In this chapter, we will go beyond the simple imperative language to consider a \texttt{fail} command and operations for output and input. These extensions will require significant changes to the semantic framework developed in previous chapters. First we will develop a generalization of the “direct” denotational semantics used in Chapter 2; then we will introduce an alternative approach called continuation semantics.

In investigating these topics, we will encounter a richer variety of domains than in previous chapters. In particular, to describe computations whose input and output can go on forever, we will use recursively defined domains.

5.1 The \texttt{fail} Command

Suppose we wish to augment the simple imperative language with a command \texttt{fail} that causes a program to cease execution. The abstract syntax is obvious:

\[
\langle \text{comm} \rangle ::= \texttt{fail}
\]

but the semantics raises a serious problem: How do we insure that, regardless of the context in which it is embedded, if the \texttt{fail} command is executed, it determines the final result of the program?

Consider, for example, the composition \(c_0; c_1\), and ignore for the moment the possibility of nontermination, so that the semantic equation is simply

\[
\llbracket c_0 ; c_1 \rrbracket_{\text{comm}} \sigma = \llbracket c_1 \rrbracket_{\text{comm}}(\llbracket c_0 \rrbracket_{\text{comm}} \sigma).
\]

If the command \(c_0\) fails, then the composition \(c_0; c_1\) also fails, regardless of \(c_1\). However, if \(\llbracket c_1 \rrbracket_{\text{comm}}\) can be any function from \(\Sigma\) to \(\Sigma\), there is no value we can give to \(\llbracket c_0 \rrbracket_{\text{comm}} \sigma\) that is guaranteed to be the value of \(\llbracket c_1 \rrbracket_{\text{comm}}(\llbracket c_0 \rrbracket_{\text{comm}} \sigma)\), because, if \(\llbracket c_1 \rrbracket_{\text{comm}}\) is a constant function, its result will be unaffected by \(\llbracket c_0 \rrbracket_{\text{comm}} \sigma\).

Actually, even nontermination raises this problem, since, if \(c_0\) runs forever, then so must \(c_0; c_1\), regardless of \(c_1\). In Chapter 2 we solved this problem by
taking the domain of command meanings to be $\Sigma \to \Sigma_\bot$ and using the semantic equation

$$\llbracket c_0 ; c_1 \rrbracket_{\text{comm}} \sigma = (\llbracket c_1 \rrbracket_{\text{comm}})_{\bot} (\llbracket c_0 \rrbracket_{\text{comm}} \sigma),$$

where the subscript $\bot$ indicates the extension of a function $f \in \Sigma \to \Sigma_\bot$ to $f_{\bot} \in \Sigma_\bot \to \Sigma_\bot$ such that

$$f_{\bot} \bot = \bot$$

$$f_{\bot} \sigma = f \sigma,$$

which guarantees that $(\llbracket c_1 \rrbracket_{\text{comm}})_{\bot}$ will preserve $\bot$, regardless of $c_1$.

A similar approach can be used for failure. We take $\llbracket c \rrbracket_{\text{comm}} \sigma$ to be $\bot$ if $c$ does not terminate, the state $\sigma'$ if $c$ terminates normally (without failure) in the state $\sigma'$, and the pair $(\text{abort}, \sigma')$ if $c$ terminates by executing $\text{fail}$ in the state $\sigma'$. Thus the meaning of commands satisfies

$$\llbracket \bot \rrbracket_{\text{comm}} \in \langle \text{comm} \rangle \to \Sigma \to \hat{\Sigma}_\bot,$$

where $\hat{\Sigma}_\bot = (\hat{\Sigma})_\bot$ and

$$\hat{\Sigma} = \Sigma \cup \{ (\text{abort}, \sigma) \mid \sigma \in \Sigma \} = \Sigma \cup \{ \text{abort} \} \times \Sigma.$$

(More abstractly, $\hat{\Sigma}$ must be the union of two disjoint sets that are each isomorphic to $\Sigma$. A more canonical choice might be $\hat{\Sigma} = \{ \text{finish, abort} \} \times \Sigma$ or $\hat{\Sigma} = \Sigma + \Sigma$, where $+$ is the disjoint union operator described in Sections A.3 and A.5 of the Appendix. But by making $\Sigma$ a subset of $\hat{\Sigma}$, we avoid having to change the semantic equations we have already given for assignment commands and $\text{skip}$.)

Then the semantics of $\text{fail}$ is given by

$$\text{DR SEM EQ: } \text{fail}$$

$$\llbracket \text{fail} \rrbracket_{\text{comm}} \sigma = (\text{abort}, \sigma),$$

while the semantic equation for composition becomes

$$\text{DR SEM EQ: Sequential Composition}$$

$$\llbracket c_0 ; c_1 \rrbracket_{\text{comm}} \sigma = (\llbracket c_1 \rrbracket_{\text{comm}})_* (\llbracket c_0 \rrbracket_{\text{comm}} \sigma),$$

where $*$ indicates the extension of a function $f \in \Sigma \to \hat{\Sigma}_\bot$ to $f_* \in \hat{\Sigma}_\bot \to \hat{\Sigma}_\bot$ such that

$$f_* \bot = \bot$$

$$f_* \sigma = f \sigma$$

$$f_*(\text{abort}, \sigma) = (\text{abort}, \sigma).$$
This definition guarantees that \( ([c]_{\text{comm}})_* \) preserves both \( \bot \) and the results \( \langle \text{abort}, \sigma \rangle \) of \text{fail} commands.

A similar generalization from \( \bot \) to \( * \) occurs in the semantic equations for \text{while} commands:

\[
\text{DR SEM EQ: while}
\]

\[
[\text{while } b \text{ do } c]_{\text{comm}} = \gamma_{\Sigma \rightarrow \Sigma \bot} F
\]

where \( F(f)\sigma = \text{if } [b]_{\text{boolexp}} \sigma \text{ then } f_*([c]_{\text{comm}} \sigma) \text{ else } \sigma. \)

The remaining semantic equations in Section 2.2 are unchanged, except for the equation for variable declarations, where one might expect

\[
[\text{newvar } v := e \text{ in } c]_{\text{comm}} \sigma
\]

\[
= (\lambda \sigma' \in \Sigma. [\sigma' | v: \sigma v])_*([c]_{\text{comm}}[\sigma | v: [e]_{\text{intexp}} \sigma]) \quad (5.1)
\]

Here, however, there is a serious problem. For example, we would have

\[
[\text{x := 0 ; newvar } x := 1 \text{ in fail}]_{\text{comm}} \sigma = \langle \text{abort}, [\sigma | x: 1] \rangle
\]

and

\[
[\text{x := 0 ; newvar } y := 1 \text{ in fail}]_{\text{comm}} \sigma = \langle \text{abort}, [\sigma | x: 0 | y: 1] \rangle,
\]

but the commands in these equations can be obtained from one another by renaming, so that they should have the same meaning.

To avoid this problem, the result of executing a command that fails must be a state that records the values of the free variables of that command, rather than of the local variables that are active when a \text{fail} instruction is executed. For this reason, when a \text{fail} command is executed, each variable in the local state that is bound by a declaration whose scope encloses the \text{fail} must revert to the value it had before the declaration took effect.

Thus variable declarations must reset final states that arise from failure as well as from normal termination. To indicate this, we change the kind of function extension used in the semantic equation for variable declarations:

\[
\text{DR SEM EQ: Variable Declaration}
\]

\[
[\text{newvar } v := e \text{ in } c]_{\text{comm}} \sigma
\]

\[
= (\lambda \sigma' \in \Sigma. [\sigma' | v: \sigma v])_{\uparrow}([c]_{\text{comm}}[\sigma | v: [e]_{\text{intexp}} \sigma]). \quad (5.2)
\]

Here, if \( f \in \Sigma \rightarrow \Sigma \), then \( f_{\uparrow} \in \hat{\Sigma}_\bot \rightarrow \hat{\Sigma}_\bot \) is the extension such that

\[
f_{\downarrow} \downarrow = \bot
\]

\[
f_{\uparrow} \sigma = f \sigma
\]

\[
f_{\uparrow} \langle \text{abort}, \sigma \rangle = \langle \text{abort}, f \sigma \rangle.
\]
We must also extend the meaning of specifications to accommodate the introduction of \texttt{fail}. The simplest approach is to treat failure in the same way as nontermination, so that it is permitted by partial correctness but prohibited by total correctness. More precisely, the semantic equations at the beginning of Section 3.1 become

\begin{align*}
\text{DR SEM EQ: Partial Correctness Specification} \\
\langle \{p\} \ c \ \{q\} \rangle_{\text{spec}} &= \forall \sigma \in \Sigma. \ [p]_{\text{assert}} \sigma \Rightarrow \\
&\quad (\langle [c]_{\text{comm}} \sigma = \bot \text{ or } [c]_{\text{comm}} \sigma \in \{\text{abort}\} \times \Sigma \text{ or } [q]_{\text{assert}}(\langle [c]_{\text{comm}} \sigma \rangle) \rangle).
\end{align*}

\begin{align*}
\text{DR SEM EQ: Total Correctness Specification} \\
\langle [p] \ c \ [q] \rangle_{\text{spec}} &= \forall \sigma \in \Sigma. \ [p]_{\text{assert}} \sigma \Rightarrow \\
&\quad (\langle [c]_{\text{comm}} \sigma \neq \bot \text{ and } [c]_{\text{comm}} \sigma \notin \{\text{abort}\} \times \Sigma \text{ and } [q]_{\text{assert}}(\langle [c]_{\text{comm}} \sigma \rangle) \rangle).
\end{align*}

From these equations, it is easy to see the soundness of the following inference rules for specifications:

\begin{itemize}
\item \textbf{SP RULE: Partial Correctness of fail (FLP)}
\begin{align*}
\{\text{true}\} \ \texttt{fail} \ \{\text{false}\},
\end{align*}
\item \textbf{SP RULE: Total Correctness of fail (FLT)}
\begin{align*}
\{\text{false}\} \ \texttt{fail} \ \{\text{false}\}.
\end{align*}
\end{itemize}

In conclusion, we note that the addition of the \texttt{fail} command, although it might na"ively seem to be a minor language extension, is in fact a serious change in the imperative language. A symptom of its seriousness is that it cannot be described by merely adding another semantic equation; instead, we had to change the underlying domain containing the range of the semantic function, from $\Sigma \rightarrow \Sigma_\bot$ to $\Sigma \rightarrow \hat{\Sigma}_\bot$. (Analogously, when we consider the operational semantics of \texttt{fail} in Section 6.3, we will have to change the set of terminal configurations.)

When a language extension requires such a global change to the underlying semantic framework, one should suspect that it may have surprising interactions with the rest of the language, changing properties that might na"ively seem to be unrelated to the extension. In fact, this is the case at present: The language described in previous chapters satisfies, for example, the law that

\begin{align*}
c \ ; \ \texttt{while true do skip} \text{ and } \texttt{while true do skip}
\end{align*}

have the same meaning for any command $c$. But when \texttt{fail} is added to the language, this seemingly unrelated equivalence law ceases to be valid.

In the following sections, we will consider extensions to intermediate output and input that also require changes to the underlying semantic framework.
5.2 Intermediate Output and a Domain of Sequences

We next consider extending the simple imperative language with a command

\[
\langle \text{comm} \rangle ::= \text{!} \langle \text{intexp} \rangle
\]

that causes output of the value of its operand without terminating program execution. (We will give the same precedence as the operator := for assignment.) At the outset, it should be emphasized that this is a major change in our view of how programs can behave. Until now, we have assumed that all programs that fail to terminate (for some particular initial state) have indistinguishable behavior (for that initial state). But now such programs can produce output, which in the most interesting case can be endless. Thus we are moving closer to a situation that characterizes a large part of real-world computer applications, where the most interesting and useful programs never terminate. (We assume that output cannot be revoked, that is, it becomes available to a user who can be confident that it will not be rescinded by anything like a "rewind" operation.)

Specifically, once intermediate output is introduced, there are three possibilities for program execution:

(a) The program may output a finite sequence of integers and then run on forever without further output.
(b) The program may output a finite sequence of integers and then terminate (normally or abortively) in a final state.
(c) The program may output an endless sequence of integers.

In each of these cases, the total output of the computation can be described by a sequence:

(a) a finite sequence of integers,
(b) a finite sequence where the last element is a state (or other member of \( \hat{\Sigma} \)) and the preceding elements are integers,
(c) an infinite sequence of integers.

Thus we define the output domain \( \Omega \) to consist of these three kinds of sequences. But now we encounter a surprise: Not only can the members of \( \Omega \) be used to describe the total output of a computation at "infinite" time, but they also can be used to describe the accumulated output at some finite time (perhaps before the computation terminates). This suggests how \( \Omega \) should be ordered: \( \omega \sqsubseteq \omega' \) should hold just when \( \omega \) and \( \omega' \) could be outputs of the same computation at earlier and later (or equal) times. In other words, \( \omega \sqsubseteq \omega' \) if and only if \( \omega \) is an initial subsequence of \( \omega' \) (which includes the possibility that \( \omega = \omega' \)).

Now consider a sequence of domain members that give "snapshots" of the accumulated output of a computation at increasing (but finite) times. For the cases listed above, we would have:
(a) An uninteresting chain whose elements are finite sequences of integers. For example:

\[
\langle \rangle \sqsubseteq \langle 3, 1 \rangle \sqsubseteq \langle 3, 1, 4 \rangle \sqsubseteq \langle 3, 1, 4 \rangle \sqsubseteq \langle 3, 1, 4 \rangle \sqsubseteq \cdots .
\]

(b) An uninteresting chain whose elements are finite sequences of integers, except for the final element, which is a finite sequence where the last distinct element is a state (or other member of \( \hat{S} \)) and the preceding elements are integers. For example:

\[
\langle \rangle \sqsubseteq \langle 3, 1 \rangle \sqsubseteq \langle 3, 1, 4 \rangle \sqsubseteq \langle 3, 1, 4, \sigma \rangle \sqsubseteq \langle 3, 1, 4, \sigma \rangle \sqsubseteq \cdots .
\]

(c) An interesting chain whose elements are finite sequences of integers. For example:

\[
\langle \rangle \sqsubseteq \langle 3, 1 \rangle \sqsubseteq \langle 3, 1, 4 \rangle \sqsubseteq \langle 3, 1, 4, 1, 5 \rangle \sqsubseteq \langle 3, 1, 4, 1, 5 \rangle \sqsubseteq \langle 3, 1, 4, 1, 5, 9 \rangle \sqsubseteq \cdots .
\]

In the first two cases, the limit of the chain is its last distinct element. In the third case, the limit is the infinite sequence of integers whose \( i \)th component is the \( i \)th component of every element of the infinite sequence that has at least \( i \) components:

\[
\langle 3, 1, 4, 1, 5, 9, \ldots \rangle.
\]

In all cases, however, the limit of the chain of snapshots is the domain element that describes the total output of the computation. (Of course, one can delete or replicate elements in any of these chains to obtain a differently paced "movie" of the same computation.)

It is worthwhile to spell out in more detail the argument about the limit in the third case. If \( \omega \) and \( \omega' \) are distinct members of \( \Omega \) such that \( \omega \sqsubseteq \omega' \), then \( \omega \) is a finite sequence of integers, \( \omega' \) is a longer sequence, and, whenever \( \omega \) has an \( i \)th component, \( \omega' \) has the same \( i \)th component. Thus the distinct members of the interesting chain must have ever-increasing length and any upper bound of the entire chain must be infinite. Moreover, the \( i \)th component of an upper bound must be the \( i \)th component of the infinitely many chain elements that have \( i \)th components. Thus the upper bound of an interesting chain of finite sequences of integers is uniquely determined, and therefore must be the least upper bound.

Finally, we note that, whenever a chain element is not a finite sequence of integers, there is no distinct member of \( \Omega \) that extends it, so that the chain must be uninteresting. Thus every interesting chain is an instance of our third case and has a least upper bound, so that \( \Omega \) is indeed a domain. (The least element \( \bot \) is the empty sequence, which represents the result of a computation that runs on forever without any output.)

We can now give a denotational semantics for our extended language where the meaning of a command maps an initial state into a sequence in \( \Omega \) that describes
the output behavior and possible final termination of the resulting execution:

\[ [-]_{\text{comm}} \in (\text{comm}) \rightarrow \Sigma \rightarrow \Omega. \]

The semantic equations for assignment commands, \texttt{skip}, and \texttt{fail} are the same as before, except that the final state (or \texttt{abort}-state pair) must be injected into a single-element sequence:

\textbf{DR SEM EQ: Assignment}

\[ [v := e]_{\text{comm}} \sigma = ([\sigma | v] [e]_{\text{intexp}} \sigma), \]

\textbf{DR SEM EQ: skip}

\[ [\text{skip}]_{\text{comm}} \sigma = \langle \sigma \rangle, \]

\textbf{DR SEM EQ: fail}

\[ [\text{fail}]_{\text{comm}} \sigma = \langle \langle \text{abort}, \sigma \rangle \rangle. \]

On the other hand, an output command gives an integer followed by the initial state (since this state is not altered by the command):

\textbf{DR SEM EQ: Output}

\[ [! e]_{\text{comm}} \sigma = \langle [e]_{\text{intexp}} \sigma, \sigma \rangle. \]

When we come to the sequential composition \( c_0 ; c_1 \) of commands, the situation is more complicated. If the execution of \( c_0 \), starting in some initial state, runs on forever (with either finite or infinite output) or results in an abortion, then the execution of \( c_0 ; c_1 \) will behave similarly. But if the execution of \( c_0 \) outputs the integers \( n_0, \ldots, n_{k-1} \) and terminates normally in state \( \sigma' \), then the result of executing \( c_0 ; c_1 \) is obtained by prefixing \( n_0, \ldots, n_{k-1} \) to the result of executing \( c_1 \) in the state \( \sigma' \). This can still be described by the semantic equation

\textbf{DR SEM EQ: Sequential Composition}

\[ [c_0 ; c_1]_{\text{comm}} \sigma = ([c_1]_{\text{comm}}) \ast ([c_0]_{\text{comm}} \sigma). \]

But now the asterisk describes the extension from \( f \in \Sigma \rightarrow \Omega \) to \( f_* \in \Omega \rightarrow \Omega \) such that

\[ f_* \langle n_0, \ldots, n_{k-1} \rangle = \langle n_0, \ldots, n_{k-1} \rangle \]

\[ f_* \langle n_0, \ldots, n_{k-1}, \sigma \rangle = \langle n_0, \ldots, n_{k-1} \rangle \circ (f \sigma) \]

\[ f_* \langle n_0, \ldots, n_{k-1}, \langle \text{abort}, \sigma \rangle \rangle = \langle n_0, \ldots, n_{k-1}, \langle \text{abort}, \sigma \rangle \rangle \]

\[ f_* \langle n_0, n_1, \ldots \rangle = \langle n_0, n_1, \ldots \rangle, \]

where \( \circ \) denotes the concatenation of sequences.
The semantic equation

**DR SEM EQ: Variable Declaration**

\[
\text{[newvar } v := e \text{ in c]}_{\text{comm}} \sigma \\
= (\lambda \sigma' \in \Sigma. [\sigma' \mid v: \sigma v])_t([c]_{\text{comm}}[\sigma \mid v: \text{[e]}_{\text{intexp}}\sigma])
\]

requires a similar treatment. Here \(t\) describes the extension of \(f \in \Sigma \rightarrow \Sigma\) to \(f_t \in \Omega \rightarrow \Omega\) such that

\[
f_t\langle n_0, \ldots, n_{k-1}, \langle\text{abort}, \sigma\rangle\rangle = \langle n_0, \ldots, n_{k-1}, \langle\text{abort}, f(\sigma)\rangle\rangle
\]

The remaining semantic equations are straightforward: The equation for conditionals remains unchanged from the previous section, but the equation for while commands is changed slightly to inject the final state into a single-element sequence:

**DR SEM EQ: while**

\[
\text{[while } b \text{ do c]}_{\text{comm}} = \text{Y}_{\Sigma \rightarrow \Omega} F
\]

where \(F(f)\sigma = \text{if } [b]_{\text{boolop}}\sigma \text{ then } f_*([c]_{\text{comm}}\sigma) \text{ else } \langle\sigma\rangle\).

In devising the denotational semantics in this section, our task has been eased by the concrete nature of the domain \(\Omega\) as a partially ordered set of sequences. Unfortunately, when we consider input in Section 5.6, the domain \(\Omega\) will become a much more abstract object that is defined by a recursive isomorphism. To prepare for this development, it is useful to rewrite our semantic equations and the relevant function extensions in terms of certain injections into \(\Omega\).

From Equations (5.3), it is easy to show that \(f_*\) is continuous and satisfies the following equations:

\[
f_*\langle\rangle = \langle\rangle
\]

\[
f_*\langle\sigma\rangle = f(\sigma)
\]

\[
f_*\langle\langle\text{abort}, \sigma\rangle\rangle = \langle\langle\text{abort}, \sigma\rangle\rangle
\]

\[
f_*\langle\langle n \circ \omega\rangle\rangle = \langle n \circ f_*\omega\rangle.
\]
If we introduce the following functions into $\Omega$, which have disjoint ranges and are each injective:

\[
\begin{align*}
\iota_\bot &\in \{\} \to \Omega \quad \text{such that} \quad \iota_\bot() = \{} = \bot_
\Omega \\
\iota_{\text{term}} &\in \Sigma \to \Omega \quad \text{such that} \quad \iota_{\text{term}}(\sigma) = \langle \sigma \rangle \\
\iota_{\text{abort}} &\in \Sigma \to \Omega \quad \text{such that} \quad \iota_{\text{abort}}(\sigma) = \langle \langle \text{abort}, \sigma \rangle \rangle \\
\iota_{\text{out}} &\in \mathbb{Z} \times \Omega \to \Omega \quad \text{such that} \quad \iota_{\text{out}}(n, \omega) = \langle n \rangle \circ \omega,
\end{align*}
\]

then we can rewrite Equations (5.5) as

\[
\begin{align*}
f_\ast \bot &= \bot \\
f_\ast(\iota_{\text{term}}\sigma) &= f\sigma \\
f_\ast(\iota_{\text{abort}}\sigma) &= \iota_{\text{abort}}\sigma \\
f_\ast(\iota_{\text{out}}(n, \omega)) &= \iota_{\text{out}}(n, f_\ast\omega).
\end{align*}
\]

In fact, $f_\ast$ is the unique continuous solution of these equations. Despite the fact that $\Omega$ is not a very syntactic entity, this can be shown by an extension of the arguments about abstract syntax and semantic equations in Sections 1.1 and 1.2. Since the functions defined by Equations (5.6) are injective and have disjoint ranges, and the sequences that can be obtained by applying these functions a finite number of times are exactly the finite sequences in $\Omega$, we can regard these functions as the constructors of an abstract syntax whose phrases are the finite sequences in $\Omega$ (where $\Sigma$ and $\mathbb{Z}$ are predefined sets). With respect to these constructors, Equations (5.7) are syntax-directed, so that they have a unique solution over the finite sequences in $\Omega$. (In (5.7) we have simplified the first equation by writing $\bot$ instead of the equivalent $\iota_\bot()$.)

By itself, this argument does not extend to the infinite sequences in $\Omega$, which cannot be obtained by applying the constructors any finite number of times. However, since every infinite sequence in $\Omega$ is a limit of finite sequences, requiring the solution to be continuous extends its uniqueness to the entirety of $\Omega$. (The domain $\Omega$ is an example of an initial continuous algebra, for which the above constructors are the operations and $f_\ast$ is a homomorphism.)

In a similar manner, the function $f_\ddagger$ defined by Equations (5.4) satisfies

\[
\begin{align*}
f_\ddagger\{} &= \{} \\
f_\ddagger\langle \sigma \rangle &= \langle f\sigma \rangle \\
f_\ddagger\langle \langle \text{abort}, \sigma \rangle \rangle &= \langle \langle \text{abort}, f\sigma \rangle \rangle \\
f_\ddagger(\langle n \rangle \circ \omega) &= \langle n \rangle \circ f_\ddagger\omega
\end{align*}
\]
or, in terms of the injections in Equations (5.6),

\[
\begin{align*}
\varphi \downarrow &= \bot \\
\varphi (\iota_{\text{term}}(\sigma)) &= \iota_{\text{term}}(f(\sigma)) \\
\varphi (\iota_{\text{abort}}(\sigma)) &= \iota_{\text{abort}}(f(\sigma)) \\
\varphi (\iota_{\text{out}}(n, \omega)) &= \iota_{\text{out}}(n, f(\omega)).
\end{align*}
\]

By a similar argument, \( \varphi \) can be shown to be the unique continuous solution of these equations.

Finally, we rewrite the semantic equations themselves in terms of the injections into \( \Omega \), rather than operations on sequences:

**DR SEM EQ: Assignment**

\[
[v := e]_{\text{comm}}(\sigma) = \iota_{\text{term}}[\sigma | v: [e]_{\text{intexp}}],
\]

**DR SEM EQ: skip**

\[
[[\text{skip}]]_{\text{comm}}(\sigma) = \iota_{\text{term}}(\sigma),
\]

**DR SEM EQ: fail**

\[
[[\text{fail}]]_{\text{comm}}(\sigma) = \iota_{\text{abort}}(\sigma),
\]

**DR SEM EQ: Output**

\[
[[!e]]_{\text{comm}}(\sigma) = \iota_{\text{out}}([e]_{\text{intexp}}(\sigma), \iota_{\text{term}}(\sigma)),
\]

**DR SEM EQ: Sequential Composition**

\[
[[c_0; c_1]]_{\text{comm}}(\sigma) = ([c_1]_{\text{comm}})\circ ([c_0]_{\text{comm}}),
\]

**DR SEM EQ: Variable Declaration**

\[
[\text{newvar } v := e \text{ in } c]_{\text{comm}}(\sigma) = (\lambda \sigma' \in \Sigma. [\sigma' | v: \sigma v]) \circ ([c]_{\text{comm}}[\sigma | v: [e]_{\text{intexp}}]),
\]

**DR SEM EQ: Conditional**

\[
[[\text{if } b \text{ then } c_0 \text{ else } c_1]]_{\text{comm}}(\sigma) = [b]_{\text{boolexp}}(\sigma) \text{ then } [c_0]_{\text{comm}}(\sigma) \text{ else } [c_1]_{\text{comm}}(\sigma),
\]

**DR SEM EQ: while**

\[
[[\text{while } b \text{ do } c]]_{\text{comm}} = \mathbf{Y}_{\Sigma \to \Omega} F
\]

where \( F(f)(\sigma) = \text{if } [b]_{\text{boolexp}}(\sigma) \text{ then } f(\sigma)([c]_{\text{comm}}(\sigma)) \text{ else } \iota_{\text{term}}(\sigma). \)
Here \( f_* \) and \( f_1 \) are the unique solutions of Equations (5.7) and (5.9).

In conclusion, some general remarks about output domains are appropriate. Although domains such as \( \Omega \) are needed to describe the output of physical processes, others are needed to describe the meanings of such output from the viewpoint of a user. For example, a common use of computations that can generate endless sequences of integers is the enumeration of sets of integers. Here the user is not interested in the order or the number of times each integer occurs in the physical output sequence \( s \), but only in the set \( \mu s \), where \( \mu \) maps each \( s \in \Omega \) into the set of integers that occur in \( s \). The sense of increasing information is given by the inclusion relation, so that the relevant domain is \( \mathcal{P} \mathbb{Z} \). It is easily seen that \( \mu \) is a continuous function from \( \Omega \) to \( \mathcal{P} \mathbb{Z} \).

These domains also provide illustrations of the notions of safety and liveness properties. A property defined over the elements of a domain is called a

- **safety property** if, whenever it holds for \( x \), it holds for all \( x' \subseteq x \), and, whenever it holds for a chain, it holds for the limit of the chain,
- **liveness property** if, whenever it holds for \( x \), it holds for all \( x' \supseteq x \).

Intuitively, a safety property insures that certain events will not occur, while a liveness property insures that certain events will occur.

For example, consider a computation in our language with intermediate output (starting from some fixed initial state). A typical safety property over \( \Omega \) is that only primes are output, that is, every integer occurring in the output \( s \) is prime. A typical liveness property is that every prime is output, that is, every prime occurs in \( s \). Equally well, one can define analogous properties over \( \mathcal{P} \mathbb{Z} \) and apply them to \( \mu s \): the safety property of a set that every member is prime, or the liveness property that every prime is a member.

As remarked in Section 3.1, the partial and total correctness specifications defined by Equations (3.1) and (3.2) are safety and liveness properties, respectively, of the meaning of the commands being specified.

### 5.3 The Physical Argument for Continuity

The domain \( \Omega \) of sequences introduced in the previous section is rich enough to illustrate clearly what we mean by an ordering of "increasing information": \( s \subseteq s' \) holds when observing the output \( s \) at some time during the computation is compatible with observing the output \( s' \) later (or with \( s' \) being the total output of the computation). Chains are "movies" of possible computations, that is, time-ordered sequences of snapshots of the output. When such a chain is interesting, it depicts a computation with endless output and provides enough information to determine the output completely.

This domain can be used to illustrate the basic argument as to why only continuous functions are physically realizable. Imagine an observer (either human...
or mechanical) whose input is the output of a computation of the kind we have just described, and whose own output is of a similar kind. This observer might be given tasks such as the following:

- Output one if you receive an infinite sequence of integers, or zero otherwise.
- Output one if you never receive the integer 37, or zero otherwise.
- Output one if you receive the digit expansion of a rational number, or zero otherwise.

In each case, as long as the input consists of, say, single-digit integers, there is never anything that the observer can safely output. Thus the above tasks, each of which describes the computation of a noncontinuous function, are impossible.

The general argument (for a domain like $\Omega$) is the following: Suppose that the task is to evaluate some function $f$ and, for a particular execution, $x_0 \subseteq x_1 \subseteq \cdots$ is a sequence of snapshots of the input at increasing times and $y_0 \subseteq y_1 \subseteq \cdots$ is a sequence of snapshots of the output at the same times. Then the ultimate input and output are

$$x = \bigcup_{n=0}^{\infty} x_n \quad \text{and} \quad y = \bigcup_{n=0}^{\infty} y_n,$$

and the function $f$ must satisfy $f(x) = y$.

To see that $f$ is monotonic, suppose $x \subseteq x'$, and consider an execution where $x$ is the limit of the input chain. At time $n$, all the observer knows about the input is that it is approximated by $x_n$, which holds for $x'$ as well as $x$. Thus the output $y_n$ must approximate $f(x_n)$ as well as $f(x)$. Since this situation holds for all $n$, $f(x')$ must be an upper bound on all of the $y_n$, and thus must extend the least upper bound $f(x)$.

To see that $f$ is also continuous, let $x_0 \subseteq x_1 \subseteq \cdots$ be any interesting chain, and suppose that this chain describes the input to a computation of $f$. At time $n$, all the observer knows about the input is that it is approximated by $x_n$, which holds for $x_n$ as well as $x$. Thus the output $y_n$ must approximate $f(x_n)$ and, since this situation holds for all $n$,

$$\bigcup_{n=0}^{\infty} y_n \subseteq \bigcup_{n=0}^{\infty} f(x_n).$$

But the left side here is $y = f(x) = f(\bigcup_{n=0}^{\infty} x_n)$.

Notice that this argument has nothing to do with the Turing concept of computability, which is based on the observer being controlled by a finite program. Instead, it is based on the physical limitations of communication: one cannot predict the future of input, nor receive an infinite amount of information in a finite amount of time, nor produce output except at finite times.
5.4 Products and Disjoint Unions of Predomains

To understand how domains such as $\Omega$ can be described by recursive domain isomorphisms, we must first extend the concepts of product and disjoint union (defined in Sections A.3 and A.5 of the Appendix) from sets to predomains. Although this generalization actually applies to products and disjoint unions over arbitrary sets, we limit our discussion to $n$-ary products and disjoint unions.

When $P_0, \ldots, P_{n-1}$ are predomains, we write $P_0 \times \cdots \times P_{n-1}$ for the partially ordered set obtained by equipping the Cartesian product of sets,

$$P_0 \times \cdots \times P_{n-1} = \{ \langle x_0, \ldots, x_{n-1} \rangle \mid x_0 \in P_0 \text{ and } \cdots \text{ and } x_{n-1} \in P_{n-1} \},$$

with the componentwise ordering

$$\langle x_0, \ldots, x_{n-1} \rangle \sqsubseteq \langle y_0, \ldots, y_{n-1} \rangle$$

if and only if $x_0 \sqsubseteq y_0$ and $\cdots$ and $x_{n-1} \sqsubseteq y_{n-1}$.

This ordering gives a predomain for which the limit of a chain of $n$-tuples is computed componentwise:

$$\bigsqcup_{k=0}^{\infty} \langle x_0^{(k)}, \ldots, x_{n-1}^{(k)} \rangle = \langle \bigsqcup_{k=0}^{\infty} x_0^{(k)}, \ldots, \bigsqcup_{k=0}^{n-1} x_{n-1}^{(k)} \rangle. \quad (5.10)$$

Moreover, if the $P_i$ are all domains, then $P_0 \times \cdots \times P_{n-1}$ is a domain whose least element is the $n$-tuple $\bot = \langle \bot_0, \ldots, \bot_{n-1} \rangle$.

If the $P_i$ are all discrete, then so is $P_0 \times \cdots \times P_{n-1}$; thus the componentwise ordering of products is compatible with the convention that sets are discretely ordered.

As with sets, we write $P^n$ for the product of $P$ with itself $n$ times.

The projection functions associated with the product are continuous, and the function constructors preserve continuity. In particular:

**Proposition 5.1** Suppose $P$, $Q$, and, for $0 \leq i \leq n-1$, $P_i$ and $Q_i$ are predomains. Then

(a) The projection functions $\pi_i$ satisfying $\pi_i(x_0, \ldots, x_{n-1}) = x_i$ are continuous functions from $P_0 \times \cdots \times P_{n-1}$ to $P_i$.

(b) If the $n$ functions $f_i \in P \to P_i$ are continuous, then the "target-tupling" function $f_0 \otimes \cdots \otimes f_{n-1}$ satisfying

$$(f_0 \otimes \cdots \otimes f_{n-1})x = \langle f_0 x, \ldots, f_{n-1} x \rangle$$

is a continuous function from $P$ to $P_0 \times \cdots \times P_{n-1}$.

(c) If the $n$ functions $f_i \in P_i \to Q_i$ are continuous, then the function $f_0 \times \cdots \times f_{n-1}$ satisfying

$$(f_0 \times \cdots \times f_{n-1})\langle x_0, \ldots, x_{n-1} \rangle = \langle f_0 x_0, \ldots, f_{n-1} x_{n-1} \rangle$$

is a continuous function from $P_0 \times \cdots \times P_{n-1}$ to $Q_0 \times \cdots \times Q_{n-1}$.
In a similar spirit, when $P_0, \ldots, P_{n-1}$ are predomains, we want to equip the disjoint union,

$$P_0 + \cdots + P_{n-1} = \{ \langle 0, x \rangle \mid x \in P_0 \} \cup \cdots \cup \{ \langle n-1, x \rangle \mid x \in P_{n-1} \},$$

with an appropriate ordering that will make it a predomain. We want each component $\{ \langle i, x \rangle \mid x \in P_i \}$ to have the same ordering as $P_i$, and members of distinct components to be incomparable. Thus we use the ordering

$$\langle i, x \rangle \sqsubseteq \langle j, y \rangle \text{ if and only if } i = j \text{ and } x \sqsubseteq_i y.$$

Any chain in the disjoint union must be a sequence of pairs with the same first component; the limit of such a chain is

$$\bigcup_{k=0}^\infty \langle i, x_k \rangle = \langle i, \bigcup_{k=0}^i x_k \rangle.$$

Even when all of the $P_i$ are domains, their disjoint union is only a predomain, since (except when $n = 1$) there is no least element.

As with the product construction, if the $P_i$ are all discrete, then so is $P_0 + \cdots + P_{n-1}$; thus the ordering of the disjoint union is compatible with the convention that sets are discretely ordered.

The injection functions associated with the disjoint union are continuous, and the function constructors preserve continuity. In particular:

**Proposition 5.2** Suppose $P$, $Q$, and, for $0 \leq i \leq n - 1$, $P_i$ and $Q_i$ are predomains. Then

(a) The injection functions $\iota_i$ satisfying

$$\iota_i(x) = \langle i, x \rangle$$

are continuous functions from $P_i$ to $P_0 + \cdots + P_{n-1}$.

(b) If the $n$ functions $f_i \in P_i \to P$ are continuous, then the “source-tupling” function $f_0 \oplus \cdots \oplus f_{n-1}$ satisfying

$$(f_0 \oplus \cdots \oplus f_{n-1})\langle i, x \rangle = f_i x$$

is a continuous function from $P_0 + \cdots + P_{n-1}$ to $P$.

(c) If the $n$ functions $f_i \in P_i \to Q_i$ are continuous, then the function $f_0 + \cdots + f_{n-1}$ satisfying

$$(f_0 + \cdots + f_{n-1})\langle i, x \rangle = \langle i, f_i x \rangle$$

is a continuous function from $P_0 + \cdots + P_{n-1}$ to $Q_0 + \cdots + Q_{n-1}$. 
5.5 Recursive Domain Isomorphisms

Pictorially, the domain $\Omega$ looks like this:

$\langle \sigma_0 \rangle \; \langle \sigma_1 \rangle \; \langle \sigma_2 \rangle \; \ldots$  
\[  \Omega \; \Omega \; \Omega \; \ldots \]

$\bot = \langle \rangle$

The least element is the empty sequence, the $\langle \sigma_i \rangle$ on the left are single-element sequences containing states or abort-state pairs, and each diamond on the right contains all sequences that begin with a particular integer. But then, since the sequences in $\Omega$ that begin with, say 17, are the sequences that are obtained by prefixing 17 to arbitrary sequences in $\Omega$, each diamond is isomorphic to $\Omega$ itself.

Now consider the product $\mathbb{Z} \times \Omega$, where $\mathbb{Z}$ is discretely ordered. This predomain consists of a copy of $\Omega$ for each $n \in \mathbb{Z}$, formed by pairing $n$ with the members of $\Omega$. Moreover, the componentwise ordering is

$\langle n, \omega \rangle \sqsubseteq \langle n', \omega' \rangle$ if and only if $n = n'$ and $\omega \sqsubseteq \omega'$,

so that each copy is ordered in the same way as $\Omega$ itself, and members of different copies are incomparable. Thus the collection of diamonds in the above diagram is isomorphic to $\mathbb{Z} \times \Omega$, and $\Omega$ itself is isomorphic to

$\hat{\Sigma} \quad \mathbb{Z} \times \Omega$

The least element is $\langle \rangle$.

To within an isomorphism, nothing in this diagram changes if we pair each member of $\hat{\Sigma}$ with zero and each member of $\mathbb{Z} \times \Omega$ with one, as long as we maintain the ordering within these components and keep members of distinct components incomparable. Thus $\Omega$ is isomorphic to $(\hat{\Sigma} + \mathbb{Z} \times \Omega)_\perp$, which we indicate by writing

$\Omega \approx (\hat{\Sigma} + \mathbb{Z} \times \Omega)_\perp$.

This means that there are continuous functions

$\Omega \xrightarrow{\phi} (\hat{\Sigma} + \mathbb{Z} \times \Omega)_\perp$

such that the compositions $\psi \cdot \phi$ and $\phi \cdot \psi$ are both identity functions.
The injections defined in Equations (5.6) in Section 5.2 can be redefined in terms of this isomorphism. We begin with the following diagram of functions, all of which are injections:

\[
\begin{array}{c}
\Sigma \\
\downarrow \iota_{\text{norm}} \\
\hat{\Sigma} \\
\downarrow \iota_0 \\
\Sigma + \mathbb{Z} \times \Omega \\
\downarrow \iota_{\text{abnorm}} \\
\mathbb{Z} \times \Omega
\end{array}
\xrightarrow{\iota_{\text{term}}} \hat{\Sigma} + \mathbb{Z} \times \Omega \xrightarrow{\iota_{\uparrow}} (\hat{\Sigma} + \mathbb{Z} \times \Omega)_{\perp} \xrightarrow{\psi} \Omega
\]

Here, \(\iota_{\text{norm}} \sigma = \sigma\), \(\iota_{\text{abnorm}} \sigma = (\text{abort}, \sigma)\), \(\iota_0\) and \(\iota_1\) are the injections associated with the disjoint union, \(\iota_{\uparrow}\) is the injection associated with lifting (defined in Section 2.3), and \(\psi\) is part of the isomorphism (and therefore a bijection). Then we define the compositions:

\[
\begin{align*}
\iota_{\text{term}} &= \psi \circ \iota_{\uparrow} \circ \iota_0 \circ \iota_{\text{norm}} \in \Sigma \to \Omega \\
\iota_{\text{abort}} &= \psi \circ \iota_{\uparrow} \circ \iota_0 \circ \iota_{\text{abnorm}} \in \Sigma \to \Omega \\
\iota_{\text{out}} &= \psi \circ \iota_{\uparrow} \circ \iota_1 \in \mathbb{Z} \times \Omega \to \Omega.
\end{align*}
\]

By using these definitions instead of Equations (5.6), we free our semantic equations from a dependency on the specific construction of \(\Omega\) as a domain of sequences of integers and states. (The fact that these functions, along with \(\iota_{\perp} \in \{()\} \to \Omega\) such that \(\iota_{\perp}() = \perp\), are continuous injections with disjoint ranges is enough to insure that they have the properties we need.)

The isomorphism \(\Omega \approx (\hat{\Sigma} + \mathbb{Z} \times \Omega)_{\perp}\) is our first encounter with an instance of a large class of domain isomorphisms that possess solutions. Specifically, if \(T\) is any function mapping domains into domains that is constructed from constant predomains and operations such as +, \(\times\), \(\to\), and \((-)_{\perp}\) (and others that will be introduced later), then there is a domain \(D\) satisfying the isomorphism \(D \approx T(D)\). Although we will repeatedly rely on this fact, the “inverse limit” construction that proves it is beyond the scope of this book.

This kind of domain isomorphism is similar to a fixed-point equation for an element of a domain, in that it always has at least one solution but often has more than one. One can even define a sense of “least” that singles out the solution given by the construction and insures that such equations as (5.7) and (5.9) in Section 5.2 have unique solutions. But even this definition of “least” (initiality in an appropriate category) is beyond the scope of this book.
5.6 Intermediate Input and a Domain of Resumptions

Next we consider extending the language of Section 5.2 with a command

\[
\langle \text{comm} \rangle ::= ?(\text{var})
\]

that causes an integer to be read from the input medium and to become the value of the operand. (We will give the operator \( ? \) the same precedence as the assignment operator \( := \).)

To see how this extension affects our semantics, suppose that applying the meaning of a program to an initial state produces some \( \omega \in \Omega \). Instead of thinking of \( \omega \) as an "output", we can think of it as describing the behavior of the process of executing the command. In the absence of input commands, when \( \Omega \approx (\Sigma + \mathbb{Z} \times \Omega)_\perp \) as in Section 5.5, there are four possibilities:

(a) If \( \omega = \bot \), the process executes forever without performing input or output.
(b) If \( \omega = \iota_{\text{term}} \sigma \), the process terminates normally in the state \( \sigma \).
(c) If \( \omega = \iota_{\text{abort}} \sigma \), the process aborts in the state \( \sigma \).
(d) If \( \omega = \iota_{\text{out}}(n, \omega') \), the process outputs the integer \( n \) and thereafter behaves as described by \( \omega' \).

To describe an input operation, we must introduce a fifth possibility, where behavior depending on the input is described by a function \( g \) that maps the input into a member of \( \Omega \):

(e) If \( \omega = \iota_{\text{in}} g \), where \( g \in \mathbb{Z} \rightarrow \Omega \), the process inputs an integer \( k \) and thereafter behaves as described by \( g k \).

To accommodate this extra alternative, we take \( \Omega \) to be a solution of

\[
\Omega \approx (\Sigma + (\mathbb{Z} \times \Omega) + (\mathbb{Z} \rightarrow \Omega))_\perp.
\]

A domain satisfying such an isomorphism is often called a domain of resumptions, since the occurrences of \( \Omega \) on the right refer to behavior when the process resumes execution after input or output. The new injection is the composition

\[
\iota_{\text{in}} = \psi \cdot \iota_1 \cdot \iota_2 \in (\mathbb{Z} \rightarrow \Omega) \rightarrow \Omega.
\]

When executed in the state \( \sigma \), the input command \( ? v \) yields a function (injected into \( \Omega \)) that maps an input \( k \) into the altered state \( [\sigma | v:k] \). Thus the new semantic equation is
The previous semantic equations in Section 5.2 remain unchanged, except that the definitions of $f_*$ by Equations (5.7) and $f_\dagger$ by Equations (5.9) must be augmented to deal with the new injection. In both cases, when the extended function is applied to $\iota_{\text{in}} g$, it is recursively applied pointwise, that is, to the results of $g$:

$$f_* \perp = \perp$$

$$f_*(\iota_{\text{term}} \sigma) = f \sigma$$

$$f_*(\iota_{\text{abort}} \sigma) = \iota_{\text{abort}} \sigma \tag{5.11}$$

$$f_*(\iota_{\text{out}} (n, \omega)) = \iota_{\text{out}} (n, f_* \omega)$$

$$f_*(\iota_{\text{in}} g) = \iota_{\text{in}} (\lambda k \in \mathbb{Z}. f_*(g k)),$$

$$f_\dagger \perp = \perp$$

$$f_\dagger(\iota_{\text{term}} \sigma) = \iota_{\text{term}} (f \sigma)$$

$$f_\dagger(\iota_{\text{abort}} \sigma) = \iota_{\text{abort}} (f \sigma) \tag{5.12}$$

$$f_\dagger(\iota_{\text{out}} (n, \omega)) = \iota_{\text{out}} (n, f_\dagger \omega)$$

$$f_\dagger(\iota_{\text{in}} g) = \iota_{\text{in}} (\lambda k \in \mathbb{Z}. f_\dagger (g k)).$$

As an example, consider the command

$$x := 0 \ ; \ \textbf{while true do} \ (? y \ ; \ x := x + y \ ; ! x).$$

By repeated application of the semantic equations and the above equations for $f_*$, one obtains

$$[x := 0 \ ; \ \textbf{while true do} \ (? y \ ; \ x := x + y \ ; ! x)]_{\text{comm}} \sigma$$

$$= ([\textbf{while true do} \ (? y \ ; \ x := x + y \ ; ! x)]_{\text{comm}})_* ([x := 0]_{\text{comm}} \sigma)$$

$$= ([\textbf{while true do} \ (? y \ ; \ x := x + y \ ; ! x)]_{\text{comm}})_* (\iota_{\text{term}} [\sigma | x: 0])$$

$$= [\textbf{while true do} \ (? y \ ; \ x := x + y \ ; ! x)]_{\text{comm}} [\sigma | x: 0]$$

$$= (Y_{\Sigma \rightarrow \Omega} F)[\sigma | x: 0]$$

$$= \left( \bigcup_{n=0}^{\infty} \Omega F^n(\perp) \right)[\sigma | x: 0]$$

$$= \bigcup_{n=0}^{\infty} F^n(\perp)[\sigma | x: 0],$$
where

\[
F(f)\sigma = \text{if } [\text{true}]_{\text{boolexp}}\sigma \text{ then } f_*(\llbracket ?y ; x := x + y ; !x \rrbracket_{\text{comm}}\sigma) \text{ else } \iota_{\text{term}}\sigma
\]

\[
= f_*(\llbracket ?y ; x := x + y ; !x \rrbracket_{\text{comm}}\sigma)
\]

\[
= f_*(\llbracket [x := x + y ; !x]_{\text{comm}} \rrbracket_*(\llbracket [?y]_{\text{comm}}\rrbracket))
\]

\[
= f_*(\llbracket [x := x + y ; !x]_{\text{comm}} \rrbracket_*(\iota_{\text{term}}(\lambda k \in \mathbb{Z}. \iota_{\text{term}}(\sigma | y : k))))
\]

\[
= f_*(\iota_{\text{term}}(\lambda k \in \mathbb{Z}. \llbracket [x := x + y ; !x]_{\text{comm}} \rrbracket_*(\iota_{\text{term}}(\sigma | y : k))))
\]

\[
= f_*(\iota_{\text{term}}(\lambda k \in \mathbb{Z}. \llbracket [x := x + y ; !x]_{\text{comm}} \rrbracket_*(\sigma | y : k))
\]

\[
= f_*(\iota_{\text{term}}(\lambda k \in \mathbb{Z}. \iota_{\text{out}}(\sigma x + k, \iota_{\text{term}}(\sigma | y : k))\sigma x + k)))
\]

\[
= f_*(\iota_{\text{term}}(\lambda k \in \mathbb{Z}. f_*(\iota_{\text{out}}(\sigma x + k, \iota_{\text{term}}(\sigma | y : k))\sigma x + k)))
\]

\[
= f_*(\iota_{\text{term}}(\lambda k \in \mathbb{Z}. \iota_{\text{out}}(\sigma x + k, f_*(\iota_{\text{term}}(\sigma | y : k))\sigma x + k))\).
\]

From this result, it is straightforward to show by induction on \(n\) that

\[
F^n(\bot)[\sigma | x : 0] = \iota_{\text{term}}(\lambda k_0 \in \mathbb{Z}. \iota_{\text{out}}(k_0, \iota_{\text{term}}(\lambda k_1 \in \mathbb{Z}. \iota_{\text{out}}(k_0 + k_1, \ldots \iota_{\text{term}}(\lambda k_{n-1} \in \mathbb{Z}. \iota_{\text{out}}(k_0 + \cdots + k_{n-1}, \bot) \ldots))))),
\]

which has the limit

\[
[\llbracket x := 0 ; \text{while true do } (?y ; x := x + y ; !x) \rrbracket_{\text{comm}}\sigma
\]

\[
= \bigcap_{n=0}^{\infty} F^n(\bot)[\sigma | x : 0]
\]

\[
= \iota_{\text{term}}(\lambda k_0 \in \mathbb{Z}. \iota_{\text{out}}(k_0, \iota_{\text{term}}(\lambda k_1 \in \mathbb{Z}. \iota_{\text{out}}(k_0 + k_1, \ldots \iota_{\text{term}}(\lambda k_2 \in \mathbb{Z.} \iota_{\text{out}}(k_0 + k_1 + k_2, \ldots)))))).
\]

### 5.7 Continuation Semantics

One can argue that the denotational semantics given in this chapter is unnatural. A particularly disturbing symptom is that, when introducing language features that seem unrelated to the sequencing of commands, we have repeatedly had to complicate the definition of command sequencing by changing the definition of the extension \(f_*.\) By the last section, \(f_*\) has become an elaborate function that is a far cry from the way any implementor would think about command sequencing.

Fundamentally, the composition \((\llbracket c_1 \rrbracket_{\text{comm}})_* \cdot \llbracket c_0 \rrbracket_{\text{comm}}\), where \(\llbracket c_1 \rrbracket_{\text{comm}}\) determines the result of the computation except as it is constrained by the operation \((-)_*,\) does not reflect the way in which \(c_0 ; c_1\) is actually implemented on a computer, where \(c_0\) has complete control over the computation until if and when it
passes control to \( c_1 \). The latter situation would be more closely mirrored by the opposite composition \([c_0]_{\text{comm}} \cdot [c_1]_{\text{comm}}\), where \([c_0]_{\text{comm}}\) would be a constant function if \( c_0 \) never relinquished control.

The key to achieving such a semantics is to make the meaning of every command a function whose result is the final result of the entire program, and to provide an extra argument to the command meaning, called a \textit{continuation}, that is a function from states to final results describing the behavior of the "rest of the program" that will occur if the command relinquishes control. Thus the semantic function for commands has the type

\[
[-]_{\text{comm}} \in \langle \text{comm} \rangle \rightarrow (\Sigma \rightarrow \Omega) \rightarrow \Sigma \rightarrow \Omega,
\]

where \( \Omega \) is the domain of final results (which we leave unspecified for the moment) and \( \Sigma \rightarrow \Omega \) is the domain of continuations. If the command \( c \), when executed in the state \( \sigma \), never relinquishes control, then the final result \([c]_{\text{comm}} \kappa \sigma \) is independent of the continuation \( \kappa \); if it produces a state \( \sigma' \) and relinquishes control, then \([c]_{\text{comm}} \kappa \sigma \) is \( \kappa \sigma' \).

A denotational semantics of this kind is called a \textit{continuation} semantics, as opposed to the kind of semantics discussed heretofore, which is often called \textit{direct} semantics. We will begin by giving a continuation semantics for the simple imperative language of Chapter 2, and then we will consider the various extensions discussed in this chapter.

Assignment and \texttt{skip} commands always relinquish control, so the final result is obtained by applying the continuation argument to the appropriate state. (We write \texttt{CN SEM EQ} to abbreviate "continuation semantic equation".)

**CN SEM EQ: Assignment**

\[
[u := e]_{\text{comm}} \kappa \sigma = \kappa [\sigma | u : [e]_{\text{intexp}} \sigma],
\]

**CN SEM EQ: \texttt{skip}**

\[
[\texttt{skip}]_{\text{comm}} \kappa \sigma = \kappa \sigma.
\]

(Note that, although we are changing from a direct to a continuation semantics of commands, the semantics of expressions remains the same. A continuation semantics for expressions will be introduced, in the context of functional languages, in Section 12.1.)

The composition \( c_0 ; c_1 \) is more subtle. Since \( c_0 \) gains control first, the final result given by \([c_0 ; c_1]_{\text{comm}}\) when applied to a continuation \( \kappa \) and a state \( \sigma \) should be the final result given by \([c_0]_{\text{comm}}\) when applied to some continuation \( \kappa' \) and the initial state \( \sigma \). If \( c_0 \) relinquishes control in a state \( \sigma' \), the final result will be \( \kappa' \sigma' \), which should be given by \([c_1]_{\text{comm}}\) applied to some continuation \( \kappa'' \) and the intermediate state \( \sigma' \). Then, if \( c_1 \) relinquishes control in a state \( \sigma'' \), the final result will be \( \kappa'' \sigma'' \). But when \( c_1 \) relinquishes control the entire composition will
have finished execution, so that the continuation \( \kappa'' \) should be the continuation \( \kappa \) to which \([c_0 ; c_1]_{\text{comm}}\) was applied. Thus

\[
[c_0 ; c_1]_{\text{comm}}^{\kappa \sigma} = [c_0]_{\text{comm}}(\lambda \sigma' \in \Sigma. [c_1]_{\text{comm}}^{\kappa \sigma'})\sigma.
\]

The apparent complexity of this equation vanishes from a more abstract viewpoint. If \( f \sigma = g \sigma \) holds for all \( \sigma \in \Sigma \), then \( f = g \). Applying this fact to both \( \sigma \) and \( \sigma' \) in the above equation gives

**CN SEM EQ: Sequential Composition**

\[
[c_0 ; c_1]_{\text{comm}}^{\kappa} = [c_0]_{\text{comm}}([c_1]_{\text{comm}}^{\kappa})
\]

or, as suggested earlier,

\[
[c_0 ; c_1]_{\text{comm}} = [c_0]_{\text{comm}} \cdot [c_1]_{\text{comm}}.
\]

Conditionals are more straightforward. Depending on whether the initial state satisfies \( b \), one or the other of the subcommands is executed in the initial state; if it relinquishes control, the rest of the computation is the same as the rest of the computation after the entire conditional. Thus

**CN SEM EQ: Conditional**

\[
[c_0 ; c_1]_{\text{comm}}^{\kappa \sigma} = [c_0]_{\text{comm}}([c_1]_{\text{comm}}^{\kappa \sigma})
\]

Just as with direct semantics, a semantic equation for the \textbf{while} command can be obtained by applying the fixed-point theorem to an “unwinding” equation:

\[
[\text{while } b \text{ do } c]_{\text{comm}}^{\kappa \sigma}
\]

\[
= [\text{if } b \text{ boolean } \sigma \text{ then } [c_0]_{\text{comm}}^{\kappa \sigma} \text{ else } [c_1]_{\text{comm}}^{\kappa \sigma}]
\]

This equation expresses \([\text{while } b \text{ do } c]_{\text{comm}}^{\kappa} \) as a function of itself, specifically as a fixed point of type \( (\Sigma \rightarrow \Omega) \rightarrow \Sigma \rightarrow \Omega \). However, the equation has the special property that every occurrence of \([\text{while } b \text{ do } c]_{\text{comm}}^{\kappa} \) is applied to the same continuation \( \kappa \), so that the equation also expresses \([\text{while } b \text{ do } c]_{\text{comm}}^{\kappa} \) as a function of itself. This fixed-point characterization is simpler because it has the simpler type \( \Sigma \rightarrow \Omega \):

**CN SEM EQ: while**

\[
[\text{while } b \text{ do } c]_{\text{comm}}^{\kappa} = Y_{\Sigma \rightarrow \Omega} F,
\]

where, for all \( w \in \Sigma \rightarrow \Omega \) and \( \sigma \in \Sigma \),

\[
Fw\sigma = \text{if } [b]_{\text{boolean } \sigma} \text{ then } [c]_{\text{comm}}^{w \sigma} \text{ else } \kappa \sigma.
\]
(We can take the fixed point at the lower type because, operationally, the "rest of the program" is the same after each execution of the while body. This is what distinguishes a loop from an arbitrary recursion.)

Finally, we have the semantic equation for variable declarations, where the continuation restores the declared variable to its initial value:

\[
\text{CN SEM EQ: Variable Declaration} \\
\text{[newvar } v := e \text{ in } c\]_\text{comm}^{K_\sigma} = [c]_\text{comm}^{(\lambda \sigma' \in \Sigma. \ k[\sigma' | v; \sigma v]) | \sigma | v: \text{intexp } \sigma].
\]

It can be shown that this continuation semantics bears the following relationship to the direct semantics given in Chapter 2: For all commands \( c \) of the (unextended) simple imperative language, continuations \( K \), and states \( \sigma \),

\[
[c]_\text{continuation}^{K_\sigma} = K_{\Sigma_{\text{direct}}}([c]_\text{comm}^{\sigma}).
\]

In particular, one can take the domain \( \Omega \) of final results to be \( \Sigma_{\perp} \), and \( k \) to be the "final continuation" that injects \( \Sigma \) into \( \Sigma_{\perp} \); then \( K_{\Sigma_{\perp}} \) is the identity function on \( \Sigma_{\perp} \), so that the continuation semantics applied to the final continuation and an initial state coincides with the direct semantics applied to the same initial state.

In fact, the equation displayed above holds for any choice of the domain \( \Omega \). (In other words, it is "polymorphic" in \( \Omega \).) The nature of \( \Omega \) will become specific, however, when we consider extensions to the simple imperative language.

\section*{5.8 Continuation Semantics of Extensions}

The various language features introduced in this chapter can be described by continuation semantics much more simply than by direct semantics. In particular, there is no need to introduce the function extensions \( f_* \) and \( f_\Sigma \).

However, we must specify the domain \( \Omega \) of "final results". In particular, we take \( \Omega \) to be the domain of resumptions defined in Section 5.6 (so that "total behavior" might be a more accurate term than "final result"). We also take

\[
\iota_{\text{term}} \in \Sigma \rightarrow \Omega \quad \iota_{\text{abort}} \in \Sigma \rightarrow \Omega \quad \iota_{\text{out}} \in \mathbb{Z} \times \Omega \rightarrow \Omega \quad \iota_{\text{in}} \in (\mathbb{Z} \rightarrow \Omega) \rightarrow \Omega
\]

to be the injections defined in Sections 5.6 and 5.5.

Now consider the fail command. Essentially, its final result is the state in which it is executed — which is clearly expressed by the semantic equation

\[
[f\text{ail}]_\text{comm}^{K_\sigma} = \iota_{\text{abort}}^{\Sigma}. \]

Here, the fact that \([f\text{ail}]_\text{comm}^{K_\sigma}\) is independent of the behavior \( k \) of the rest of the program makes it obvious that fail does not execute the rest of the program.

Unfortunately, however, by taking the final result to be the current state, we raise the problem with variable declarations that was discussed at the end of
Section 5.1. When an occurrence of fail within the scope of a variable declaration is executed, if the current state is taken to be the final result without resetting the declared variable to its more global value, then variable renaming will not preserve meaning.

We will describe the solution to this problem at the end of this section, but first we consider the simpler problem of treating intermediate output and input.

The final result of a program that executes the output command \( !e \) consists of the value of \( e \) followed by the final result of the rest of the program. This is captured by the semantic equation

\[
\text{CN SEM EQ: Output} \\
[!e]_{\text{comm}}^{\kappa \sigma} = \iota_{\text{out}}([e]_{\text{inexp}}^{\sigma}, \kappa \sigma).
\]

On the other hand, the final result of a program that executes the input command \( ?v \) is a function mapping the input integer into the final result for an appropriately altered state. Thus

\[
\text{CN SEM EQ: Input} \\
[?v]_{\text{comm}}^{\kappa \sigma} = \iota_{\text{in}}(\lambda k \in \mathbb{Z}. \kappa[\sigma \mid v; k]).
\]

It can be shown that this continuation semantics bears the following relationship to the direct semantics described earlier in this chapter (provided that the direct semantics of variable declarations is defined by Equation (5.1) in Section 5.1, even though the preservation of meaning under renaming is violated):

\[
[\ell]_{\text{comm}}^{\kappa \sigma} = \kappa_*([\ell]_{\text{comm}}^{\text{direct}}),
\]

where \((\cdot)_*\) is defined by Equations (5.11) in Section 5.6. As a special case, where \(\kappa\) is taken to be the “final” continuation \(\iota_{\text{term}}\), we have

\[
[c]_{\text{continuation}}^{\text{direct}} \iota_{\text{term}} \sigma = [c]_{\text{comm}}^{\text{direct}} \sigma,
\]

since \((\iota_{\text{term}})_*\) can be shown to be the identity function on \(\Omega\).

Finally, we return to the problem of resetting local variables in failure states to preserve meaning under the renaming of variables. The solution here is a global, though straightforward, change in the semantics: The meanings of commands become functions accepting two continuations: \(\kappa_f\), which is used to map a state into a final result if the command terminates normally in that state, and \(\kappa_f\), which is used to map a state into a final result if the command fails in that state. Then the new continuation \(\kappa_f\) is used to reset a local variable when a fail occurring within the scope of a variable declaration is executed.

Thus the continuation semantics of commands satisfies

\[
[\ell]_{\text{comm}} \in (\text{comm}) \rightarrow (\Sigma \rightarrow \Omega) \rightarrow (\Sigma \rightarrow \Omega) \rightarrow \Sigma \rightarrow \Omega,
\]

and the relevant semantic equations are
Failure, Input-Output, and Continuations

CN SEM EQ: Assignment
\[ [v := e]_{\text{comm}}^{\kappa_t \kappa_f \sigma} = \kappa_t[\sigma | v: [e]_{\text{intexp}}] \],

CN SEM EQ: skip
\[ [\text{skip}]_{\text{comm}}^{\kappa_t \kappa_f \sigma} = \kappa_t \sigma \],

CN SEM EQ: Sequential Composition
\[ [c_0 ; c_1]_{\text{comm}}^{\kappa_t \kappa_f \sigma} = [c_0]_{\text{comm}}^{\kappa_t \kappa_f \sigma} ([c_1]_{\text{comm}}^{\kappa_t \kappa_f \sigma}) \]

CN SEM EQ: Conditional
\[ [\text{if } b \text{ then } c_0 \text{ else } c_1]_{\text{comm}}^{\kappa_t \kappa_f \sigma} = \text{if } [b]_{\text{boolexp}}^{\sigma} \text{ then } [c_0]_{\text{comm}}^{\kappa_t \kappa_f \sigma} \text{ else } [c_1]_{\text{comm}}^{\kappa_t \kappa_f \sigma} \],

CN SEM EQ: while
\[ [\text{while } b \text{ do } c]_{\text{comm}}^{\kappa_t \kappa_f \sigma} = Y_{\Sigma \to \Omega} F \]
where \( F w \sigma = \text{if } [b]_{\text{boolexp}}^{\sigma} \text{ then } [c]_{\text{comm}}^{\kappa_t \sigma} \text{ else } \kappa_t \sigma \),

CN SEM EQ: Variable Declaration
\[ [\text{newvar } v := e \text{ in } c]_{\text{comm}}^{\kappa_t \kappa_f \sigma} = [c]_{\text{comm}}^{(\lambda \sigma' \in \Sigma. \kappa_t(\sigma' | v: \sigma v)) (\lambda \sigma' \in \Sigma. \kappa_f(\sigma' | v: \sigma v))[\sigma | v: [e]_{\text{intexp}}]} \],

CN SEM EQ: fail
\[ [\text{fail}]_{\text{comm}}^{\kappa_t \kappa_f \sigma} = \kappa_f \sigma \]

CN SEM EQ: Output
\[ ![e]_{\text{comm}}^{\kappa_t \kappa_f \sigma} = \iota_{\text{out}}( [e]_{\text{intexp}}^{\sigma}, \kappa_t \sigma) \],

CN SEM EQ: Input
\[ [? v]_{\text{comm}}^{\kappa_t \kappa_f \sigma} = \iota_{\text{in}}(\lambda k \in \mathbb{Z}. \kappa_t(\sigma | v: k)) \]

It can be shown that this continuation semantics bears the following relationship to the direct semantics described earlier (provided that variable declarations are defined by Equation (5.2) in Section 5.1, so that renaming preserves meaning):
\[ [c]_{\text{comm}}^{\kappa_t \kappa_f \sigma} = (\kappa_t, \kappa_f)_*( [c]_{\text{direct}}^{\sigma}) \].
where \((\kappa_t, \kappa_f)_*\) satisfies
\[
(\kappa_t, \kappa_f)_* \perp = \perp \\
(\kappa_t, \kappa_f)_* (\iota_{\text{term}} \sigma) = \kappa_t \sigma \\
(\kappa_t, \kappa_f)_* (\iota_{\text{abort}} \sigma) = \kappa_f \sigma \\
(\kappa_t, \kappa_f)_* (\iota_{\text{out}} (n, \omega)) = \iota_{\text{out}} (n, (\kappa_t, \kappa_f)_* \omega) \\
(\kappa_t, \kappa_f)_* (\iota_{\text{in}} g) = \iota_{\text{in}} (\lambda k \in \mathbf{Z}. (\kappa_t, \kappa_f)_* (g k)).
\]

It can also be shown that \((\iota_{\text{term}}, \iota_{\text{abort}})_*\) is the identity function on \(\Omega\). Thus, as a special case,
\[
\llbracket c \rrbracket_{\text{continuation}} \iota_{\text{term}} \iota_{\text{abort}} \sigma = \llbracket c \rrbracket_{\text{direct}} \sigma,
\]
where \(\iota_{\text{term}}\) and \(\iota_{\text{abort}}\) can be thought of as the final continuations for normal and abortive termination.

**Bibliographic Notes**

The concept of continuous algebras was developed by Goguen, Thatcher, Wagner, and Wright [1977].

There are a variety of approaches to solving domain isomorphisms. The earliest was the inverse limit construction, which was originally devised by Scott [1971; 1972], generalized by Reynolds [1972b], and eventually generalized and abstracted much further by Wand [1975; 1979] and by Smyth and Plotkin [1982]. More elementary accounts are given in Tennent [1991, Chapter 10], Schmidt [1986, Chapter 11], and Gunter [1992, Section 10.1].

A later method, also due to Scott [1976], expressed the solution of a domain isomorphism as the set of fixed points of an idempotent function (i.e. satisfying \(f = f \cdot f\)) on a “universal” domain. This method comes in three flavors, depending on whether one uses arbitrary idempotent functions (called retractions) or restricts such functions to be approximations of the identity function (called projections) or extensions of the identity function (called closures). It is described in Gunter and Scott [1990, Section 6] and Gunter [1992, Section 8.2].

Still more recently, Larsen and Winskel [1991] developed a method using Scott’s [1982] information systems. This approach is also described in Winskel [1993, Chapter 12].

Resumptions were introduced (in the more complex setting of concurrent computation) by Plotkin [1976] and Hennessy and Plotkin [1979].

Most of the literature on continuations discusses the concept in the setting of functional languages (where we will return to continuations in Section 12.1). However, the properties of continuation semantics for imperative languages are described, perhaps to excess, by Reynolds [1977].
Both the various extension operations, such as \((-)_{\perp}\) and \((-)_{*}\), and the use of continuations are special cases of a general treatment of computational effects as monads that was devised by Moggi [1991] in the setting of functional languages. More intuitive descriptions, also in the functional setting, have been given by Wadler [1992; 1993].

Exercises

5.1 Let \(c\) be the command

\[
\text{while } x \neq 0 \text{ do if } x = 1 \text{ then fail else } x := x - 2.
\]

For simplicity, assume that the set \langle var \rangle contains only \(x\), so that \([x: m]\) is a typical state.

(a) What is \([c]_{\text{comm}}[x: m]\), where \([-]_{\text{comm}}\) is the direct semantic function defined in Chapter 2 and extended in Section 5.1?

(b) What is \([c]_{\text{comm} \kappa f}[x: m]\), where \([-]_{\text{comm}}\) is the two-continuation semantic function defined in Section 5.8?

(c) Derive the answer to part (a) from the relevant semantic equations.

(d) Give preconditions such that

\[
\{}?{c}\{x = 0\} \quad \{}?{c}\{x = 5\} \quad \{}?{c}\{x = 0\}.
\]

Your preconditions should be as weak as possible.

(e) Prove the last specification in part (d).

5.2 Extend the continuation semantics of Section 5.7, and also the more elaborate continuation semantics of Section 5.8, by giving semantic equations for the \texttt{repeat} command described in Exercise 2.2.

5.3 Consider extending the simple imperative language (including \texttt{fail}) by adding a new command,

\[
\langle \text{comm} \rangle := \texttt{catchin} \ (\text{comm}) \ \textbf{with} \ (\text{comm})
\]

that resumes computation after failure. Specifically, \texttt{catchin c0 with c1} causes \(c_0\) to be executed, and, if \(c_0\) terminates normally (that is, without any failure that is not caught by a lower-level occurrence of \texttt{catchin}), then \texttt{catchin c0 with c1} terminates normally without executing \(c_1\). But if \(c_0\) terminates with an uncaught failure, then \(c_1\) is executed (beginning in the state that resulted from the execution of \(c_0\)) and \texttt{catchin c0 with c1} terminates normally or fails, depending on whether \(c_1\) terminates normally or fails.

(a) Extend the direct semantics of Section 5.1 to describe this extension. The type of meaning of commands should remain the same.
(b) Extend the continuation semantics of Section 5.8 (using two continuations $\kappa_t$ and $\kappa_f$) to describe this extension. Again, the type of meaning of commands should remain the same.

5.4 Consider a further extension of the simple imperative language where failures have labels:

\[ (\text{comm}) := \text{fail} \{\text{label}\} \mid \text{catch} \{\text{label}\} \text{ in } \text{comm} \mid \text{with} \text{comm} \]

(where $\{\text{label}\}$ denotes a countably infinite predefined set of labels). Now $\text{fail} \{\ell\}$ causes a failure named $\ell$. The command $\text{catch} \{\ell\} \text{ in } c_0 \text{ with } c_1$ causes $c_0$ to be executed, and, if $c_0$ terminates normally or with an uncaught failure with a name different than $\ell$, then $\text{catch} \{\ell\} \text{ in } c_0 \text{ with } c_1$ terminates normally or with a similarly named failure, without executing $c_1$. But if $c_0$ terminates with an uncaught failure named $\ell$, then $c_1$ is executed (beginning in the state that resulted from the execution of $c_0$) and $\text{catch} \{\ell\} \text{ in } c_0 \text{ with } c_1$ terminates normally or fails, depending on whether $c_1$ terminates normally or fails. For example,

\[
\text{catch cold in catch cold in } (x := 0 ; \text{fail cold}) \text{ with } y := 0 \text{ with } z := 0
\]

is equivalent to $x := 0 ; y := 0$, but

\[
\text{catch cold in catch flu in } (x := 0 ; \text{fail cold}) \text{ with } y := 0 \text{ with } z := 0
\]

is equivalent to $x := 0 ; z := 0$.

(a) Extend the direct semantics of Section 5.1 to describe this language extension. The type of command meanings should still be $[-]_{\text{comm}} \in \langle \text{comm}\rangle \rightarrow \Sigma \rightarrow \hat{\Sigma}_\bot$, but now

\[
\hat{\Sigma} = \Sigma \cup (\{\text{label}\} \times \Sigma),
\]

where $\langle \ell, \sigma \rangle$ is the result of a command that fails with label $\ell$.

(b) Extend the continuation semantics of Section 5.8 to describe this language extension. The type of command meanings should now be

\[
[-]_{\text{comm}} \in \langle \text{comm}\rangle \rightarrow (\Sigma \rightarrow \Omega) \rightarrow ((\{\text{label}\} \rightarrow \Sigma \rightarrow \Omega) \rightarrow \Sigma \rightarrow \Omega).
\]

Here the meaning of a command $c$ is applied to a continuation $\kappa_t$ that describes the "rest of the computation" to be done in case $c$ completes execution normally, and to a function $\kappa_f$ such that $\kappa_f \ell$ describes the rest of the computation to be done in case $c$ completes execution with a failure named $\ell$. The domain $\Omega$ of final results is still the domain of resumptions used in Sections 5.6 to 5.8, except that now $\hat{\Sigma} = \Sigma \cup (\{\text{label}\} \times \Sigma)$, where a final result $\langle \ell, \sigma \rangle$ arises when the program terminates with an uncaught failure named $\ell$. 

---


http://dx.doi.org/10.1017/CBO9780511626364.006

5.5 Use either the direct or continuation semantics that you defined in Exercise 5.4 to prove that, for all commands \(c\) and labels \(\ell\),

\[
\left[\text{catch } \ell \text{ in } c \text{ with fail } \ell\right]_{\text{comm}} = \left[c\right]_{\text{comm}}.
\]

5.6 In Section 5.1, we redefined program specifications to treat failure in the same way as nontermination. This makes it impossible to assert anything about the state in which a failure occurs; as a consequence, these forms of specification are inadequate for reasoning about the \text{catchin} commands introduced in Exercise 5.3.

To avoid these limitations, one can work with a more elaborate kind of specification containing two consequent. These new specifications have the form \([p]c[q][r]\), where \(q\) describes the final states in which normal termination occurs, while \(r\) describes the final states in which failure occurs. More precisely (for determinate commands with no input or output),

\[
[p]c[q][r] = \forall \sigma \in \Sigma. [p]_{\text{assert}} \sigma \Rightarrow ([c]_{\text{comm}} \sigma \neq \bot \text{ and }
((\exists \sigma' \in \Sigma. [c]_{\text{comm}} \sigma = \sigma' \text{ and } [q]_{\text{assert}} \sigma')
\]

or \((\exists \sigma' \in \Sigma. [c]_{\text{comm}} \sigma = (\text{abort}, \sigma') \text{ and } [r]_{\text{assert}} \sigma'))\).

Give inference rules for inferring specifications of this new kind for assignment commands, \text{skip}, \text{fail}, sequential composition, \text{catchin}, and conditional commands.

5.7 Suppose \(f \in \Sigma \rightarrow \Omega\) and let \(f_* \in \Omega \rightarrow \Omega\) be the function defined by Equations 5.3 in Section 5.2. Prove that \(f_*\) is continuous.

5.8 Let \(S\) be the domain of finite and infinite sequences of integers with \(x \sqsubseteq y\) if and only if \(x\) is a prefix of \(y\), that is, if there is a sequence \(z\) such that \(y\) is the concatenation of \(x\) and \(z\). Let \(\mathcal{P} \mathbb{Z}\) be the powerset of the integers, ordered under inclusion. Prove that the function that maps a sequence of integers into the set of its elements is a continuous function from \(S\) to \(\mathcal{P} \mathbb{Z}\). (Note that this function is essentially a simplification of the function \(\mu\) described in Section 5.2.)

5.9 Prove that the composition function \(\text{cmp}\) such that

\[
\text{cmp}(f, g) = g \cdot f
\]

is a continuous function from \((P \rightarrow P') \times (P' \rightarrow P'')\) to \(P \rightarrow P''\).

5.10 Suppose we have domains \(D\) and \(D'\) and a pair of monotone functions \(\phi \in D \rightarrow D'\) and \(\psi \in D' \rightarrow D\) that satisfy the equations for an isomorphism:

\[
\psi \cdot \phi = I_D \quad \text{and} \quad \phi \cdot \psi = I_{D'}.
\]

Show that \(\phi\) and \(\psi\) are strict and continuous.
5.11 In Section A.3 of the Appendix, it is stated that

\[ S + \cdots + S = n \times S, \]

where \( n \) is interpreted as the set 0 to \( n - 1 \). Show that this equality remains true when \( S \) is an arbitrary predomain, provided that 0 to \( n - 1 \) is discretely ordered.