree, we find that zero or more of its leaves are nonterminals. We write a hypothetical context as $C[\ ]$. When we use an expression E to fill the holes in a context $C[\ ]$, giving a well-formed expression, we write $C[E]$. We fill the holes by attaching E's derivation tree to all the nonterminal leaves in $C[\ ]$'s tree. The formalization of contexts is left as an exercise.

A context provides an operating environment for an expression and gives us a criterion for judging information content and behavior. For example, we can use structural induction to prove that for expressions M and N and context $C[\ ]$, $M \sqsubseteq N$ implies that $C[M] \sqsubseteq C[N]$. We write $M \tilde{\sqsubseteq} N$ if, for all contexts $C[\ ]$ and $a \in \mathbf{Ans}$, $C[M] \Rightarrow^* a$ implies that $C[N] \Rightarrow^* a'$ and $a \equiv a'$. We write $M \tilde{\simeq} N$ if $M \tilde{\sqsubseteq} N$ and $N \tilde{\sqsubseteq} M$ and say that M and N are *operationally equivalent.* The following result is due to Plotkin (1977).

10.4 Proposition:

If $\Rightarrow$ is faithful to the semantics of Function-expr, then $\Rightarrow$ is terminating iff for all $M, N \in$ Function-expr, $M \equiv N$ implies $M \tilde{\simeq} N$.

Proof: Left as an exercise. □

This result implies that denotational semantics equality implies operational equivalence under

the assumption that the operational semantics is faithful and terminating. Does the converse hold? That is, for a faithful and terminating operational semantics, does operational equivalence imply semantic equality? If it does, we say that the denotational semantics of *Function-expr* is *fully abstract* in relation to its operational semantics.

Plotkin (1977) has shown that the answer to the full abstractness question for *Function-expr* and its usual interpreter is *no.* Let $F_b: (Tr_\perp \times Tr_\perp) \to Tr_\perp$ be the function expression:

$\lambda a.$ let $t_1 = a(true, \perp)$ in
  $t_1 \to$ (let $t_2 = a(\perp, true)$ in
    $t_2 \to$ (let $t_3 = a(false, false)$ in
      $t_3 \to \perp \,[]\, b$)
    $[]\, \perp$)
  $[]\, \perp$

We see that $F_{true} \not\equiv F_{false}$ (and vice versa) by considering the continuous function $v: Tr_\perp \times Tr_\perp \to Tr_\perp$ whose graph is $\{((\perp, true), true), ((true, \perp), true), ((true, true), true), ((true, false), true), ((false, true), true), ((false, false), false)\}$ (as usual, pairs $((m, n), \perp)$ are omitted). The function v is the ''parallel or'' function, and $F_{true}(v) = true$ but $F_{false}(v) = false$. But Plotkin proved, for the language of function expressions and its usual operational semantics, that $F_{true} \simeq F_{false}$! The reason that the two expressions are operationally equivalent is that ''parallel'' functions like v are not representable in the function notation; the notation and its interpreter are inherently ''sequential.''

There are two possibilities for making the denotational semantics of *Function-expr* fully abstract in relation to its interpreter. One is to extend *Function-expr* so that ''parallel'' functions are representable. Plotkin did this by restricting the **Ans** set to be the set of constants for *Nat* and *Tr* and introducing a parallel conditional operation *par-cond*: $Tr_\perp \times D_\perp \times D_\perp \to D_\perp$, for $D \in \{Nat, Tr\}$, with reduction rules:

par-cond(*true*, d_1, d_2) $\Rightarrow d_1$
par-cond(*false*, d_1, d_2) $\Rightarrow d_2$
par-cond($\perp$, *d*, *d*) $\Rightarrow d$

Then $v = \lambda(t_1, t_2).$ *par-cond*(t_1, *true*, t_2). Plotkin showed that the extended notation and its denotational semantics are fully abstract in relation to the operational semantics.

Another possibility for creating a fully abstract semantics is to reduce the size of the semantic domains so that the nonrepresentable ''parallel'' functions are no longer present. A number of researchers have proposed domain constructions that do this. The constructions are nontrivial, and you should research the suggested readings for further information.

SUGGESTED READINGS

General compiler generating systems: Appel 1985; Ganzinger, Ripken & Wilhelm 1977; Jones 1980; Mosses 1975, 1976, 1979; Paulson 1982, 1984; Pleban 1984; Wand 1983
Static semantics processing: Ershov 1978; Jones et al. 1985; Mosses 1975; Paulson 1982

Sethi 1981

Rewriting rules & evaluators for function notation: Berry & Levy 1979; Burge 1975; Hoffman & O'Donnell 1983; Huet & Oppen 1980; Landin 1964; Vegdahl 1984

Combinator systems & evaluators: Christiansen & Jones 1983; Clarke et al. 1980; Curry & Feys 1958; Hudak & Krantz 1984; Hughes 1982; Mosses 1979a, 1980, 1983a, 1984; Raoult & Sethi 1982; Sethi 1983; Turner 1979

Transformation methods: Bjørner & Jones 1978, 1982; Georgeff 1984; Hoare 1972; Raoult & Sethi 1984; Raskovsky & Collier 1980; Raskovsky 1982; Reynolds 1972; Schmidt 1985a, 1985b; Steele 1977; Steele & Sussman 1976a, 1976b

Continuation-based implementation techniques: Clinger 1984; Henson & Turner 1982; Milne & Strachey 1976; Nielson 1979; Polak 1981; Sethi 1981; Wand 1980a, 1982a, 1982b, 1983, 1985b

Full abstraction: Berry, Curien, & Levy 1983; Milner 1977; Mulmuley 1985; Plotkin 1977; Stoughton 1986

EXERCISES

1. a. Using the semantics of Figure 5.2, evaluate the program **P**⟦Z:=A+1⟧*four* using the compile-evaluate method and using the interpreter method.
   b. Repeat part a for the program **P**⟦**begin var** A; A:=A+1 **end**⟧ l_0 ($\lambda l.\, zero$) and the language of Figures 7.1 and 7.2.

2. If you have access to a parser generator system such as YACC and a functional language implementation such as Scheme, ML, or LISP, implement an SPS-like compiler generating system for denotational semantics.

3. Using the criteria for determining unfrozen expressions, work in detail the static semantics processing of:
   a. The example in Section 10.2.
   b. The program in part b of Exercise 1 (with the *Store*, *Nat*, and *Tr* algebras frozen).
   c. The program in Figure 7.6 (with the *Function*, *List*, and *Environment* algebras frozen; and again with just the *Function* and *List* algebras frozen).

4. Let the *Store* algebra be unfrozen; redo the static semantics of parts a and b of Exercise 3. Why don't real life compilers perform a similar service? Would this approach work well on programs containing loops?

5. The example of static semantics simplification in Section 10.2 showed combinations of the form $(\underline{\lambda} s.M)s$ simplified to M. Why is this acceptable in the example? When is it not?

6. Recall that the simplification rule for the *fix* operation is $(fix\, F) = F(fix\, F)$. How should this rule be applied during static semantics analysis? Consider in particular the cases of:

i. Recursively defined environments $e = fix(\lambda e_1. \cdots)$.
ii. Recursively defined commands and procedures $p = fix(\lambda p.\underline{\lambda} s. \cdots)$.

7. A number of researchers have proposed that a language's valuation function be divided into a static semantics valuation function and a dynamic semantics valuation function:

$$\mathbf{C}_S\text{: Command} \to \mathit{Environment} \to \mathit{Tr}$$
$$\mathbf{C}_D\text{: Command} \to \mathit{Environment}_1 \to ((\mathit{Store} \to \mathit{Poststore}) + \mathit{Err})$$

such that a well typed command $[\![C]\!]$ has denotation $\mathbf{C}_S[\![C]\!] = \mathit{true}$ and a denotation $\mathbf{C}_D[\![C]\!]$ in the summand $(\mathit{Store} \to \mathit{Poststore})$. Similarly, an ill typed program has a denotation of *false* with respect to $\mathbf{C}_S$ and an in*Err*() denotation with respect to $\mathbf{C}_D$.

a. Formulate these valuation functions for the language in Figures 7.1 and 7.2.
b. Comment on the advantages of this approach for implementing the language. Are there any disadvantages?

8. a. Perform leftmost-outermost reductions on the following expressions. If the expression does not appear to have a normal form, halt your reduction and justify your decision.

i. $(\lambda x.\, zero)((\lambda x.\, x\, x)(\lambda x.\, x\, x))$
ii. $(\underline{\lambda} x.\, zero)((\lambda x.\, x\, x)(\lambda x.\, x\, x))$
iii. $((\lambda x.\lambda y.\, y(y\, x))(one\ plus\ one))(\lambda z.\, z\ times\ z)$
iv. $((\underline{\lambda} x.\underline{\lambda} y.\, y(y\, x))(one\ plus\ one))(\lambda z.\, z\ times\ z)$
v. $((\underline{\lambda} x.\underline{\lambda} y.\, y(y\, x))(one\ plus\ one))(\underline{\lambda} z.\, z\ times\ z)$

b. Redo part a, representing the expressions as trees. Show the traversal through the trees that a tree evaluator would take.

9. Implement the tree evaluator for function notation. (It is easiest to implement it in a language that supports list processing.) Next, improve the evaluator to use an environment table. Finally, improve the evaluator to use call-by-need evaluation.

10. Evaluate the code segment $pushclosure(\Delta\text{: } return)\text{: } \Delta\text{: } call$ from Figure 10.4 on the VEC-machine.

11. Translate the expressions in exercise 8 into VEC-machine code and evaluate them on the VEC-machine.

12. Improve the code generation map **T** for the VEC-machine so that it generates assembly code that contains jumps and conditional jumps.

13. Improve the VEC-machine so that *push x* becomes *push* ''*offset*,'' where ''*offset*'' is the offset into the environment where the value bound to *x* can be found. Improve the code generation map so it correctly calculates the offsets for binding identifiers.

14. Augment the VEC-machine with instructions for handling pairs and sum values. Write the translations $\mathbf{T}[\![(E_1, E_2)]\!]$ and $\mathbf{T}[\![\text{cases } E_1 \text{ of } G]\!]$, where G ::= in$I_1(I_2) \rightarrow E$ [] G | end.

15. Compile the programs in Exercise 3 to VEC-code after static semantics has been performed. Can you suggest an efficient way to perform static semantics?

16. a. Augment the combinator language definition in Figure 10.7 with combinators for building environments.
    b. Rewrite the semantics of the language in Figures 7.1 and 7.2 in combinator form.
    c. Propose a method for doing static semantics analysis on the semantics in part b.

17. Convert the *Store* algebra in Figure 10.7 into one that manipulates ordered tree values. Are the new versions of *access* and *update* more efficient than the existing ones?

18. a. Verify that the **C** and **E** valuation functions of the language of Figure 5.2 are single-threaded. Does the **P** function satisfy the criteria of Definition 10.1? Is it single-threaded?
    b. Extend Definition 10.1 so that a judgement can be made about the single-threadedness of the language in Figures 7.1 and 7.2. Is that language single-threaded? Derive control combinators for the language.

19. An alternative method of generating a compiler from a continuation semantics of a language is to replace the command and expression continuation algebras by algebras of machine code. Let *Code* be the domain of machine code programs for a VEC-like stack machine, and use the following two algebras in place of the command and expression continuation algebras in Figure 9.5 (note: the ‘‘:’’ denotes the code concatenation operator):

    XII'. Command code
    Domain $c \in Cmdcont = Code$
    Operations
    finish = **stop**
    error = $\lambda t.$ **pushconst** t

    XIII'. Expression code
    Domain $k \in Exprcont = Code$
    Operations
    return-value = $\lambda n.\lambda k.$ **pushconst** n: k
    save-arg = $\lambda f.\lambda g.\ f(g)$
    add = $\lambda k.$ **add**: k
    fetch = $\lambda l.\lambda k.$ **pushvar** l: k
    assign = $\lambda l.\lambda c.$ **storevar** l: c
    choose = $\lambda c_1.\lambda c_2.$ **jumpzero** L_1: c_1: **jump** L_2: **label** L_1: c_2: **label** L_2

a. Generate the denotations of several programs using the new algebras.
b. Outline a proof that shows the new algebras are faithful to the originals in an operational sense.

20. Revise the stack semantics in Figure 10.13 so that the expressible value stack isn't used by command continuations; that is, the *Cmdcont* domain remains as it is in Figure 9.5, but the stack is still used by the *Exprcont* domain, that is, *Exprcont* is defined as in Figure 10.13. What are the pragmatics of this semantics and its implementation?

21. a. Prove that the evaluator with an environment in Section 10.3 is faithful and terminating in relation to the evaluator in Section 10.1.
    b. Prove that the VEC-machine in Section 10.3.1 is faithful and terminating in relation to the evaluator with environment in Section 10.3 when the answer set is limited to first-order normal forms.
    c. Prove that the defunctionalized version of a semantic algebra is faithful and terminating in relation to the original version of the algebra.

22. Prove that the properties of faithfulness and termination compose; that is, for operational semantics *A*, *B*, and *C*, if *C* is faithful/terminating in relation to *B*, and *B* is faithful/terminating in relation to *A*, then *C* is faithful/terminating in relation to *A*.

23. a. Give the graph of the function $par\text{-}cond : Tr_\perp \times D_\perp \times D_\perp \to D_\perp$ for $D \in \{ Nat, Tr \}$, prove that the function is continuous, and prove that its rewriting rules are sound.
    b. Attempt to define the graph of a continuous function $par\text{-}cond : Tr_\perp \times D \times D \to D$ for arbitrary D and show that its rewriting rules in Section 10.7 are sound. What goes wrong?