Chapter 9

Control as a Semantic Domain

The sequencing found in programs' denotations is somewhat illusionary. Sequencing is an operational concept, and function definitions contain no actual ''sequencing,'' regardless of their format. The sequencing is suggested by the simplification strategies for function notation. A simplification step corresponds roughly to an operational evaluation step, and the order in which the simplifications occur suggests an operational semantics. More than one simplification strategy may be acceptable, however, so the operational ideas do not predominate the definition.

The style of denotational semantics we have been using is called *direct semantics.* Direct semantics definitions tend to use lower-order expressions (that is, nonfunctional values and functions on them) and emphasize the compositional structure of a language. The equation:

$$\mathbf{E}[\![E_1+E_2]\!] = \lambda s.\ \mathbf{E}[\![E_1]\!]s\ \mathit{plus}\ \mathbf{E}[\![E_2]\!]s$$

is a good example. The equation gives a simple exposition of side-effect-free addition. The order of evaluation of the operands isn't important, and any simplification strategy works fine. One of the aims of the denotational semantics method is to make the meanings of program constructs clear without relying on operational mechanisms, and in this regard direct semantics performs well.

Languages designed for low-level or general application confront their users with the concept of control. *Control* might be defined as the evaluation ordering of a program's constructs. A language that promotes control as a primary feature provides the user with the ability to affect the control; that is, to change the order of evaluation. Control is an argument that the user can seize and alter. An example of a control argument is the stack of activation records maintained in support of an executing program. The stack contains the sequencing information that ''drives'' the program, and altering the stack alters the program's future order of evaluation.

The semantic argument that models control is called a *continuation.* Continuations were first developed for modelling unrestricted branches (''gotos'') in general purpose languages, but their utility in developing nonstandard evaluation orderings has made them worthy of study in their own right. This chapter presents a number of forms of continuations and their uses.

9.1 CONTINUATIONS

We begin with a small example that uses a control argument. Consider an imperative language similar to the one in Figure 7.2 augmented with a FORTRAN-like **stop** command. The evaluation of a **stop** in a program causes a branch to the very end of the program, cancelling the evaluation of all remaining statements. The output store that the program produces is

the one that is supplied as an argument to **stop.** The semantics of a **stop** command can be handled within direct semantics by applying the technique used in Figure 7.1 to trap error values, but we wish to model the change of control more directly. We add a control stack argument to the semantic function. The control stack keeps a list of all the commands that need to be evaluated. The valuation function repeatedly accesses the stack, popping off the top command and executing it. An empty stack means that evaluation is complete, and a **stop** command found at the top of the stack causes the remainder of the stack to be discarded. The valuation function for commands has functionality:

$$\mathbf{C}: \text{Command} \to \mathit{Environment} \to \mathit{Control\text{-}stack} \to \mathit{Store} \to \mathit{Store}_{\perp}$$

where $c \in \mathit{Control\text{-}stack} = (\mathit{Control\text{-}stack} \to \mathit{Store} \to \mathit{Store}_{\perp})^{*}_{\perp}$. The expression $(\mathbf{C}[\![\mathrm{C}]\!]e\,c\,s)$ resembles an interpreter-like configuration where $(\mathbf{C}[\![\mathrm{C}]\!]e)$ is the control stack top, c is the remainder of the stack, and s is the usual store argument. A fragment of the **C** function reads:

$$\mathbf{C}[\![\mathrm{C}_1;\mathrm{C}_2]\!] = \lambda e.\lambda c.\lambda s.\ \mathbf{C}[\![\mathrm{C}_1]\!]e\ ((\mathbf{C}[\![\mathrm{C}_2]\!]e)\ \mathit{cons}\ c)\ s$$
$$\mathbf{C}[\![\mathrm{I}:=\mathrm{E}]\!] = \lambda e.\lambda c.\lambda s.\ (\mathit{hd}\ c)\ (\mathit{tl}\ c)\ (\mathit{update}(\mathit{accessenv}[\![\mathrm{I}]\!]\ e)\ (\mathbf{E}[\![\mathrm{E}]\!]e\ s)\ s)$$
$$\ldots$$
$$\mathbf{C}[\![\mathbf{stop}]\!] = \lambda e.\lambda c.\lambda s.\ s$$

We obtain a neat definition for the **while**-loop:

$$\mathbf{C}[\![\mathbf{while}\ \mathrm{B}\ \mathbf{do}\ \mathrm{C}]\!] = \lambda e.\lambda c.\lambda s.\ \mathbf{B}[\![\mathrm{B}]\!]e\ s \to \mathbf{C}[\![\mathrm{C}]\!]e\ ((\mathbf{C}[\![\mathbf{while}\ \mathrm{B}\ \mathbf{do}\ \mathrm{C}]\!]e)\ \mathit{cons}\ c)\ s \ [\!]\ (\mathit{hd}\ c)\ (\mathit{tl}\ c)\ s$$

which makes clear that control returns to the top of the loop after evaluating the loop's body.

Whenever the c stack is popped, $(\mathit{hd}\ c)$ is always given $(\mathit{tl}\ c)$ as its argument. The simplification steps are shortened if the semantics of $[\![\mathrm{C}_1;\mathrm{C}_2]\!]$ is written so that $(\mathbf{C}[\![\mathrm{C}_2]\!]e)$ takes c as an argument at ''push-time.'' The stack can be replaced by a function. This new function is a (*command*) *continuation.* Its domain is $c \in \mathit{Cmdcont} = \mathit{Store} \to \mathit{Store}_{\perp}$. The language fragment now reads:

$$\mathbf{C}: \text{Command} \to \mathit{Environment} \to \mathit{Cmdcont} \to \mathit{Cmdcont}$$
$$\mathbf{C}[\![\mathrm{C}_1;\mathrm{C}_2]\!] = \lambda e.\lambda c.\ \mathbf{C}[\![\mathrm{C}_1]\!]e\ (\mathbf{C}[\![\mathrm{C}_2]\!]e\ c)$$
$$\mathbf{C}[\![\mathrm{I}:=\mathrm{E}]\!] = \lambda e.\lambda c.\lambda s.\ c(\mathit{update}(\mathit{accessenv}[\![\mathrm{I}]\!]\ e)\ (\mathbf{E}[\![\mathrm{E}]\!]e\ s)\ s)$$
$$\mathbf{C}[\![\mathbf{stop}]\!] = \lambda e.\lambda c.\lambda s.\ s$$
$$\mathbf{C}[\![\mathbf{if}\ \mathrm{B}\ \mathbf{then}\ \mathrm{C}_1\ \mathbf{else}\ \mathrm{C}_2]\!] = \lambda e.\lambda c.\ \mathit{choose}(\mathbf{B}[\![\mathrm{B}]\!]e)\ (\mathbf{C}[\![\mathrm{C}_1]\!]e\ c)\ (\mathbf{C}[\![\mathrm{C}_2]\!]e\ c)$$
$$\mathbf{C}[\![\mathbf{while}\ \mathrm{B}\ \mathbf{do}\ \mathrm{C}]\!] = \lambda e.\lambda c.\ \mathit{fix}(\lambda c'.\ \mathit{choose}(\mathbf{B}[\![\mathrm{B}]\!]e)\ (\mathbf{C}[\![\mathrm{C}]\!]e\ c')\ c)$$
where $\mathit{choose}: (\mathit{Store} \to \mathit{Tr}) \to \mathit{Cmdcont} \to \mathit{Cmdcont} \to \mathit{Cmdcont}$
$$\mathit{choose}\ b\ c_1\ c_2 = \lambda s.\ (b\ s) \to (c_1\ s)\ [\!]\ (c_2\ s)$$

The continuation argument c represents the ''remainder of the program'' in each of the clauses. The **while**-loop equation is now a least fixed point over the continuation argument rather than the store, for the loop problem is stated as ''how does the remainder of the program appear if the **while**-loop can reappear in it an unbounded number of times?'' Comparing this equation with the one defined using the control stack will make things clear.

Figure 9.1 shows a program and its continuation semantics. The nested continuation

Figure 9.1

Let $C_0 = C_1;C_2;\textbf{stop};C_1$
$C_1 = X{:=}1$
$C_2 = \textbf{while}\ X > 0\ \textbf{do}\ C_3$
$C_3 = X{:=}X-1$

$\mathbf{C}[\![C_0]\!]e_1\ \mathit{finish}\ s_1$

where $e_1 = (\mathit{updateenv}\ [\![X]\!]\ l_0\ e_0)$
and $\mathit{finish} = (\lambda s.\ s)$

$\mathbf{C}[\![C_1]\!]e_1\ (\mathbf{C}[\![C_2;\textbf{stop};C_1]\!]e_1\ \mathit{finish})\ s_1$

$(\lambda c.\lambda s.\ c(\mathit{update}(\mathit{accessenv}[\![X]\!]\ e_1)\ (\mathbf{E}[\![1]\!]e_1\ s)\ s))\ (\mathbf{C}[\![C_2;\textbf{stop};C_1]\!]e_1\ \mathit{finish})\ s_1$

$(\mathbf{C}[\![C_2]\!]e_1\ (\mathbf{C}[\![\textbf{stop};C_1]\!]e_1\ \mathit{finish}))\ (\mathit{update}\ l_0\ \mathit{one}\ s_1)$

s_2

$\mathit{fixF}\ s_2$

where $F = \lambda c'.\mathit{choose}\ (\mathbf{B}[\![X > 0]\!]e_1)\ (\mathbf{C}[\![C_3]\!]e_1\ c')\ c_1$
where $c_1 = \mathbf{C}[\![\textbf{stop};C_1]\!]e_1\ \mathit{finish}$

$(\mathbf{B}[\![X > 0]\!]e_1\ s_2) \rightarrow (\mathbf{C}[\![C_3]\!]e_1\ (\mathit{fix}\ F)\ s_2)\ [\!]\ (c_1\ s_2)$

$\mathbf{C}[\![C_3]\!]e_1\ (\mathit{fix}\ F)\ s_2$

$(\mathit{fix}\ F)\ (\mathit{update}\ l_0\ \mathit{zero}\ s_2)$

s_3

$(\mathbf{B}[\![X > 0]\!]e_1\ s_3) \rightarrow (\mathbf{C}[\![C_3]\!]e_1\ (\mathit{fix}\ F)\ s_3)\ [\!]\ (c_1\ s_3)$

$(c_1\ s_3)$

$\mathbf{C}[\![\textbf{stop}]\!]e_1\ (\mathbf{C}[\![C_1]\!]e_1\ \mathit{finish})\ s_3$

s_3

resembles a form of left-to-right function composition that can be overridden when an

extraordinary condition (e.g., a **stop** command) occurs. Herein lies the utility of continuations, for normal function composition could not be overidden in a direct semantics definition.

The abstractions in the semantic equations are all nonstrict. The continuations eliminate the need for strict abstractions on the store arguments. You can see the reason in the definition for the **while**-loop: the value of ($\mathbf{C}[\![$**while** B **do** C$]\!] e\, c\, s$) is undefined, iff all finite expansions of the loop map s to undefined, iff c is not applied to any store in any finite expansion. The remainder of the program (that is, c) is never reached when a ''loop forever'' situation is encountered. A semantic equation $\mathbf{C}[\![\mathrm{C}]\!] = \lambda c.\lambda s.\, c(\cdots s \cdots)$ defines a construct $[\![\mathrm{C}]\!]$ that is guaranteed to terminate, for c is applied to the updated store. An equation $\mathbf{C}[\![\mathrm{C}]\!] = \lambda c.\lambda s.\, (\cdots)\, c\, s$ defines a construct whose termination is not guaranteed, so both c and s are carried along for use by $(\cdots)$.

Since a continuation represents a program's complete computation upon a store, the continuation may contain some final ''cleaning up'' instructions that produce a final output. For example, the *finish* continuation used in Figure 9.1 might also be defined as $finish = (\lambda s.\, \textit{"done"})$, which would make all the command continuations into mappings from stores to character strings. The general form of the command continuation domain is:

$$c \in Cmdcont = Store \to Answer$$

where *Answer* can be the domain of stores, output buffers, messages, or whatever. This generalization makes continuations especially suitable for handling unusual outputs.

9.1.1 Other Levels of Continuations

The semantic equations defined in the previous section show that the command valuation function can be written in continuation style and coexist with other valuation functions written in the direct style. Nonetheless, let's consider representing the valuation function for expressions in the continuation style. Recall that $\mathbf{E}$: Expression $\to$ *Environment* $\to$ *Store* $\to$ *Expressible-value* is the functionality of the valuation function. In continuation form, expression evaluation breaks into explicit steps. In terms of the control stack analogy, an *expression continuation* resembles a stack of evaluation steps for computing the value of an expression. Expression continuations for some expressions will create intermediate values that must be saved along the way. This suggests:

$$k \in Exprcont = Expressible\text{-}value \to Store \to Answer'$$

The expressible value argument to an expression continuation is the intermediate value of the partially evaluated expression. The *Answer'* domain will be considered shortly.

The semantic equations for some of the expression constructs read as follows:

$\mathbf{E}$: Expression $\to$ *Environment* $\to$ *Exprcont* $\to$ *Store* $\to$ *Answer'*

$\mathbf{E}[\![\mathrm{E}_1 + \mathrm{E}_2]\!] = \lambda e.\lambda k.\, \mathbf{E}[\![\mathrm{E}_1]\!] e\, (\lambda n_1.\, \mathbf{E}[\![\mathrm{E}_2]\!] e\, (\lambda n_2.\, k(n_1\ plus\ n_2)))$

$\mathbf{E}[\![\mathrm{I}]\!] = \lambda e.\lambda k.\lambda s.\, k(access(accessenv[\![\mathrm{I}]\!]\, e)\, s)\, s$

$\mathbf{E}[\![\mathrm{N}]\!] = \lambda e.\lambda k.\, k(\mathbf{N}[\![\mathrm{N}]\!])$

Notice how the steps in the addition expression are spelled out by the nested continuation:

$[\![E_1]\!]$ evaluates and binds to n_1; $[\![E_2]\!]$ evaluates and binds to n_2; and k, the subsequent evaluation, carries on with (n_1 *plus* n_2).

How do the expression continuations integrate with the command continuations? The answer is tied to the structure of *Answer'*. If the ultimate answer of an expression is the value of the expression, that is, *Answer'* = *Expressible-value*, then two different levels of control result: expression level control and command level control. The interface between the two is a bit awkward:

$$\mathbf{C}[\![I{:=}E]\!] = \lambda e.\lambda c.\lambda s.\ c(\mathit{update}(\mathit{accessenv}[\![I]\!]\,e)\ (\mathbf{E}[\![E]\!]e\ \mathit{fin}\ s)\ s)$$

where $\mathit{fin} \in \mathit{Exprcont}$ is $\mathit{fin} = \lambda n.\lambda s.\ n$

If *Answer'* = *Answer*, then the two levels of control integrate nicely:

$$\mathbf{C}[\![I{:=}E]\!] = \lambda e.\lambda c.\lambda s.\ \mathbf{E}[\![E]\!]e\ (\lambda n.\lambda s'.\ c(\mathit{update}(\mathit{accessenv}[\![I]\!]\,e)\,n\,s'))\ s$$

Now *Exprcont* = *Expressible-value* → *Cmdcont*, which makes clear that the purpose of a series of expression evaluation steps is to produce a value and return back to the level of command control. In an implementation, the code for evaluating expressions exists on the same control stack as the code for evaluating commands.

In a similar fashion, continuations can be introduced into the other valuation functions of a language. Even the operations of the semantic algebras can be converted. As an example, a completely sequentialized version of assignment reads:

$$\mathbf{C}[\![I{:=}E]\!] = \lambda e.\lambda c.\ \mathit{accessenv}'[\![I]\!]\,e\ (\lambda l.\ \mathbf{E}[\![E]\!]e\ (\lambda n.\ \mathit{update}'\,l\,n\,c))$$

where

$\mathit{accessenv}'$: Identifier → *Environment* → (*Location* → *Cmdcont*) → *Cmdcont*

$\mathit{accessenv}' = \lambda i.\lambda e.\lambda m.\ m(e(i))$

and update' : *Location* → *Expressible-value* → *Cmdcont* → *Cmdcont*

$\mathit{update}' = \lambda l.\lambda n.\lambda c.\lambda s.\ c([\,l \mapsto n\,]s)$

An assignment statement determines its left-hand-side value first, then its right-hand-side value, and then the update.

Figure 9.5 shows a complete imperative language using command and expression continuations.

9.2 EXCEPTION MECHANISMS

We can use continuations to develop exception-handling mechanisms. An *exception handler* is a procedure that is invoked when an extraordinary situation occurs that cannot be handled by the usual order of evaluation. Control transfers to the handler procedure, which adjusts the state so that computation may resume. Exception handlers of various forms can be found in PL/1, ML, CLU, Ada, Scheme, and many other languages.

On the invocation of an exception handler, the continuation that owns the store surrenders the store to the handler, which is also a continuation. The handler repairs the store and

relinquishes control to a continuation representing some remainder of the computation. Figure 9.2 presents a simple version that meshes with the language in Section 9.1.

An exception is signaled by ⟦**raise** I⟧, which discards the existing continuation and extracts the continuation for ⟦I⟧ from the environment. The handler ⟦**on** I **do** C⟧ applies its body to the store and yields control to the continuation that represents the commands following the enclosing **begin-end** block.

It is disconcerting that the continuation *c* in the configuration (**C**⟦**raise** I⟧*e c s*) is discarded. It suggests that the complete plan of evaluation is abandoned when an exception handler is invoked. Actually, this is not true: the continuation assigned to the handler ''overlaps'' the one that was discarded; the commands following the current active block are evaluated as planned. You are left with the exercise of revising the definition so that this property is explicit.

9.3 BACKTRACKING MECHANISMS

A useful variant of exception handling is the ''undoing'' of evaluation steps back to a configuration that is ''safe.'' This version of exception handling is called *backtracking.* Backtracking is an integral feature of programming languages designed for heuristic problem-solving: if a problem solving strategy fails, a backtrack is taken to a configuration that allows an alternative strategy to be applied. These ''strategies'' are continuations.

We integrate a backtracking facility into a language by using a *failure continuation.* The continuation representing the usual evaluation sequence is called the *success continuation.* The failure continuation is invoked when backtracking is needed. Maintenance of the failure continuation is done by certain constructs in the language: exception-handler definitions, choice constructs, cut points, and so on. Figure 9.3 presents a programming language that resembles a propositional version of PROLOG.

The success continuation is built from the subgoals in the conjunctive construct ⟦C_1,C_2⟧. The failure continuation is updated by the choice construct ⟦C_1 **or** C_2⟧, which chooses the strategy indicated by goal ⟦C_1⟧ and saves alternative strategy ⟦C_2⟧ in the failure continuation; and by the break point construct ⟦**cut**⟧, which disallows backtracking past the point marked by the break point. The success continuation is applied when an endpoint ⟦**succeedwith** F⟧ is encountered, and the store is updated about the achievement. Similarly, the failure continuation is applied when a ⟦**fail**⟧ construct is encountered. The **or** construct saves the store in its failure continuation so that the updates done in an unsuccessful strategy are undone. This treatment of the store violates the usual ''sequentiality'' of store processing, and you are left with the exercise of finding an alternative semantics that is ''sequential.''

9.4 COROUTINE MECHANISMS

Section 9.3 generalized from using one continuation to two; now we generalize to a family of them. A system of continuations that activate one another can be used to design a *coroutine*

Figure 9.2

Abstract syntax:

D∈ Declaration
C∈ Command
I∈ Identifier

D ::= $D_1;D_2$ | ··· | **on** I **do** C
C ::= **begin** D;C **end** | ··· | **raise** I

Semantic algebras:

I. Program outputs
Domain $Answer = (Store + String)_\perp$

II. Command continuations
Domain $c \in Cmdcont = Store \to Answer$
Operations

$fin : Cmdcont$
$fin = \lambda s.\ \text{in}Store(s)$

$err : Cmdcont$
$err = \lambda s.\ \text{in}String(\text{"error"})$

III. Denotable values
Domain $Denotable\text{-}value = Cmdcont + Nat + \cdots$

IV. Environments
Domain $e \in Environment =$ Identifier $\to Denotable\text{-}value$
Operations (usual)

Valuation functions:

D: Declaration $\to Environment \to Cmdcont \to Environment$
$\mathbf{D}[\![D_1;D_2]\!] = \lambda e.\lambda c.\ \mathbf{D}[\![D_2]\!](\mathbf{D}[\![D_1]\!]e\ c)\ c$
$\mathbf{D}[\![\textbf{on}\ I\ \textbf{do}\ C]\!] = \lambda e.\lambda c.\ update[\![I]\!]\ \text{in}Cmdcont(\mathbf{C}[\![C]\!]e\ c)\ e$

C: Command $\to Environment \to Cmdcont \to Cmdcont$
$\mathbf{C}[\![\textbf{begin}\ D;C\ \textbf{end}]\!] = \lambda e.\lambda c.\ \mathbf{C}[\![C]\!]\ (\mathbf{D}[\![D]\!]e\ c)\ c$
$\mathbf{C}[\![\textbf{raise}\ I]\!] = \lambda e.\lambda c.$ cases $(accessenv[\![I]\!]\ e)$ of
$\quad \text{is}Cmdcont(c') \to c'$
$\quad [] \ \text{is}Nat(n) \to err$
$\quad [] \ \cdots$ end

Figure 9.3

Abstract syntax:

P∈ Program
D∈ Declaration
C∈ Command
I∈ Identifier
F∈ Primitive-operator

P ::= D. ?C

D ::= D_1.D_2 | I ← C

C ::= C_1,C_2 | C_1 **or** C_2 | I | **succeedwith** F | **fail** | **cut**

Semantic algebras:

I. Program outputs
Domain *Answer* = *Store* + *String*

II. Stores
Domain *s*∈ *Store*

Figure 9.3 (continued)

III. Continuations
Domain $c \in Cmdcont = Store \to Answer$
$fc \in Failure\text{-}cont = Cmdcont$
$sc \in Success\text{-}cont = Failure\text{-}cont \to Cmdcont$
Operations
succeeded : *Success-cont*
$succeeded = \lambda fc.\lambda s.\ \text{in}Store(s)$
failed : *Failure-cont*
$failed = \lambda s.\ \text{in}String(\text{"failure"})$

IV. Evaluation strategies
Domain $Strategy = Success\text{-}cont \to Failure\text{-}cont \to Cmdcont$

V. Environments
Domain $e \in Environment = \text{Identifier} \to Strategy$
Operations
emptyenv: *Environment*
$emptyenv = \lambda i.\ (\lambda sc.\lambda fc.\ fc)$
accessenv: Identifier $\to$ *Environment* $\to$ *Strategy* (usual)
updateenv: Identifier $\to$ *Strategy* $\to$ *Environment* $\to$ *Environment* (usual)

Valuation functions:

P: Program $\to$ *Cmdcont*
$\mathbf{P}[\![\text{D. ?C}]\!] = \mathbf{C}[\![\text{C}]\!]\ (\mathbf{D}[\![\text{D}]\!]\ emptyenv)\ succeeded\ failed$

D: Declaration $\to$ *Environment* $\to$ *Environment*
$\mathbf{D}[\![\text{D}_1\text{.D}_2]\!] = \mathbf{D}[\![\text{D}_2]\!] \circ \mathbf{D}[\![\text{D}_1]\!]$
$\mathbf{D}[\![\text{I} \leftarrow \text{C}]\!] = \lambda e.\ updateenv[\![\text{I}]\!]\ (\mathbf{C}[\![\text{C}]\!]e)\ e$

C: Command $\to$ *Environment* $\to$ *Strategy*
$\mathbf{C}[\![\text{C}_1\text{,C}_2]\!] = \lambda e.\lambda sc.\ \mathbf{C}[\![\text{C}_1]\!]e\ (\mathbf{C}[\![\text{C}_2]\!]e\ sc)$
$\mathbf{C}[\![\text{C}_1\ \textbf{or}\ \text{C}_2]\!] = \lambda e.\lambda sc.\lambda fc.\lambda s.\ \mathbf{C}[\![\text{C}_1]\!]e\ sc\ (\lambda s_1.\ \mathbf{C}[\![\text{C}_2]\!]e\ sc\ fc\ s)\ s$
$\mathbf{C}[\![\text{I}]\!] = accessenv\ [\![\text{I}]\!]$
$\mathbf{C}[\![\textbf{succeedwith}\ \text{F}]\!] = \lambda e.\lambda sc.\lambda fc.\ \lambda s.\ sc\ fc\ (\mathbf{F}[\![\text{F}]\!]s)$
$\mathbf{C}[\![\textbf{fail}]\!] = \lambda e.\lambda sc.\lambda fc.\ fc$
$\mathbf{C}[\![\textbf{cut}]\!] = \lambda e.\lambda sc.\lambda fc.\ sc\ failed$

F: Primitive-operator $\to$ *Store* $\to$ *Store* (omitted)

system. Unlike a subroutine, a coroutine need not complete all the steps in its continuation

before relinquishing control to another. A program configuration carries along a collection of partially completed continuations, representing the coroutines in the system. Let us call this collection a *coroutine environment.* When a coroutine is invoked, the current active continuation is saved in the coroutine environment. The invoked continuation is selected from the coroutine environment and placed in control of the configuration.

A language supporting coroutines is presented in Figure 9.4.

In addition to the usual command continuation domain and the newly introduced coroutine environment domain, a domain of coroutine continuations is needed to handle the coroutine environment. The *resume* operation invokes a coroutine by storing the current coroutine continuation and extracting the invoked one. The identifier carried in the coroutine environment is set to the name of the coroutine now in control of the configuration.

9.5 UNRESTRICTED BRANCHING MECHANISMS

We can generalize the coroutine mechanism so that it does not save the continuation of the calling coroutine when another coroutine is invoked. This creates the form of branching known as the **goto.** Without the promise to resume a coroutine at its point of release, the domain of coroutine environments becomes unnecessary, and the coroutine continuation domain becomes the command continuation domain. From here on, we speak not of coroutines, but of labeled commands; the ⟦**resume** I⟧ command is now ⟦**goto** I⟧.

The continuation associated with a label is kept in the usual environment, which is a static object (unlike the coroutine environment), because the command continuation associated with a label is determined by the label's textual position in the program. We handle a branch by placing the continuation associated with the destination label in control: $\mathbf{C}[\![\textbf{goto}\ \mathrm{I}]\!] = \lambda e.\lambda c.\ accessenv[\![\mathrm{I}]\!]\ e$.

Figure 9.5 presents a definition for a language with unrestricted branches.

So far we have ignored mutual recursion in invocations, but we must now confront the issue if backwards branches are to be allowed. What does a branch continuation look like? Since continuations model the remainder of a program, a continuation for a label ⟦I⟧ must not only contain the denotation for the one command labeled by ⟦I⟧, but the denotation of the remainder of the program that follows ⟦I⟧: if ⟦I⟧ labels command $[\![\mathrm{C}_i]\!]$, the continuation c_i associated with ⟦I⟧ is $(\mathbf{C}[\![\mathrm{C}_i]\!]e\ c_{i+1})$, where c_{i+1} is the continuation for the commands that follow $[\![\mathrm{C}_i]\!]$.

Now consider a block with n distinct labels: ⟦**begin** D; I_1:C_1; I_2:C_2; $\cdots$ I_n:C_n **end**⟧. The continuations are:

$$
\begin{aligned}
c_1 &= (\mathbf{C}[\![\mathrm{C}_1]\!]e\prime\ c_2)\\
c_2 &= (\mathbf{C}[\![\mathrm{C}_2]\!]e\prime\ c_3)\\
&\cdots\\
c_{n-1} &= (\mathbf{C}[\![\mathrm{C}_{n-1}]\!]e\prime\ c_n)\\
c_n &= (\mathbf{C}[\![\mathrm{C}_n]\!]e\prime\ c)\\
&\quad \text{where } e\prime = (updateenv[\![\mathrm{I}_1]\!]\ \text{in}Cmdcont(c_1)\\
&\qquad\qquad (updateenv[\![\mathrm{I}_2]\!]\ \text{in}Cmdcont(c_2)
\end{aligned}
$$

Figure 9.4

Abstract syntax:

B∈ Block
D∈ Declaration
C∈ Command
I∈ Identifier

B ::= D; **initiate** I
D ::= D_1;D_2 | **coroutine** I=C
C ::= C_1;C_2 | **resume** I | I:=E

Semantic algebras:

I. Command continuations
Domain $Cmdcont = Store \to Answer_\perp$

II. Coroutine continuations and the environments holding them
Domains $c \in Coroutine\text{-}cont = Coroutine\text{-}env \to Cmdcont$
$e \in Coroutine\text{-}env = ((\text{Identifier} \to Coroutine\text{-}cont) \times \text{Identifier})_\perp$
Operations
quit: *Coroutine-cont*
err: *Coroutine-cont*
empty-env: Identifier→ *Coroutine-env*
$empty\text{-}env = \lambda i.\ ((\lambda i'.\ err),\ i)$
initialize: Identifier → *Coroutine-cont* → *Coroutine-env* → *Coroutine-env*
$initialize = \lambda i.\lambda c.\lambda e.$ let $(map, caller) = e$ in $([\, i \mapsto c\,]map, caller)$
resume: Identifier → *Coroutine-cont* → *Coroutine-env* → *Cmdcont*
$resume = \lambda i.\lambda c.\lambda e.$ let $(map, caller) = e$ in
let $map' = [\, caller \mapsto c\,]map$
in $(map'\ i)\ (map', i)$

Valuation functions:

B: Block → *Cmdcont*
$\mathbf{B}[\![\text{D; }\mathbf{initiate}\ \text{I}]\!] = resume[\![\text{I}]\!]\ \ quit\ \ (\mathbf{D}[\![\text{D}]\!](empty\text{-}env[\![\text{I}]\!]))$

D: Declaration → *Coroutine-env* → *Coroutine-env*
$\mathbf{D}[\![D_1;D_2]\!] = \mathbf{D}[\![D_2]\!] \circ \mathbf{D}[\![D_1]\!]$
$\mathbf{D}[\![\mathbf{coroutine}\ \text{I=C}]\!] = initialize[\![\text{I}]\!]\ (\mathbf{C}[\![\text{C}]\!]\ quit)$

C: Command → *Coroutine-cont* → *Coroutine-env* → *Cmdcont*
$\mathbf{C}[\![C_1;C_2]\!] = \mathbf{C}[\![C_1]\!] \circ \mathbf{C}[\![C_2]\!]$
$\mathbf{C}[\![\mathbf{resume}\ \text{I}]\!] = resume[\![\text{I}]\!]$
$\mathbf{C}[\![\text{I:=E}]\!] = \lambda c.\lambda e.\lambda s.\ c\ e\ (update[\![\text{I}]\!]\ (\mathbf{E}[\![\text{E}]\!]s)\ s)$

Figure 9.5

Abstract syntax:

P∈ Program
B∈ Block
D∈ Declaration
C∈ Command
E∈ Expression
I∈ Identifier
N∈ Numeral

P ::= B.

B ::= **begin** D; I_1:C_1; I_2:C_2; · · · ; I_n:C_n **end**

D ::= D_1;D_2 | **const** I=N | **var** I

C ::= C_1;C_2 | I:=E | **if** E **then** C_1 **else** C_2 | **while** E **do** C | B | **goto** I

E ::= E_1+E_2 | I | N | **do** C **resultis** E | (E)

Semantic algebras:

I.-V. Natural numbers, truth values, locations, identifiers, and character strings
(as usual)

VI. Semantic outputs
Domain $a \in Answer = (OK + Err)_\perp$
where $OK = Store$ and $Err = String$

VII.-IX. Expressible, denotable, and storable values
Domains $n \in Exprval = Storable\text{-}value = Nat$
$d \in Denotable\text{-}value = Nat + Location + Cmdcont + Errvalue$
where $Errvalue = Unit$

X. Environments
Domain $e \in Environment = (\text{Identifier} \to Denotable\text{-}value) \times Location$
Operations
(defined in Figure 7.1)

XI. Stores
Domain $s \in Store = Location \to Storable\text{-}value$
Operations
(defined in Figure 7.1)

XII. Command continuations
Domain $c \in Cmdcont = Store \to Answer$

Figure 9.5 (continued)

Operations

finish : *Cmdcont*
finish = λ*s*. in*OK*(*s*)

error : *String* → *Cmdcont*
error = λ*t*.λ*s*. in*Err*(*t*)

XIII. Expression continuations

Domain $k \in Exprcont = Exprval \to Cmdcont$

Operations

return-value : *Exprval* → *Exprcont* → *Cmdcont*
return-value = λ*n*.λ*k*. *k*(*n*)

save-arg = (*Exprcont* → *Cmdcont*) → (*Exprval* → *Exprcont*) → *Exprcont*
save-arg = λ*f*.λ*g*.λ*n*. *f*(*g n*)

add : *Exprcont* → *Exprval* → *Exprval* → *Cmdcont*
add = λ*k*.λn_1.λn_2. *k*(n_1 *plus* n_2)

fetch : *Location* → *Exprcont* → *Cmdcont*
fetch = λ*l*.λ*k*.λ*s*. *k*(*access l s*) *s*

assign : *Location* → *Cmdcont* → *Exprcont*
assign = λ*l*.λ*c*.λ*n*.λ*s*. *c*(*update l n s*)

choose : *Cmdcont* → *Cmdcont* → *Exprcont*
choose = λc_1.λc_2.λ*n*. *n greaterthan zero* → c_1 [] c_2

Valuation functions:

P: Program → *Location* → *Cmdcont*
P⟦B.⟧ = λ*l*. **B**⟦B⟧ (*emptyenv l*) *finish*

B: Block → *Environment* → *Cmdcont* → *Cmdcont*

$$\begin{aligned}
&\mathbf{B}[\![\textbf{begin}\ \mathrm{D};\ \mathrm{I}_1{:}\mathrm{C}_1;\ \mathrm{I}_2{:}\mathrm{C}_2;\ \cdots;\ \mathrm{I}_n{:}\mathrm{C}_n\ \textbf{end}]\!] = \\
&\quad \lambda e.\lambda c.\ (\mathit{fix}(\lambda \mathit{ctuple}.\ (\ (\mathbf{C}[\![\mathrm{C}_1]\!] e' (\mathit{ctuple}{\downarrow}2)), \\
&\qquad\qquad (\mathbf{C}[\![\mathrm{C}_2]\!] e' (\mathit{ctuple}{\downarrow}3)), \\
&\qquad\qquad \cdots, \\
&\qquad\qquad (\mathbf{C}[\![\mathrm{C}_n]\!] e' c)\ )\ ))\!\downarrow\! 1
\end{aligned}$$

where $e' = (\mathit{updateenv}[\![\mathrm{I}_1]\!]\ \mathrm{in}\mathit{Cmdcont}(\mathit{ctuple}{\downarrow}1)$

$$\begin{aligned}
&\qquad (\mathit{updateenv}[\![\mathrm{I}_2]\!]\ \mathrm{in}\mathit{Cmdcont}(\mathit{ctuple}{\downarrow}2) \\
&\qquad\quad \cdots \\
&\qquad\qquad (\mathit{updateenv}[\![\mathrm{I}_n]\!]\ \mathrm{in}\mathit{Cmdcont}(\mathit{ctuple}{\downarrow}n)\ (\mathbf{D}[\![\mathrm{D}]\!]e)) \cdots))
\end{aligned}$$

Figure 9.5 (continued)

D: Declaration → *Environment* → *Environment*
(defined in Figure 7.2)

C: Command → *Environment* → *Cmdcont* → *Cmdcont*

C⟦C_1;C_2⟧ = λ*e*. **C**⟦C_1⟧*e* ∘ **C**⟦C_2⟧*e*

C⟦I:=E⟧ = λ*e*.λ*c*. cases (*accessenv*⟦I⟧ *e*) of
is*Nat*(*n*) → *error* "const used on lhs"
[] is*Location*(*l*) → **E**⟦E⟧*e* (*assign l c*)
[] is*Cmdcont*(*c*) → *error* "label used on lhs"
[] is*Errvalue*() → *error* "lhs undeclared"
end

C⟦**if** E **then** C_1 **else** C_2⟧ = λ*e*.λ*c*. **E**⟦E⟧*e* (*choose* (**C**⟦C_1⟧*e c*) (**C**⟦C_2⟧*e c*))

C⟦**while** E **do** C⟧ = λ*e*.λ*c*. *fix*(λc_1. **E**⟦E⟧*e* (*choose* (**C**⟦C⟧*e* c_1) *c*))

C⟦B⟧ = **B**⟦B⟧

C⟦**goto** I⟧ = λ*e*.λ*c*. cases (*accessenv*⟦I⟧ *e*) of
is*Nat*(*n*) → *error* "const used as label"
[] is*Location*(*l*) → *error* "var used as label"
[] is*Cmdcont*(c_1) → c_1
[] is*Errvalue*() → *error* "unknown id"
end

E: Expression → *Environment* → *Exprcont* → *Cmdcont*

E⟦E_1+E_2⟧ = λ*e*.λ*k*. **E**⟦E_1⟧*e* (*save-arg* (**E**⟦E_2⟧*e*) (*add k*))

E⟦I⟧ = λ*e*.λ*k*. cases (*accessenv*⟦I⟧ *e*) of
is*Nat*(*n*) → *return-value n k*
[] is*Location*(*l*) → *fetch l k*
[] is*Cmdcont*(*c*) → *error* "label used in expr"
[] is*Errvalue*() → *error* "undeclared iden"
end

E⟦N⟧ = λ*e*.λ*k*. *return-value* (**N**⟦N⟧) *k*

E⟦**do** C **resultis** E⟧ = λ*e*.λ*k*. **C**⟦C⟧*e* (**E**⟦E⟧*e k*)

E⟦(E)⟧ = **E**⟦E⟧

N: Numeral → *Nat* (omitted)

. . .

(*updateenv*⟦I_n⟧ in*Cmdcont*(c_n) (**D**⟦D⟧*e*)) · · ·))

Each c_i possesses the environment that contains the denotations of all the labels. But to define the environment, each c_i must be defined. The mutual recursion is resolved by the *fix* operation. The least fixed point is an *n*-tuple of continuations, one for each label. The denotation of the entire block is the continuation associated with the first label.

You are encouraged to construct example programs and determine their denotations. A program for computing the factorial function is given a denotation in Figure 9.6. (Assume that semantic equations **E**⟦E_1∗E_2⟧ and **E**⟦E_1–E_2⟧ for multiplication and subtraction are added to

Figure 9.6

P⟦**begin const** A=*a*; **var** X; **var** TOTAL; **var** FAC;
L1: X:=A; TOTAL:=1; **goto** L2;
L2: **while** X **do** (TOTAL:=TOTAL∗X; X:=X-1);
L3: FAC:=TOTAL
end.⟧ =

```
λl. (fix(λctuple.
    ( return-value a
        ( assign l
          ( return-value one
            ( assign next-locn(l)
              ( ctuple↓2) )))),
      fix(λc'. ( fetch l
          ( choose
              ( fetch next-locn(l)
                (save-arg (fetch l)
                  ( mult
                    ( assign next-locn(l)
                      ( fetch l
                        ( save-arg (return-value one)
                          ( sub
                            ( assign l
                              c' ))))))))
              (ctuple↓3) ))),
      ( fetch next-locn(l)
          ( assign (next-locn(next-locn(l)))
            finish ))
    ) ) ↓1
```

the language. The equations have the same format as the one for addition, using operations *mult* and *sub* respectively instead of *add*.) The denotation in Figure 9.6 is simplified to the stage where all abstract syntax pieces and environment arguments have been written away. The denotation of the factorial program is the first component of the least fixed point of a functional; the functional maps a triple of command continuations to a triple of command continuations. Examining the functional's body, we see that component number *i* of the triple is the denotation of the commands labeled by identifier *Li* in the program. A jump to label *Lk* has the denotation (*ctuple*$\downarrow k$). Each component of the tuple is a deeply nested continuation whose actions upon the store can be read from left to right. The actions are low level and resemble conventional assembly language instructions. This feature is exploited in the next chapter.

9.6 THE RELATIONSHIP BETWEEN DIRECT AND CONTINUATION SEMANTICS

A question of interest is the exact relationship between a language's direct semantics definition and its continuation semantics definition. We would like to prove that the two definitions map the same program to equal denotations. But, since the internal structures of the semantics definitions are so dissimilar, the proof can be quite difficult. A related question is whether or not we can derive a continuation semantics definition for a language from its direct semantics definition. Both questions are studied in the following example.

Let $\mathbf{C}_D$: Command $\rightarrow$ *Store*$_\perp \rightarrow$ *Store*$_\perp$ be a valuation function written in direct semantics style. Say that we desire a valuation function $\mathbf{C}_C$: Command $\rightarrow$ *Cmdcont* $\rightarrow$ *Cmdcont*, *Cmdcont* = *Store*$_\perp \rightarrow$ *Store*$_\perp$, in continuation style such that $\mathbf{C}_D$ is equivalent to $\mathbf{C}_C$. The equivalence can be stated as $\mathbf{C}_D[\![\mathrm{C}]\!] = \mathbf{C}_C[\![\mathrm{C}]\!](\lambda s.\, s)$. But this property will be difficult to prove by induction. Recall that in Section 9.1 we saw the format $\mathbf{C}_C[\![\mathrm{C}]\!] = \lambda c.\lambda s.\, c(f(s))$ for a terminating command $[\![\mathrm{C}]\!]$ that performs *f* to the store. Since $\mathbf{C}_D[\![\mathrm{C}]\!]$ describes $[\![\mathrm{C}]\!]$'s actions in isolation, the equality:

$$\mathbf{C}_C[\![\mathrm{C}]\!]c\,s = c(\mathbf{C}_D[\![\mathrm{C}]\!]s)$$

holds. This equality generalizes the earlier definition of equivalence; it is called a *congruence*, and two definitions that satisfy a congruence are called *congruent*.

What about a command that might not terminate? Then $\mathbf{C}_D[\![\mathrm{C}]\!]s$ might be $\perp$, so what should $\mathbf{C}_C[\![\mathrm{C}]\!]c\,s$ be? If we require *strict* continuations, then the congruence still holds: $\mathbf{C}_C[\![\mathrm{C}]\!]c\,s = c(\mathbf{C}_D[\![\mathrm{C}]\!]s) = c(\perp) = \perp$. Sethi and Tang (1980) suggest that a continuation semantics for a simple language be derived from its direct semantics definition by *defining* $\mathbf{C}_C[\![\mathrm{C}]\!]c\,s = c(\mathbf{C}_D[\![\mathrm{C}]\!]s)$ for each construct $[\![\mathrm{C}]\!]$ in the language. We then apply an inductive hypothesis to simplify $c(\mathbf{C}_D[\![\mathrm{C}]\!]s)$ into a more satisfactory form. Some derivations follow.

First, for $\mathbf{C}_D[\![\mathrm{C}_1;\mathrm{C}_2]\!] = \underline{\lambda} s.\, \mathbf{C}_D[\![\mathrm{C}_2]\!](\mathbf{C}_D[\![\mathrm{C}_1]\!]s)$, define:

$$\mathbf{C}_C[\![\mathrm{C}_1;\mathrm{C}_2]\!] = \lambda c.\underline{\lambda} s.\, c(\mathbf{C}_D[\![\mathrm{C}_1;\mathrm{C}_2]\!]s)$$
$$= \lambda c.\underline{\lambda} s.\, c(\mathbf{C}_D[\![\mathrm{C}_2]\!](\mathbf{C}_D[\![\mathrm{C}_1]\!]s))$$

Since $\mathbf{C}_D[\![\mathrm{C}_1]\!]s \in$ *Store*$_\perp$, we can use the inductive hypothesis that $\mathbf{C}_C[\![\mathrm{C}_2]\!]c\,s = c(\mathbf{C}_D[\![\mathrm{C}_2]\!]s)$

to obtain the value $\lambda c.\underline{\lambda} s.\ \mathbf{C}_C[\![\mathrm{C}_2]\!]c\,(\mathbf{C}_D[\![\mathrm{C}_1]\!]s)$. But $\mathbf{C}_C[\![\mathrm{C}_2]\!]c$ is a command continuation, so we apply the inductive hypothesis for $[\![\mathrm{C}_1]\!]$ to obtain $\lambda c.\underline{\lambda} s.\ \mathbf{C}_C[\![\mathrm{C}_1]\!]\,(\mathbf{C}_C[\![\mathrm{C}_2]\!]c)\,s$. By extensionality and the inductive hypothesis that $\mathbf{C}_C[\![\mathrm{C}_2]\!]$ is strict, we obtain:

$$\mathbf{C}_C[\![\mathrm{C}_1;\mathrm{C}_2]\!] = \mathbf{C}_C[\![\mathrm{C}_1]\!] \circ \mathbf{C}_C[\![\mathrm{C}_2]\!]$$

Second, for $\mathbf{C}_D[\![\textbf{if}\ \mathrm{B}\ \textbf{then}\ \mathrm{C}_1\ \textbf{else}\ \mathrm{C}_2]\!] = \underline{\lambda} s.\ \mathbf{B}[\![\mathrm{B}]\!]s \to \mathbf{C}_D[\![\mathrm{C}_1]\!]s \,[]\, \mathbf{C}_D[\![\mathrm{C}_2]\!]s$, we define:

$$\begin{aligned}\mathbf{C}_C[\![\textbf{if}\ \mathrm{B}\ \textbf{then}\ \mathrm{C}_1\ \textbf{else}\ \mathrm{C}_2]\!] &= \lambda c.\underline{\lambda} s.\ c(\mathbf{C}_D[\![\textbf{if}\ \mathrm{B}\ \textbf{then}\ \mathrm{C}_1\ \textbf{else}\ \mathrm{C}_2]\!]s)\\ &= \lambda c.\underline{\lambda} s.\ c(\mathbf{B}[\![\mathrm{B}]\!]s \to \mathbf{C}_D[\![\mathrm{C}_1]\!]s \,[]\, \mathbf{C}_D[\![\mathrm{C}_2]\!]s)\end{aligned}$$

$\mathbf{B}[\![\mathrm{B}]\!]s$ is a defined truth value, so a cases analysis gives the value $\lambda c.\underline{\lambda} s.\ \mathbf{B}[\![\mathrm{B}]\!]s \to c(\mathbf{C}_D[\![\mathrm{C}_1]\!]s) \,[]\, c(\mathbf{C}_D[\![\mathrm{C}_2]\!]s)$. By the inductive hypothesis for $[\![\mathrm{C}_1]\!]$ and $[\![\mathrm{C}_2]\!]$, we obtain:

$$\mathbf{C}_C[\![\textbf{if}\ \mathrm{B}\ \textbf{then}\ \mathrm{C}_1\ \textbf{else}\ \mathrm{C}_2]\!] = \lambda c.\underline{\lambda} s.\ \mathbf{B}[\![\mathrm{B}]\!]s \to \mathbf{C}_C[\![\mathrm{C}_1]\!]c\,s \,[]\, \mathbf{C}_C[\![\mathrm{C}_2]\!]c\,s$$

Deriving the continuation semantics for a **while**-loop is a bit more involved. For $\mathbf{C}_D[\![\textbf{while}\ \mathrm{B}\ \textbf{do}\ \mathrm{C}]\!] = \mathit{fix}(\lambda f.\underline{\lambda} s.\ \mathbf{B}[\![\mathrm{B}]\!]s \to f(\mathbf{C}_D[\![\mathrm{C}]\!]s) \,[]\, s)$, we define:

$$\mathbf{C}_C[\![\textbf{while}\ \mathrm{B}\ \textbf{do}\ \mathrm{C}]\!] = \lambda c.\underline{\lambda} s.\ c(\mathbf{C}_D[\![\textbf{while}\ \mathrm{B}\ \textbf{do}\ \mathrm{C}]\!]s)$$

The presence of *fix* thwarts our attempts at simplification. Let:

$$F_D = \lambda f.\underline{\lambda} s.\ \mathbf{B}[\![\mathrm{B}]\!]s \to f(\mathbf{C}_D[\![\mathrm{C}]\!]s) \,[]\, s$$

We hope to find a corresponding functional F_C such that $(\mathit{fix}\,F_C)c\,s = c((\mathit{fix}\,F_D)s)$. Even though we haven't a clue as to what F_C should be, we begin constructing a fixed point induction proof of this equality and derive F_C as we go. The admissible predicate we use is:

$$P(f_C, f_D) = \text{``for all } c \in \mathit{Cmdcont} \text{ and } s \in \mathit{Store}_\perp,\ (f_C\,c\,s) = c(f_D\,s)\text{''}$$

where $f_C : \mathit{Cmdcont} \to \mathit{Cmdcont}$ and $f_D : \mathit{Store}_\perp \to \mathit{Store}_\perp$.

For the basis step, we have $f_C = (\lambda c.\underline{\lambda} s.\perp)$ and $f_D = (\underline{\lambda} s.\ \perp)$; the proof follows immediately from strictness. For the inductive step, the inductive hypothesis is $(f_C\,c\,s) = c(f_D\,s)$; we wish to prove that $(F_C\,f_C)\,c\,s = c((F_D\,f_D)s)$, deriving F_C in the process. We derive:

$$\begin{aligned}&c((F_D\,f_D)s)\\ &= c((\underline{\lambda} s.\ \mathbf{B}[\![\mathrm{B}]\!]s \to f_D(\mathbf{C}_D[\![\mathrm{C}]\!]s) \,[]\, s)s)\\ &= c(\mathbf{B}[\![\mathrm{B}]\!]s \to f_D(\mathbf{C}_D[\![\mathrm{C}]\!]s) \,[]\, s)\end{aligned}$$

when s is proper. This equals:

$$\begin{aligned}&\mathbf{B}[\![\mathrm{B}]\!]s \to c(f_D(\mathbf{C}_D[\![\mathrm{C}]\!]s)) \,[]\, (c\,s),\ \text{by distributing } c \text{ across the conditional}\\ &= \mathbf{B}[\![\mathrm{B}]\!]s \to (f_C\,c)(\mathbf{C}_D[\![\mathrm{C}]\!]s) \,[]\, (c\,s),\ \text{by the inductive hypothesis}\\ &= \mathbf{B}[\![\mathrm{B}]\!]s \to \mathbf{C}_C[\![\mathrm{C}]\!](f_C\,c)s \,[]\, (c\,s)\end{aligned}$$

by the structural induction hypothesis on $\mathbf{C}_D[\![\mathrm{C}]\!]$, because $(f_C\,c) \in \mathit{Cmdcont}$. If we let:

$$F_C = \lambda g.\lambda c.\underline{\lambda} s.\ \mathbf{B}[\![\mathrm{B}]\!]s \to \mathbf{C}_C[\![\mathrm{C}]\!](g\,c)\,s \,[]\, (c\,s)$$

then we are finished:

$$\mathbf{C}_C[\![\textbf{while}\ \mathrm{B}\ \textbf{do}\ \mathrm{C}]\!] = \mathit{fix}(\lambda g.\lambda c.\underline{\lambda} s.\ \mathbf{B}[\![\mathrm{B}]\!]s \rightarrow \mathbf{C}_C[\![\mathrm{C}]\!](g\,c)\,s\,[]\ (c\,s))$$

The definition of **while** used in Figure 9.5 can be proved equal to this one with another fixed point induction proof.

The continuation semantics $\mathbf{C}_C$ is congruent to $\mathbf{C}_D$ because it was defined directly from the congruence predicate. The proof of congruence is just the derivation steps read backwards. Few congruences between semantic definitions are as easy to prove as the one given here. Milne and Strachey (1976), Reynolds (1974b), and Stoy (1981) give examples of nontrivial semantics definitions and proofs of congruence.

SUGGESTED READINGS

Continuations: Abdali 1975; Jensen 1978; Mazurkiewicz 1971; Milne & Strachey 1976; Strachey & Wadsworth 1974; Stoy 1977

Control mechanisms: Bjørner & Jones 1982; Friedman et al. 1984; Jones 1982b; Reynolds 1972; Strachey & Wadsworth 1974

Congruences between definitions: Meyer & Wand 1985; Morris 1973; Milne & Strachey 1976; Royer 1985; Reynolds 1974b; Sethi & Tang 1980; Stoy 1981

EXERCISES

1. Add to the syntax of the language in Figure 7.2 the command **exitblock**, which causes a forward branch to the end of the current block.

   a. Without using continuations, integrate the **exitblock** construct into the semantics with as little fuss as possible. (Hint: adjust the *Poststore* domain and *check* operation.)
   b. Rewrite the semantics in continuation style and handle **exitblock** by discarding the current continuation and replacing it by another.
   c. Repeat parts a and b for an **exitloop** command that causes a branch out of the innermost loop; for a **jump** *L* command that causes a forward branch to a command labeled by identifier *L*.

2. Convert the operations in the *Nat*, *Environment*, and *Store* algebras of Figure 7.1 into continuation style.

   a. Modify the *access* operation so that an access of an uninitialized storage cell leads directly to an error message.
   b. Introduce a division operation that handles division by zero with an error message.
   c. Rewrite the semantic equations in Figure 7.2 to use the new algebras.

3. A language's control features can be determined from the continuation domains that it

uses.

a. Propose the forms of branching mechanisms that will likely appear when a semantic definition uses each of the following domains:

   i. $Declaration\text{-}cont = Environment \to Cmdcont$
   ii. $Denotable\text{-}value\text{-}cont = Denotable\text{-}value \to Cmdcont$
   iii. $Nat\text{-}cont = Nat \to Cmdcont$
   iv. $Location\text{-}cont = Location \to Exprcont$

b. A reasonable functionality for the continuation version of natural number addition is $add : Nat \to Nat\text{-}cont$. For each of parts i through iv, propose operations that use the continuation domain defined.

4. Newcomers to the continuation semantics method often remark that the denotation of a program appears to be ''built backwards.''

   a. How does this idea relate to the loading of a control stack prior to interpretation? To the compilation of a program?
   b. Notice that the semantic equations of Figure 9.5 do not mention any *Store*-valued objects. Consider replacing the *Cmdcont* algebra by a version of the *Store* algebra; formulate operations for the domain $Cmdcont = Store$. Do programs in the new semantics ''compute backwards''?
   c. Jensen (1978) noted a strong resemblance between continuation style semantics and weakest precondition semantics (Dijkstra 1976). Let Pred be the syntax domain of predicate calculus expressions. The symbols ''B'' and ''*p*'' stand for elements of Pred. Here is the weakest precondition semantics of a small language:

$$\begin{aligned}
&wp([\![C_1;C_2]\!], p) = wp([\![C_1]\!], wp([\![C_2]\!], p))\\
&wp([\![I{:=}E]\!], p) = [E/I]p\\
&wp([\![\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2]\!], p) = ([\![B]\!] \textit{ and } wp([\![C_1]\!], p))\\
&\quad \textit{or } ((\textit{not}[\![B]\!]) \textit{ and } wp([\![C_2]\!], p))\\
&wp([\![\textbf{while } B \textbf{ do } C]\!], p) = (\text{there exists } i \geq 0 \text{ such that } H_i(p))\\
&\quad \text{where } H_0(p) = (\textit{not}([\![B]\!]) \textit{ and } p)\\
&\quad \text{and } H_{i+1}(p) = wp([\![\textbf{if } B \textbf{ then } C \textbf{ else skip}]\!], H_i(p))\\
&\quad \text{and } wp([\![\textbf{skip}]\!], p) = p
\end{aligned}$$

   Now consider the continuation semantics of the language. In particular, let $\mathbf{P} : \text{Pred} \to Predicate$ be the valuation function for predicates, where $Predicate = Store \to Tr_1$ and $Tr_1 = Unit_\perp$. Let $true : Tr_1$ be () and $false : Tr_1$ be $\perp$. Define $Cmdcont = Predicate$. Using the semantic equations in Section 9.1, show that $\mathbf{C}[\![C]\!]p = \mathbf{P}(wp([\![C]\!], p))$.

5. Rework the semantics of Figure 9.5 so that a distinction is made between compile-time errors and run-time computations. In particular, create the following domains:

$$\begin{aligned}
&Pgmcont = Compile\text{-}err + (Location \to Computation)\\
&Cmdcont = Compile\text{-}err + Computation
\end{aligned}$$

$$Exprcont = Compile\text{-}err + (Expressible\text{-}value \rightarrow Computation)$$
$$Compile\text{-}err = String$$
$$Computation = Store \rightarrow Answer$$
$$Answer = (Store + Run\text{-}err)_{\perp}$$
$$Run\text{-}err = String$$

Formulate the semantics so that a denotable or expressible value error in a program ⟦P⟧ implies that $\mathbf{P}[\![P]\!] = \text{in}Compile\text{-}err(t)$, for some message *t*, and a type-correct program has denotation $\mathbf{P}[\![P]\!] = \text{in}(Location \rightarrow Computation)(f)$.

6. Design an imperative language that establishes control at the expression level but not at the command level. That is, the **E** valuation function uses expression continuations, but the **C** valuation function is in direct semantics style. What pragmatic advantages and disadvantages do you see?

7. Apply the principles of abstraction, parameterization, correspondence, and qualification to the language in Figure 9.5.

8. PL/1 supports exception handlers that are invoked by machine level faults. For example, a user can code the handler ⟦**on zerodivide do** C⟧, which is raised automatically when a division by zero occurs.

   a. Add the zero division exception handler to the language defined by Figures 9.2 and 9.5.
   b. The user can disable an exception handler by the command ⟦**no** I⟧, where ⟦I⟧ is the name of an exception handler, either built in or user defined. Add this feature to the language.

9. In ML, exception handlers are dynamically scoped. Revise the definition in Figure 9.2 to use dynamic scoping of handlers. How does this affect the raising of exceptions and exits from blocks? (Consider exceptions raised from within invoked procedures.)

10. One form of coroutine structure places a hierarchy on the coroutines; a coroutine can ''own'' other coroutines. Call these the *parent* and *child* coroutines, respectively. Child coroutines are declared local to the parent, and only a parent can call a child. A child coroutine can pause and return control to its parent but can-not resume its siblings or other nonrelated coroutines. Design a language with hierarchical coroutines.

11. Modify the semantics of the backtracking language in Figure 9.3 so that the commands can recursively invoke one another.

12. Extend the list processing language in Figure 7.5 to allow jumps in expression evaluation. Augment the syntax of expressions by:

    E ::= ··· | **catch** E | **throw** E

    The ⟦**catch** E⟧ construct is the intended destination of any ⟦**throw** E′⟧ evaluated within

⟦E⟧. The value **catch** produces is the value of ⟦E′⟧. Evaluation of ⟦**throw** E⟧ aborts normal evaluation and the value of ⟦E⟧ is communicated to the nearest enclosing ⟦**catch**⟧. Give the semantics of these constructs.

13. Derive the continuation semantics corresponding to the direct semantics of expressions, using the method in Section 9.6 and the congruence $\mathbf{E}_C[\![\mathrm{E}]\!]k\,s = k(\mathbf{E}_D[\![\mathrm{E}]\!]s)\,s$, for:

    a. The **E** valuation function in Figure 5.2.
    b. The **E** valuation function in Figure 7.5.

14. Prove that the direct and continuation semantics of the language in Section 9.6 are also congruent in an operational sense: prove that $\mathbf{C}_D[\![\mathrm{C}]\!]s$ simplifies to s' iff $\mathbf{C}_C[\![\mathrm{C}]\!]c\,s$ simplifies to $c(s')$.

15. Consider the conditions under which a designer uses continuation domains in a language definition.

    a. What motivates their introduction into the definition?
    b. Under what conditions should some valuation functions map to continuations and others to noncontinuation values?
    c. What characteristics of a language *must* result if continuation domains are placed in the language definition?
    d. Are languages with continuation definitions easier to reason about (e.g., in program equivalence proofs) than languages with direct definitions?
    e. What freedom of choice of implementations is lost when continuation domains are used?