Chapter 7

Fixed Points

"Bless thee, Bottom! bless thee! thou art translated!"

— *A Midsummer Night's Dream (III.1.99)*

"Bottom! O most courageous day! O most happy hour!"

— *ibid., (IV.2.26)*

Recursive definitions are a powerful and elegant tool for specifying complex structures and processes. While such definitions are second nature to experienced programmers, novices are often mystified by recursive definitions. Their confusion often centers on the following question: "how can something be defined in terms of itself?" Sometimes there is a justifiable cause for confusion — not all recursive definitions make sense!

In this chapter, we carve out a class of recursive definitions that *do* make sense, and present a technique for assigning meaning to them. The technique involves finding a fixed point of a function derived from the recursive definition. We will make extensive use of this technique in our denotational descriptions of programming languages to define recursive valuation functions and recursive domains.

7.1 The Fixed Point Game

7.1.1 Recursive Definitions

For our purposes, a **recursive definition** is an equation of the form

$$x = \ldots x \ldots$$

where ... x ... designates a mathematical expression that contains occurrences of the defined variable x. Mutually recursive definitions of the form

$$\begin{aligned} x_1 &= \ldots x_1 \ldots x_2 \ldots x_n \ldots \\ x_2 &= \ldots x_1 \ldots x_2 \ldots x_n \ldots \\ &\vdots \\ x_n &= \ldots x_1 \ldots x_2 \ldots x_n \ldots \end{aligned}$$

can always be rephrased as a single recursive definition,

$$\begin{aligned} x = \langle &\ldots (\mathit{Proj}\,1\ \ x\ ) \ldots (\mathit{Proj}\,2\ \ x\ ) \ldots (\mathit{Proj}\,n\ \ x\ ) \ldots, \\ &\ldots (\mathit{Proj}\,1\ \ x\ ) \ldots (\mathit{Proj}\,2\ \ x\ ) \ldots (\mathit{Proj}\,n\ \ x\ ) \ldots, \\ &\vdots \\ &\ldots (\mathit{Proj}\,1\ \ x\ ) \ldots (\mathit{Proj}\,2\ \ x\ ) \ldots (\mathit{Proj}\,n\ \ x\ ) \ldots \rangle \end{aligned}$$

where x stands for the tuple $\langle x_1, x_2, \ldots, x_n \rangle$ and $\mathit{Proj}\,i$ extracts the ith element of the tuple. For this reason, it is sufficient to focus on recursive definitions involving a single variable.

A **solution** to a recursive definition is a value that makes the equation true when substituted for all occurences of the defined variable. A recursive definition may have zero, one, or more solutions. For example, suppose that x ranges over the integers. Then:

- $x = 1 + x$ has no solutions;
- $x = 4 - x$ has exactly one solution;
- $x = 9 \div x$ has two solutions;
- $x = x$ has an infinite number of solutions.

It is important to specify the domain of the defined variable in a recursive definition, since the set of solutions depends on this domain. For example, the recursive definition $x = 1 \div 16x^3$ has

- zero solutions over the integers[1] ;
- one solution over the positive rationals ($\frac{1}{2}$);
- two solutions over the rationals ($\frac{1}{2}$, $-\frac{1}{2}$);
- four solutions over the complex numbers ($\frac{1}{2}$, $-\frac{1}{2}$, $\frac{i}{2}$, $-\frac{i}{2}$).

In fact, many numerical domains were invented precisely to solve classes of equations that were insoluble with existing domains.

Although we are most familiar with equations that involve numeric variables, equations can involve variables from *any* domain, including product, sum, sequence, and function domains. For example, consider the following recursive definitions involving an element s of the sequence domain $Nat*$:

- The equation $s\ = (cons\ 3\ (cons\ (head\ s)\ [\,]))$ has the unique solution $[3, 3]$.
- The equation $s\ = (cons\ (head\ s)\ (cons\ (1\ +\ (head\ s))\ [\,]))$ has an infinite number of solutions of the form $[n, n + 1]$, where $n : Nat$.
- The equation $s\ =\ (cons\ (1\ +\ (head\ s))\ (cons\ (head\ s)\ [\,]))$ has no solutions in $Nat*$. The first element i of a solution s would have to satisfy the equation $n = n + 1$, and this equation has no solution.

We will be especially interested in recursive definitions over function domains. Suppose that f is an element of the domain $Nat \rightarrow Nat$. Consider the following recursive function definition of f:

$$f = \lambda n\ .\ \mathbf{if}\ (n\ =\ 0)\ \mathbf{then}\ 0\ \mathbf{else}\ (2\ +\ (f\ \ (n\ -\ 1)))\ \mathbf{fi}$$

Intuitively, this equation is solved when f is a doubling function, but how do we show this more formally? Recall that a function in $Nat \rightarrow Nat$ can be viewed as its graph, the set of input/output pairs for the function. The graph associated with the lambda expression is

$$\begin{aligned}
\{&\langle 0,\ \mathbf{if}\ (0\ =\ 0)\ \mathbf{then}\ 0\ \mathbf{else}\ (2\ +\ (f\ \ 0))\rangle, \\
&\langle 1,\ \mathbf{if}\ (1\ =\ 0)\ \mathbf{then}\ 0\ \mathbf{else}\ (2\ +\ (f\ \ 0))\rangle, \\
&\langle 2,\ \mathbf{if}\ (2\ =\ 0)\ \mathbf{then}\ 0\ \mathbf{else}\ (2\ +\ (f\ \ 1))\rangle, \\
&\langle 3,\ \mathbf{if}\ (3\ =\ 0)\ \mathbf{then}\ 0\ \mathbf{else}\ (2\ +\ (f\ \ 2))\rangle, \\
&\ldots\}.
\end{aligned}$$

[1] In this case, $\div$ is interpreted as a quotient function on integers.

After simplification, this becomes

$$\begin{array}{l}\{\langle 0,\ 0\rangle, \\ \ \langle 1,\ (2\ +\ (f\ \ 0))\rangle, \\ \ \langle 2,\ (2\ +\ (f\ \ 1))\rangle, \\ \ \langle 3,\ (2\ +\ (f\ \ 2))\rangle, \\ \ \ldots\}.\end{array}$$

If f is a doubling function, then the graph of the right-hand side can be further simplified to

$$\{\langle 0,\ 0\rangle,\ \langle 1,\ 2\rangle,\ \langle 2,\ 4\rangle,\ \langle 3,\ 6\rangle,\ \ldots\}$$

This is exactly the graph of the doubling function f on the left-hand side of the equation, so the equation holds true. It is not difficult to show that the doubling function is the *only* solution to the equation; we leave this as an exercise.

As with recursive definitions over other domains, recursive definitions of functions may have zero, one, or more solutions. Maintaining the assumption that f is in $\mathit{Nat} \to \mathit{Nat}$, the definition

$$f \;=\; \lambda n\,.\,(1\ +\ (f\ \ n))$$

has zero solutions, because the result n_{r} for any given input would have to satisfy $n_{\mathrm{r}} = n_{\mathrm{r}} + 1$. On the other hand, the definition

$$f \;=\; \lambda n\,.\,(f\ \ (1\ +\ n))$$

has an infinite number of solutions; for any given constant n_{c}, a function with the graph $\{\langle n,\ n_{\mathrm{c}}\rangle \mid n : \mathit{Nat}\}$ is a solution to the equation.

7.1.2 Fixed Points

If d ranges over domain D, then a recursive definition

$$d \;=\; \ldots d \ldots$$

can always be encoded as the $D \to D$ function

$$\lambda d\,.\ \ldots d \ldots$$

We will call this the **generating function** for the recursive definition. For example, if $r : Real$, the numeric equation

$$r \;=\; 1 - r^2$$

can be represented by the $Real \rightarrow Real$ generating function

$$\lambda r \,.\, (1 \;-\; (r \;*\; r)).$$

Similarly, the recursive function definition

$$dbl : Nat \rightarrow Nat = \lambda n \,.\, \textbf{if}\, (n \;=\; 0)\, \textbf{then}\, 0\, \textbf{else}\, (2 \;+\; (dbl\; (n \;-\; 1)))\, \textbf{fi}$$

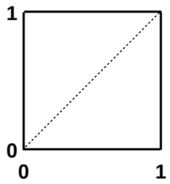
can be represented by the generating function

$$\begin{aligned}&g_{dbl} : (Nat \rightarrow Nat) \rightarrow (Nat \rightarrow Nat)\\ &= \lambda f \,.\, \lambda n \,.\, \textbf{if}\, (n \;=\; 0)\;\; \textbf{then}\, 0\;\; \textbf{else}\, (2 \;+\; (f\;\; (n \;-\; 1)))\;\; \textbf{fi}\end{aligned}$$

where $f : Nat \rightarrow Nat$. A generating function is not recursive, so its meaning can be straightforwardly determined from its component parts.

A solution to a recursive definition is a fixed point of its associated generating function. A **fixed point** of a function $g : D \rightarrow D$ is an element $d : D$ such that $(g\; d) = d$. If a function in $D \rightarrow D$ is viewed as moving elements around the space D, elements satisfying this definition are the only ones that remain stationary; hence the name "fixed point".

To build intuitions about fixed points, it is helpful to consider functions from the unit interval[2] $[0, 1]$ to itself. Such a function can be graphed in the following box:



Every point where the function graph intersects the $y = x$ diagonal is a fixed point of the function. For example, Figure 7.1 shows the graphs of functions with zero, one, and two fixed points.

It is especially worthwhile to consider how a generating function like g_{dbl} moves elements around a domain of functions. Here are a few examples of how g_{dbl} maps various functions $f : Nat \rightarrow Nat$:

[2]The unit interval is the set of real numbers between 0 and 1, inclusive.

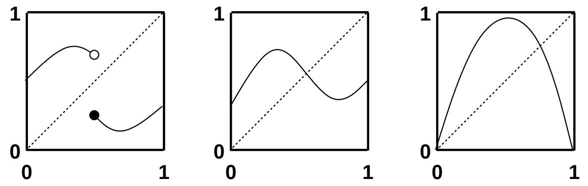


Figure 7.1: Functions on the unit interval with zero, one, and two fixed points.

- If f is the identity function $\lambda n \,.\, n$, then $(g_{dbl}\ f)$ is the function that increments positive numbers and returns 0 for 0:

$$\lambda n \,.\, \textbf{if}\ (n\ =\ 0)\ \textbf{then}\ 0\ \textbf{else}\ (n\ +\ 1)\ \textbf{fi}$$

- If f is the function $\lambda n \,.\ ((n\ +\ 1)^2\ -\ 2)$ then $(g_{dbl}\ f)$ is the function $\lambda n \,.\, n^2$
- If f is a doubling function, then $(g_{dbl}\ f)$ is also the doubling function, so the doubling function is a fixed point of g_{dbl}. Indeed, it is the only fixed point of g_{dbl}.

Since functions $D \to D$ correspond to recursive definitions, their fixed points have all the properties of solutions to recursive definitions. In particular, such a function may have zero, one, or more fixed points, and the existence and character of fixed points depends on the details of the function and the nature of the domain D.

7.1.3 The Iterative Fixed Point Technique

Above, we saw that recursive definitions can make sense over any domain. However, the methods we used to find and/or verify solutions in the examples were rather ad hoc. In the case of numeric definitions, there are many familiar techniques for manipulating equations to find solutions. Are there any techniques that will help us solve recursive definitions over more general domains?

It turns out that there is a class of recursive definitions for which an **iterative fixed point technique** will find a distinguished solution of the definition. This technique finds a unique fixed point to the generating function of the recursive definition. Here, we sketch the basic idea of this technique; later sections flesh out the details and present more examples.

The iterative fixed point technique is motivated by the observation that it is often possible to find a fixed point for a generating function by iterating the function starting with an appropriate initial value. For example: in Section 6.5.3 the `fixed-point` procedure is used to iterate a rewriting process over an initial instruction sequence until no more rules are applicable. The final instruction sequence (if it exists) is a fixed point of the rewriting process.

As a more graphical example of the iteration technique, consider a transformation T on two-dimensional line drawings that is the sequential composition of the following three steps:

1. Rotate the drawing 90 degrees counter-clockwise about the origin.

2. Translate the drawing right by one unit.

3. Add a line from (0,0) to (0,1).

Figure 7.2 shows what happens when T is iterated starting with the empty drawing. Each of the first four applications of T adds a new line until the unit square is produced. Subsequent applications of T do not modify the square; it is a fixed point of T.

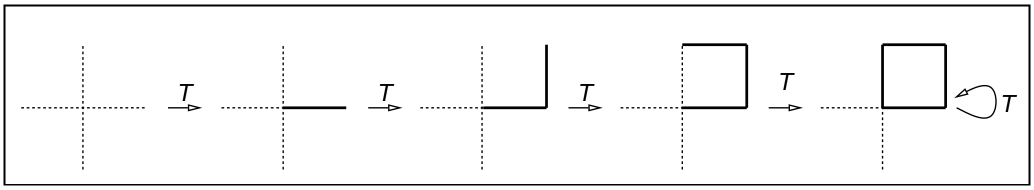


Figure 7.2: Iterating the transformation T starting with an empty line drawing leads to a fixed point in fours steps.

In the line drawing example, a fixed point was reached after four iterations of the transformation. Often, iterating a generation function does not yield a fixed point in a finite number of steps, but only approaches one in the limit. A classic numerical example is finding square roots. The square root of a non-negative rational n is a solution of the recursive definition

$$x = \frac{x + \frac{n}{x}}{2}.$$

Iterating the generation function for this definition starting with n yields a sequence of approximations that converge to $\sqrt{n}$. For example, for $n = 3$ the first few iteration steps are:

$$\begin{array}{rcl}
(g_{sqrt3}{}^{0}\ 3) & = & 3 \\
(g_{sqrt3}{}^{1}\ 3) & = & 2 \\
(g_{sqrt3}{}^{2}\ 3) & = & \frac{7}{4} = 1.75 \\
(g_{sqrt3}{}^{3}\ 3) & = & \frac{97}{56} \approx 1.7321428571428572 \\
(g_{sqrt3}{}^{4}\ 3) & = & \frac{18817}{10864} \approx 1.7320508100147276 \\
 & \vdots &
\end{array}$$

Even in non-numeric domains, generating functions can produce sequences of values approaching a limiting fixed point. For example, consider the following recursive definition of the even natural numbers:

$$evens = \{0\} \cup \{x + 2 \mid x \in evens\}$$

The associated generating function is

$$g_{evens} : \mathcal{P}(Nat) \to \mathcal{P}(Nat) = \lambda s\ .\ \{0\} \cup \{(n\ +\ 2) \mid n \in s\}$$

where s ranges over the powerset of Nat. Then iterating g_{evens} starting with the empty set yields a sequence of sets that approaches the set of even numbers in the limit:

$$\begin{array}{rcl}
(g_{evens}{}^{0}\ \{\}) & = & \{\} \\
(g_{evens}{}^{1}\ \{\}) & = & \{0\} \\
(g_{evens}{}^{2}\ \{\}) & = & \{0,\ 2\} \\
(g_{evens}{}^{3}\ \{\}) & = & \{0,\ 2,\ 4\} \\
(g_{evens}{}^{4}\ \{\}) & = & \{0,\ 2,\ 4,\ 6\} \\
 & \vdots &
\end{array}$$

The above examples of the iterative fixed point technique involve different domains but exhibit a common structure. In each case, the generating function maps an approximation of the fixed point into a better approximation, where the notion of "better" depends on the details of the function:

- In the program simplifier, program a is better than program b if the instruction count of a is less than or equal to the instruction count of b.

- In the line drawing example, picture a is better than picture b if a contains at least as many lines of the unit square as b.
- In the square root example, number a is better than number b if $|a^2 - n| \leq |b^2 - n|$.
- In the even number example, set a is better than set b if $b \subseteq a$.

Moreover, in each of the examples, the sequence of approximations produced by the generating functions converges to a fixed point in the limit. This doesn't necessarily follow from the fact that each approximation is better than the previous one. For example, each element of the series $0, 0.9, 0.99, 0.999, \ldots$ is closer to $\sqrt{2}$ than the previous element, but the series converges to 1, not to $\sqrt{2}$. The notion of approaching a limiting value is central to the iterative fixed point technique.

The basic structure of the iterative fixed point technique is depicted in Figure 7.3. The generating function $g : D \to D$ is defined over a domain D whose values are assumed to be ordered by their information content. A line connects two values when the lower value is an approximation to the higher value; that is, the higher value contains all the information of the lower value plus some extra information. What counts as "information" and "approximation" depends on the problem domain. When values are sets, for instance, a line from a up to b might indicate that $a \subset b$.

In the iterative fixed point technique, iteratively applying g from an appropriate starting value d_0 yields a sequence of values with increasing information content. Intuitively, iterative applications of g climb up through the ordered values by refining the information of successive approximations. If this process reaches a value d_i such that $d_i = (g\ d_i)$, then the fixed point d_i has been found. If this process never actually reaches a fixed point, it should at least approach a fixed point as a limiting value.

We emphasize that the iterative fixed point technique does not work for every generating function — it depends on the details of the generating function as well as the starting point. The technique must certainly fail for generating functions that have no fixed points. Even when a generating function has a fixed point, iteration won't necessarily find it. For example, iterating the generating function for $x = \frac{3}{x}$ starting with any non-zero rational number yields an alternating sequence that never gets any closer to the fixed point, $\sqrt{3}$. To see that starting point matters, consider calling g_{evens} on an initial set that contains the number 1. The iteration of g_{evens} will converge to the set of *all* natural numbers, not just the even numbers. This is a fixed point of g_{evens}, but not the necessarily the one that we were trying to find!

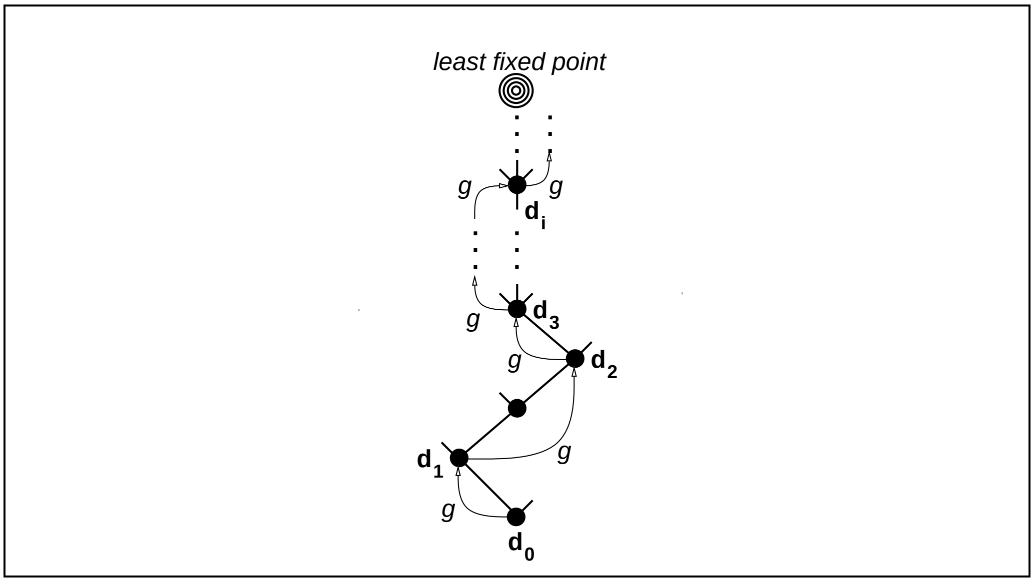


Figure 7.3: The "game board" for the iterative fixed point technique.

In the next section, we will describe an important class of generating functions that are *guaranteed* to have a fixed point. A fixed point of these functions (the so-called **continuous functions on pointed CPOs**) can be found by applying the iterative fixed point technique starting with a special informationless element called **bottom**. Such functions may have more than one fixed point, but the one found by iterating from bottom has less information than all the others — it is the **least fixed point**. We will choose this distinguished fixed point as *the* solution of the associated recursive definition. It turns out that this solution matches our operational intuitions about what solution the computer will find when the recursive definition is expressed as a program. As long as we restrict our attention to this special class of functions, we are guaranteed to be able to solve any recursive definition.

▷ **Exercise 7.1** Above, we showed that the line drawing transformation T discussed above has the unit square as a fixed point. How many fixed points does T have? ◁

▷ **Exercise 7.2** For each of the following classes of functions from the unit interval to itself, indicate the minimum and maximum number of fixed points of functions in the class.

a. constant functions (i.e., functions of the form $\lambda x \,.\, a$);

b. linear functions (i.e., functions of the form $\lambda x \,.\, ax + b$);

c. quadratic functions (i.e., functions of the form $\lambda x \, . \, ax^2 + bx + c$);

d. continuous functions (i.e., functions whose graph is an unbroken curve);

e. non-decreasing functions (i.e., functions f for which $a \leq b$ implies $(f \;\, a) \leq (f \;\, b)$.

f. non-increasing functions (i.e., functions f for which $a \leq b$ implies $(f \;\, a) \geq (f \;\, b)$.

$\triangleleft$

7.2 Fixed Point Machinery

Here we present the machinery necessary to define a class of functions for which a distinguished fixed point always exists. Our presentation is influenced by Schmidt [Sch86] and by Gunter and Scott [GS90].

7.2.1 Partial Orders

A **partial order** is a pair $\langle D, \sqsubseteq \rangle$ of a domain D and a relation $\sqsubseteq$ that is reflexive, transitive, and anti-symmetric. A relation is **anti-symmetric** if $a \sqsubseteq b$ and $b \sqsubseteq a$ together imply $a = b$. The notation $a \sqsubseteq b$ is pronounced "a is weaker than b" or "b is stronger than a". Later, we shall be ordering elements by information content, so we will also pronounce $a \sqsubseteq b$ as "a approximates b". When the relation $\sqsubseteq$ is understood from context, it is common to refer to the partial order $\langle D, \sqsubseteq \rangle$ as D.

Partial orders are commonly depicted by **Hasse diagrams** in which elements (represented by points) are connected by lines. In such a diagram, $a \sqsubseteq b$ if and only if there is a path from the point representing a to the point representing b such that each link of the path goes upward on the page. For example, Figure 7.4 shows the Hasse diagram for the partial order PO on six symbols whose relation is defined by the following graph:

$$\{\langle \texttt{a, a}\rangle, \langle \texttt{a, d}\rangle, \langle \texttt{a, e}\rangle, \langle \texttt{a, f}\rangle, \ \langle \texttt{b, b}\rangle, \langle \texttt{b, d}\rangle, \langle \texttt{b, e}\rangle, \langle \texttt{b, f}\rangle, \\ \langle \texttt{c, c}\rangle, \langle \texttt{c, e}\rangle, \langle \texttt{c, f}\rangle, \ \langle \texttt{d, d}\rangle, \langle \texttt{d, e}\rangle, \langle \texttt{d, f}\rangle, \ \langle \texttt{e, e}\rangle, \ \langle \texttt{f, f}\rangle \ \}$$

Elements of a partial order are not necessarily related. Two elements of a partial order that are unrelated by $\sqsubseteq$ are said to be **incomparable**. For example, here is a listing of all the pairs of incomparable elements in PO: $\{\texttt{a, b}\}$, $\{\texttt{a, c}\}$, $\{\texttt{b, c}\}$, $\{\texttt{c, d}\}$, and $\{\texttt{e, f}\}$.

An **upper bound** of a subset X of a partial order D is an element $u \in D$ that is stronger than every element of X; i.e., for every x in X, $x \sqsubseteq u$. In PO, the subset $\{\texttt{a}, \texttt{b}\}$ has upper bounds `d`, `e`, and `f`; the subset $\{\texttt{a}, \texttt{b}, \texttt{c}\}$ has upper

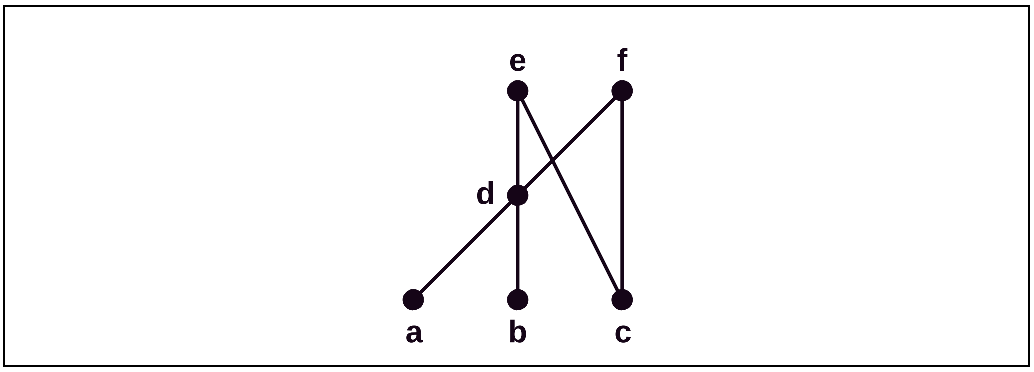


Figure 7.4: A Hasse diagram for the partial order PO.

bounds `e` and `f`; and the subset $\{\texttt{e}, \texttt{f}\}$ has no upper bounds. The **least upper bound (lub)** of a subset X of D, written $\bigsqcup_D X$, is the upper bound of X that is weaker than every other upper bound of X; such an element may not exist. In PO, the lub of $\{\texttt{a}, \texttt{b}\}$ is `d`, but neither $\{\texttt{a}, \texttt{b}, \texttt{c}\}$ nor $\{\texttt{e}, \texttt{f}\}$ has a lub. There are symmetric notions of **lower bound** and **greatest lower bound (glb**[3] **)**, but our fixed point machinery will mainly use upper bounds.

An element that is weaker than all other elements in a partial order D is called the **bottom** element and is denoted $\bot_D$. Symmetrically, an element that is stronger that all other elements in D is the **top** element (written $\top_D$). Bottom and top elements do not necessarily exist. For example, PO has neither.

Any partial order D can be **lifted** to another partial order $D_\bot$ that has all the elements and orderings of D, but includes a new element $\bot_{D_\bot}$ that is weaker than all elements of D. If D already has a bottom element $\bot_D$, then $\bot_D$ and $\bot_{D_\bot}$ are disinct, with $\bot_{D_\bot}$ being the weaker of the two. Symmetrically, the notation $D^\top$ designates the result of extending D with a new top element.

A **discrete** partial order is one in which every pair of elements is incomparable. By default, we will assume that primitive semantic domains have the discrete ordering. For example, Figure 7.5 depicts the discrete ordering for *Nat*. In this partial order, numbers are not ordered by their value, but by their information content. Each number approximates only itself.

A **flat** partial order D is a lifted discrete partial order. Flat partial orders will play an important role in our treatment of semantic domains. Figure 7.6 depicts the flat partial order $Nat_\bot$ of natural numbers. Note that $\bot_{Nat_\bot}$ acts as an "unknown natural number" that approximates every natural number.

[3]Pronounced "glub".

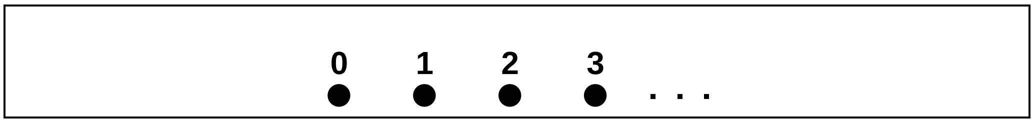


Figure 7.5: The domain *Nat* is assumed to have the discrete ordering.

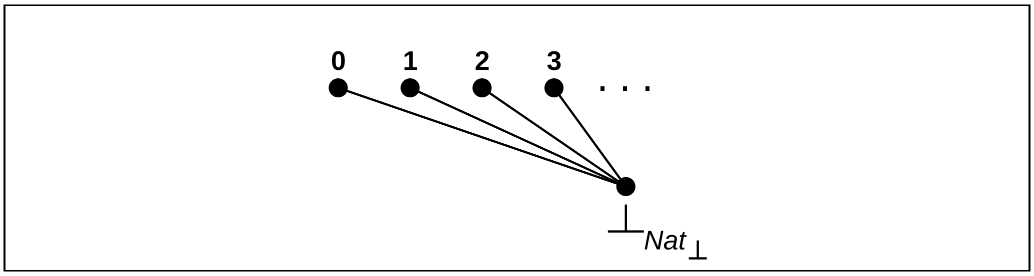


Figure 7.6: The lifted flat partial order $Nat_\perp$.

A **total order** is a partial order in which every two elements are related (i.e., no two elements are incomparable). For example, the natural numbers under the traditional value-based ordering form a total order called ω. The elements of a total order can be arranged in a vertical line in a Hasse diagram (see Figure 7.7).

A **chain** is a totally-ordered non-empty subset of a partial order. The chains of *PO* include $\{\mathtt{a},\mathtt{d},\mathtt{e}\}$, $\{\mathtt{c},\mathtt{f}\}$, $\{\mathtt{b},\mathtt{f}\}$, and $\{\mathtt{d}\}$. In $Nat_\perp$, the only chains are (1) singleton sets and (2) doubleton sets containing $\perp_{Nat_\perp}$.

Given partially ordered domains, we would like to define orderings on product, sum, sequence, and function domains such that the resulting domains are also partially ordered. That way, we will be able to view our semantic domains as partial orders. In the following definitions, assume that D and E are arbitrary partial orders ordered by $\sqsubseteq_D$ and $\sqsubseteq_E$, respectively. We will illustrate the definitions with examples involving the two concrete partial orders G and H in Figure 7.8.

Product Domains

$D \times E$ is a partial order under the following ordering:

$$\langle d_1,\ e_1\rangle \sqsubseteq_{D\times E} \langle d_2,\ e_2\rangle \text{ if } d_1 \sqsubseteq_D d_2 \text{ and } e_1 \sqsubseteq_E e_2.$$

The partial order $G \times H$ is depicted in Figure 7.9. Note how the Haase diagram for $G \times H$ is visually the product of the Hasse diagrams for G and H. That is

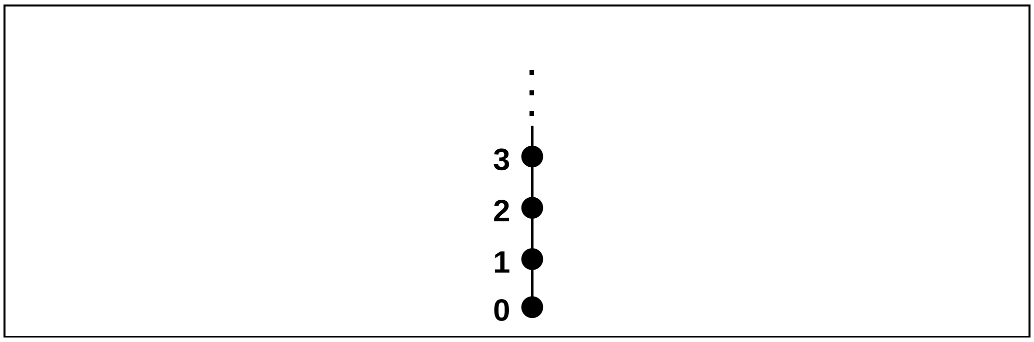


Figure 7.7: The partial order ω of natural numbers under the traditional value-based ordering.

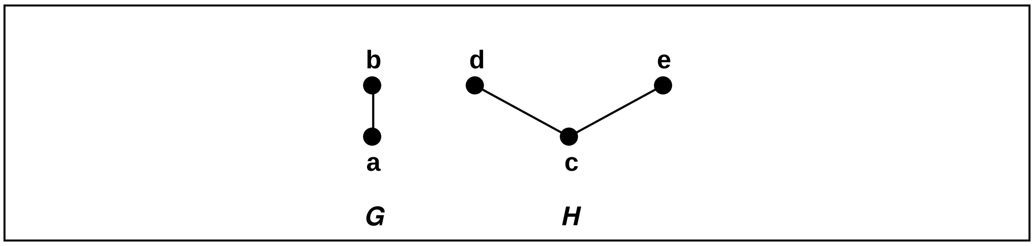


Figure 7.8: Two simple partial orders.

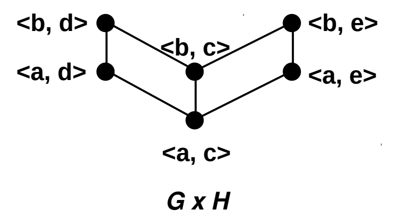


Figure 7.9: The product partial order $G \times H$.

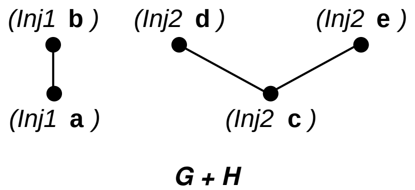


Figure 7.10: The sum partial order $G + H$.

$G \times H$ results from making a copy of G at every point of H (or, symmetrically, making a copy of H at every point of G) and adding the extra lines specified by the ordering.

Sum Domains

$D + E$ is a partial order under the following ordering:

$$(\mathit{Inj1}_{D,E}\ d_1) \sqsubseteq_{D\,+\,E} (\mathit{Inj1}_{D,E}\ d_2) \text{ if } d_1 \sqsubseteq_D d_2$$

$$(\mathit{Inj2}_{D,E}\ e_1) \sqsubseteq_{D\,+\,E} (\mathit{Inj2}_{D,E}\ e_2) \text{ if } e_1 \sqsubseteq_E e_2$$

This ordering preserves the order between elements of the same summand, but treats elements from different summands as incomparable. The Hasse diagram for a sum partial order is simply the juxtaposition of the digrams for the summands (see Figure 7.10).

Function Domains

$D \rightarrow E$ is a partial order under the following ordering:

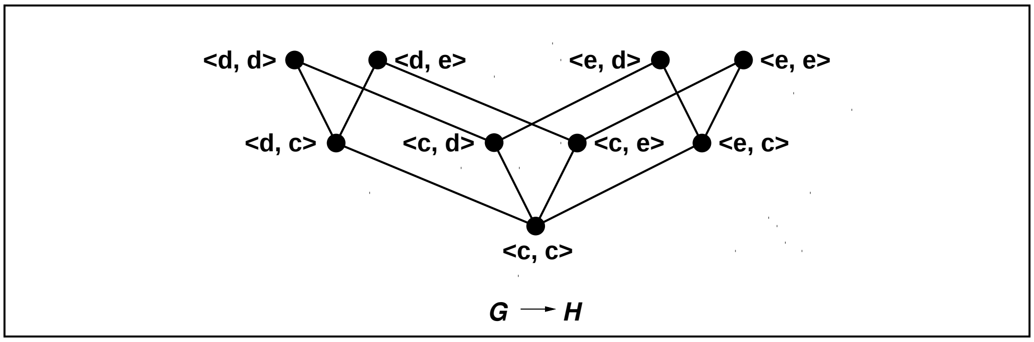


Figure 7.11: The function partial order $G \rightarrow H$.

$$f_1 \sqsubseteq_{D \rightarrow E} f_2 \text{ if, for all } d \text{ in } D, (f_1\ d) \sqsubseteq_E (f_2\ d).$$

Consider using this ordering on the elements of $G \rightarrow H$. As usual, a total function from G to H can be represented by a graph of input/output pairs, but here we employ a more compact notation in which such a function is represented as a pair of the elements that `a` and `b` map to, respectively. Thus, the function with graph $\{\langle$`a, c`$\rangle, \langle$`b, d`$\rangle\}$ can be abbreviated as $\langle$`c, d`$\rangle$. Using this notation, the partial order $G \rightarrow H$ is isomorphic[4] to the partial order $H \times H$ (see Figure 7.11). This sort of isomorphism holds whenever D is a finite domain. That is, if D has n elements, then $D \rightarrow E$ is isomorphic to E^n.

Sequence Domains

There are two common ways to order the elements of $D*$. These differ in whether sequence elements of different lengths are comparable.

- Under the **prefix ordering**,

  $$[d_1,\ d_2,\ \ldots,\ d_k] \sqsubseteq_{D*} [d_1{}',\ d_2{}',\ \ldots,\ d_l{}'] \text{ if } k \leq l \text{ and } d_i \sqsubseteq_D d_i{}' \text{ for all } 1 \leq i \leq k$$

  If D is a discrete domain, this implies that a sequence s_1 is weaker than s_2 if s_1 is a prefix of s_2 — i.e., $s_2 = s_1 @ s'$ for some sequence s'.

[4]Informally, two partial orders are isomorphic if their Hasse diagrams can be rearranged to have the same shape (ignoring the labels on the vertices). Formally, two partial orders A and B are isomorphic if there is a bijective function $f : A \rightarrow B$ that preserves ordering in both directions. That is, $a \sqsubseteq_A a'$ implies $(f\ a) \sqsubseteq_B (f\ a')$ and $b \sqsubseteq_B b'$ implies $(f^{-1}\ b) \sqsubseteq_A (f^{-1}\ b')$.

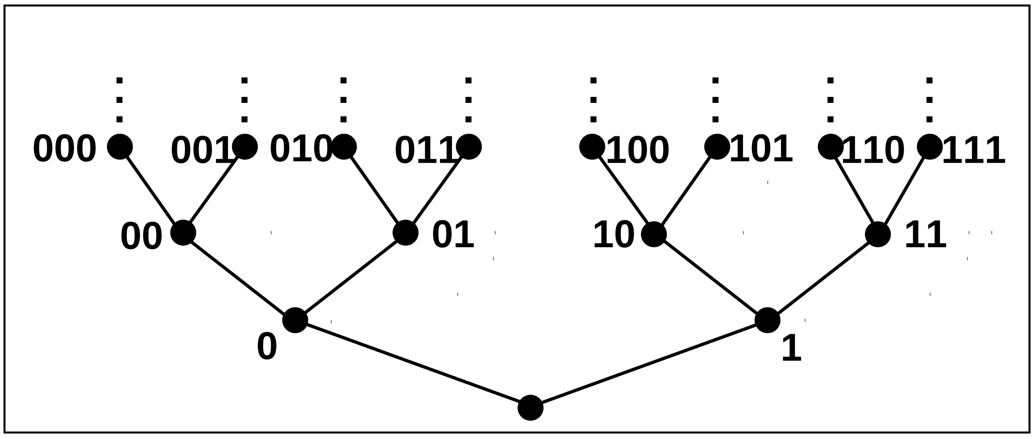


Figure 7.12: The sequence partial order Bit^* under the prefix ordering.

As an example, suppose that Bit is the discrete partial order of the binary digits `0` and `1`. Then Bit^* under the prefix order is isomorphic to the partial order of binary numerals shown in Figure 7.12. (For example, the numeral `110` corresponds to the sequence $[\mathtt{1}, \mathtt{1}, \mathtt{0}]$.) This partial order is an infinite binary tree rooted at the empty sequence. Each element of the tree can be viewed as an approximation to all of the elements of the subtree rooted at it. For example, `110` is an approximation to `1100`, `1101`, `11000`, `11001`, `11010`, etc. In computational terms, this notion of approximation corresponds to the behavior of a computation process that produces its answer by printing out a string of `0`s and `1`s from left to right, one character at a time. At any point in time, the characters already printed are the current approximation to the final string that will be produced by the process.

Note that if D has some non-trivial ordering relations, the prefix ordering of D^* is more complex than a simple tree.

- Under the **sum-of-products ordering**, D^* is treated as isomorphic to the infinite sum of products

$$D^0 + D^1 + D^2 + D^3 + \cdots$$

As in the prefix ordering, sequences are ordered componentwise by their elements, but the sum-of-products ordering treats sequences of different lengths as incomparable. For example, under the sum-of-products ordering, $Bit_\perp{}^*$ is isomorphic to the partial order depicted in Figure 7.13.

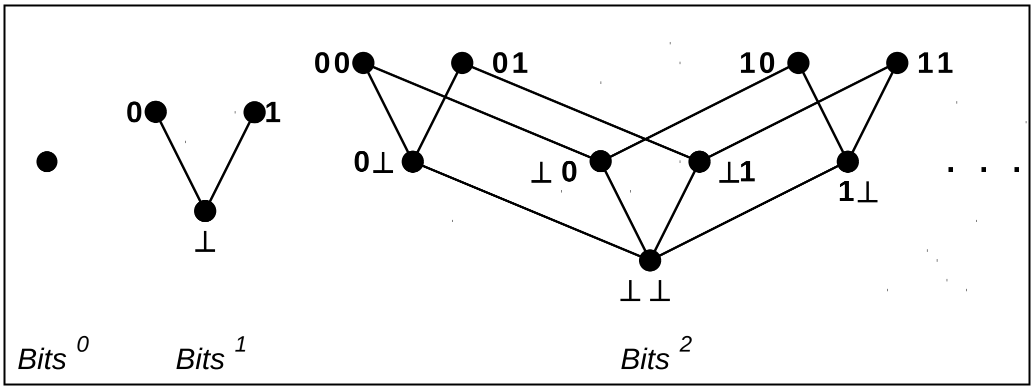


Figure 7.13: The sequence partial order G^* under the sum-of-products ordering.

Although we have stated that the above definitions are partial orders, we have not argued that each ordering is in fact reflexive, transitive, and antisymmetric. We encourage the reader to show that these properties hold for each of the definitions.

The orderings defined above are not the only ways to order compound domains, but they are relatively natural and are useful in many situations. Later, we will refine some of these orderings (particularly in the case of function domains). But, for the most part, these are the orderings that will prove useful for our study of semantic domains.

▷ **Exercise 7.3** Using the partial orders G and H in Figure 7.8, draw a Hasse diagram for each of the following compound partial orders:

a. $G \times G$

b. $H \times H$

c. $G \to G$

d. $H \to H$

e. $H \to G$

f. G^* under the prefix ordering (show the first four levels)

g. H^* under the prefix ordering (show the first four levels)

h. G^* under the sum-of-products ordering (show the first three summands)

i. H^* under the sum-of-products ordering (show the first three summands)

◁

▷ **Exercise 7.4** Suppose that A and B are finite partial orders with the same number of elements, but they are not isomorphic. Partition the following partial orders into equivalence classes based on isomorphism. That is, each class should contain all the partial orders that are isomorphic to each other.

$$\begin{array}{l} A \times A,\ A \times B,\ \ B \times A,\ B \times B, \\ A + A,\ A + B,\ \ B + A,\ B + B, \\ A \to A,\ A \to B,\ \ B \to A,\ B \to B \end{array}$$

◁

▷ **Exercise 7.5** Given a discretely ordered domain D, the powerset $\mathcal{P}(D)$ is a partial order under the **subset ordering**:

$$S \sqsubseteq_{\mathcal{P}(D)} S' \text{ if } S \subseteq S'$$

Draw the Hasse diagram for the partial order $\mathcal{P}(\{\mathtt{a}, \mathtt{b}, \mathtt{c}\})$ under the subset ordering.

If D is a partial order that is not discrete, it turns out that there are many "natural" ways to order the elements of the **powerdomain** $\mathcal{P}(D)$, each of which is useful for different purposes. See [Sch86] or [GS90] for details. ◁

▷ **Exercise 7.6** For each ordering on a compound domain defined above, show that the ordering is indeed a partial order. I.e., show that it is reflexive, transitive, and anti-symmetric. ◁

7.2.2 Complete Partial Orders (CPOs)

A partial order D is **complete** if every chain in D has a least upper bound in D. The term "complete partial order" is usually abbreviated **CPO**. Intuitively, completeness means that any sequence of elements visited on a upward path through a Hasse diagram must converge to a limit. Completeness is important because it will guarantee that the iterative fixed point technique will converge on a limiting value.

Here are some examples of CPOs:

- Any partial order with a finite number of elements is a CPO because every chain is finite and contains its lub. PO, G, and H from the previous section are all examples finite CPOs.

- Any flat partial order is a CPO because every chain has at most two elements, one of which must be the lub. So $Nat_{\perp}$ is an example of a CPO with an infinite number of elements.

- $\mathcal{P}(Nat)$ is a CPO in which the elements (each of which is a subset of the naturals) are ordered by subset inclusion. It is complete because the lub of every chain C is the (possibly infinite) union of the elements of C. Unlike the previous examples of CPOs, this is one in which a chain may be infinite and not contain its own lub. Consider the chain C with elements c_i, where c_i is defined to be $\{n \mid n \leq i, n : Nat\}$ Then:

$$\bigsqcup_{\mathcal{P}(Nat)} C = \bigcup \{\{0\}, \{0,1\}, \{0,1,2\}, \ldots\} = Nat$$

  The lub of C is the entire set of natural numbers, but no individual c_i is equal to this set.

- The unit interval under the usual ordering of real numbers is a CPO. It is complete because the construction of the reals guarantees that it contains the least upper bound of every subset of the interval. The unit interval is another CPO in which chains do not necessarily contain their own lubs. For example, the set of all rational numbers less than $\sqrt{.5}$ does not contain $\sqrt{.5}$.

- The partial functions from Nat to Nat (denoted $Nat \rightharpoonup Nat$) form a CPO. Recall that a partial function can be represented by a graph of input/ouput pairs. So the function that is undefined everywhere is represented by $\{\}$, the function that returns 23 given 17 and is elsewhere undefined is represented by $\{\langle 17,\ 23 \rangle\}$, and so on. The ordering of elements in this CPO is just subset inclusion on the graphs of the functions. It is complete for the same reason that $\mathcal{P}(Nat)$ is complete.

It is worthwhile to consider examples of partial orders that are *not* CPOs:

- The total order ω depicted in Figure 7.7 is not a CPO because the chain consisting of the entire set has no least upper bound (i.e., there is no largest natural number). This partial order can be turned into a CPO $\omega^\top$ by extending it with a top element $\top_{\omega^\top}$ that by definition is larger than every natural number (see Figure 7.14.)

- The partial order of rational numbers (under the usual ordering) between 0 and 1, inclusive, is not complete because it does not contain irrational numbers like $\sqrt{.5}$. It can be made complete by extending it with the irrationals between 0 and 1; this results in the unit interval $[0, 1]$.

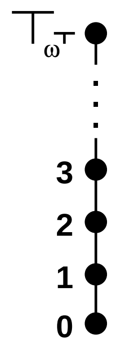


Figure 7.14: The partial order ω is the partial order ω of natural numbers extended with a largest element $\top_{\omega^\top}$.

- The partial order of sequences *Bit** under the prefix ordering is not a CPO. By definition, $D*$ is the set of *finite* sequences whose elements are taken from D. But the chain $\{[\,],\ [\mathtt{1}],\ [\mathtt{1},\mathtt{1}],\ [\mathtt{1},\mathtt{1},\mathtt{1}],\ \ldots\}$ has as its lub an infinite sequence of `1`s, which is not an element of *Bit**. To make this partial order complete, it is necessary to extend it with the set of infinite sequences over `0` and `1`, written Bit^{∞}. So the set of strings $Bit* \cup Bit^{\infty}$ under the prefix ordering is a CPO.

The CPO $D* \cup D^{\infty}$ introduced in the last example is common enough to warrant a simpler notation. We will use $\overline{D*}$ to stand for $D* \cup D^{\infty}$ under the prefix ordering.

As with partial orders, we are interested in combination properties of CPOs. As indicated by the following facts, we can use $\perp$, $\times$, $+$, $\rightarrow$, and $*$ to build new CPOs out of existing CPOs. Suppose that D and E are CPOs, then:

- $D_{\perp}$ is a CPO.
- $D \times E$ is a CPO under the partial order for products.
- $D + E$ is a CPO under the partial order for sums.
- $D \rightarrow E$ is a CPO under the partial order for functions.
- $D*$ is a CPO under the sum-of-products odering for sequences.
- $\overline{D*}$ is a CPO.

$\triangleright$ **Exercise 7.7** For each of the compound CPOs described above, show that the compound partial order is indeed complete. That is, show that the completeness property of D and E implies that each chain of the compound domain has a lub in the compound domain. $\triangleleft$

7.2.3 Pointedness

A partial order is **pointed** if it has a bottom element. Pointedness is important because the bottom element of a CPO is the natural place for the iterative fixed point technique to start. Of the CPO examples discussed in the previous section, all are pointed except for PO. Here are the pointed CPOs listed with their bottom elements:

- G, bottom $=$ `a`;
- H, bottom $=$ `c`;
- $Nat_\perp$, bottom $= \perp_{Nat}$;
- $\mathcal{P}(Nat)$, bottom $= \{\}$;
- $[0,1]$, bottom $= 0$;
- $Nat \rightharpoonup Nat$, bottom $=$ the function whose graph is $\{\}$;
- $\omega^\top$, bottom $= 0$;
- Bit^*, bottom $= [\,]$.

In the iterative fixed point technique, the bottom element of a pointed CPO is treated as the element with least information — the "worst" approximation to the desired value. For example, $\perp_{Nat_\perp}$ is the unknown natural number, $[\,]$ is a (very bad) approximation to any sequence of `0`s and `1`s, and $\{\}$ is a (very bad) approximation to the graph of any partial function from Nat to Nat.

In computational terms, the bottom element of a CPO can informally be viewed as representing a process that **diverges** (i.e., gets caught in an infinite loop). For example, a procedure that returns a boolean for even numbers but diverges on odd numbers can be modelled as an element of the domain $Int \to Bool_\perp$ that maps every odd number to $\perp_{Bool_\perp}$.

Pointed CPOs are commonly used to encode partial functions as total functions. Any partial function f in $D \rightharpoonup E$ can be represented as a total function f' in $D \rightharpoonup E_\perp$ by having f' map every targetless element $d : D$ of f to $\perp_{E_\perp}$. For example, the partial function in $PO \to PO$ with graph

$$\{\langle \texttt{a},\ \texttt{d}\rangle,\ \langle \texttt{c},\ \texttt{b}\rangle,\ \langle \texttt{f},\ \texttt{f}\rangle\}$$

can be represented as the total function in $PO \to PO_\perp$ with graph

$$\{\langle \texttt{a},\ \texttt{d}\rangle,\ \langle \texttt{b},\ \perp_{PO_\perp}\rangle,\ \langle \texttt{c},\ \texttt{b}\rangle,\ \langle \texttt{d},\ \perp_{PO_\perp}\rangle,\ \langle \texttt{e},\ \perp_{PO_\perp}\rangle,\ \langle \texttt{f},\ \texttt{f}\rangle\}$$

Because of the isomorphism between $D \rightharpoonup E$ and $D \to E_\perp$, we will casually perform implicit conversions between the two representations.

The following are handy facts about the pointedness of partial orders constructed out of parts. Suppose that D and E are arbitrary partial orders (not necessarily pointed). Then:

- $D_\perp$ is pointed.
- $D \times E$ is pointed if D and E are pointed.
- $D + E$ is never pointed.
- $D \to E$ is pointed if E is pointed.
- D^* under the sum-of-products ordering is never pointed.
- $\overline{D^*}$ under the prefix ordering is pointed.

Unpointed compound domains like $D + E$ and D^* can always be made pointed by lifting them with a new bottom element.

▷ **Exercise 7.8** Prove each of the facts about pointedness claimed above. ◁

7.2.4 Monotonicity and Continuity

Suppose that $f : D \to E$, where D and E are CPOs (not necessarily pointed). Then

- f is **monotonic** if $d_1 \sqsubseteq_D d_2$ implies $(f\ d_1) \sqsubseteq_E (f\ d_2)$.
- f is **continuous** if, for all chains C in D, $(f\ \bigsqcup_D C) = \bigsqcup_E \{(f\ c) \mid c \in C\}$.

A monotonic function preserves order between CPOs, while a continuous function preserves limits. In the iterative fixed point technique, monotonicity is important because when $f : D \to D$ is monotonic, the set of values

$$\{(f\ d),\ (f\ (f\ d)),\ (f\ (f\ (f\ d))),\ \ldots\}$$

is guaranteed to form a chain. Continuity guarantees that this chain approaches a limit.

As an example of these properties, consider the CPO of functions $G \to H$ depicted in Figure 7.11. Any function represented by the pair $\langle x, y\rangle$ is monotonic if and only if $x \sqsubseteq y$. Although there are $3^2 = 9$ total functions from G to H, only five of these are monotonic:

$$\{\langle \mathtt{c},\ \mathtt{c}\rangle,\ \langle \mathtt{c},\ \mathtt{d}\rangle,\ \langle \mathtt{d},\ \mathtt{d}\rangle,\ \langle \mathtt{c},\ \mathtt{e}\rangle,\ \langle \mathtt{e},\ \mathtt{e}\rangle\}$$

The reason that there are fewer monotonic functions than total functions is that choosing the target t for a particular source element s constrains all the source elements stronger than s to map to a target stronger than t. For example, a monotonic function that maps `a` to `e` must necessarily map `b` to `e`. With larger domains, the reduction from total functions to monotonic functions can be more dramatic.

What functions from G to H are continuous? The only non-singleton chain in G is $\{\mathtt{a}, \mathtt{b}\}$. By the definition of continuity, this means that a function $f : D \to E$ is continuous if

$$(f\ \bigsqcup_D \{\mathtt{a}, \mathtt{b}\}) = \bigsqcup_E \{(f\ \mathtt{a}), (f\ \mathtt{b})\}.$$

In this case, this condition simplifies to $(f\ \mathtt{a}) \sqsubseteq_E (f\ \mathtt{b})$, which is equivalent to saying that f is monotonic. Thus, the continuous functions from G to H are exactly the five monotonic functions listed above.

The relationship between monotonic and continuous functions in this example is more than coincidence. Monotonicity and continuity are not orthogonal, as indicated by the following facts:

- On finite CPOs (and even infinite CPOs with only finite chains), monotonicity implies continuity.
- On *any* CPO, continuity implies monotonicity.

We leave the proof of these facts as exercises.

Although monotonicity and continuity coincide on finite-chain CPOs, monotonicity *does not* imply continuity in general. To see this, consider the following function from $\omega^\top$ to the two-point CPO $\mathit{Two\text{-}Pts} = \{\bot, \top\}$:

$$mon\text{-}not\text{-}con : \omega^\top \rightarrow Two\text{-}Pts = \lambda n \ . \ \mathbf{if}\ (n \ = \ \top_{\omega^\top})\ \mathbf{then}\ \top\ \mathbf{else}\ \bot\ \mathbf{fi}$$

(See Figure 7.15 for a depiction of this function.) This function is clearly monotonic, but it is not continuous because on the subset ω of $\omega^\top$,

$$(f \ \bigsqcup\omega) = (f \ \ \top_{\omega^\top}) = \top \neq \bot = \bigsqcup_{Two\text{-}Pts}\{\bot\} = \bigsqcup_{Two\text{-}Pts}\{(f \ \ n) \mid n \in \omega\}$$

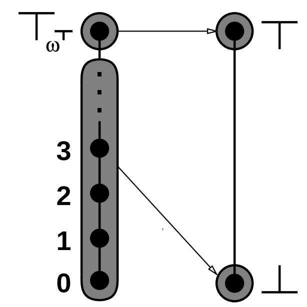


Figure 7.15: An example of a function that is monotonic but not continuous.

An important fact about continuous functions is that the set of continuous functions between CPOs D and E is itself a CPO. For example, Figure 7.16 depicts the CPO of the five continuous functions between G and H. If E is pointed, the function that maps all elements of D to $\bot_E$ is continuous and serves as the bottom element of the continuous function CPO.

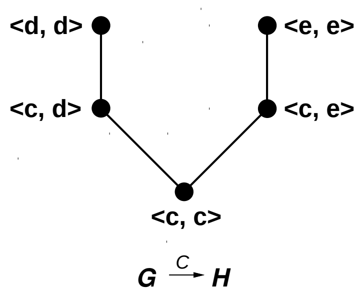


Figure 7.16: The CPO of continuous functions between G and H.

Since the CPO of total functions between D and E and the CPO of continuous functions between D and E are usually distinct, it will be helpful to have a notation that distinguishes them. We will use $D \stackrel{T}{\to} E$ to designate the CPO of total functions from D to E. and $D \stackrel{C}{\to} E$ to designate the CPO of continuous functions from D to E. It turns out that the CPO of continuous functions is almost always the "right thing" in semantics, so we adopt the convention that, throughout the rest of this text, any unannotated $\to$ should be interpreted as $\stackrel{C}{\to}$. We shall use $\stackrel{T}{\to}$ whenever we wish to discuss set-theoretic functions, and will explicitly use $\stackrel{C}{\to}$ only when we wish to emphasize the difference between $\stackrel{T}{\to}$ and $\stackrel{C}{\to}$.

$\triangleright$ **Exercise 7.9** Using the CPOs G and H from Figure 7.8, draw Hasse diagrams for the following CPOs:

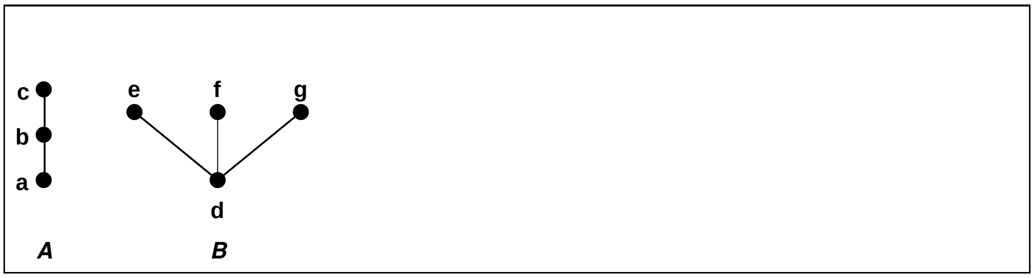
a. $G \stackrel{C}{\to} G$

b. $H \stackrel{C}{\to} H$

c. $H \stackrel{C}{\to} G$

$\triangleleft$

$\triangleright$ **Exercise 7.10** Consider the CPOs A and B pictured below:



For each of the following functional domains, give the number of the (1) total, (2) monotonic, and (3) continuous functions in the domain:

a. $A \stackrel{T}{\to} A$

b. $B \stackrel{T}{\to} B$

c. $A \stackrel{T}{\to} B$

d. $B \stackrel{T}{\to} A$

$\triangleleft$

$\triangleright$ **Exercise 7.11**

a. Show that a continuous function between CPOs is necessarily monotonic.

b. Show that a monotonic function must also be continuous if its source is a CPO all of whose chains are finite.

◁

▷ **Exercise 7.12** This problem considers functions f from $[0,1]$ to itself. We will say that f is *continuous in the CPO sense* if it is a member of $[0,1] \overset{C}{\to} [0,1]$, where $[0,1]$ is assumed to have the traditional ordering. We will say that f is *continuous in the classical sense* if for all x and ϵ there exists a δ such that

$$(f\ \ [x-\delta, x+\delta]) \subseteq [(f\ \ x)-\epsilon, (f\ \ x)+\epsilon].$$

(Here we are abusing the function call notation to designate the image of all of the elements of the interval.)

a. Does classical continuity imply CPO continuity? If so, give a proof; if not, provide a counter-example of a function that is continuous in the classical sense but not in the CPO sense.

b. Does CPO continuity imply classical continuity? If so, give a proof; if not, provide a counter-example of a function that is continuous in the CPO sense but not in the classical sense.

◁

7.2.5 The Least Fixed Point Theorem

Suppose D is a domain and $f : D \to D$. Then $d : D$ is a **fixed point** of f if $(f\ \ d) = d$. If $\langle D, \sqsubseteq \rangle$ is a partial order, then $d : D$ is the **least fixed point** of f if it is a fixed point of f and $d \sqsubseteq d'$ for every fixed point d' of f.

Everything is now in place to prove the following fixed point theorem:

> **Least Fixed Point Theorem:** If D is a pointed CPO, then a continuous function $f : D \to D$ has a least fixed point $(\mathbf{fix}_D\ f)$ defined by $\bigsqcup_D \{(f^n\ \ \bot_D) \mid n \geq 0\}$.

Proof:

First we show that the above definition of $(\mathbf{fix}_D\ f)$ is a fixed point of f:

- Since $\bot_D$ is the least element in D, $\bot_D \sqsubseteq (f\ \ \bot_D)$.
- Since f is monotonic (recall that continuity implies monotonicity), $\bot_D \sqsubseteq (f\ \ \bot_D)$ implies $(f\ \ \bot_D) \sqsubseteq (f\ \ (f\ \ \bot_D))$. By induction, $(f^n\ \ \bot_D) \sqsubseteq (f^{n+1}\ \ \bot_D)$ for every $n \geq 0$, so $\{(f^n\ \ \bot_D) \mid n \geq 0\}$ is a chain in D.

- Now,

$$
\begin{array}{lll}
(f\ (\mathbf{fix}_D\ f)) & & \\
\quad = & (f\ \bigsqcup_D\{(f^n\ \bot_D) \mid n \geq 0\}) & \text{By definition of } \mathbf{fix}_D. \\
\quad = & \bigsqcup_D\{(f\ (f^n\ \bot_D)) \mid n \geq 0\} & \text{By continuity of } f. \\
\quad = & \bigsqcup_D\{(f^n\ \bot_D) \mid n \geq 1\} & \\
\quad = & \bigsqcup_D\{(f^n\ \bot_D) \mid n \geq 0\} & (f^0\ \bot_D) = \bot_D \text{ can't change lub.} \\
\quad = & (\mathbf{fix}_D\ f) & \text{By definition of } \mathbf{fix}_D.
\end{array}
$$

Thus, $(f\ (\mathbf{fix}_D\ f)) = (\mathbf{fix}_D\ f)$, showing that $(\mathbf{fix}_D\ f)$ is indeed a fixed point of f.

To see that this is the *least* fixed point of f, suppose d' is some other fixed point. Then clearly $\bot_D \sqsubseteq d'$, and by the monotonicity of f, $(f^n\ \bot_D) \sqsubseteq (f^n\ d') = d'$. So d' is an upper bound of the set $S = \{(f^n\ d') \mid n \geq 0\}$. But then, by the definition of least upper bound, $(\mathbf{fix}_D\ f) = \bigsqcup_D S \sqsubseteq d'$.

◇

We can treat $\mathbf{fix}_D$ as a function of type $(D \to D) \to D$. It turns out that $\mathbf{fix}_D$ is itself a continuous function, and satisfies some other properties that make it "the right thing" for many semantic purposes (see Gunter and Scott [GS90]).

The Least Fixed Point Theorem describes an important class of situations in which fixed points exist, and we shall use it to specify the meaning of various recursive definitions. However, we emphasize that there are many generating functions that have least fixed points but do not satisfy the conditions of the Least Fixed Point Theorem. In these cases, some other means must be used to find the least fixed point.

7.2.6 Fixed Point Examples

Here we present several brief examples of the Least Fixed Point Theorem in action. We have discussed many of these examples informally already, but will show how the fixed point machinery formalizes the intuition underlying the iterative fixed point technique.

Sequence Examples

As a first application of the Least Fixed Point Theorem, we will revisit some sequence problems posed earlier. In order to model sequences of natural numbers, we will use the domain

$$s \in Natseq = \overline{Nat_\bot{}^*}$$

We use the flat domain $Nat_\perp$ instead of Nat to model the elements of a sequence so that there is a distinguished bottom element to which *head* can map the empty sequence. We will assume that $(tail\ [\,]) = [\,]$, though we could alternately introduce a new bottom element for sequences if we wanted to distinguish $(tail\ [\,])$ from $[\,]$. We use $\overline{Nat_\perp{}^*}$ rather than $Nat_\perp{}^*$ because the former is a pointed CPO that contains all the limiting values that are missing from the latter.

The equation $s = (cons\ 3\ (cons\ (head\ s)\ [\,]))$ has as its associated generation function the following:

$$g_{seq1} : Natseq \to Natseq = \lambda s\,.\,(cons\ 3\ (cons\ (head\ s)\ [\,]))$$

$Natseq$ is a pointed CPO with bottom element $[\,]$, and it is not hard to show that g_{seq1} is continuous. Thus, the Least Fixed Point Theorem applies, and the least fixed point can be found by iterating g starting with $[\,]$:

$$\begin{array}{l} (\mathbf{fix}_{Natseq}\ g_{seq1}) \\ \quad = \bigsqcup_{Natseq}\{(g_{seq1}{}^0\ [\,]),\ (g_{seq1}{}^1\ [\,]),\ (g_{seq1}{}^2\ [\,]),\ (g_{seq1}{}^3\ [\,]),\ \ldots\} \\ \quad = \bigsqcup_{Natseq}\{[\,],\ [3, \perp_{Nat_\perp}],\ [3,3]\} \\ \quad = [3,3] \end{array}$$

In this case, the unique fixed point [3,3] of g_{seq1} is reached after two iterations of g_{seq1}.

What happens when we apply this technique to an equation like

$$s = (cons\ (head\ s)\ (cons\ (1\ +\ (head\ s))\ [\,]))$$

which has an infinite number of fixed points? The corresponding generating function is

$$g_{seq2} : Natseq \to Natseq = \lambda s\,.\,(cons\ (head\ s)\ (cons\ (1\ +\ (head\ s))\ [\,])).$$

This function is continuous as long as $+$ returns $\perp_{Nat_\perp}$ when one of its arguments is $\perp_{Nat_\perp}$. The Least Fixed Point Theorem applies, and iterating g_{seq2} on $[\,]$ gives:

$$\begin{array}{l} (\mathbf{fix}_{Natseq}\ g_{seq2}) \\ \quad = \bigsqcup_{Natseq}\{(g_{seq2}{}^0\ [\,]),\ (g_{seq2}{}^1\ [\,]),\ (g_{seq2}{}^2\ [\,]),\ (g_{seq2}{}^3\ [\,]),\ \ldots\} \\ \quad = \bigsqcup_{Natseq}\{[\,],\ [\perp_{Nat_\perp}, \perp_{Nat_\perp}],\ \} \\ \quad = [\perp_{Nat_\perp}, \perp_{Nat_\perp}] \end{array}$$

After one iteration, the iterative fixed point technique finds the fixed point $[\perp_{Nat_\perp}, \perp_{Nat_\perp}]$, which is indeed less than all the other fixed points $[n, (n\ +\ 1)]$. Intuitively, this result indicates that the solution is a sequence of two numbers, but that the value of those numbers cannot be determined without making an arbitrary decision. Note the crucial roles that the bottom elements [] and $\perp_{Nat_\perp}$ play in this example. Each represents the value of a domain with the least information. Iterative application of the generating function may or may not refine these values by adding information.

A similar story holds for equations like

$$s = (cons\ (1\ +\ (head\ s))\ (cons\ (head\ s)\ [\,]))$$

that have no solutions in Nat^*. The reader can verify that this equation *does* have the unique solution $[\perp_{Nat_\perp}, \perp_{Nat_\perp}]$ in *Natseq* and that this solution can be found by an application of the Least Fixed Point Theorem.

As a final sequence example, we consider the equation $s = (cons\ 1\ s)$, whose associated generating function is

$$g_{seq3} : Natseq \to Natseq = \lambda s\,.\,(cons\ 1\ s).$$

This function is continuous, and the Least Fixed Point Theorem can be invoked to find a solution to the original equation:

$$\begin{aligned} &(\mathbf{fix}_{Natseq}\ g_{seq3}) \\ &= \bigsqcup_{Natseq}\{(g_{seq3}{}^0\ [\,]),\ (g_{seq3}{}^1\ [\,]),\ (g_{seq3}{}^2\ [\,]),\ (g_{seq3}{}^3\ [\,]),\ \ldots\} \\ &= \bigsqcup_{Natseq}\{[\,],\ [1],\ [1,\ 1],\ [1,\ 1,\ 1],\ \ldots\} \\ &= [1,\ 1,\ 1,\ \ldots] \end{aligned}$$

In this case, the unique fixed point of g_{seq3} is an infinite sequence of 1s. This fixed point is not reached in a finite number of iterations, but is the limit of the sequence of approximations $(g_{seq3}{}^n\ [\,])$. This example underscores why it is necessary to extend $Nat_\perp{}^*$ with $Nat_\perp{}^\infty$ to make *Natseq* a CPO. Without the infinite sequences in $Nat_\perp{}^\infty$, the iterative fixed point technique could not find a solution to some equations.

Function Examples

In the remainder of this book, we will typically apply the iterative fixed point technique to generating functions over function domains. Here we consider a few examples involving fixed points over the following domain of functions:

$$f \in Natfun = Nat \to Nat_\perp$$

Since $\to$ now designates *continuous* functions, *Natfun* is a domain of the continuous functions between *Nat* and $Nat_\perp$. *Natfun* is a CPO because the set of continuous functions between CPOs is itself a CPO under the usual ordering of functions. Furthermore, *Natfun* is pointed because $Nat_\perp$ is pointed. Recall that $Nat \to Nat_\perp$ is isomorphic to $Nat \rightharpoonup Nat$, so elements of *Natfun* can be represented by a function graph in which pairs whose target is $\perp_{Nat_\perp}$ are omitted.

Our first example is the definition of the doubling function studied earlier:

$$dbl = \lambda n \,.\, \mathbf{if}\,(n\ =\ 0)\ \mathbf{then}\ 0\ \mathbf{else}\ (2\ +\ (dbl\ (n\ -\ 1)))\ \mathbf{fi}$$

A solution to this definition is the fixed point of the generating function g_{dbl}:

$$\begin{array}{l} g_{dbl} : Natfun \to Natfun \\ = \lambda f\ .\ \lambda n\ .\ \ \mathbf{if}\,(n\ =\ 0)\ \ \mathbf{then}\ 0\ \ \mathbf{else}\ (2\ +\ (f\ \ (n\ -\ 1)))\ \ \mathbf{fi} \end{array}$$

Natfun is a pointed CPO in whose bottom element is the function whose graph is $\{\}$. In this CPO, $\bigsqcup$ on a chain of functions in $Nat \to Nat$ is equivalent to $\cup$ on a chain of graphs of functions in $Nat \rightharpoonup Nat$. It can be shown that g_{dbl} is continuous, so the Least Fixed Point Theorem applies:

$$\begin{array}{ll} \multicolumn{2}{l}{(\mathbf{fix}_{Natfun}\ g_{dbl})} \\ = & \bigsqcup_{Natfun}\{(g_{dbl}{}^0\ \{\}),\ (g_{dbl}{}^1\ \{\}),\ (g_{dbl}{}^2\ \{\}),\ (g_{dbl}{}^3\ \{\}),\ \ldots\} \\ = & \bigsqcup_{Natfun}\{\{\},\ \{\langle 0,\ 0\rangle\},\ \{\langle 0,\ 0\rangle,\ \langle 1,\ 2\rangle\},\ \{\langle 0,\ 0\rangle,\ \langle 1,\ 2\rangle,\ \langle 2,\ 4\rangle\},\ \ldots\} \\ = & \{\langle n,\ 2n\rangle \mid n : Nat\} \end{array}$$

Each $(g_{dbl}{}^n\ \{\})$ is a finite approximation of the doubling function that is only defined on the naturals $0 \le i \le n-1$. The least (and only) fixed point is the limit of these approximations: a doubling functin defined on all naturals.

As an example of a function with an infinite number of fixed points, consider the following recursive definition of a function in *Natfun*:

$$even0 : Natfun = \lambda n\ .\ \mathbf{if}\,(n\ =\ 0)\ \mathbf{then}\ 0\ \mathbf{else}\ (even0\ \ (n\ \ mod\ \ 2))\ \mathbf{fi}.$$

Here $(a\ \ mod\ \ b)$ returns the remainder of dividing a by b. For each constant c in $Nat_\perp$, the function whose graph is

$$\bigcup\nolimits_{n\ :\ Nat}\{\langle 2n,\ 0\rangle, \langle 2n+1,\ c\rangle\}$$

is a solution for *even0*. Each solution maps all even numbers to zero, but maps every odd number to the same constant c, where c is a parameter that distinguishes one solution from another. Each of these solutions is a fixed point of the generating function g_{even0}:

$$\begin{array}{l} g_{even0} : Natfun \to Natfun \\ = \lambda f\ .\ \lambda n\ .\ \ \mathbf{if}\,(n\ =\ 0)\ \ \mathbf{then}\ 0\ \ \mathbf{else}\ (f\ (n\ \ mod\ \ 2))\ \ \mathbf{fi} \end{array}$$

It turns out that this function is continuous, so the Least Fixed Point Theorem gives:

$$\begin{aligned}
&(\mathbf{fix}_{Natfun}\ g_{even0}) \\
&\quad = \bigsqcup_{Natfun}\{(g_{even0}{}^0\ \{\}),\ (g_{even0}{}^1\ \{\}),\ (g_{even0}{}^2\ \{\}),\ (g_{even0}{}^3\ \{\}),\ \ldots\} \\
&\quad = \bigsqcup_{Natseq}\{\{\},\ \{\langle 0,\ 0\rangle\},\ \{\langle 0,\ 0\rangle,\ \langle 2,\ 0\rangle\},\ \{\langle 0,\ 0\rangle,\ \langle 2,\ 0\rangle,\ \langle 4,\ 0\rangle\},\ \ldots\} \\
&\quad = \{\langle 2n,\ 0\rangle \mid n : Nat\}
\end{aligned}$$

The least fixed point is a function that maps every even number to zero, but is undefined (i.e., yields $\bot_{Nat_\bot}$) on the odd numbers. Indeed, this the least element of the class of fixed points described above; it uses the least arbitrary value for the constant c.

The solution for *even0* matches our intuitions about the operational behavior of programming language procedures for computing *even0*. For example, the definition for *even0* can be transliterated into the following Scheme procedure:

```
(define (even0 n)
  (if (= n 0)
      0
      (even0 (mod n 2))))
```

We expect this procedure to return zero in a finite number of steps for an even natural number, but to diverge for an odd natural number. The fact that the function *even0* maps odd numbers to $\bot_{Nat_\bot}$ can be interpreted as signifying that the procedure `even0` diverges on odd-numbered inputs.

▷ **Exercise 7.13** For each of the following equations:

- Characterize the set of all solutions to the equation in the specified solution domain;
- Use the iterative fixed point technique to determine the least solution to the equation.

Assume that s : *Natseq*, $p : \mathcal{P}(Nat)$, f : *Natfun*, and $h : Int \to Int_\bot$.

a. $s = (cons\ 2\ (cons\ (head\ (tail\ s))\ s))$

b. $s = (cons\ (1\ +\ (head\ (tail\ s)))\ (cons\ 3\ s))$

c. $s = (cons\ 5\ (mapinc\ s))$ where *mapinc* is a function in $Natseq \to Natseq$ that maps every sequence $[n_1,\ n_2,\ n_3,\ \ldots]$ into the sequence $[(1\ +\ n_1),\ (1\ +\ n_2),\ (1\ +\ n_3),\ \ldots]$

d. $p = \{1\} \cup \{x + 3 \mid x \in p\}$

e. $p = \{1\} \cup \{2x \mid x \in p\}$

f. $p = \{1\} \cup \{|2x - 4| \mid x \in p\}$

g. $f = \lambda n \,.\, (f\ \ n)$

h. $f = \lambda n \,.\, (f\ \ (1 + n))$

i. $f = \lambda n \,.\, (1 + (f\ \ n))$

j. $f = \lambda n \,.\ $ **if** $(n = 1)$
then 0
else if $(\mathit{even?}\ n)$
then $(1 + (f\ \ (n\ /\ 2)))$
else $(f\ \ (n + 2))$
fi
fi,

where *even?* is a predicate determining if a number is even.

k. $h = \lambda i \,.\ $ **if** $(i = 0)$ **then** 0 **else** $(h\ \ (i - 2))$ **fi**

◁

▷ **Exercise 7.14** Section 7.1.3 sketches an example involving the solution of an equation on line drawings involving the transformation T. Formalize this example by completing the following steps:

a. Represent line drawings as an appropriate pointed CPO *Lines*.

b. Express the transformation T as a continuous function g_T in $\mathit{Lines} \rightarrow \mathit{Lines}$.

c. Use the iterative fixed point technique to find the least fixed point of g_T.

◁

▷ **Exercise 7.15** A binary relation R on a set A is a subset of $A \times A$. The **reflexive transitive closure** of R is the smallest subset R' of $A \times A$ satisfying the following properties:

- If $a \in A$, then $\langle a, a\rangle \in R'$;
- If $\langle a, b\rangle$ is in R' and $\langle b, c\rangle$ is in R, then $\langle a, c\rangle$ is in R'.

a. Describe how the reflexive transitive closure of a binary relation can be expressed as an instance of the Least Fixed Point Theorem. What is the pointed CPO? What is the bottom element? What is the generating function?

b. Use the iterative fixed point technique to determine the reflexive transitive closure of the following relation on the set $\{\texttt{a},\ \texttt{b},\ \texttt{c},\ \texttt{d},\ \texttt{e}\}$:

$$\{\langle \texttt{a},\ \texttt{c}\rangle,\ \langle \texttt{c},\ \texttt{e}\rangle,\ \langle \texttt{d},\ \texttt{a}\rangle,\ \langle \texttt{d},\ \texttt{b}\rangle,\ \langle \texttt{e},\ \texttt{c}\rangle\}$$

◁

▷ **Exercise 7.16** Show that each of the generating functions g_{seq1}, g_{seq2}, g_{seq3}, g_{dbl}, g_{even0} is continuous.

◁

7.2.7 Continuity and Strictness

We have seen how compound CPOs can be assembled out of component CPOs using the domain operators $_\perp$, $\times$, $+$, *, and $\rightarrow$. We have also seen how the pointedness of compound CPOs is in some cases dependent on the pointedness of its components.

But a pointed CPO D is not the only prerequisite of the Least Fixed Point Theorem. The other prerequisite is that the generating function $f : D \rightarrow D$ must be continuous. In the examples of the previous section, we waved our hands about the continuity of the generating functions, but did not actually prove continuity in any of the cases. The proofs are not difficult, but they are tedious. Below, we argue that all generating functions that can be expressed in the metalanguage summarized in Section A.4 are guaranteed to be continuous, so we generally do not need to worry about the continuity of generating functions. We also introduce strictness, an important property for characterizing functions on pointed domains.

Recall that metalanguage expressions include:

- constants (both primitive values and primitive functions on such values);
- variables;
- assembly and disassembly operators for compound domains (e.g., $\langle \ldots \rangle$ and *Proj*i notation for products, *Inj*i and **matching** notation for sums, *cons*, *empty?*, *head*, and *tail* for sequences, λ abstraction and application for functions);
- syntactic sugar like **if** and the generalized pattern-matching version of **matching**.

It turns out that all of the assembly and disassembly operators for compound domains are continuous and that the composition of continuous functions is continuous (see [Sch86] for the details). This implies that any function expressed as a composition of assembly and disassembly operators is continuous. As long as primitive functions are continuous and the **if** and **matching** notations preserve continuity, all functions expressible in this metalanguage subset must be continuous. Below, we refine our interpretation of primitive functions and the sugar notations so that continuity is guaranteed.

Assume for now that all primitive domains are flat CPOs. What does it mean for a function between primitive domains to be continuous? Since all chains on a flat domain D can contain at most two elements ($\bot_D$ and a non-bottom element d), the continuity of a function $f : D \to E$ between flat domains D and E is equivalent to the following monotonicity condition:

$$(f\ \ \bot_D) \sqsubseteq_E (f\ \ d).$$

This condition is only satisfied in the following two cases:

- f maps $\bot_D$ to $\bot_E$, in which case d can map to any element of E;
- f maps *all* elements of D to the same non-bottom element of E.

In particular, f is *not* continuous if it maps $\bot_D$ and d to distinct non-bottom elements of E.

For example, a function *sqr* in $Nat_\bot \to Nat_\bot$ that maps $\bot_{Nat_\bot}$ to $\bot_{Nat_\bot}$ and every number to its square is continuous. So is the constant function *three* that maps every element of $Nat_\bot$ (includindg $\bot_{Nat_\bot}$) to 3. But a function f that maps every non-bottom number n to its square and maps $\bot_{Nat_\bot}$ to 3 is *not* continuous, because the $(f\ \ n)$ is not a refinement of the approximation $(f\ \ \bot_{Nat_\bot})\ = 3$.

From a computational perspective, the continuity restriction makes sense because it only permits the modelling of computable functions. Uncomputable functions cannot be expressed without resorting to non-continuous functions. The celebrated halting function, which determines whether or not a program halts on a given input, is an example of an uncomputable function. Intuitively, the halting function requires a mechanism for detecting whether a computation is caught in an infinite loop; such a mechanism must map $\bot$ to one non-bottom element and other inputs to different non-bottom elements.

If D and E are pointed domains, a function $f : D \to E$ is **strict** if $(f\ \ \bot_D)$ $= \bot_E$. Otherwise, f is **non-strict**. For example, the *sqr* function described above is strict, while the *three* function is non-strict. Although strictness and continuity are orthogonal properties in general, strictness does imply continuity for functions between flat domains (see Exercise 7.17).

Strictness is important because it captures the operational notion that a computation will diverge if it depends on an input that diverges. For example, strictness models the parameter-passing strategies of most modern languages, in which a procedure call will diverge if the evaluation of any of its arguments diverges. Non-strictness models the parameter-passing strategies of so-called lazy and eager languages. See Chapters 9 and 12 for a discussion of these parameter-passing mechanisms.

When pointed CPOs are manipulated in our metalanguage, we shall assume the strictness of various operations:

- All the primitive functions on flat domains are strict. When such a function has multiple arguments, we will assume it is strict in each of its arguments. Thus, $+_{Nat_\perp}$ returns $\perp_{Nat_\perp}$ if either argument is $\perp_{Nat_\perp}$, and $=_{Nat_\perp}$ returns $\perp_{Bool_\perp}$ if either argument is $\perp_{Nat_\perp}$.

- An **if** expression is strict in its predicate whenever it is an element of $Bool_\perp$ rather than $Bool$. Thus the expression

  $$\textbf{if}\ x =_{Nat_\perp} y\ \textbf{then}\ 3\ \textbf{else}\ 3\ \textbf{fi}$$

  is guaranteed to return $\perp_{Nat_\perp}$ (*not* 3) if either x or y is $\perp_{Nat_\perp}$. Together with the strictness of $=_{Natnum_\perp}$, the strictnes of **if** predicates thwarts attempts to express non-computable functions. For example, the expression

  $$\textbf{if}\ x = \perp_{Nat_\perp}\ \textbf{then}\ \mathit{true}\ \textbf{else}\ \mathit{false}\ \textbf{fi}$$

  will always return $\perp_{Bool_\perp}$.

- A **matching** expression is strict in its discriminant whenever it is an element of a pointed CPO. As with the strictness of **if** predicates, this restriction matches computational intuitions and prevents the expression of non-computable functions.

- If D is a pointed domain, we require the *head* operation on sequences to be strict on D^* under the prefix ordering. That is, (*head* [ ]) must equal $\perp_D$. If D is not pointed, or if D^* has the sum-of-products ordering, *head* is undefined for [ ]; i.e., it is only a partial function.

With the above provisions for strictness, it turns out that, all functions expressible in the metalanguage are continuous.

Since we often want to specify new strict functions, it is helpful to have a convenient notation for expressing strictness. If f is any function between pointed domains D and E, then $(\mathbf{strict}_{D,E}\ f)$ is a strict version of f. That is, $(\mathbf{strict}_{D,E}\ f)$ maps $\bot_D$ to $\bot_E$ and maps every non-bottom element d of D to $(f\ \ d)$. As usual, we will omit the subscripts on **strict** when they are clear from context. For example, a strict function in $Nat_\bot \to Nat_\bot$ that returns 3 for all non-bottom inputs can be defined as:

$$\mathit{strict\text{-}three} = (\mathbf{strict}\ \ (\lambda n\ .\ 3))$$

We adopt the abbreviation $\underline{\lambda}\ .\ \ \ldots$ for $(\mathbf{strict}\ \ (\lambda\ .\ \ \ldots))$, so $\underline{\lambda} n\ .\ \ 3$ is another way to write the above function.

▷ **Exercise 7.17**

a. Show that strictness and continuity are orthogonal by exhibiting functions in $D \to D$ that have the properties listed below. You may choose different Ds for different parts.

   i. Strict and continuous;
   ii. Non-strict and continuous;
   iii. Strict and non-continuous;
   iv. Non-strict and non-continuous.

b. Which combinations of properties from the previous part cannot be achieved if D is required to be a flat domain? Justify your answer.

◁

7.3 Reflexive Domains

This section is under development. In the future, it will include a sketch of the technique for solving recursive domain definitions.

Reflexive domains are those that are defined by recursive domain equations. Finding the fixed point of the definition function associated with a recursive domain equation involves complexities that we did not encounter above, and requires the introduction of techniques that are beyond the scope of this book. For details, see Chapter 11 of Schmidt [Sch86], Chapter 6 of Stoy [Sto85], and Section 7 of Gunter and Scott [GS90].

7.4 Summary

Here are the "big ideas" of this chapter:

- The meaning of a recursive definition over a domain D can be understood as the fixed point of a function $D \to D$.
- Complete partial orders (CPOs) model domain elements as approximations that are ordered by information. In a CPO, every sequence of information-consistent approximations has a well-defined limit.
- A CPO D is pointed if it has a least element (bottom, written $\bot_D$). The bottom element, which stands for "no information" is used as a starting point for the fixed point process. Bottom can be used to represent a partial function as a total function. It is often used to model computations that diverge (go into an infinite loop). A function between CPOs is strict if it preserves bottom.
- Functions between CPOs are monotonic if they preserve the information ordering and continuous if they preserve the limits. Continuity implies monotonicity, but not vice versa.
- If D is a pointed CPO, every continuous function $f : D \to D$ has a least fixed point $(\mathbf{fix}_d\ f)$ that is defined as the limit of iterating f starting at $\bot_D$.
- The domain constructors $_\bot$, $\times$, $+$, $\to$, and $\overline{}^*$ can be viewed as operators on CPOs. In particular, $D_1 \to D_2$ is interpreted as the CPO of *continuous* functions from D_1 to D_2. Only some of these constructors preserve pointedness. The new domain constructor $_\bot$ extends a domain with a new bottom element, guaranteeing that it is pointed.
- Functions that can be expressed in the metalanguage of Section A.4 are guaranteed to be continuous. Intuitively, such functions correspond to the computable functions.
- Recursive domain equations that are not solvable when domains are viewed as sets can become solvable when domains are viewed as CPOs. The key ideas (due to Scott) are to interpret equality as isomorphism and to focus only on continuous functions rather than all set-theoretic functions. There are restricted kinds of CPOs for which any domain equations over a rich set of operators are guaranteed to have a solution.

7.5 Reading

This chapter is based largely on Schmidt's presentation in Chapter 6 of [Sch86]. The excellent overview article by Gunter and Scott [GS90] presents alternate approaches involving more restricted domains and touches upon many technical details omitted above. See Mosses's article on denotational semantics [Mos90] to see how these more restricted domains are used in practice. Gunter's book [Gun92] discusses many domain issues in detail.

For an introduction to the techniques of solving recursive domain equations, see Chapter 11 of Schmidt's book [Sch86], Chapter 6 of Stoy's book [Sto85], or Section 7 of Gunter and Scott's article [GS90].