Chapter 12

Nondeterminism and Concurrency

A program is *deterministic* if its evaluations on the same input always produce the same output. The evaluation strategy for a deterministic program might not be unique. For example, side effect-free arithmetic addition can be implemented in more than one fashion:

1. Evaluate the left operand; evaluate the right operand; add.
2. Evaluate the right operand; evaluate the left operand; add.
3. Evaluate the two operands in parallel; add.

A program is *nondeterministic* if it has more than one allowable evaluation strategy and different evaluation strategies lead to different outputs. One example of a nondeterministic construct is addition *with* side effects, using the three evaluation strategies listed above. If an operand contains a side effect, then the order of evaluation of the operands can affect the final result. This situation is considered a result of bad language design, because elementary arithmetic is better behaved. It is somewhat surprising that the situation is typically resolved by outlawing all but one of the allowable evaluation strategies and embracing hidden side effects!

There are situations where nondeterminism is acceptable. Consider an error-handling routine that contains a number of commands, each indexed by a specific error condition. If a run-time error occurs, the handler is invoked to diagnose the problem and to compensate for it. Perhaps the diagnosis yields multiple candidate error conditions. Only one correction command is executed within the handler, so the choice of which one to use may be made nondeterministically.

A concept related to nondeterminism is *parallel evaluation*. Some language constructs can be naturally evaluated in parallel fashion, such as side effect-free addition using the third strategy noted above. This ''nice'' form of parallelism, where the simultaneous evaluation of subparts of the construct do not interact, is called *noninterfering parallelism.* In *interfering parallelism,* there is interaction, and the relative speeds of the evaluations of the subparts do affect the final result. We call a *concurrent language* one that uses interfering parallelism in its evaluation of programs. The classic example of a concurrent language is an imperative language that evaluates in parallel commands that share access and update rights to a common variable.

We require new tools to specify the semantics of nondeterministic and concurrent languages. A program's answer denotation is no longer a single value d from a domain D, but a set of values $\{ d_0, d_1, \cdots, d_i, \cdots \}$ describing all the results possible from the different evaluations. The set of values is an element from the *powerdomain* $\mathbb{P}(D)$. The powerdomain corresponds to the powerset in Chapter 2, but the underlying mathematics of the domain-based version is more involved. The members of $\mathbb{P}(D)$ must be related in terms of both subset properties and the partial ordering properties of D. Unfortunately, there is no best powerdomain construction, and a number of serious questions remain regarding the theory.

Section 12.1 describes the properties of the powerdomain construction, and Section 12.2

uses it to model a nondeterministic language. Section 12.3 presents one approach to modelling interfering parallelism. Section 12.4 presents an alternative approach to nondeterministic and parallel evaluation, and Section 12.5 gives an overview to the mathematics underlying powerdomain construction.

12.1 POWERDOMAINS

The *powerdomain* construction builds a domain of sets of elements. For domain A, the powerdomain builder $\mathbb{P}(\_)$ creates the domain $\mathbb{P}(A)$, a collection whose members are sets $X \subseteq A$. The associated assembly operations are:

$\emptyset : \mathbb{P}(A)$, a constant that denotes the smallest element in $\mathbb{P}(A)$.

$\{\ \_\ \} : A \to \mathbb{P}(A)$, which maps its argument $a \in A$ to the *singleton set* $\{\ a\ \}$.

$\_ \cup \_ : \mathbb{P}(A) \times \mathbb{P}(A) \to \mathbb{P}(A)$, the *binary union* operation, which combines its two arguments $M = \{\ a_0, a_1, \cdots\ \}$ and $N = \{\ b_0, b_1, \cdots\ \}$ into the set $M \cup N = \{\ a_0, a_1, \cdots, b_0, b_1, \cdots\ \}$.

The disassembly operation builder for powerdomains converts an operation on A-elements into one on $\mathbb{P}(A)$-elements.

For $f: A \to \mathbb{P}(B)$, there exists a unique operation $f^+ : \mathbb{P}(A) \to \mathbb{P}(B)$ such that for any $M \in \mathbb{P}(A)$, $f^+(M) = \bigcup \{\ f(m) \mid m \in M\ \}$.

The operation builder can be applied to operations $g: A \to B$ to produce a function in the domain $\mathbb{P}(A) \to \mathbb{P}(B)$: use $(\lambda a.\ \{\ g(a)\ \})^+$.

12.2 THE GUARDED COMMAND LANGUAGE

A well-defined programming language depends on explicit evaluation strategies as little as possible. Imperative languages require sequencing at the command level to clarify the order of updates to the store argument. The sequencing is critical to understanding command composition, but it need not be imposed on the other command builders. As an example, a useful generalization of the conditional command **if** B **then** C_1 **else** C_2 is the multichoice **cases** command:

cases
    $B_1 : C_1$;
    $B_2 : C_2$;
    $\cdots$
    $B_n : C_n$
end

A command C_i is executed when test B_i evaluates to *true.* A problem is that more than one B_i may hold. Normally, we want only one command in the **cases** construct to be evaluated, so we must make a choice. The traditional choice is to execute the ''first'' C_i, reading from ''top to bottom,'' whose test B_i holds, but this choice adds little to the language. A better solution is to nondeterministically choose any one of the candidate commands whose test holds. In Dijkstra (1976), this form of conditional naturally meshes with the development of programs from formal specifications. As an exercise, we define a denotational semantics of the imperative language proposed by Dijkstra.

Dijkstra's language, called the *guarded command language,* is an assignment language augmented by the nondeterministic conditional command and a nondeterministic multitest loop, which iterates as long as one of its tests is true. The language is presented in Figure 12.1.

The domain of possible answers of a nondeterministic computation is a powerdomain of post-store elements. The operation of primary interest is *then,* which sequences two nondeterministic commands. The semantics of an expression $(f_1\ then f_2)(s)$ says that f_1 operates on s, producing a set of post-stores. Each post-store is passed through f_2 to produce an answer set. The answer sets are unioned.

The functionality of the **C** valuation function points out that a command represents a nondeterministic computation. The semantic equations for the conditional and loop commands both use an auxiliary valuation function **T** to determine if at least one of the tests (*guards*) of the construct holds. In the case of the conditional, failure of all guards causes an abortion; failure of all the guards of the loop construct causes exit of the loop. The **G** function defines the meaning of a conditional/loop body. The updates of all the guarded commands whose tests hold are joined together, and a set of stores result.

Here is a small example. For:

$C_0 = G_1 \,[]\, G_2$
$G_1 = X \geq 0 \rightarrow Y := 1$
$G_2 = X = 0 \rightarrow Y := 0$

we derive:

$\mathbf{C}[\![C_0]\!] = \mathbf{C}[\![G_1 \,[]\, G_2]\!]$
$= \lambda s.\ \mathbf{T}[\![G_1 \,[]\, G_2]\!]s \rightarrow \mathbf{G}[\![G_1 \,[]\, G_2]\!]s \,[]\, abort\, s$
$= \lambda s.\ (\mathbf{B}[\![X \geq 0]\!]s)\ or\ (\mathbf{B}[\![X = 0]\!]s) \rightarrow \mathbf{G}[\![G_1 \,[]\, G_2]\!]s \,[]\, abort\, s$

$(\mathbf{B}[\![X \geq 0]\!]s)\ or\ (\mathbf{B}[\![X = 0]\!]s)$ must be *true,* so we simplify to:

$\lambda s.\ \mathbf{G}[\![G_1 \,[]\, G_2]\!]s = \lambda s.\ (\mathbf{B}[\![X \geq 0]\!]s \rightarrow \mathbf{C}[\![Y := 1]\!]s \,[]\, noanswer)$
$\qquad\qquad join\ (\mathbf{B}[\![X = 0]\!]s \rightarrow \mathbf{C}[\![Y := 0]\!]s \,[]\, noanswer)$

Consider a store s_0 such that $(access[\![X]\!]\, s_0)$ is greater than *zero.* Then the above expression simplifies to:

$(\mathbf{C}[\![Y := 1]\!]s_0)\ join\ noanswer$
$= return(s_1) \cup \varnothing$, where $s_1 = (update[\![Y]\!]\ one\ s_0)$
$= \{\ in Store(s_1)\ \} \cup \varnothing$

Similarly, for a store $s_{'0}$ such that $(access[\![X]\!]\, s_{'0})$ is *zero,* we see that the expression

Figure 12.1

C∈ Command
G∈ Guarded-command
E∈ Expression
B∈ Boolean-expression
I∈ Identifier

C ::= C_1;C_2 | I:=E | **if** G **fi** | **do** G **od**
G ::= G_1 [] G_2 | B→ C

Semantic algebras:

I.-IV. Truth values, identifiers, natural numbers, and stores
(the usual definitions)

V. Results of nondeterministic computations
Domains $p \in Poststore = (Store + Errvalue)$
where $Errvalue = Unit$
$a \in Answer = \mathbb{P}(Poststore_{\perp})$
Operations

$no\text{-}answer : Answer$
$no\text{-}answer = \emptyset$

$return : Store \to Answer$
$return = \lambda s.\ \{ \text{in}Store(s) \}$

$abort : Store \to Answer$
$abort = \lambda s.\ \{ \text{in}Errvalue() \}$

$join : Answer \times Answer \to Answer$
$a_1\ join\ a_2 = a_1 \cup a_2$

$then : (Store \to Answer) \times (Store \to Answer) \to (Store \to Answer)$
$f_1\ then\ f_2 = (\underline{\lambda} p.$ cases p of
is$Store(s) \to f_2(s)$
[] is$Errvalue() \to \{ \text{in}Errvalue() \}$
end$)^{+} \circ f_1$

Valuation functions:

C: Command → *Store* → *Answer*
$\mathbf{C}[\![C_1;C_2]\!] = \mathbf{C}[\![C_1]\!]\ then\ \mathbf{C}[\![C_2]\!]$
$\mathbf{C}[\![I:=E]\!] = \lambda s.\ return(update[\![I]\!](\mathbf{E}[\![E]\!]s)\ s)$
$\mathbf{C}[\![\textbf{if}\ G\ \textbf{fi}]\!] = \lambda s.\ \mathbf{T}[\![G]\!]s \to \mathbf{G}[\![G]\!]s\ []\ abort\ s$
$\mathbf{C}[\![\textbf{do}\ G\ \textbf{od}]\!] = fix(\lambda f.\lambda s.\ \mathbf{T}[\![G]\!]s \to (\mathbf{G}[\![G]\!]\ then f)(s)\ []\ return\ s)$

Figure 12.1 (continued)

T: Guarded-command → *Store* → *Tr*
  $\mathbf{T}[\![G_1\ []\ G_2]\!] = \lambda s.\ (\mathbf{T}[\![G_1]\!]s)\ or\ (\mathbf{T}[\![G_2]\!]s)$
  $\mathbf{T}[\![B \rightarrow C]\!] = \mathbf{B}[\![B]\!]$

G: Guarded-command → *Store* → *Answer*
  $\mathbf{G}[\![G_1\ []\ G_2]\!] = \lambda s.\ (\mathbf{G}[\![G_1]\!]s)\ join\ (\mathbf{G}[\![G_2]\!]s)$
  $\mathbf{G}[\![B \rightarrow C]\!] = \lambda s.\ \mathbf{B}[\![B]\!]s \rightarrow \mathbf{C}[\![C]\!]s\ []\ no\text{-}answer$

E:Expression → *Store* → *Nat* (usual)

B: Boolean-expr → *Store* → *Tr* (usual)

simplifies to:

$$
\begin{aligned}
&(\mathbf{C}[\![Y{:=}1]\!]s_{'0})\ join\ (\mathbf{C}[\![Y{:=}0]\!]s_{'0}) \\
&= return(s_{'1})\ join\ return(s_{'2}) \\
&\qquad \text{where } s_{'1} = (update[\![Y]\!]\ one\ s_{'0}) \text{ and } s_{'2} = (update[\![Y]\!]\ zero\ s_{'0}) \\
&= \{\ in_{Store}(s_{'1})\ \} \cup \{\ in_{Store}(s_{'2})\ \} \\
&= \{\ in_{Store}(s_{'1}),\ in_{Store}(s_{'2})\ \}
\end{aligned}
$$

You may have noticed that the phrase $\{\ in_{Store}(s_1)\ \} \cup \varnothing$ was not simplified to $\{\ in_{Store}(s_1)\ \}$ in the first simplification. This step was omitted, for the property $a \cup \varnothing = a$, for $a \in \mathbb{P}(A)$, does *not* hold for all of the versions of powerdomains! This discouraging result is discussed in Section 12.5.

12.3 CONCURRENCY AND RESUMPTION SEMANTICS

As mentioned in the introduction, there are two kinds of parallelism: noninterfering and interfering. The modelling of noninterfering parallelism requires no new concepts, but modelling interfering parallelism does. When assignments evaluate concurrently on the same store, the result is a set of possible result stores, and the powerdomain construction is needed. Further, we require a technique for representing the operational aspects of concurrency in the semantics. We follow Plotkin's method, which depicts the concurrent evaluation of commands C_1 and C_2 by interleaving the evaluation steps of C_1 with those of C_2. This leads to a form of denotational definition called *resumption semantics.*

Figure 12.2 shows a simple imperative language augmented by a parallel evaluation operator $||$. The language's assignment statement is *noninterruptable* and has exclusive rights to the store. We treat a noninterruptable action as the ''evaluation step'' mentioned above. The evaluation of $C_1 || C_2$ interleaves the assignments of C_1 with those of C_2. A semantics of $C_1 || C_2$ generates the set of results of all possible interleavings. Here are some example

Figure 12.2

Abstract syntax:

P∈ Program
C∈ Command
E∈ Expression
B∈ Boolean-expr
I∈ Identifier

P ::= C.

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

program fragments and an informal description of their actions.

12.1 Example:

⟦X:=X+1⟧ is an ordinary assignment. Given a store argument, the command updates ⟦X⟧'s cell. No other command may access or alter the store while the assignment is performed.

12.2 Example:

⟦X:=X+2; X:=X−1⟧ is a compound command. Although each of the two assignments receives exclusive rights to the store when executing, another command operating in parallel may interrupt the composition after the evaluation of the first assignment and before the evaluation of the second. Thus, the composition is *not* semantically equivalent to ⟦X:=X+1⟧. Consider ⟦(X:=X+2; X:=X−1) || (X:=3)⟧. The possible interleavings of this concurrent command are:

X:=X+2; X:=X-1; X:=3
X:=X+2; X:=3; X:=X-1
X:=3; X:=X+2; X:=X-1

Each of the interleavings yields a different output store. Command composition must have a denotational semantics that is different from the ones used in previous chapters.

12.3 Example:

⟦(**if** X=0 **then** Y:=1 **else** Y:=2) || (X:=X+1)⟧. The evaluation of the test of a conditional is noninterruptable. The possible evaluation sequences are:

for X having an initial value of *zero*:

test X=0; Y:=1; X:=X+1
test X=0; X:=X+1; Y:=1
X:=X+1; test X=0; Y:=2

for X having an initial positive value:
test X=0; Y:=2; X:=X+1
test X=0; X:=X+1; Y:=2
X:=X+1; test X=0; Y:=2

12.4 Example:

⟦(X:=1; **while** X> 0 **do** Y:=Y+1) || (X:=0)⟧. Like Example 12.3, a command executing concurrently with a **while**-loop may be interleaved with the loop's evaluation sequence. When ⟦X⟧ is zero, the loop terminates, so the interleaving of ⟦X:=0⟧ into the loop's evaluation becomes critical to the result of the program. The possible evaluation sequences are:

X:=0; X:=1; test X>0; Y:=Y+1; test X>0; Y:=Y+1; ···
X:=1; X:=0; test X>0
X:=1; test X>0; X:=0; Y:=Y+1; test X>0
X:=1; test X>0; Y:=Y+1; X:=0; test X>0
X:=1; test X>0; Y:=Y+1; test X>0; X:=0; Y:=Y+1; test X>0
X:=1; test X>0; Y:=Y+1; test X>0; Y:=Y+1; X:=0; test X>0
...

These evaluation sequences are called *fair* because the assignment ⟦X:=0⟧ eventually appears in the evaluation sequence. A fair evaluation of $C_1 || C_2$ eventually evaluates both C_1 and C_2. An unfair sequence would evaluate the loop all its nonterminating way and *then* ''perform'' ⟦X:=0⟧. The resumption semantics assigned to this example will include the fair sequences plus the unfair sequence just mentioned.

As we saw in Example 12.2, even apparently sequential constructs are impacted by possible outside interference. The denotation of a command can no longer be a map from an input to an output store but must become a new entity, a *resumption.* Figure 12.3 presents the semantic algebra of resumptions.

A resumption can be thought of as a set of interruptable evaluation sequences. Let r be a resumption and s be a store. If r consists of just a single step, as in Example 12.1, $r(s)$ is (a set containing) a new store, that is, $\{ \text{in}_{Store}(s') \}$. If r is a single sequence of steps, as in Example 12.2, $r(s)$ is not the application of all the steps to s, but the application of just the first of the steps, producing (a set containing) a new store plus the remaining steps that need to be done, that is, $\{ \text{in}_{Store \times Res}(s', r') \}$, where s' is the store resulting from the first step and r' is the remainder of the steps. This structure is necessary so that the interleaving of other evaluation sequences, that is, other resumptions, can be handled if necessary. When resumption r depicts a parallel evaluation, as in Examples 12.3 and 12.4, r contains a number of evaluation sequences, and $r(s)$ is a nonsingleton set of partial computations.

Figure 12.3

IV. Resumptions

Domains $p \in Pgm\text{-}state = (Store + (Store \times Res))$

$r \in Res = Store \to \mathbb{P}(Pgm\text{-}state_\perp)$

Operations

$step : (Store \to Store) \to Res$

$step = \lambda f.\lambda s.\ \{\ \text{in}Store(f\,s)\ \}$

$pause : Res \to Res$

$pause = \lambda r.\lambda s.\ \{\ \text{in}Store\times Res(s, r)\ \}$

$\_ * \_ : Res \times Res \to Res$

$r_1 * r_2 = (\underline{\lambda} p.$ cases p of

$\quad \text{is}Store(s_1) \to \{\ \text{in}Store\times Res(s_1, r_2)\ \}$

$\quad [] \ \text{is}Store\times Res(s_1, r_1) \to \{\ \text{in}Store\times Res(s_1, r_1 * r_2)\ \}$

$\quad \text{end})^+ \circ r_1$

$par : Res \times Res \to Res$

$r_1\ par\ r_2 = \lambda s.\ (r_1\ then\ r_2)(s) \cup (r_2\ then\ r_1)(s)$

where $then : Res \times Res \to Res$ is

$r_1\ then\ r_2 = (\underline{\lambda} p.$ cases p of

$\quad \text{is}Store(s_1) \to \{\ \text{in}Store\times Res(s_1, r_2)\ \}$

$\quad [] \ \text{is}Store\times Res(s_1, r_1) \to \{\ \text{in}Store\times Res(s_1, r_1\ then\ r_2)\ \}$

$\quad\quad \cup \{\ \text{in}Store\times Res(s_1, r_2\ then\ r_1)\ \}$

$\quad \text{end})^+ \circ r_1$

$flatten : Res \to Store \to \mathbb{P}(Store_\perp)$

$flatten = \lambda r.(\underline{\lambda} p.$ cases p of

$\quad \text{is}Store(s_1) \to \{\ s_1\ \}$

$\quad [] \ \text{is}Store\times Res(s_1, r_1) \to (flatten\ r_1)(s_1)$

$\quad \text{end})^+ \circ r$

The operations upon resumptions show how resumptions are defined from atomic actions, paused to allow interleavings, sequentially composed, interleaved, and evaluated to an answer set. The first of these constructions is *step*, which builds a single step evaluation sequence that immediately and noninterruptedly performs its actions upon a store. A sequence r that can be interrupted before it performs any action at all is defined by (*pause r*), which holds its store argument without applying any of r to it. The expression $r_1 * r_2$ is the composition of the evaluation steps of resumption r_1 with the steps of r_2. The interleaving of resumptions is defined by *par*. A resumption is converted into a set of noninterruptable evaluation sequences by *flatten*. The expression (*flatten r*)(*s*) evaluates each of the sequences in r with s to an output store. The result is a set of stores.

The valuation functions are specified in Figure 12.4.

The denotation of a command in the language is a resumption because the command might be embedded in a parallel evaluation. If so, the command's evaluation sequence would have to be interleaved with the other parts of the parallel construction. Once the command is completely built into a program, the resumption is flattened into the family of possible evaluation sequences that it represents. Since a conditional can be interrupted after its test and before its clauses are evaluated, the *pause* operation must be inserted to create an explicit interrupt point. The loop is handled similarly.

The **E** and **B** functions are not written in the resumption style. As we saw in Chapter 9, the sequencing aspects of a language may be specified at certain levels and ignored at others. The language in Figure 12.4 interleaves evaluation steps only at the command level. Nonetheless, expression resumptions can be introduced, just as expression continuations were introduced to augment command continuations in Chapter 9. This is left as an exercise.

You should determine the denotations of the commands in Examples 12.1 through 12.4.

12.4 AN ALTERNATIVE SEMANTICS FOR CONCURRENCY

Although we have given a denotational semantics to a concurrent language, the definitions of the resumption operations are far too complex. The problem is that our function notation is ill-suited for representing multivalued objects; at most, one function can ''own'' an argument. In a description of a concurrent language, more than one function competes for the same argument— in Figure 12.4, it was the computer store. We seek a natural way of describing

Figure 12.4

$\mathbf{P}$: Program $\to$ *Store* $\to$ $\mathbb{P}$(*Store*$_\perp$)
$\mathbf{P}[\![\text{C.}]\!] = \mathit{flatten}(\mathbf{C}[\![\text{C}]\!])$

$\mathbf{C}$: Command $\to$ *Res*
$\mathbf{C}[\![\text{I:=E}]\!] = \mathit{step}(\lambda s.\ \mathit{update}[\![\text{I}]\!]\ (\mathbf{E}[\![\text{E}]\!]s)\ s)$
$\mathbf{C}[\![\text{C}_1\text{;C}_2]\!] = \mathbf{C}[\![\text{C}_1]\!] * \mathbf{C}[\![\text{C}_2]\!]$
$\mathbf{C}[\![\text{C}_1 \mid\mid \text{C}_2]\!] = \mathbf{C}[\![\text{C}_1]\!]\ \mathit{par}\ \mathbf{C}[\![\text{C}_2]\!]$
$\mathbf{C}[\![\textbf{if}\ \text{B}\ \textbf{then}\ \text{C}_1\ \textbf{else}\ \text{C}_2]\!] = \lambda s.\mathbf{B}[\![\text{B}]\!]s \to (\mathit{pause}(\mathbf{C}[\![\text{C}_1]\!])\ s)\ [\!]\ (\mathit{pause}(\mathbf{C}[\![\text{C}_2]\!])\ s)$
$\mathbf{C}[\![\textbf{while}\ \text{B}\ \textbf{do}\ \text{C}]\!] = \mathit{fix}(\lambda f.\lambda s.\ \mathbf{B}[\![\text{B}]\!]s \to (\mathit{pause}(\mathbf{C}[\![\text{C}]\!] * f)\ s)\ [\!]\ (\mathit{step}(\lambda s.\ s)\ s))$

$\mathbf{E}$: Expression $\to$ *Store* $\to$ *Nat* (like Figure 5.2)

$\mathbf{B}$: Boolean-expr $\to$ *Store* $\to$ *Tr* (like Figure 5.2)

this competition.

Consider a generalization of function notation such that more than one function can be applied to, or choose to communicate with, an argument. Further, the argument may choose which function it wishes to interact with. This recalls Hewitt's actor theory (Hewitt & Baker 1978).

We must make some notational changes. First, the application of a function to an argument is no longer written as $f(a)$, as we will have situations in which both f_1 and f_2 desire the same a. We ''specialize'' the λ in $f = \lambda x.M$ to name a *port* or argument path by using Greek letters α, β, γ, $\cdots$ in place of the λ. The argument is marked with the same Greek letter with a bar over it. Thus, the application $(\lambda x.M)(a)$ becomes $(\alpha x.M) \mid (\bar{\alpha}a)$, and, in the general case, $(\alpha x_1.M_1) \mid \cdots \mid (\alpha x_n.M_n) \mid (\bar{\alpha}a)$ describes the situation where all of the functions $(\lambda x_1.M_1)$, $\cdots$, $(\lambda x_n.M_n)$ wish to use a, but only one of them will receive it.

The notation that we are developing is called a *Calculus for Communicating Systems* (CCS), and it was defined by Milner. Lack of space prevents us from giving a complete presentation of CCS, so you should read Milner's book (Milner 1980). We will study the basic features of CCS and see how the parallel language of the previous section can be described with it.

The syntax of CCS *behavior expressions* is given in Figure 12.5.

Consider the BNF rule for behavior expressions. The first two forms in the rule are the generalized versions of function abstraction and argument application that we were considering. Note that the argument construct $\bar{P}E.B$ is generalized to have an argument E and a body B, just like an abstraction has. Once the argument part E binds to some abstraction, the body B evaluates. Abstractions and arguments are symmetrical and autonomous objects in CCS. The third form, $B_1 \mid B_2$, is parallel composition; behavior expressions B_1 and B_2 may evaluate in parallel and pass values back and forth. The behavior expression $B_1 + B_2$ represents nondeterministic choice: either B_1 or B_2 may evaluate, but not both. $B_1 * B_2$ represents sequential evaluation; at the end of B_1's evaluation, B_2 may proceed. The *if* construct is the usual conditional. The behavior expression $B \S P$ hides the port P in B from outside communication; $B \S P$ cannot send or receive values along port P, so any use of P must be totally within B. $B[P_1/P_2]$

Figure 12.5

B∈ Behavior-expression
E∈ Function-expression
P∈ Port
I∈ Identifier

B ::= $PI.B \mid \bar{P}E.B \mid B_1 \mid B_2 \mid B_1 + B_2 \mid B_1 * B_2$
$\quad \mid$ *if* E *then* B_1 *else* $B_2 \mid B \S P \mid B[P_1/P_2] \mid$ *nil*

E ::= (defined in Chapter 3)

renames all nonhidden occurrences of port P_2 to P_1. Finally, *nil* is the inactive behavior expression.

What is the meaning of a behavior expression? We could provide a resumption semantics, but since we wish to forgo resumptions, we follow Milner's lead: he suggests that the meaning of a behavior expression is a tree showing all the possible evaluation paths that the expression might take. The arcs of such a *communication tree* are labeled with the values that can be sent or received by the behavior expression. Some examples are seen in Figure 12.6. Nondeterministic and parallel composition cause branches in a communication tree. An internal communication of a value within a behavior expression produces an arc labeled by a τ symbol. Actually, the trees in Figure 12.6 are overly simplistic because they use identifiers on the arcs instead of values. A completely expanded rendition of expression 2 in that figure is:

$$\begin{array}{cccccccc} \alpha zero & & \alpha one & \cdots & \alpha i & \cdots \\ \\ \bar{\beta} zero & \bar{\gamma} one & \bar{\beta} one & \bar{\gamma} two & \bar{\beta} i & \bar{\gamma}(i\ plus\ one) \end{array}$$

because the behavior expression is a function of $n \in Nat$. The communication tree represents the ''graph'' of the behavior expression.

Like function graphs, communication trees are somewhat awkward to use. Just as we use simplification rules on function expressions to determine the unique meaning of an expression, we use inference rules on behavior expressions to determine an evaluation path in a behavior expression's communication tree. Inference rules for CCS and some useful equivalences for behavior expressions are given in Figure 12.7. An axiom $B \xrightarrow{\mu v} B'$ says that on reception of a v value along port μ, behavior expression B progresses to B'. An inference rule:

$$\frac{B'_1 \xrightarrow{\mu v} B'_1}{B_2 \xrightarrow{\mu v} B'_2}$$

says that if behavior expression B_1 can progress to B'_1 via communication μv, then expression B_2 can progress to B'_2 with the same communication. An equivalence $B \equiv B'$ says that behavior expression B may be rewritten to B' (without any communication) since both expressions represent equivalent communication trees. The descriptions of the rules and trees are necessarily brief, and you are referred to Milner's book.

We can put the inference rules to good use. Here is a derivation of a path in the communication tree for expression 3 in Figure 12.6:

$(\alpha n.\ nil) \mid (\beta m.\ \bar{\alpha}(m\ plus\ one).\ nil)$
$\xrightarrow{\beta two}\ (\alpha n.\ nil) \mid (\bar{\alpha}(two\ plus\ one).\ nil)$, by Com$\rightarrow$ (1), Act$\rightarrow$ (1)
$\equiv\ (\alpha n.\ nil) \mid (\bar{\alpha}(three).\ nil)$

Figure 12.6

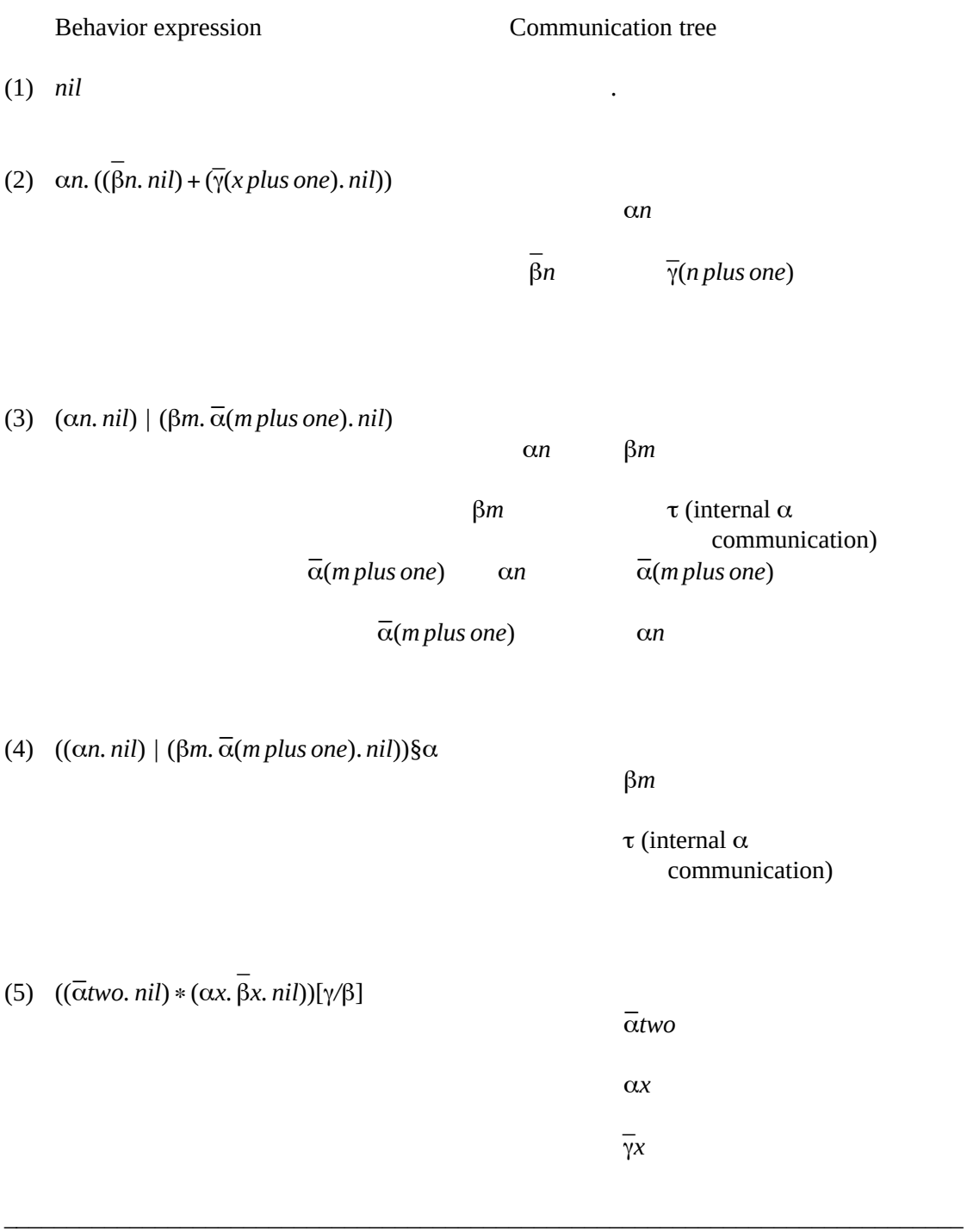

Behavior expression
Communication tree
(1) nil
.
(2) αn. ((β̄n. nil) + (γ̄(x plus one). nil))
αn
β̄n
γ̄(n plus one)
(3) (αn. nil) | (βm. ᾱ(m plus one). nil)
αn
βm
βm
τ (internal α communication)
ᾱ(m plus one)
αn
ᾱ(m plus one)
ᾱ(m plus one)
αn
(4) ((αn. nil) | (βm. ᾱ(m plus one). nil))§α
βm
τ (internal α communication)
(5) ((ᾱtwo. nil) ∗ (αx. β̄x. nil))[γ/β]
ᾱtwo
αx
γ̄x

Figure 12.7

Inference rules:

Act→ (1) $\alpha x.\, B \xrightarrow{\alpha v} [\,v/x\,]B$

(2) $\bar{\alpha} v.\, B \xrightarrow{\bar{\alpha} v} B$

Com→ (1) $\dfrac{B_1 \xrightarrow{\mu v} B'_1}{B_1 \mid B_2 \xrightarrow{\mu v} B'_1 \mid B_2}$ where μ may be either α or $\bar{\alpha}$, $\alpha \in P$

(2) $\dfrac{B_2 \xrightarrow{\mu v} B'_2}{B_1 \mid B_2 \xrightarrow{\mu v} B_1 \mid B'_2}$ (3) $\dfrac{B_1 \xrightarrow{\alpha v} B'_1 \qquad B_2 \xrightarrow{\bar{\alpha} v} B'_2}{B_1 \mid B_2 \xrightarrow{\tau} B'_1 \mid B'_2}$

Sum→ (1) $\dfrac{B_1 \xrightarrow{\mu v} B'_1}{B_1 + B_2 \xrightarrow{\mu v} B'_1}$ (2) $\dfrac{B_2 \xrightarrow{\mu v} B'_2}{B_1 + B_2 \xrightarrow{\mu v} B'_2}$

Seq→ $\dfrac{B_1 \xrightarrow{\mu v} B'_1}{B_1 * B_2 \xrightarrow{\mu v} B'_1 * B_2}$

Con→ (1) $\dfrac{B_1 \xrightarrow{\mu v} B'_1}{\textit{if true then } B_1 \textit{ else } B_2 \xrightarrow{\mu v} B'_1}$ (2) $\dfrac{B_2 \xrightarrow{\mu v} B'_2}{\textit{if false then } B_1 \textit{ else } B_2 \xrightarrow{\mu v} B'_2}$

Res→ $\dfrac{B \xrightarrow{\mu v} B'}{B \S \alpha \xrightarrow{\mu v} B' \S \alpha}$ where μ is not in $\{\, \alpha, \bar{\alpha} \,\}$

Rel→ $\dfrac{B \xrightarrow{\mu v} B'}{B[\,\gamma/\mu\,] \xrightarrow{\gamma v} B'[\,\gamma/\mu\,]}$

Equivalences:

$B_1 \mid B_2 \equiv B_2 \mid B_1$
$(B_1 \mid B_2) \mid B_3 \equiv B_1 \mid (B_2 \mid B_3)$
$nil \mid B \equiv B$
$B_1 + B_2 \equiv B_2 + B_1$
$(B_1 + B_2) + B_3 \equiv B_1 + (B_2 + B_3)$
$nil + B \equiv B$
$nil * B \equiv B$
$(B_1 * B_2) * B_3 \equiv B_1 * (B_2 * B_3)$

$\xrightarrow{\tau} nil \mid nil$, by Com→ (3), Act→ (3)

$\equiv$ *nil*

and the path is β*two*, τ. The path shows a result of supplying the argument *two* on the β port to the behavior expression. From here on, we will only be interested in deriving paths that contain no instances of external input or output; all value communication will be internal. These ''closed derivations'' correspond to the simplification sequences built for function expressions. The behavior expression just given does not have a closed derivation to *nil*, for some value must be given to the β port. The following expression does have a closed derivation to *nil:*

$(\alpha n.\ nil) \mid (\beta m.\ \bar{\alpha}(m\ plus\ one).\ nil) \mid (\bar{\beta} two.\ nil)$
$\xrightarrow{\tau} (\alpha n.\ nil) \mid (\bar{\alpha}.\ (two\ plus\ one).\ nil) \mid nil$
$\equiv (\alpha n.\ nil) \mid (\bar{\alpha}(three).\ nil) \mid nil$
$\xrightarrow{\tau} nil \mid nil \mid nil$
$\equiv nil$

We will also allow named behavior expressions, and the namings may be recursive. For example:

binary-semaphore = $\bar{\alpha}().\ \beta().$ *binary-semaphore*

describes a simple binary semaphore. The argument values transmitted along the α- and β-ports are from the *Unit* domain. Named behavior expressions can be abstracted on function expression values. For example:

counting-sem(*n*) = *if n equals zero*
then $\beta().$ *counting-sem*(*one*)
else $((\bar{\alpha}().$ *counting-sem*(*n minus one*)))
$+ (\beta().$ *counting-sem*(*n plus one*))

is a definition of a counting semaphore. The rewriting rule for recursively named behavior expressions is the usual unfolding rule for recursively defined expressions.

The CCS-based semantics of the language in Figure 12.4 is given in Figure 12.8. The store is managed by a semaphore-like behavior expression, which transmits and receives the store from communicating commands. The new semantics is faithful to the one in Figure 12.4 because it treats assignments as noninterruptable primitive commands, allows interleaving of commands in parallel evaluation, and admits interleaving into a conditional command between the test and the selected clause. A derivation of a program denotation is seen in Figure 12.9.

The CCS-based denotation makes the competition for the store easier to see and the possible outcomes easier to determine. Although the resumption semantics provides a precise function meaning for a parallel program, the CCS version provides a depiction that is easier to read, contains many operational analogies, and has a precise meaning in communication tree form.

Figure 12.8

Abstract syntax:

P∈ Program
C∈ Command
E∈ Expression
B∈ Boolean-expr

P ::= C.

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

Semantic algebras: (usual)

Store manager behavior expression:

$sem(s) = \bar{\alpha}s.\ \mu s'.\ sem(s')$

Valuation functions:

P: Program → *Behavior-expression*
$\mathbf{P}[\![C.]\!] = \lambda s.\ \mathbf{C}[\![C]\!] \mid sem(s)$

C: Command → *Behavior-expression*
$\mathbf{C}[\![C_1;C_2]\!] = \mathbf{C}[\![C_1]\!] * \mathbf{C}[\![C_2]\!]$
$\mathbf{C}[\![C_1 || C_2]\!] = \mathbf{C}[\![C_1]\!] \mid \mathbf{C}[\![C_2]\!]$
$\mathbf{C}[\![I:=E]\!] = \alpha s.\ \bar{\mu}(update[\![I]\!](\mathbf{E}[\![E]\!]s)\ s).\ nil$
$\mathbf{C}[\![\textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2]\!] = \alpha s.\ if\ \mathbf{B}[\![B]\!]s\ then\ \bar{\mu}s.\ \mathbf{C}[\![C_1]\!]\ else\ \bar{\mu}s.\ \mathbf{C}[\![C_2]\!]$
$\mathbf{C}[\![\textbf{while}\ B\ \textbf{do}\ C]\!] = f$
where $f = \alpha s.\ if\ \mathbf{B}[\![B]\!]s\ then\ \bar{\mu}s.\ (\mathbf{C}[\![C_1]\!] * f)\ else\ \bar{\mu}s.\ nil$

Figure 12.9

Let

B_{IN} stand for $\alpha s.\ \bar{\mu}(update[\![\mathrm{I}]\!]\ \mathbf{N}[\![\mathrm{N}]\!]\ s).\ nil$

in

$\mathbf{P}[\![\mathrm{X}{:=}0\ ||\ \mathrm{X}{:=}1;\ \mathbf{if}\ \mathrm{X}{=}0\ \mathbf{then}\ \mathrm{Y}{:=}0\ \mathbf{else}\ \mathrm{Y}{:=}1]\!] =$

$\lambda s.\ B_{\mathrm{X0}} \mid B_{\mathrm{X1}} * B_{if} \mid sem(s)$

where $B_{if} = \alpha s.\ if\ (access[\![\mathrm{X}]\!]\ s)\ then\ \bar{\mu}s.\ B_{\mathrm{Y0}}\ else\ \bar{\mu}s.B_{\mathrm{Y1}}$

A derivation is:

$(\lambda s.\ B_{\mathrm{X0}} \mid B_{\mathrm{X1}} * B_{if} \mid sem(s))(s_0)$

$\equiv\ B_{\mathrm{X0}} \mid B_{\mathrm{X1}} * B_{if} \mid sem(s_0)$

$\equiv\ B_{\mathrm{X0}} \mid B_{\mathrm{X1}} * B_{if} \mid \bar{\alpha}s_0.\ \mu s'.\ sem(s')$

$\xrightarrow{\tau} B_{\mathrm{X0}} \mid \bar{\mu}(s_1).\ nil * B_{if} \mid \mu s'.\ sem(s')$ where $s_1 = (update[\![\mathrm{X}]\!]\ one\ s_0)$

$\xrightarrow{\tau} B_{\mathrm{X0}} \mid nil * B_{if} \mid sem(s_1)$

$\equiv\ B_{\mathrm{X0}} \mid \alpha s.\ if\ (access[\![\mathrm{X}]\!]\ s)\ equals\ zero\ then\ \cdots\ \mid \bar{\alpha}s_1.\ \mu s'.\ sem(s')$

$\xrightarrow{\tau} B_{\mathrm{X0}} \mid if\ (access[\![\mathrm{X}]\!]\ s_1)\ equals\ zero\ then\ \cdots\ \mid \mu s'.\ sem(s')$

$\equiv\ B_{\mathrm{X0}} \mid if\ false\ then\ \bar{\mu}s_1.\ B_{\mathrm{Y0}}\ else\ \bar{\mu}s_1.\ B_{\mathrm{Y1}} \mid \mu s'.\ sem(s')$

$\xrightarrow{\tau} B_{\mathrm{X0}} \mid B_{\mathrm{Y1}} \mid sem(s_1)$

$\equiv\ \alpha s.\ \bar{\mu}(update[\![\mathrm{X}]\!]\ zero\ s).\ nil \mid B_{\mathrm{Y1}} \mid \bar{\alpha}s_1.\mu s'.\ sem(s')$

$\xrightarrow{\tau} \bar{\mu}(s_2).\ nil \mid B_{\mathrm{Y1}} \mid \mu s'.\ sem(s')$ where $s_2 = (update\ [\![\mathrm{X}]\!]\ zero\ s_1)$

$\xrightarrow{\tau} nil \mid B_{\mathrm{Y1}} \mid sem(s_2)$

$\equiv\ \alpha s.\ \bar{\mu}(update[\![\mathrm{Y}]\!]\ one\ s).\ nil \mid \bar{\alpha}s_2.\mu s'.\ sem(s')$

$\xrightarrow{\tau} \bar{\mu}(s_3).\ nil \mid \mu s'.\ sem(s')$ where $s_3 = (update[\![\mathrm{Y}]\!]\ one\ s_2)$

$\xrightarrow{\tau} nil \mid sem(s_3)$

$\equiv\ sem(s_3).$

12.5 THE POWERDOMAIN STRUCTURE

We conclude this chapter by studying the mathematics of powerdomain construction. As mentioned in the introduction, there is no ''best'' version of a powerdomain. This is because the clash between the subset properties of the powerdomain and the partial ordering properties of its component domain can be resolved in several ways. We present the methods in two stages: we first construct powerdomains from domains that have trivial partial order structure, and then we generalize to arbitrary domains. The first stage is straightforward, but the second is somewhat involved and only an overview is given. Plotkin's and Smyth's original articles give a complete presentation.

12.5.1 Discrete Powerdomains

A domain A with a trivial partial ordering (that is, for all $a,b \in A$, $a \sqsubseteq b$ iff $a = b$ or $a = \bot$) is called a *flat domain.* Powerdomains built from flat domains are called *discrete powerdomains.* Examples of flat domains are $\mathbb{N}$, $\mathbb{N}_\bot$, $\mathbb{N}\times\mathbb{N}$, *Identifier*, *Identifier* $\to \mathbb{N}$, and (*Identifier* $\to \mathbb{N})_\bot$, but *not Identifier* $\to \mathbb{N}_\bot$ or $\mathbb{N}_\bot \times \mathbb{N}$. A flat domain has almost no internal structure.

The first method builds the set-of-all-sets construction from a flat domain.

12.5 Definition:

For a flat domain D, the discrete relational powerdomain of D, written $\mathbb{P}_R(D)$*, is the collection of all subsets of proper (that is, non-*$\bot$*) elements of D, partially ordered by the subset relation* $\subseteq$ *.*

Let $\sqsubseteq_R$ stand for the partial ordering relation $\subseteq$ on $\mathbb{P}_R(D)$. In preparation for the construction of relational powerdomains from nonflat domains, we note that for all $A,B \in \mathbb{P}_R(D)$:

$A \sqsubseteq_R B$ iff for every $a \in A$ there exists some $b \in B$ such that $a \sqsubseteq_D b$

This property ties together the structural and subset properties of elements in the powerdomains. The only element in the flat domain that might have caused structural problems, $\bot \in D$, is handled by making it ''disappear'' from the construction of $\mathbb{P}_R(D)$. Thus, the domain $\mathbb{P}_R(\mathbb{N})$ is identical to $\mathbb{P}_R(\mathbb{N}_\bot)$: both are just the powerset of $\mathbb{N}$, partially ordered by $\subseteq$.

In the relational powerdomain, the constant $\varnothing$ does indeed stand for the empty set in the domain. As an exercise, you are asked to show that the associated assembly and disassembly operations are continuous.

The relational powerdomain construction is a natural one for a cpo lacking $\bot$. When used with a pointed cpo, it ignores the possibility of nontermination as a viable answer. For this reason, the relational powerdomain is useful in those cases where only partial correctness issues are of primary concern. The domain works well with the semantics in Figure 12.1, because the property $\varnothing \cup d = d$ is necessary to supply the expected semantics for conditional and loop commands.

The relational powerdomain is inadequate for modelling operational concerns. If the $\bot$ element of a flat domain is introduced into the elements of the powerdomain, the result is an Egli-Milner powerdomain.

12.6 Definition:

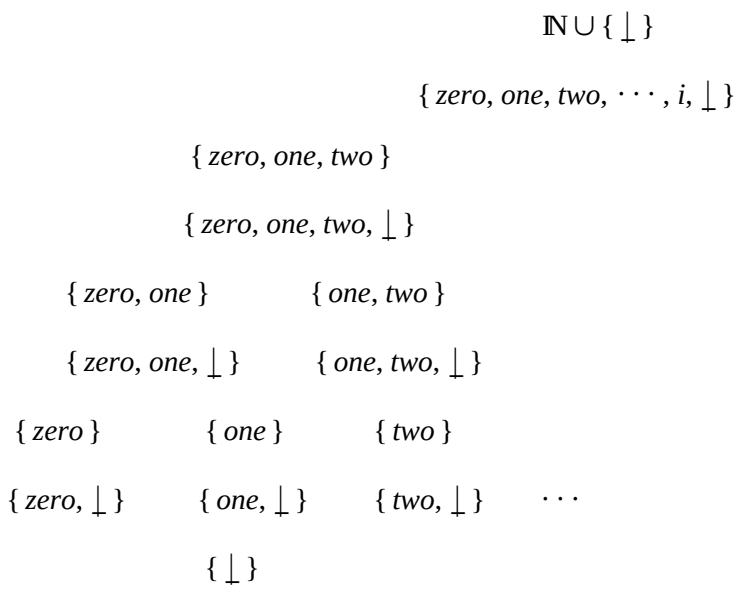
For a pointed cpo D, the discrete Egli-Milner powerdomain of D, written $\mathbb{P}_{EM}(D)$*, is the collection of nonempty subsets of D which are either finite or contain* $\bot$*, partially ordered as follows: for all* $A,B \in \mathbb{P}_{EM}(D)$*,* $A \sqsubseteq_{EM} B$ *iff:*

1. *For every* $a \in A$*, there exists some* $b \in B$ *such that* $a \sqsubseteq_D b$*.*
2. *For every* $b \in B$*, there exists some* $a \in A$ *such that* $a \sqsubseteq_D b$*.*

The construction only operates upon pointed cpos. All sets in the powerdomain are nonempty, because an element denotes a set of possible results of a computation, and the empty set has

no significance, for it contains no results, not even $\bot$. The infinite elements of the powerdomain contain $\bot$ to show that, if a computation has an infinite set of possible results, it will have to run forever to cover all the possibilities, hence nontermination is also a viable result.

A partial drawing of $\mathbb{P}_{\mathrm{EM}}(\mathbb{N}_\bot)$ is:



The subset ordering is restricted to those relations that are computationally feasible. We read an element $\{ m_1, m_2, \cdots, m_n \}$ not containing $\bot$ as the final result of a computation. An element $\{ m_1, m_2, \cdots, m_n, \bot \}$ may be read as either a partial result of a computation, where $\bot$ denotes a lack of knowledge about the remaining output values, or as the final result of a computation that might not terminate. Thus, $\{ one, \bot \} \sqsubseteq \{ one, two, three \}$, as $\bot$ is ''completed'' to $\{ two, three \}$, but $\{ one \} \not\sqsubseteq \{ one, two \}$, as the output information in the set $\{ one \}$ is complete. Also, the least upper bound of the chain $\{ \bot \}$, $\{ zero, \bot \}$, $\{ zero, one, \bot \}$, $\cdots$, must be $\mathbb{N} \cup \{ \bot \}$, rather than $\mathbb{N}$ (if $\mathbb{N}$ were indeed in $\mathbb{P}_{\mathrm{EM}}(\mathbb{N}_\bot)$), for $(\mathbb{N} \cup \{ \bot \}) \sqsubseteq_{\mathrm{EM}} \mathbb{N}$, and since all elements of the chain possess the property of ''noncompletion of output,'' so must the least upper bound.

In the Egli-Milner powerdomain, the constant $\varnothing$ represents the set $\{ \bot \}$. A consequence is that $\varnothing \cup d$ does *not* equal *d*. You should show that the singleton and union operations are continuous and that the operation f^+ is continuous when f is continuous and strict.

The Egli-Milner powerdomain is useful for analyzing the operational properties of a language. For this reason, it is the choice for supplying the semantics of the concurrent language in Section 12.3.

The final example of discrete powerdomain uses the third variant of partial ordering on set elements.

12.7 Definition:

For a flat domain D, the discrete Smyth powerdomain of D, written $\mathbb{P}_{\mathrm{S}}(D)$*, is the collection of finite, nonempty sets of proper elements of D along with D itself, partially ordered*

as follows: for all $A,B \in \mathbb{P}_S(D)$, $A \sqsubseteq_S B$ *iff for every* $b \in B$ *there exists some* $a \in A$ *such that* $a \sqsubseteq_D b$

Since $\sqsubseteq_S$ is the inverse of $\sqsubseteq_R$, $A \sqsubseteq_S B$ iff $B \subseteq A$. The reverse subset ordering suggests that a set B is better defined than A when B's information is more specific than A's. Computation upon a Smyth powerdomain can be viewed as the process of determining what *cannot* be an answer. A nonterminating computation rules out nothing; that is, virtually anything might result. Thus, D is the least defined element in $\mathbb{P}_S(D)$. A partial drawing of $\mathbb{P}_S(\mathbb{N}_\perp)$ is:

$$\{ \textit{two} \} \qquad \{ \textit{one} \} \qquad \{ \textit{zero} \}$$

$$\{ \textit{one, two} \} \quad \{ \textit{zero, two} \} \quad \{ \textit{zero, one} \} \quad \cdots$$

$$\{ \textit{zero, one, two} \} \qquad \{ \textit{zero, one, three} \} \quad \cdots$$

$$\mathbb{N} \cup \{ \perp \}$$

As with the Egli-Milner powerdomain, the value of $\emptyset \cup d$ is not d— it is $\emptyset$! This is because $\emptyset$ represents the set D. The Smyth powerdomain is appropriate for total correctness studies, that is, results that are valid only if the program examined always terminates. The guarded command language of Section 12.2 was designed by Dijkstra to be understood in terms of total correctness. He introduced an assertion language and described the actions of commands in terms of their assertion transformation properties. The semantics of the language in Figure 12.1 is *not* faithful to Dijkstra's ideas when the Smyth powerdomain is used. You are given the exercise of rewriting the semantics of the language to match Dijkstra's intentions.

12.5.2 General Powerdomains

We now generalize the discrete powerdomain constructions to handle nonflat domains. The problems inherent in handling nonflat domains are examined first, and the general versions of the three powerdomains are presented.

Let us begin with the generalization of the relational powerdomain. We would like to define the relational powerdomain of an arbitrary domain D in a fashion similar to the discrete version: the elements of $\mathbb{P}_R(D)$ should be the subsets of proper elements of D, ordered by the relation formulated earlier: $A \sqsubseteq_R B$ iff for all $a \in A$ there exists some $b \in B$ such that $a \sqsubseteq_D b$. Unfortunately, this or-dering leads to:

12.8 Problem:

$\sqsubseteq_R$ is not a partial ordering. As an example, for proper elements $d_1, d_2 \in D$ such that $d_1 \sqsubseteq_D d_2$, both $\{ d_2 \} \sqsubseteq_R \{ d_1, d_2 \}$ and $\{ d_1, d_2 \} \sqsubseteq_R \{ d_2 \}$. The reason for this equivalence is that the total information contents of the two sets are identical. This

example shows the clash of the structure of D with the subset properties of the powerset.

We might attempt a solution by grouping together those sets that are equivalent with respect to the ordering $\sqsubseteq_R$. Let $\mathbb{P}(D)/\sqsubseteq_R$, the *quotient of* $\mathbb{P}(D)$ *with respect to* $\sqsubseteq_R$, be the sets of proper elements of D grouped into collections called *equivalence classes.* Two sets A and B are in the same equivalence class iff $A \sqsubseteq_R B$ and $B \sqsubseteq_R A$. The equivalence classes are partially ordered by $\sqsubseteq_R$: for equivalence classes $P, Q \in \mathbb{P}(D)/\sqsubseteq_R$, $P \sqsubseteq Q$ iff for all $A \in P$ and $B \in Q$, $A \sqsubseteq_R B$. Let $[A]$ represent the equivalence class containing the set $A \in \mathbb{P}(D)$. We can define the operations:

$\varnothing : \mathbb{P}(D)/\sqsubseteq_R$ denotes $[\{\,\}]$
$\{\_\} : D \to \mathbb{P}(D)/\sqsubseteq_R$ maps $d \in D$ to $[\{d\}]$
$\_ \cup \_ : \mathbb{P}(D)/\sqsubseteq_R \times \mathbb{P}(D)/\sqsubseteq_R \to \mathbb{P}(D)/\sqsubseteq_R$ is $[A] \cup [B] = [A \cup B]$

Least upper bounds in the domain are determined by set union: for a chain $C = \{[A_i] \mid i \in I\}$, $\bigsqcup C$ is $[\bigcup \{A_i \mid i \in I\}]$; the proof is left as an exercise. Unfortunately, this quotient domain isn't good enough.

12.9 Problem:

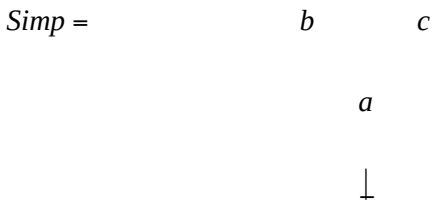
The singleton operation is not continuous. For example, if $c \in D$ is the least upper bound of the chain $\{d_i \mid i \in I\}$ in D, then $\bigsqcup \{[\{d_i\}] \mid i \in I\} = [\bigcup \{\{d_i\} \mid i \in I\}] = [\{d_i \mid i \in I\}]$, but this *not* the same equivalence class as $[\{c\}]$. We can also show that the usual definition of f^+ for continuous f is also discontinuous. The quotient relation is inadequate; a set such as $\{c\}$ must belong to the same equivalence class as $\{d_i \mid i \in I\}$, because both have the same information content.

We must define a quotient relation that better describes the total information content of sets of elements. The best measure of information content was introduced in exercise 20 of Chapter 6: it is the topological open set. Recall that the *Scott-topology* upon a domain D is a collection of subsets of D known as *open sets.* A set $U \subseteq D$ is open in the Scott-topology on D iff:

1. U is closed upwards, that is, for every $d_2 \in D$, if there exists some $d_1 \in U$ such that $d_1 \sqsubseteq_D d_2$, then $d_2 \in U$.
2. If $d \in U$ is the least upper bound of a chain C in D, then some $c \in C$ is in U.

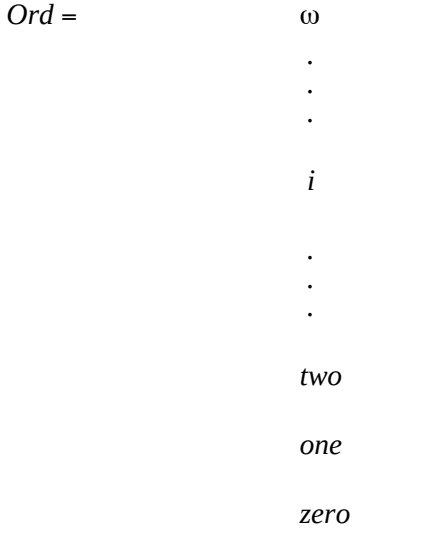
An open set represents a property or an information level. Clause 1 says that if $d_1 \in U$ has enough information to fulfill property U, then so must any d_2 such that $d_1 \sqsubseteq_D d_2$. Clause 2 says that if $\bigsqcup C \in U$ satisifes a property, it is only because it contains some piece of information $c \in C$ that makes it so, and c must satisfy the property, too. These intuitions are justified by exercise 20 of Chapter 6, which shows that a function $f: D \to E$ is partial order continuous iff f is topologically continuous on the Scott-topologies for D and E.

Here is a domain with its open set structure ''drawn in'':



Each semicircular region represents an open set. The open sets of *Simp* are { *b* }, { *c* }, { *a*, *b*, *c* }, { ⊥, *a*, *b*, *c* }, { *b*, *c* } (why?), and ∅ (why?).

A more interesting example is:



Note that { ω } is *not* an open set, for it is the least upper bound of the chain { *zero*, *one*, *two*, ⋯ }, and whenever ω belongs to an open set, so must one of the members of the chain. An open set in *Ord* is either empty or has the structure { *j*, (*j plus one*), (*j plus two*), ⋯ , ω }.

The open sets of a domain *D* define all the properties on *D*. The total information content of a set of elements from *D* can be precisely stated by listing all the open sets to which the elements of the set belong. For sets $A, B \subseteq D$, say that:

$A \sqsubseteq_R B$ iff for every $a \in A$ and open set $U \subseteq D$, if $a \in U$, then there exists a $b \in B$ such that $b \in U$ as well

Further, say that:

$A \simeq_R B$ iff $A \sqsubseteq_R B$ and $B \sqsubseteq_R A$

That is, the elements of *A* and *B* belong to exactly the same collection of open sets in *D*. Note that $A \sqsubseteq_R B$ implies $A \sqsubseteq_R B$. We use the relation $\simeq_R$ to define the equivalence classes in the

general powerdomain construction. The relation equates a chain with a set containing the chain's least upper bound. This solves Problem 12.9.

An alternative presentation of $\sqsubseteq_R$ is done without topological concepts; for $A,B \subseteq D$:

$A \sqsubseteq_R B$ iff for all continuous functions f: $D \to Unit_\perp$, $f(A) \sqsubseteq_R f(B)$

The definition is equivalent to the topological one, for the open sets of a domain D are in one-to-one correspondence with the continuous functions in $D \to Unit_\perp$.

12.10 Definition:

For domain D, the (general) relational powerdomain of D, written $\mathbb{P}_R(D)$*, is the collection of the subsets of the proper elements of D, quotiented by the relation* $\simeq_R$*, partially ordered by* $\sqsubseteq_R$*. The associated operations are defined as:*

$\varnothing : \mathbb{P}_R(D)$ *denotes* $[\{\,\}]$
$\{\_\} : D \to \mathbb{P}_R(D)$ *maps* $d \in D$ *to* $[\{d\}]$
$\_ \cup \_ : \mathbb{P}_R(D) \times \mathbb{P}_R(D) \to \mathbb{P}_R(D)$ *is* $[A] \cup [B] = [A \cup B]$
for f: $D \to \mathbb{P}_R(E)$, $f^+ : \mathbb{P}_R(D) \to \mathbb{P}_R(E)$ *is* $f^+[A] = [\bigcup \{ f(a) \mid a \in A \}]$

The operations are well defined and continuous, and least upper bound corresponds to set union: $\bigsqcup \{ [A_i] \mid i \in I \} = [\bigcup \{ A_i \mid i \in I \}]$. Examples of relational powerdomains are:

$\mathbb{P}_R(Simp) =$

$[\{a,b,c\}, \{b,c\}]$

$[\{a,b\}, \{b\}]$ $\quad$ $[\{a,c\}, \{c\}]$

$[\{a\}]$

$[\{\,\}]$

$\mathbb{P}_R(Ord) =$

$[\{\omega\}, \{j, \ldots, k, \omega\},$
$\{j, j\ plus\ one, j\ plus\ two,$
$\cdots\}, \cdots]$
$\vdots$

$[\{j\}, \{one, j\},$
$\{one, two, j\}, \cdots]$
$\vdots$

$[\{two\}, \{one, two\}]$

$[\{one\}]$

$[\{\,\}]$

Note the difference between $\mathbb{P}(Ord)/\sqsubseteq_R$ and $\mathbb{P}_R(Ord)$: the former makes a distinction between sets containing ω and infinite sets without it, but the latter does not, since both kinds of sets have the same total information content. It is exactly this identification of sets that solves the continuity problem. You should construct $\mathbb{P}_R(D)$ for various examples of flat domains and verify that the domains are identical to the ones built in Section 12.4.

In summary, we can think of the powerdomain $\mathbb{P}_R(D)$ as the set of all subsets of D but must also remember that some sets are equivalent to others in terms of total information content. This equivalence becomes important when the assembly and disassembly operations associated with the powerdomain are defined, for they must be consistent in their mapping of equivalent sets to equivalent answers.

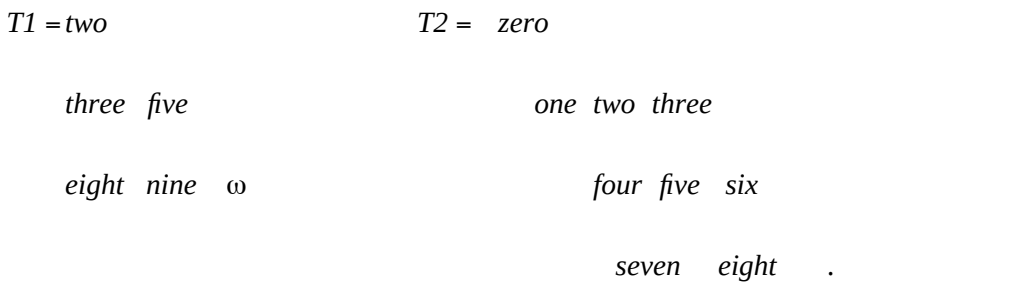
The general Egli-Milner powerdomain, also called the Plotkin powerdomain, is constructed along the same lines as the general relational powerdomain. Differences exist in the sets of elements included and in the quotient relation applied. Since the powerdomain is operationally oriented, sets of elements are chosen that are computationally feasible. Recall that not all subsets of D-elements were used in the discrete Egli-Milner powerdomain: infinite sets included $\perp$. A general definition of acceptable set starts from the notion of a finitely branching generating tree:

12.11 Definition:

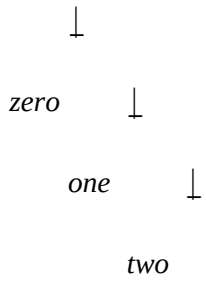
A finitely branching generating tree for domain D is a finitely branching, possibly infinite tree whose nodes are labeled with elements of D such that for all nodes m and n in the tree, if m is an ancestor to n, them m's label is $\sqsubseteq_D$ *n's label.*

Such a tree represents a computation history of a nondeterministic program. The requirement of finite branching forces *bounded nondeterminism*— at any point in the program there exists at most a finite number of possible next computation steps. A path in the tree is a possible computation path; the sequence of labels along a path represent partial outputs; and the least upper bound of the labels along a path represents the final output of that computation. The set of possible outputs for a program is the set of least upper bounds of all the paths of the generating tree. Call this set a *finitely generable set,* and let $\mathbf{F}_g(D)$ be all the finitely generable sets of domain D. The sets used to build the Egli-Milner powerdomain are $\mathbf{F}_g(D)$.

Here are some examples of finitely branching generating trees and their corresponding finitely generable sets. For domain *Ord* and trees:



The paths generated from *T1* are *two, three; two, five, eight; two, five, nine;* and *two, five,* ω. The finitely generable set is { *three, eight, nine,* ω }. For *T2,* the finitely generable set is $\{ n \mid (n \bmod three) \neq zero \} \cup \{ \omega \}$. The ω element is the least upper bound of the infinite path *zero, three, six, nine,* $\cdots$. For domain $\mathbb{N}_\perp$ and tree:



the finitely generable set is { *zero, one, two,* $\cdots$, $\bot$ }. We can prove that any finitely generable infinite set for domain $\mathbb{N}_\bot$ must contain $\bot$: if the set is infinite, its generating tree has an infinite number of nodes; by Konig's lemma, the tree must have an infinite path. The proof that this path must be $\bot$, $\bot$, $\bot$, $\cdots$ is left as an exercise.

Problems 12.8 and 12.9 also arise if the elements $\mathbf{F}_g(D)$ are ordered by $\sqsubseteq_{EM}$. The topology of domain D again comes to the rescue: for $A, B \subseteq D$, $A \sqsubseteq_{EM} B$ iff:

1. For every $a \in A$ and open set $U \subseteq D$, if $a \in U$, then there exists some $b \in B$ such that $b \in U$ as well.
2. For every $b \in B$ and open set $U \subseteq D$, if b is in U's complement $\overline{U}$, then there exists some $a \in A$ such that $a \in \overline{U}$ as well.

Condition 1 was seen in the previous section. Condition 2 states that if B is inadequate in information content with respect to one of its elements, A is inadequate in a similar way. Thus, A can reach an answer in ''no better way'' than B can. The two conditions are embodied in the claim:

$A \sqsubseteq_{EM} B$ iff for all continuous functions $f: D \to Unit_\bot$, $f(A) \sqsubseteq_{EM} f(B)$

We say that $A \simeq_{EM} B$ iff $A \sqsubseteq_{EM} B$ and $B \sqsubseteq_{EM} A$.

12.12 Definition:

For a pointed cpo D, the general Egli-Milner powerdomain of D, written $\mathbb{P}_{EM}(D)$*, is the collection* $\mathbf{F}_g(D)$ *quotiented by the relation* $\simeq_{EM}$*, partially ordered by* $\sqsubseteq_{EM}$.

The definitions of the associated operations are left as exercises.

The general version of the Smyth construction follows the lines of the Egli-Milner construction. The sets used to build the domain are again $\mathbf{F}_g(D)$. For all $A, B \subseteq D$, say that:

$A \sqsubseteq_S B$ iff for every $b \in B$ and open set $U \subseteq D$, if $b \in \overline{U}$, then there exists some $a \in A$ such that $a \in U$ as well

This is clause 2 of the $\sqsubseteq_{EM}$ definition, so:

$A \sqsubseteq_S B$ iff for all continuous function $f: D \to Unit_\bot$, $f(A) \sqsubseteq_S f(B)$

Let $A \simeq_S B$ iff $A \sqsubseteq_S B$ and $B \sqsubseteq_S A$.

12.13 Definition:

For pointed cpo D, the general Smyth powerdomain of D, written $\mathbb{P}_S(D)$*, is the collection* $\mathbf{F}_g(D)$ *quotiented by the relation* $\simeq_S$*, partially ordered by* $\sqsubseteq_S$.

The operations upon $\mathbb{P}_S(D)$ are the expected ones.

SUGGESTED READINGS

Nondeterminism & parallelism: Apt & Olderog 1984; Apt & Plotkin 1981; Dijkstra 1976; Hennessy & Plotkin 1979; Hewitt & Baker 1978; Main & Benson 1984; Park 1981
CCS: Milner 1980, 1983, 1985
Powerdomains: Abramsky 1983; Nielsen, Plotkin, & Winskel 1981; Plotkin 1976, 1982a, 1982b; Smyth 1978, 1983

EXERCISES

1. a. Draw $\mathbb{P}_R(D)$, $\mathbb{P}_{EM}(D)$, and $\mathbb{P}_S(D)$ for each of the following D:
      i. $\mathbb{N}$
      ii. $\mathbb{N}_\perp$
      iii. $\mathbb{B}_\perp \times \mathbb{B}_\perp$
      iv. $\mathbb{B} \to \mathbb{B}$
      v. $\mathbb{B} \to Unit_\perp$
   b. Draw:
      i. $\mathbb{P}_R(\mathbb{P}_R(\mathbb{B}_\perp))$
      ii. $\mathbb{P}_{EM}(\mathbb{P}_{EM}(\mathbb{B}_\perp))$
      iii. $\mathbb{P}_S(\mathbb{P}_S(\mathbb{B}_\perp))$

2. For a flat domain D, model a set $A \in \mathbb{P}(D)$ as a function $A : D \to Unit_\perp$ such that $d \in D$ belongs to A iff $A(d) = ()$.
   a. Define the appropriate functions for $\emptyset$, $\{\_\}$, $\_\cup\_$. To which version of discrete powerdomain is $D \to Unit_\perp$ isomorphic?
   b. What goes wrong when using $A : D \to \mathbb{B}$ and saying that $d \in A$ iff $A(d) = true$? What goes wrong when the definition in part a is applied to nonflat domains?

3. Revise the semantics of the guarded command language of Figure 12.1 so that $Answer = \mathbb{P}_S(Poststore)$. Rewrite the valuation functions so that a command $[\![C]\!]$ that always terminates with a store s_0 has denotation $\mathbf{C}[\![C]\!]s_0 \neq \emptyset$.

4. Redefine the semantics of the **if** statement in Figure 12.4 so that interruption and

interleaving may not occur between evaluation of the test and the first step of evaluation of the chosen clause. Do the same with the semantics of **if** in Figure 12.8.

5. (Plotkin) Extend the syntax of the language of Figure 12.2 to include [[**critical** C]], a critical region construct. Define the resumption and CCS semantics for the construct so that [[C]] evaluates to completion without interruption.

6. a. Let $\mathbf{C}[\![\mathbf{skip}]\!] = \mathit{step}(\lambda s.\, s)$ be added to the language in Figure 12.4. Show that $\mathbf{C}[\![\mathbf{skip};\mathbf{skip}]\!] \neq \mathbf{C}[\![\mathbf{skip}]\!]$, but that $\mathbf{P}[\![\mathbf{skip};\mathbf{skip}]\!] = \mathbf{P}[\![\mathbf{skip}]\!]$.
   b. Let the CCS semantics of the construct be $\mathbf{C}[\![\mathbf{skip}]\!] = \mathit{nil}$; show that $\mathbf{C}[\![\mathbf{skip};\mathbf{skip}]\!] = \mathbf{C}[\![\mathbf{skip}]\!]$.

7. Give two commands in the language of Figure 12.8:
   a. That have the same semantics for the relational powerdomain but different semantics for the Egli-Milner and Smyth powerdomains.
   b. That have the same semantics for the Smyth powerdomain but different semantics for the Egli-Milner and relational powerdomains.

8. Consider the domain of expression resumptions:

$$\mathit{Expr\text{-}res} = \mathit{Store} \to \mathbb{P}((\mathit{Expressible\text{-}value} \times \mathit{Store}) + (\mathit{Store} \times \mathit{Expr\text{-}res}))$$

   a. Use *Expr-res* to define the semantics of interleaved evaluation of expressions E ::= E_1+E_2 | N | I .
   b. Integrate the semantics you defined in part a with the language in Figure 12.4.
   c. Repeat parts a and b using CCS semantics and the semantics in Figure 12.8.

9. a. Show (or describe) all the closed derivations of the program in Figure 12.9. Draw the corresponding behavior tree, showing just the closed derivations.
   b. Using the semantics in Figure 12.8, draw the behavior tree denotations for **C**[[X:=0]] and **C**[[**if** X:=0 **then** Y:=0 **else** Y:=1]].

10. Rewrite the semantics of the guarded command language using CCS.

11. Use resumption semantics to redefine the semantics of the PROLOG-like language of Figure 9.3 so that the denotation of a program is a set of all the possible successful evaluation strategies that a program can take. Repeat the exercise for CCS semantics.

12. Give a resumption semantics to the CCS notation. Prove the soundness of the derivation rules in Figure 12.7.

13. a. What similarities exist between the resumption semantics method of Section 12.3 and the behavior trees model of Section 12.4? What are the primary differences?
   b. What similarities exist between behavior trees and finite generating trees? What are the primary differences?

14. Prove that the $\{\_\}$, $\_\cup\_$ and f^+ operations are well defined in Definition 12.10; that is, show that the choice of representatives A and B in $[A]\cup[B]$ and $f^+[A]$ do not affect the result.

15. Attempt to solve Problem 12.9 by defining $A \sqsubseteq_R B$ iff $\bigsqcup\{a \mid a\in A\} \sqsubseteq \bigsqcup\{b \mid b\in B\}$. What goes wrong?

16. A set $U\subseteq A$ is (Scott-) *closed* if $A - U$ is open in the Scott-topology on A.

    a. Prove that $U\subseteq A$ is closed iff:

       i. For all $d, e\in A$, if $d\in U$ and $e \sqsubseteq d$ then $e\in U$.
       ii. For all directed $D\subseteq A$, $D\subseteq U$ implies $\bigsqcup D\in U$.

    b. Prove that if cpo A is pointed, the relational powerdomain $\mathbb{P}_R(A)$ is isomorphic to the collection of all the nonempty closed subsets of A partially ordered by subset inclusion.

17. a. Why must $f: D \to \mathbb{P}(E)$ be strict to build $f^+ : \mathbb{P}(D) \to \mathbb{P}(E)$ in the case of the Egli-Milner and Smyth powerdomains?
    b. What problems arise in building and using the Egli-Milner and Smyth powerdomains when D is not pointed?

18. For $x,y\in\mathbb{P}(D)$, for which versions of the powerdomains do the following hold?

    a. $x \sqsubseteq x\cup y$
    b. $x\cup y \sqsubseteq x$
    c. $x = \{x\}\cup\varnothing$
    c. $\varnothing = x\cup\varnothing$
    d. $x\cup x = x$

19. a. Consider $\mathbb{P}_{R'}(D)$, a variant of the relational powerdomain such that the elements consist of all subsets of a cpo D, quotiented by $\simeq_R$, partially ordered by $\sqsubseteq_R$. What is unsatisfactory about $\mathbb{P}_{R'}(D)$?
    b. In a similar fashion, comment on the suitability of $\mathbb{P}_{EM'}(D)$, built from all subsets of D, quotiented by $\simeq_{EM}$, partially ordered by $\sqsubseteq_{EM}$.
    c. In a similar fashion, comment on the suitability of $\mathbb{P}_{S'}(D)$, built from all the subsets of D, quotiented by $\simeq_S$, partially ordered by $\sqsubseteq_S$.

20. a. For each of the powerdomain constructions, attempt to define a continuous function $\_\mathbf{in}\_ : D\times\mathbb{P}(D)\to\mathbb{B}$ such that for all $d\in D$, $U\subseteq D$, $d\,\mathbf{in}\,[U] = \mathit{true}$ iff $d\in U$. (Hint: first attempt to define **in** for the discrete powerdomains and then generalize.)
    b. For each of the powerdomain constructions upon which you succeeded in defining **in**, for all $d\in D$, $U,V\subseteq D$:

       i. Attempt to show $d\,\mathbf{in}\,([U]\cup[V]) = \mathit{true}$ iff $d\,\mathbf{in}\,[U] = \mathit{true}$ or $d\,\mathbf{in}\,[V] = \mathit{true}$.
       ii. Attempt to define a continuous function $\cap : \mathbb{P}(D)\times\mathbb{P}(D)\to\mathbb{P}(D)$ such that $d\,\mathbf{in}\,([U]\cap[V]) = \mathit{true}$ iff $d\,\mathbf{in}\,[U] = \mathit{true}$ and $d\,\mathbf{in}\,[V] = \mathit{true}$.