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:

\[ E[[E_1 + E_2]] = \lambda s. E[[E_1]]s \text{ plus } E[[E_2]]s \]

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, canceling 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:

\[ C : \text{Command} \rightarrow \text{Environment} \rightarrow \text{Control-stack} \rightarrow \text{Store} \rightarrow \text{Store}_1 \]

where \( c \in \text{Control-stack} = (\text{Control-stack} \rightarrow \text{Store} \rightarrow \text{Store}_1)_0 \). The expression \((C[e] c s)\) resembles an interpreter-like configuration where \(C[e]\) 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:

\[
\begin{align*}
C[1;C_2] &= \lambda e. \lambda c. \lambda s. C[C_1][e] ((C[C_2][e] \text{cons } c) s) \\
C[1];E &= \lambda e. \lambda c. \lambda s. (\text{hd } c) \ (tl \ c) \ (\text{update}(\text{accessenv}[[1]]) \ e) \ (E[E][e] s) s \\
& \ldots \\
C[\text{stop}] &= \lambda e. \lambda c. \lambda s. s
\end{align*}
\]

We obtain a neat definition for the while-loop:

\[
\begin{align*}
C[\text{while } B \ do \ C] &= \lambda e. \lambda c. \lambda s. B[B][e] s \rightarrow C[C][e] ((C[\text{while } B \ do \ C][e] \text{cons } c) s) \\\n& \ (\text{hd } c) \ (tl \ c) \ s
\end{align*}
\]

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

Whenever the \(c\) stack is popped, \((\text{hd } c)\) is always given \((tl \ c)\) as its argument. The simplification steps are shortened if the semantics of \( [[C_1;C_2]] \) is written so that \( (C[C][e]) \) 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 \( c \in \text{Cmdcont} = \text{Store} \rightarrow \text{Store}_1 \). The language fragment now reads:

\[
\begin{align*}
C : \text{Command} & \rightarrow \text{Environment} \rightarrow \text{Cmdcont} \rightarrow \text{Cmdcont} \\
C[C_1;C_2] &= \lambda e. \lambda c. \ C[C_1][e] \ (C[C_2][e] c) \\
C[1];E &= \lambda e. \lambda c. \lambda s. (\text{update}(\text{accessenv}[[1]]) \ e) \ (E[E][e] s) s \\
C[\text{stop}] &= \lambda e. \lambda c. \lambda s. s \\
C[\text{if } B \ then \ C_1 \ else \ C_2] &= \lambda e. \lambda c. \ \text{choose}(B[B][e] \ (C[C_1][e] c) \ (C[C_2][e] c)) \\
C[\text{while } B \ do \ C] &= \lambda e. \lambda c. \ \text{fix}(\lambda c. \ \text{choose}(B[B][e] \ (C[C][e] c) \ c)) \\
\text{where choose: } (\text{Store} \rightarrow \text{Tr}) & \rightarrow \text{Cmdcont} \rightarrow \text{Cmdcont} \\
\text{choose } b c_1 c_2 &= \lambda s. (b s) \rightarrow (c_1 s) \ [(c_2 s) ]
\end{align*}
\]
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 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 overridden 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 $(C[\text{while } B \text{ do } C] e \ c\ s)$ 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 $C[\text{while } B \text{ do } C] = \lambda c. \lambda s. c(\cdot \cdot \cdot s \cdot \cdot \cdot)$ defines a construct $[\text{while } C]$ that is guaranteed to terminate, for $c$ is applied to the updated store. An equation $C[[C]] = \lambda c. \lambda s. (\cdot \cdot \cdot c \cdot s$ defines a construct whose termination is not guaranteed, so both $c$ and $s$ are carried along for use by $(\cdot \cdot \cdot)$.

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 $\text{finish} = (\lambda s. \text{‘done’})$, which would make all the command continuations into mappings from stores to character strings. The general form of the command continuation domain is:

$$c \in \text{Cmdcont} = \text{Store} \rightarrow \text{Answer}$$

where $\text{Answer}$ 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 $E: \text{Expression} \rightarrow \text{Environment} \rightarrow \text{Store} \rightarrow \text{Expressible-value}$ 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:

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

The expressible value argument to an expression continuation is the intermediate value of the partially evaluated expression. The $\text{Answer}$ domain will be considered shortly.
Let \( C_0 = C_1 ; C_2 ; \text{stop} ; C_1 \)
\( C_1 = X := 1 \)
\( C_2 = \text{while } X > 0 \text{ do } C_3 \)
\( C_3 = X := X - 1 \)

\[
C\{C_0\}e_1 \text{ finish } s_1
\]
where \( e_1 = (\text{updateenv } \{X\} l_0 e_0) \)
and \( \text{finish} = (\lambda s. s) \)

\[
C\{C_1\}e_1 \ (C\{C_2; \text{stop}; C_1\}e_1 \text{ finish}) \ s_1
\]

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

\[
(C\{C_2\} e_1 \ (C\{\text{stop}; C_1\} e_1 \text{ finish})) \ (\text{update } l_0 \text{ one } s_1)
\]

\[
fixF \ s_2
\]

where \( F = \lambda c. \text{choose} (B\{X > 0\} e_1) (C\{C_3\} e_1 c_1) \ c_1 \)
where \( c_1 = C\{\text{stop}; C_1\} e_1 \text{ finish} \)

\[
(B\{X > 0\} e_1) \ s_2 \rightarrow (C\{C_3\} e_1 (fixF) \ s_2) \ [ ] (c_1 \ s_2)
\]

\[
C\{C_3\} e_1 (fixF) \ s_2
\]

\[
(fixF) \ (\text{update } l_0 \text{ zero } s_2)
\]

\[
s_3
\]

\[
(B\{X > 0\} e_1) \rightarrow (C\{C_3\} e_1 (fixF) s_3) \ [ ] (c_1 \ s_3)
\]

\[
(c_1 \ s_3)
\]

\[
C\{\text{stop}\} e_1 \ (C\{C_1\} e_1 \text{ finish}) \ s_3
\]

\[
s_3
\]

The semantic equations for some of the expression constructs read as follows:
9.1.1 Other Levels of Continuations

\[ E : \text{Expression} \rightarrow \text{Environment} \rightarrow \text{Exprcont} \rightarrow \text{Store} \rightarrow \text{Answer} \]

\[ E[[E_1+E_2]] = \lambda e. \lambda k. E[[E_1]](\lambda n_1. E[[E_2]](\lambda n_2. k(n_1+\text{plus} n_2))) \]

\[ E[[I]] = \lambda e. \lambda k. \lambda s. k(\text{access}(\text{accessenv}[I] e) s) s \]

\[ E[[N]] = \lambda e. \lambda k. k(\text{N}[N]) \]

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 \((n_1+\text{plus} n_2)\).

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

\[ C[[I:=E]] = \lambda e. \lambda c. \text{accessenv}[I] e (\lambda l. E[[E]] e(\lambda n. \text{update}(\text{accessenv}[I] e) l n c)) s \]

where \(\text{fin} \in \text{Exprcont}\) is \(\text{fin} = \lambda n. \lambda s. n\)

If \(\text{Answer}' = \text{Answer}\), then the two levels of control integrate nicely:

\[ C[[I:=E]] = \lambda e. \lambda c. \text{accessenv}[I] e (\lambda n. \lambda s. \text{E}[E][E] e(\lambda n. \lambda s. \text{c}(\text{update}(\text{accessenv}[I] e) n s))) s \]

Now \(\text{Exprcont} = \text{Expressible-value} \rightarrow \text{Cmdcont}\), 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:

\[ C[[I:=E]] = \lambda e. \lambda c. \text{accessenv}[I] e (\lambda l. E[[E]] e(\lambda n. \text{update}(\text{accessenv}[I] e) l n c)) \]

where \(\text{accessenv} : \text{Identifier} \rightarrow \text{Environment} \rightarrow (\text{Location} \rightarrow \text{Cmdcont}) \rightarrow \text{Cmdcont}\)

\(\text{accessenv} = \lambda i. \lambda e. \lambda m. m(e(i))\)

and \(\text{update} : \text{Location} \rightarrow \text{Expressible-value} \rightarrow \text{Cmdcont} \rightarrow \text{Cmdcont}\)

\(\text{update} = \lambda l. \lambda n. \lambda c. c([l \mapsto n])s\)

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
Abstract syntax:

\[ D \in \text{Declaration} \]
\[ C \in \text{Command} \]
\[ I \in \text{Identifier} \]

\[ D ::= D_1 ; D_2 | \cdots | \text{on I do } C \]
\[ C ::= \text{begin } D; C \text{ end } | \cdots | \text{raise I} \]

Semantic algebras:

I. Program outputs

Domain \( Answer = (\text{Store + String}) \)

II. Command continuations

Domain \( c \in \text{Cmdcont} = \text{Store} \rightarrow Answer \)

Operations

\[ \text{fin} : \text{Cmdcont} \]
\[ \text{fin} = \lambda s. \text{inStore}(s) \]

\[ \text{err} : \text{Cmdcont} \]
\[ \text{err} = \lambda s. \text{inString}(\text{"error"}) \]

III. Denotable values

Domain \( \text{Denotable-value} = \text{Cmdcont + Nat + \cdots} \)

IV. Environments

Domain \( e \in \text{Environment} = \text{Identifier} \rightarrow \text{Denotable-value} \)

Operations (usual)

Valuation functions:

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

\[ C: \text{Command} \rightarrow \text{Environment} \rightarrow \text{Cmdcont} \rightarrow \text{Cmdcont} \]
\[ C[\text{begin } D; C \text{ end}] = \lambda e. \lambda c. C[C] (D[D] e) c \]
\[ C[\text{raise I}] = \lambda e. \lambda c. \text{cases} (\text{accessEnv}[I] e) \text{ of} \]
\[ \quad \text{isCmdcont}(c) \rightarrow c, \]
\[ \quad [] \text{isNat}(n) \rightarrow \text{err} \]
\[ \quad \cdots \text{ end} \]
Control as a Semantic Domain

Figure 9.3

Abstract syntax:

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

P ::= D. ?C
D ::= D₁,D₂ | I ← C
C ::= C₁,C₂ | C₁ or C₂ | I | succeedwith F | fail | cut

Semantic algebras:

I. Program outputs
   Domain Answer = Store + String

II. Stores
   Domain s ∈ Store
9.3 Backtracking Mechanisms

III. Continuations

\[ c \in \text{Cmdcont} = \text{Store} \rightarrow \text{Answer} \]
\[ fc \in \text{Failure-cont} = \text{Cmdcont} \]
\[ sc \in \text{Success-cont} = \text{Failure-cont} \rightarrow \text{Cmdcont} \]

Operations

\[ \text{succeeded} : \text{Success-cont} \]
\[ \text{succeeded} = \lambda fc. \lambda s. \text{inStore}(s) \]
\[ \text{failed} : \text{Failure-cont} \]
\[ \text{failed} = \lambda s. \text{inString}(\text{"failure"}) \]

IV. Evaluation strategies

\[ \text{Domain Strategy} = \text{Success-cont} \rightarrow \text{Failure-cont} \rightarrow \text{Cmdcont} \]

V. Environments

\[ \text{Domain e} \in \text{Environment} = \text{Identifier} \rightarrow \text{Strategy} \]

Operations

\[ \text{emptyenv} : \text{Environment} \]
\[ \text{emptyenv} = \lambda i. (\lambda sc. \lambda fc. fc) \]
\[ \text{accessenv} : \text{Identifier} \rightarrow \text{Environment} \rightarrow \text{Strategy} \text{ (usual)} \]
\[ \text{updateenv} : \text{Identifier} \rightarrow \text{Strategy} \rightarrow \text{Environment} \rightarrow \text{Environment} \text{ (usual)} \]

Valuation functions:

\[ \text{P: Program} \rightarrow \text{Cmdcont} \]
\[ \text{P}[[D, ?C]] = \text{C}[[C]] \ (\text{D}[[D]] \text{emptyenv}) \text{ succeeded failed} \]

\[ \text{D: Declaration} \rightarrow \text{Environment} \rightarrow \text{Environment} \]
\[ \text{D}[[D_1, D_2]] = \text{D}[[D_2]] \circ \text{D}[[D_1]] \]
\[ \text{D}[I \leftarrow C] = \lambda e. \text{updateenv}[[I]] (\text{C}[[C]]e) e \]

\[ \text{C: Command} \rightarrow \text{Environment} \rightarrow \text{Strategy} \]
\[ \text{C}[[C_1, C_2]] = \lambda e. \lambda sc. \text{C}[[C_1]]e \ (\text{C}[[C_2]]e sc) \]
\[ \text{C}[[C_1 \text{ or } C_2]] = \lambda e. \lambda sc. \lambda fc. \lambda s. \text{C}[[C_1]]e sc \ (\lambda sv. \text{C}[[C_2]]e sc fc s) s \]
\[ \text{C}[I] = \text{accessenv}[[I]] \]
\[ \text{C}[[\text{succeed} \text{with } F]] = \lambda e. \lambda sc. \lambda fc. \lambda s. sc fc \ (\text{F}[[F]]s) \]
\[ \text{C}[[\text{fail}]] = \lambda e. \lambda sc. \lambda fc. fc \]
\[ \text{C}[[\text{cut}]] = \lambda e. \lambda sc. \lambda fc. sc \text{ failed} \]

\[ \text{F: Primitive-operator} \rightarrow \text{Store} \rightarrow \text{Store} \text{ (omitted)} \]

---

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 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: C[goto I] = \lambda e. \lambda c. accessenv[I] e.
9.5 Unrestricted Branching Mechanisms

Figure 9.4

Abstract syntax:
\[ B \in \text{Block} \]
\[ D \in \text{Declaration} \]
\[ C \in \text{Command} \]
\[ I \in \text{Identifier} \]
\[ B ::= D; \text{initiate } I \]
\[ D ::= D_1; D_2 \mid \text{coroutine } I=C \]
\[ C ::= C_1; C_2 \mid \text{resume } I \mid I:=E \]

Semantic algebras:
I. Command continuations
Domain \( \text{Cmdcont} = \text{Store} \rightarrow \text{Answer} \)
II. Coroutine continuations and the environments holding them
Domains \( e \in \text{Coroutine-env} = ((\text{Identifier} \rightarrow \text{Coroutine-cont}) \times \text{Identifier}) \)

Operations
\[ \text{quit} : \text{Coroutine-cont} \]
\[ \text{err} : \text{Coroutine-cont} \]
\[ \text{empty-env} : \text{Identifier} \rightarrow \text{Coroutine-env} \]
\[ \text{initialize} : \text{Identifier} \rightarrow \text{Coroutine-cont} \rightarrow \text{Coroutine-env} \rightarrow \text{Coroutine-env} \]
\[ \text{resume} : \text{Identifier} \rightarrow \text{Coroutine-cont} \rightarrow \text{Coroutine-env} \rightarrow \text{Cmdcont} \]

Valuation functions:
\[ B : \text{Block} \rightarrow \text{Cmdcont} \]
\[ B[D; \text{initiate } I] = \text{resume}[I] \quad \text{quit} (D[D](\text{empty-env}[I])) \]
\[ D : \text{Declaration} \rightarrow \text{Coroutine-env} \rightarrow \text{Coroutine-env} \]
\[ D[D_1; D_2] = D[D_2] \circ D[D_1] \]
\[ D[\text{coroutine } I=C] = \text{initialize}[I] \quad (C[C] \text{ quit}) \]
\[ C : \text{Command} \rightarrow \text{Coroutine-cont} \rightarrow \text{Coroutine-env} \rightarrow \text{Cmdcont} \]
\[ C[C_1; C_2] = C[C_1] \circ C[C_2] \]
\[ C[\text{resume } I] = \text{resume}[I] \]
\[ C[I:=E] = \lambda c. \lambda e. \lambda s. c \ e \ (\text{update}[I] \ (E[E]s)) \]
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 $[[C_i]]$, the continuation $c_i$ associated with $[[I]]$ is $(C[[C_i]] e_{c_{i+1}})$, where $e_{c_{i+1}}$ is the continuation for the commands that follow $[[C_i]]$.

Now consider a block with $n$ distinct labels: $[[\text{begin } D; I_1: C_1; I_2: C_2; \cdots; I_n: C_n \text{ end}]]$. The continuations are:

$$
c_1 = (C[[C_1]] e_I c_2) \\
c_2 = (C[[C_2]] e_I c_3) \\
\vdots \\
c_{n-1} = (C[[C_{n-1}]] e_I c_n) \\
c_n = (C[[C_n]] e_I c)
$$

where $e = (\text{updateenv}[[I_1]] \text{ in } \text{Cmdcont}(c_1))$ 

$$(\text{updateenv}[[I_2]] \text{ in } \text{Cmdcont}(c_2)) \\
\vdots \\
(\text{updateenv}[[I_n]] \text{ in } \text{Cmdcont}(c_n) (D[[D]] e)) \cdots ))$$

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 $E[[E_1 \ast E_2]]$ and $E[[E_1 - E_2]]$ for multiplication and subtraction are added to 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 $(\text{ctuple} \downarrow k)$. 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.
Abstract syntax:

\[ P \in \text{Program} \]
\[ B \in \text{Block} \]
\[ D \in \text{Declaration} \]
\[ C \in \text{Command} \]
\[ E \in \text{Expression} \]
\[ I \in \text{Identifier} \]
\[ N \in \text{Numeral} \]

\[ P ::= B. \]
\[ B ::= \text{begin} D_1; C_1; I_1; C_2; \ldots; I_n; C_n \text{ end} \]
\[ D ::= D_1; D_2 | \text{const} I = N | \text{var} I \]
\[ C ::= C_1; C_2 | I: E = E | \text{if} E \text{ then } C_1 \text{ else } C_2 | \text{while } E \text{ do } C | B | \text{goto } I \]
\[ E ::= E_1 + E_2 | I | N | \text{do } C \text{ result is } E | (E) \]

Semantic algebras:

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

VI. Semantic outputs

Domain \( a \in \text{Answer} = (\text{OK} + \text{Err}) \)
where \( \text{OK} = \text{Store} \) and \( \text{Err} = \text{String} \)

VII.-IX. Expressible, denotable, and storable values

Domains \( n \in \text{Exprval} = \text{Storable-value} = \text{Nat} \)
\[ d \in \text{Denotable-value} = \text{Nat} + \text{Location} + \text{Cmdcont} + \text{Errvalue} \]
where \( \text{Errvalue} = \text{Unit} \)

X. Environments

Domain \( e \in \text{Environment} = (\text{Identifier} \rightarrow \text{Denotable-value}) \times \text{Location} \)
Operations
(defined in Figure 7.1)

XI. Stores

Domain \( s \in \text{Store} = \text{Location} \rightarrow \text{Storable-value} \)
Operations
(defined in Figure 7.1)

XII. Command continuations

Domain \( c \in \text{Cmdcont} = \text{Store} \rightarrow \text{Answer} \)
Operations

\[ \text{finish} : \text{Cmdcont} \]
\[ \text{finish} = \lambda s. \text{inOK}(s) \]

\[ \text{error} : \text{String} \rightarrow \text{Cmdcont} \]
\[ \text{error} = \lambda t. \lambda s. \text{inErr}(t) \]

XIII. Expression continuations

Domain \( k \in \text{Exprcont} = \text{Exprval} \rightarrow \text{Cmdcont} \)

Operations

\[ \text{return-value} : \text{Exprval} \rightarrow \text{Exprcont} \rightarrow \text{Cmdcont} \]
\[ \text{return-value} = \lambda n. \lambda k. k(n) \]

\[ \text{save-arg} = (\text{Exprcont} \rightarrow \text{Cmdcont}) \rightarrow (\text{Exprval} \rightarrow \text{Exprcont}) \rightarrow \text{Exprcont} \]
\[ \text{save-arg} = \lambda f. \lambda g. \lambda n. f(g(n)) \]

\[ \text{add} : \text{Exprcont} \rightarrow \text{Exprval} \rightarrow \text{Exprval} \rightarrow \text{Cmdcont} \]
\[ \text{add} = \lambda k. \lambda n_1. \lambda n_2. k(n_1 \text{plus} n_2) \]

\[ \text{fetch} : \text{Location} \rightarrow \text{Exprcont} \rightarrow \text{Cmdcont} \]
\[ \text{fetch} = \lambda l. \lambda k. \lambda s. k(\text{access}(l, s)) \]

\[ \text{assign} : \text{Location} \rightarrow \text{Cmdcont} \rightarrow \text{Exprcont} \]
\[ \text{assign} = \lambda l. \lambda c. \lambda n. \lambda s. c(\text{update}(l, n, s)) \]

\[ \text{choose} : \text{Cmdcont} \rightarrow \text{Cmdcont} \rightarrow \text{Exprcont} \]
\[ \text{choose} = \lambda c_1. \lambda c_2. \lambda n. \text{greaterthan zero} \rightarrow c_1 \lbrack c_2 \rbrack \]

Valuation functions:

\[ \text{P} : \text{Program} \rightarrow \text{Location} \rightarrow \text{Cmdcont} \]
\[ \text{P}[B.] = \lambda l. \text{B}[B.] (\text{emptyenv}(l)) \text{finish} \]

\[ \text{B} : \text{Block} \rightarrow \text{Environment} \rightarrow \text{Cmdcont} \rightarrow \text{Cmdcont} \]
\[ \text{B}[\text{begin} D; I_1; C_1; I_2; C_2; \ldots ; I_n; C_n \text{ end}] = \lambda e. \lambda c. (\text{fix}(\lambda \text{ctuple. } ((\text{C}[C_1] e \circ (\text{ctuple} \downarrow 2)), \ldots, (\text{C}[C_n] e \circ c))) \downarrow 1 \]

where \( e \circ = (\text{updateenv}[I_1] \text{ inCmdcont}(\text{ctuple} \downarrow 1)) \)
\[ (\text{updateenv}[I_2] \text{ inCmdcont}(\text{ctuple} \downarrow 2)) \]
\[ \ldots \]
\[ (\text{updateenv}[I_n] \text{ inCmdcont}(\text{ctuple} \downarrow n) (D[D[e]]) \ldots) \]
Figure 9.5 (continued)

D: Declaration $\rightarrow$ Environment $\rightarrow$ Environment
(defined in Figure 7.2)

C: Command $\rightarrow$ Environment $\rightarrow$ Cmdcont $\rightarrow$ Cmdcont
$C[C_1;C_2] = \lambda e. C[C_1]e \cdot C[C_2]e$
$C[I:=E] = \lambda e.\lambda c. \text{cases (accessenv}[I]e) of$
\hspace{1em} \text{isNat}(n) \rightarrow \text{error} "\text{const used on lhs}"$
\hspace{1em} \text{isLocation}(l) \rightarrow E[E]e (\text{assign } l c)$
\hspace{1em} \text{isCmdcont}(c) \rightarrow \text{error} "\text{label used on lhs}"$
\hspace{1em} \text{isErrvalue()} \rightarrow \text{error} "\text{lhs undeclared}"
\hspace{1em}\text{end}$

$C[\text{if } E \text{ then } C_1 \text{ else } C_2] = \lambda e.\lambda c. \text{choose (}C[C_1]e c) (C[C_2]e c))$

$C[\text{while } E \text{ do } C] = \lambda e.\lambda c. \text{fix} (\lambda c. E[E]e (\text{choose (}C[C]e c' c)))$

$C[B] = B[B]$  

$C[\text{ goto } I] = \lambda e.\lambda c. \text{cases (accessenv}[I]e) of$
\hspace{1em} \text{isNat}(n) \rightarrow \text{error} "\text{const used as label}"$
\hspace{1em} \text{isLocation}(l) \rightarrow \text{error} "\text{var used as label}"$
\hspace{1em} \text{isCmdcont}(c' \rightarrow c'$)
\hspace{1em} \text{isErrvalue()} \rightarrow \text{error} "\text{unknown id}"$
\hspace{1em}\text{end}$

E: Expression $\rightarrow$ Environment $\rightarrow$ Exprcont $\rightarrow$ Cmdcont
$E[E_1+E_2] = \lambda e.\lambda k. E[E_1]e (\text{save-arg (}E[E_2]e) (\text{add } k))$

$E[I] = \lambda e.\lambda k. \text{cases (accessenv}[I]e) of$
\hspace{1em} \text{isNat}(n) \rightarrow \text{return-value } n k$
\hspace{1em} \text{isLocation}(l) \rightarrow \text{fetch } l k$
\hspace{1em} \text{isCmdcont}(c) \rightarrow \text{error} "\text{label used in expr}"
\hspace{1em} \text{isErrvalue()} \rightarrow \text{error} "\text{undeclared iden}"
\hspace{1em}\text{end}$

$E[N] = \lambda e.\lambda k. \text{return-value (}N[N]k) k$

$E[\text{do } \text{C resultis } E] = \lambda e.\lambda k. C[C]e (E[E]e k)$

$E[(E)] = E[E]$  

N: Numeral $\rightarrow$ Nat  (omitted)
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 \( C_D: \text{Command} \rightarrow \text{Store} \rightarrow \text{Store} \) be a valuation function written in direct semantics style. Say that we desire a valuation function \( C_C: \text{Command} \rightarrow \text{Cmdcont} \rightarrow \text{Cmdcont} \),
\[ C_{\text{ndcont}} = \text{Store}_1 \rightarrow \text{Store}_1, \text{ in continuation style such that } C_D \text{ is equivalent to } C_C. \text{ The equivalence can be stated as } C_D[\llbracket C \rrbracket] = C_C[\llbracket C \rrbracket](\lambda s. s). \text{ But this property will be difficult to prove by induction. Recall that in Section 9.1 we saw the format } C_C[\llbracket C \rrbracket] = \lambda c. \lambda s. c(f(s)) \text{ for a terminating command } \llbracket C \rrbracket \text{ that performs } f \text{ to the store. Since } C_D[\llbracket C \rrbracket] \text{ describes } \llbracket C \rrbracket \text{'s actions in isolation, the equality:} \]

\[
C_C[\llbracket C \rrbracket]c s = c(C_D[\llbracket C \rrbracket]s)
\]

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 \( C_D[\llbracket C \rrbracket] \) might be \( \perp \), so what should \( C_C[\llbracket C \rrbracket]c s \) be? If we require strict continuations, then the congruence still holds: \( C_C[\llbracket C \rrbracket]c s = c(C_D[\llbracket C \rrbracket]s) = c(\perp) = \perp. \) Sethi and Tang (1980) suggest that a continuation semantics for a simple language be derived from its direct semantics definition by defining \( C_C[\llbracket C \rrbracket]c s = c(C_D[\llbracket C \rrbracket]s) \) for each construct \( \llbracket C \rrbracket \) in the language. We then apply an inductive hypothesis to simplify \( c(C_D[\llbracket C \rrbracket]s) \) into a more satisfactory form. Some derivations follow.

First, for \( C_D[\llbracket C_1; C_2 \rrbracket] = \lambda s. C_D[\llbracket C_2 \rrbracket](C_D[\llbracket C_1 \rrbracket]s) \), define:

\[
C_C[\llbracket C_1; C_2 \rrbracket] = \lambda c. \lambda s. c(C_D[\llbracket C_1 \rrbracket; C_2 \rrbracket s)
\]

\[
= \lambda c. \lambda s. c(C_D[\llbracket C_2 \rrbracket](C_D[\llbracket C_1 \rrbracket]s))
\]

Since \( C_D[\llbracket C_1 \rrbracket] s \in \text{Store}_1 \), we can use the inductive hypothesis that \( C_C[\llbracket C_2 \rrbracket] s c = c(C_D[\llbracket C_2 \rrbracket]s) \) to obtain the value \( \lambda c. \lambda s. c(C_D[\llbracket C_1 \rrbracket; C_2 \rrbracket s) \). But \( C_C[\llbracket C_2 \rrbracket] c \) is a command continuation, so we apply the inductive hypothesis for \( \llbracket C_1 \rrbracket \) to obtain \( \lambda c. \lambda s. c(C_C[\llbracket C_1 \rrbracket; (C_C[\llbracket C_2 \rrbracket] c)s) \). By extensionality and the inductive hypothesis that \( C_C[\llbracket C_2 \rrbracket] \) is strict, we obtain:

\[
C_C[\llbracket C_1; C_2 \rrbracket] = C_C[\llbracket C_1 \rrbracket] \circ C_C[\llbracket C_2 \rrbracket]
\]

Second, for \( C_D[\text{if } B \text{ then } C_1 \text{ else } C_2 \rrbracket] = \lambda s. B[\llbracket B \rrbracket] s \rightarrow C_D[\llbracket C_1 \rrbracket] s \mid C_D[\llbracket C_2 \rrbracket] s \), we define:

\[
C_C[\text{if } B \text{ then } C_1 \text{ else } C_2 \rrbracket] = \lambda c. \lambda s. c(C_D[\text{if } B \text{ then } C_1 \text{ else } C_2 \rrbracket] s)
\]

\[
= \lambda c. \lambda s. c(B[\llbracket B \rrbracket] s \rightarrow C_D[\llbracket C_1 \rrbracket] s \mid C_D[\llbracket C_2 \rrbracket] s)
\]

\( B[\llbracket B \rrbracket] \) is a defined truth value, so a cases analysis gives the value \( \lambda c. \lambda s. c(B[\llbracket B \rrbracket] s \rightarrow c(C_D[\llbracket C_1 \rrbracket] s) \mid c(C_D[\llbracket C_2 \rrbracket] s) \). By the inductive hypothesis for \( \llbracket C_1 \rrbracket \) and \( \llbracket C_2 \rrbracket \), we obtain:

\[
C_C[\text{if } B \text{ then } C_1 \text{ else } C_2 \rrbracket] = \lambda c. \lambda s. B[\llbracket B \rrbracket] s \rightarrow C_C[\llbracket C_1 \rrbracket] c s \mid C_C[\llbracket C_2 \rrbracket] c s
\]

Deriving the continuation semantics for a \textbf{while}-loop is a bit more involved. For \( C_D[\text{while } B \text{ do } C \rrbracket] = \text{fix}(\lambda f. \lambda s. B[\llbracket B \rrbracket] s \rightarrow f(C_D[\llbracket C \rrbracket] s) \mid s) \), we define:

\[
C_C[\text{while } B \text{ do } C \rrbracket] = \lambda c. \lambda s. c(C_D[\text{while } B \text{ do } C \rrbracket) s)
\]

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

\[
F_D = \lambda f. \lambda s. B[\llbracket B \rrbracket] s \rightarrow f(C_D[\llbracket C \rrbracket] s) \mid s
\]

We hope to find a corresponding functional \( F_C \) such that \( (\text{fix } F_C)s = c((\text{fix } F_D)s) \). 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:

\[
P(f_C, f_D) = \{ \text{for all } c \in \text{Cmdcont and } s \in \text{Store}_1, (f_C c s) = c(f_D s) \}
\]

where \( f_C : \text{Cmdcont} \rightarrow \text{Cmdcont} \) and \( f_D : \text{Store}_1 \rightarrow \text{Store}_1 \).

For the basis step, we have \( f_C = (\lambda c. \lambda s. \bot) \) and \( f_D = (\lambda s. \bot) \); 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:

\[
c((F_D f_D) s)
= c((\lambda s. \text{B}[\text{B}]) s \rightarrow f_D (\text{C}_D[C][s]) [s] s)
= c((\text{B}[\text{B}]) s \rightarrow f_D (\text{C}_D[C][s]) [s] s)
\]

when \( s \) is proper. This equals:

\[
\text{B}[\text{B}][s] \rightarrow c((f_D (\text{C}_D[C][s]) [s] s), \text{by distributing } c \text{ across the conditional}
= \text{B}[\text{B}][s] \rightarrow (f_C c)(\text{C}_D[C][s]) [s] s), \text{by the inductive hypothesis}
= \text{B}[\text{B}][s] \rightarrow \text{C}_C[C][f_C c][s] [s] (c s)
\]

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

\[
F_C = \lambda g. \lambda c. \lambda s. \text{B}[\text{B}][s] \rightarrow \text{C}_C[C][g c][s] (c s)
\]

then we are finished:

\[
\text{C}_C[C][\text{while B do C}][s] = \text{fix}(\lambda g. \lambda c. \lambda s. \text{B}[\text{B}][s] \rightarrow \text{C}_C[C][g c][s] (c s))
\]

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

The continuation semantics \( \text{C}_C \) is congruent to \( \text{C}_D \) 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**: Björner & 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
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. $Declaration-cont = Environment \rightarrow Cmdcont$
      ii. $Denotable-value-cont = Denotable-value \rightarrow Cmdcont$
      iii. $Nat-cont = Nat \rightarrow Cmdcont$
      iv. $Location-cont = Location \rightarrow Exprcont$
b. A reasonable functionality for the continuation version of natural number addition is
\[ add : \text{Nat} \rightarrow \text{Nat-cont} \]. 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 \text{Store}-valued
objects. Consider replacing the \text{Cmdcont} algebra by a version of the \text{Store}
algebra; formulate operations for the domain \text{Cmdcont} = \text{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 \text{Pred} be the syntax domain of
predicate calculus expressions. The symbols “B” and “p” stand for elements of
\text{Pred}. Here is the weakest precondition semantics of a small language:

\[
\begin{align*}
wp([C_1;C_2], p) &= wp([C_1], wp([C_2], p)) \\
wp([I:=E], p) &= [E/A]p \\
wp([\text{if } B \text{ then } C_1 \text{ else } C_2], p) &= ([B] \land wp([C_1], p)) \\
&\quad \lor \neg([B]) \land wp([C_2], p)) \\
wp([\text{while } B \text{ do } C], p) &= (\text{there exists } i \geq 0 \text{ such that } H_i(p)) \\
&\quad \text{where } H_0(p) = (\neg([B]) \land p) \\
&\quad \text{and } H_{i+1}(p) = wp([\text{if } B \text{ then } C \text{ else skip}], H_i(p)) \\
\text{and } wp([\text{skip}], p) &= p
\end{align*}
\]

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

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

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

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 $[\text{on zerodivide do } C]$, 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 $[\text{no } I]$, where $[\text{I}]$ 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 cannot 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:

   $$E ::= \cdots \mid \text{catch } E \mid \text{throw } E$$

   The $[\text{catch } E]$ construct is the intended destination of any $[\text{throw } E]$ evaluated within $[E]$. The value catch produces is the value of $[E]$. Evaluation of $[\text{throw } E]$ aborts normal evaluation and the value of $[E]$ is communicated to the nearest enclosing $[\text{catch}]$. 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 $E_C[E][k]s = k(E_D[E][k]s)s$, 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 $C_D \llbracket C \rrbracket s$ simplifies to $s$ iff $C_C \llbracket C \rrbracket c \llbracket s \rrbracket$ 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?