Chapter 10
DOMAIN THEORY AND FIXED-POINT SEMANTICS

Although we did not stress the point in Chapter 9, the notation of denotational semantics is built upon that of the lambda calculus. The purpose of denotational semantics is to provide mathematical descriptions of programming languages independent of their operational behavior. The extended lambda calculus serves as a mathematical formalism, a metalanguage, for denotational definitions. As with all mathematical formalisms, we need to know that the lambda calculus has a model to ensure that the definitions are not meaningless.

Furthermore, denotational definitions, as well as programming languages in general, rely heavily on recursion, a mechanism whose description we deferred in the discussion of the lambda calculus in Chapter 5. Normally a user of a programming language does not care about the logical foundations of declarations, but we maintain that serious questions can be raised concerning the validity of recursion. In this chapter we justify recursive definitions to guarantee that they actually define meaningful objects.

10.1 CONCEPTS AND EXAMPLES

Programmers use recursion to define functions and procedures as subprograms that call themselves and also to define recursive data structures. Most imperative programming languages require the use of pointers to declare recursive data types such as (linked) lists and trees. In contrast, many functional programming languages allow the direct declaration of recursive types. Rather than investigating recursive types in an actual programming language, we study recursive data declarations in a wider context. In this introductory section we consider the problems inherent in recursively defined functions and data and the related issue of nontermination.

Recursive Definitions of Functions

When we define a symbol in a denotational definition or in a program, we expect that the symbol can be replaced by its meaning wherever it occurs. In particular, we expect that the symbol is defined in terms of other (preferably simpler) concepts so that any expression involving the symbol can be expressed by substituting its definition. With this concept in mind, consider two simple recursive definitions:

f(n) = if n=0 then 1 else f(n–1)

g(n) = if n=0 then 1 else g(n+1).

The purpose of these definitions is to give meaning to the symbols "f" and "g". Both definitions can be expressed in the applied lambda calculus as

define f = λn . (if (zerop n) 1 (f (sub n 1)))

define g = λn . (if (zerop n) 1 (g (succ n))).

Either way, these definitions fail the condition that the defined symbol can be replaced by its meaning, since that meaning also contains the symbol. The definitions are circular. The best we can say is that recursive "definitions" are equations in the newly defined symbol. The meaning of the symbol will be a solution to the equation, if a solution exists. If the equation has more than one solution, we need some reason for choosing one of those solutions as the meaning of the new symbol.

An analogous situation can be seen with a mathematical equation that resembles the recursive definitions:

$x = x^2 - 4x + 6$.

This "definition" of x has two solutions, x=2 and x=3. Other similar definitions of x, such as x = x+5, have no solutions at all, while $x = x^2/x$ has infinitely many solutions. We need to describe conditions on a recursive definition of a function, really a recursion equation, to guarantee that at least one solution exists and a reason for choosing one particular solution as the meaning of the function.

For the examples considered earlier, we will describe in this chapter a methodology that enables us to show that the equation in f has only one solution (λn . 1), but the equation in g has many solutions, including

(λn . 1) and (λn . if n=0 then 1 else *undefined*).

One purpose of this chapter is to develop a "fixed-point" semantics that gives a consistent meaning to recursive definitions of functions.

Recursive Definitions of Sets (Types)

Recursively defined sets occur in both programming languages and specifications of languages. Consider the following examples:

1. The BNF specification of Wren uses direct recursion in specifying the syntactic category of identifiers,

   <identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>,

   and indirect recursion in many places, such as,

   <command> ::= **if** <boolean expr> **then** <command seq> **end if**

   <command seq> ::= <command> | <command> **;** <command seq>.

2. The domain of lists of natural numbers N may be provided in a functional programming language according to the definition:

   List = $\{nil\} \cup (N \times List)$ where *nil* represents the empty list.

   Scheme lists have essentially this form using "cons" as the constructor operation forming ordered pairs in N x List. Standard ML allows data type declarations following this pattern.

3. A model for the (pure) lambda calculus requires a domain of values that are manipulated by the rules of the system. These values incorporate variables as primitive objects and functions that may act on any values in the domain, including any of the functions. If V denotes the set of variables and $D \to D$ represents the set of functions from set D to D, the domain of values for the lambda calculus can be "defined" by $D = V \cup (D \to D)$.

The third example presents major problems if we analyze the cardinality of the sets involved. We give the critical results without going into the details of measuring the cardinality of sets. It suffices to mention that the sizes of sets are compared by putting their elements into a one-to-one correspondence. We denote the cardinality of a set A by $|A|$ with the following properties:

1. $|A| \leq |B|$ if there is a one-to-one function $A \to B$.
2. $|A| = |B|$ if there is a one-to-one and onto function $A \to B$ (which can be shown to be equivalent to $|A| \leq |B|$ and $|B| \leq |A|$).
3. $|A| < |B|$ if $|A| \leq |B|$ but not $|A| \geq |B|$.

Two results about cardinalities of sets establish the problem with the recursive "definition" of D:

1. In the first use of "diagonalization" as a proof method, Georg Cantor proved that for any set A, $|A| < |P(A)|$ where $P(A)$ is the power set of A—that is, the set of all subsets of A.

2. If $|A| > 1$, then $|P(A)| \leq |A \rightarrow A|$.

Since $D \rightarrow D$ is a subset of D by the definition, $|D \rightarrow D| \leq |D|$. Therefore,

$$|D \rightarrow D| \leq |D| < |P(D)| \leq |D \rightarrow D|,$$

which is clearly a contradiction.

One way to provide a solution to the recursion equation $D = V \cup (D \rightarrow D)$ is to restrict the membership in the set of functions $D \rightarrow D$ by putting a "structure" on the sets under consideration and by requiring that functions are well-behaved relative to the structure. Although the solution to this recursion equation is beyond the scope of this book (see the further readings at the end of this chapter), we study this structure carefully for the intuition that it provides about recursively defined functions and sets.

Modeling Nontermination

Any programming language that provides (indefinite) iteration or recursively defined functions unfortunately also allows a programmer to write nonterminating programs. Specifying the semantics of such a language requires a mechanism for representing nontermination. At this point we preview domain theory by considering how it handles nontermination. Domain theory is based on a relation of definedness. We say $x \subseteq y$ if x is less defined than or equal to y. This means that the information content of x is contained in the information content of y. Each domain (structured set) contains a least element $\perp$, called bottom, representing the absence of information. Bottom can be viewed as the result of a computation that fails to terminate normally. By adding a bottom element to every domain, values that produce no outcome under a function can be represented by taking $\perp$ as the result. This simple idea enables us to avoid partial functions in describing the semantics of a programming language, since values for which a function is undefined map to the bottom element in the codomain.

Dana Scott developed domain theory to provide a model for the lambda calculus and thereby provide a consistent foundation for denotational semantics. Without such a foundation, we have no reason to believe that denotational definitions really have mathematical meaning. At the same time, domain theory gives us a valid interpretation for recursively defined functions and types.

In this chapter we first describe the structure supplied to sets by domain theory, and then we investigate the semantics of recursively defined functions via fixed-point theory. Finally, we use fixed points to give meaning to recursively defined functions in the lambda calculus, implementing them by extending the lambda calculus evaluator described in Chapter 5.

Exercises

1. Write a recursive definition of the factorial function in the lambda calculus using *define*.

2. Give a recursive definition of binary trees whose leaf nodes contain natural number values.

3. Suppose A is a finite set with $|A| = n$. Show that $|P(A)| = 2^n$ and $|A \to A| = n^n$.

4. Let A be an arbitrary set. Show that it is impossible for $f : A \to P(A)$ to be a one-to-one and onto function. *Hint*: Consider the set $X = \{a \in A \mid a \notin f(a)\}$.

5. Prove that $|P(A)| \leq |A \to A|$ for any set A with $|A| \geq 2$. *Hint*: Consider the characteristic functions of the sets in $P(A)$.

10.2 DOMAIN THEORY

The structured sets that serve as semantic domains in denotational semantics are similar to the structured sets called lattices, but these domains have several distinctive properties. Domains possess a special element $\perp$, called **bottom**, that denotes an undefined element or simply the absence of information. A computation that fails to complete normally produces $\perp$ as its result. Later in this section we describe how the bottom element of a domain can be used to represent the nontermination of programs, since a program that never halts is certainly an undefined object. But first we need to define the structural properties of the sets that serve as domains.

Definition: A **partial order** on a set S is a relation $\subseteq$ with the following properties:

1. $\subseteq$ is **reflexive**: $x \subseteq x$ for all $x \in S$.
2. $\subseteq$ is **transitive**: ($x \subseteq y$ and $y \subseteq z$) implies $x \subseteq z$ for all $x,y,z \in S$.
3. $\subseteq$ is **antisymmetric**: ($x \subseteq y$ and $y \subseteq x$) implies $x = y$ for all $x,y \in S$. ∎

Definition: Let A be a subset of S.

1. A **lower bound** of A is an element $b \in S$ such that $b \subseteq x$ for all $x \in A$.
2. An **upper bound** of A is an element $u \in S$ such that $x \subseteq u$ for all $x \in A$.
3. A **least upper bound** of A, written *lub* A, is an upper bound of A with the property that for any upper bound m of A, *lub* $A \subseteq m$. ∎

Example 1: The subset relation $\subseteq$ on the power set $P(\{1,2,3\})$ is a partial order as shown by the **Hasse diagram** in Figure 10.1. The main idea of a Hasse

diagram is to represent links between distinct items where no other values intervene. The reflexive and transitive closure of this "minimal" relation forms the ordering being defined. We know that the subset relation is reflexive, transitive, and antisymmetric. Any subset of $P(\{1,2,3\})$ has lower, upper, and least upper bounds. For example, if A = { {1}, {1,3}, {3} }, both {1,2,3} and {1,3} are upper bounds of A, $\varnothing$ is a lower bound of A, and *lub* A = {1,3}. ∎

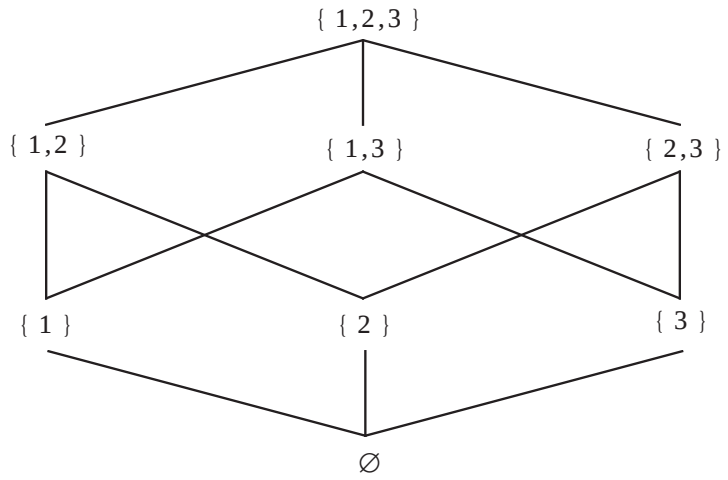


Figure 10.1: Partial order on $P(\{1,2,3\})$

In addition to possessing a partial ordering, the structured sets of domain theory require a "smallest" element and the existence of limits for sequences that admit a certain conformity.

Definition : An **ascending chain** in a partially ordered set S is a sequence of elements $\{x_1, x_2, x_3, x_4, \ldots\}$ with the property $x_1 \subseteq x_2 \subseteq x_3 \subseteq x_4 \subseteq \ldots$. ∎

Remember that the symbol $\subseteq$ stands for an arbitrary partial order, not necessarily the subset relation. Each item in an ascending chain must contain information that is consistent with its predecessor in the chain; it may be equal to its predecessor or it may provide additional information.

Definition : A **complete partial or der** (**cpo**) on a set S is a partial order $\subseteq$ with the following properties:

1. There is an element $\perp \in S$ for which $\perp \subseteq x$ for all $x \in S$.
2. Every ascending chain in S has a least upper bound in S. ∎

Sets with complete partial orders serve as the semantic domains in denotational semantics. On these domains, $\subseteq$ is thought of as the relation **approximates** or **is less defined than or equal to** . View $x \subseteq y$ as asserting

that y has at least as much information content as x does, and that the information in y is consistent with that in x. In other words, y is a consistent (possibly trivial) extension of x in terms of information.

The least upper bound of an ascending chain summarizes the information that has been accumulated in a consistent manner as the chain progresses. Since an ascending chain may have an infinite number of distinct values, the least upper bound acts as a limit value for the infinite sequence. On the other hand, a chain may have duplicate elements, since $\subseteq$ includes equality, and a chain may take a constant value from some point onward. Then the least upper bound is that constant value.

Example 1 (r evisited) : $P(\{1,2,3\})$ with $\subseteq$ is a complete partial order. If S is a set of subsets of {1,2,3}, *lub* $S = \cup\{X \mid X \in S\}$, and $\emptyset$ serves as bottom. Note that every ascending chain in $P(\{1,2,3\})$ is a finite subset of $P(\{1,2,3\})$—for example, the chain with $x_1=\{2\}$, $x_2=\{2,3\}$, $x_3=\{1,2,3\}$, and $x_i=\{1,2,3\}$ for all $i \geq 4$. ∎

Example 2 : Define $m \subseteq n$ on the set $S = \{1,2,3,5,6,15\}$ as the divides relation, $m \mid n$ (n is a multiple of m). The set S with the divides ordering is a complete partial order with 1 as the bottom element, and since each ascending chain is finite, its last element serves as the least upper bound. Figure 10.2 gives a Hasse diagram for this ordered set. Observe that the elements of the set lie on three levels. Therefore no ascending chain can have more than three distinct values. ∎

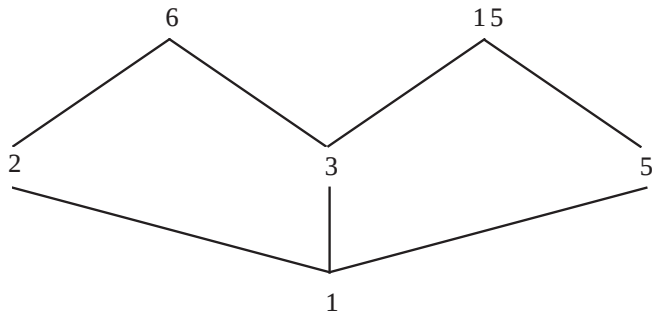


Figure 10.2: Partial order "divides" on {1,2,3,5,6,15}

These complete partially ordered sets have a lattice-like structure but need not be lattices. Lattices possess the property that any two elements have a least upper bound and a greatest lower bound. A complete lattice also satisfies the condition that *any* subset has a least upper bound and a greatest lower bound. Observe that {1,2,3,5,6,15} with "divides" is not a lattice since {6,15} has no least upper bound.

Any finite set with a partial order and a bottom element $\perp$ is a cpo since each ascending chain is a finite set and its last component will be the least upper bound. A partially ordered set with an infinite number of distinct elements that lie on an infinite number of "levels" may not be a cpo.

Elementary Domains

Common mathematical sets such as the natural numbers and the Boolean values are converted into complete partial orders by adding a bottom element $\perp$ and defining the **discrete partial order** $\subseteq$ as follows:

$$\text{for } x,y \in S,\ x \subseteq y \text{ iff } x = y \text{ or } x = \perp.$$

In denotational semantics, elementary domains correspond to "answers" or results produced by programs. A typical program produces a stream of these atomic values as its result.

Example 3 : The domain of Boolean values T has the structure shown in Figure 10.3. With a discrete partial order, bottom is called an **improper** value, and the original elements of the set are called **proper**. Each proper value, true or false, contains more information than $\perp$, but they are incomparable with each other. The value true has no more information content than false; it is just different information. ∎

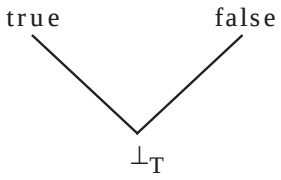


Figure 10.3: Boolean Domain

Example 4 : The domain of natural numbers N has the structure portrayed in Figure 10.4. ∎

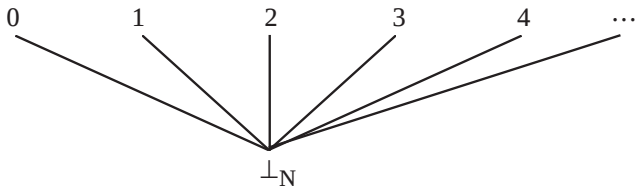


Figure 10.4: Domain of natural numbers

Do not confuse the "approximates" ordering $\subseteq$ with the numeric ordering $\leq$ on the natural numbers. Under $\subseteq$, no pair of natural numbers is comparable, since neither contains more or even the same information as the other. These primitive complete partially ordered sets are also called **elementary** or **flat domains** . More complex domains are formed by three domain constructors.

Product Domains

Definition : If A with ordering $\subseteq_A$ and B with ordering $\subseteq_B$ are complete partial orders, the **product domain** of A and B is AxB with the ordering $\subseteq_{AxB}$ where

$$A \times B = \{\langle a,b\rangle \mid a \in A \text{ and } b \in B\} \text{ and}$$

$$\langle a,b\rangle \subseteq_{AxB} \langle c,d\rangle \text{ iff } a \subseteq_A c \text{ and } b \subseteq_B d.$$ ∎

It is a simple matter to show that $\subseteq_{AxB}$ is a partial order on AxB, which we invite the reader to try as an exercise. Assuming that a product domain is a partial order, we need a bottom element and least upper bound for ascending chains to guarantee it is a cpo.

Theorem: $\subseteq_{AxB}$ is a complete partial order on AxB.

Proof: $\perp_{AxB} = \langle \perp_A, \perp_B \rangle$ acts as bottom for AxB, since $\perp_A \subseteq_A a$ and $\perp_B \subseteq_B b$ for each $a \in A$ and $b \in B$. If $\langle a_1,b_1\rangle \subseteq \langle a_2,b_2\rangle \subseteq \langle a_3,b_3\rangle \subseteq \ldots$ is an ascending chain in AxB, then $a_1 \subseteq_A a_2 \subseteq_A a_3 \subseteq_A \ldots$ is a chain in A with a least upper bound $lub\{a_i \mid i \geq 1\} \in A$, and $b_1 \subseteq_B b_2 \subseteq_B b_3 \subseteq_B \ldots$ is a chain in B with a least upper bound $lub\{b_i \mid i \geq 1\} \in B$. Therefore $lub\{\langle a_i,b_i\rangle \mid i \geq 1\} = \langle lub\{a_i \mid i \geq 1\}, lub\{b_i \mid i \geq 1\}\rangle \in AxB$ is the least upper bound for the original chain. ∎

A product domain can be constructed with any finite set of domains in the same manner. If $D_1, D_2, \ldots, D_n$ are domains (sets with complete partial orders), then $D_1 x D_2 x \ldots x D_n$ with the induced partial order is a domain. If the original domains are identical, then the product domain is written D^n.

Example 5 : Consider a classification of university students according to two domains.

1. Level = $\{\perp_L$, undergraduate, graduate, nondegree$\}$
2. Gender = $\{\perp_G$, female, male$\}$

The product domain Level x Gender allows 12 different values as depicted in the diagram in Figure 10.5, which shows the partial ordering between the elements using only the first letters to symbolize the values. Notice that six values are "partial", containing incomplete information for the classification. We can interpret these partial values by imagining two processes, one to determine the level of a student and the other to ascertain the gender. The six incomplete values fit into one of three patterns.

$\langle \perp_L, \perp_G \rangle$ Both processes fail to terminate normally.

$\langle \perp_L$,male$\rangle$ The Gender process terminates with a result but the Level process fails.

$\langle$graduate,$\perp_G\rangle$ The Level process completes but the Gender one does not terminate normally. ∎

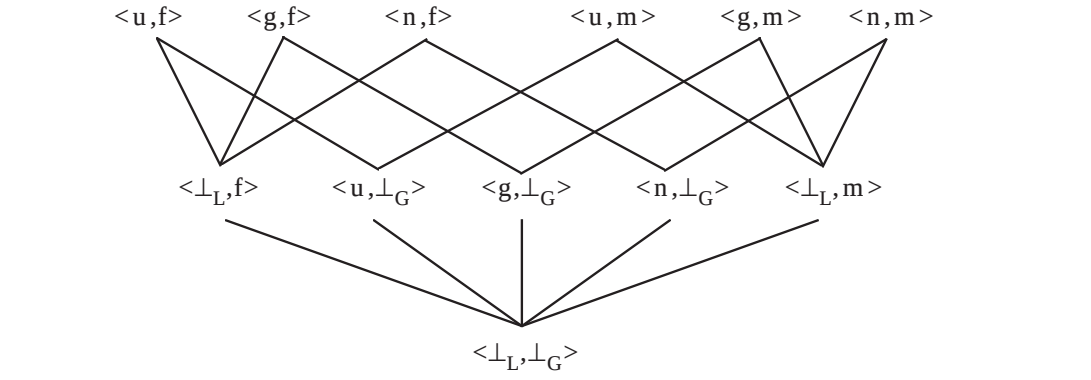


Figure 10.5: Level x Gender

To choose components from an element of a product domain, selector functions are defined on the structured domain.

Definition : Assume that for any product domain AxB, there are **projection** functions

first : AxB→A, defined by *first* <a,b> = a for any <a,b>∈AxB, and

second : AxB→B, defined by *second* <a,b> = b for any <a,b>∈AxB. ∎

Selector functions of this sort may be applied to arbitrary product domains, $D_1 x D_2 x \ldots x D_n$. As a shorthand notation we sometimes use *1st*, *2nd*, *3rd*, ..., *nth*, for the names of the selector functions.

Example 6 : We used a product domain Integer×Operation×Integer×Integer to represent the states of the calculator in Chapter 9. In the semantic equations of Figure 9.8, pattern matching simulates the projection functions—for example, the equation

meaning ⟦P⟧ = d where (a,op,d,m) = *perform* ⟦P⟧ (0,*nop*,0,0)

abbreviates the equation

meaning ⟦P⟧ = *third*(*perform* ⟦P⟧ (0,*nop*,0,0)).

Similarly,

evaluate $[\![\mathbf{M^R}]\!]$ (a,op,d,m) = (a,op,m,m)

is a more readable translation of

$$evaluate\ [\![\mathbf{M^R}]\!]\ \mathrm{st} = (first(\mathrm{st}), second(\mathrm{st}), fourth(\mathrm{st}), fourth(\mathrm{st})).$$ ∎

Generally, using pattern matching to select components from a structure leads to more understandable definitions, as witnessed by its use in Prolog and many functional programming languages, such as Standard ML.

Sum Domains (Disjoint Unions)

Definition : If A with ordering $\subseteq_A$ and B with ordering $\subseteq_B$ are complete partial orders, the **sum domain** of A and B is A+B with the ordering $\subseteq_{A+B}$ defined by

$$A+B = \{<a,1> \mid a \in A\} \cup \{<b,2> \mid b \in B\} \cup \{\bot_{A+B}\},$$

$$<a,1> \subseteq_{A+B} <c,1> \text{ if } a \subseteq_A c,$$

$$<b,2> \subseteq_{A+B} <d,2> \text{ if } b \subseteq_B d,$$

$$\bot_{A+B} \subseteq_{A+B} <a,1> \text{ for each } a \in A,$$

$$\bot_{A+B} \subseteq_{A+B} <b,2> \text{ for each } b \in B, \text{ and}$$

$$\bot_{A+B} \subseteq_{A+B} \bot_{A+B}.$$ ∎

The choice of "1" and "2" as tags in a disjoint union is purely arbitrary. Any two distinguishable values can serve the purpose. In Chapter 9 we used the symbols *int* and *bool* as tags for the sum domain of storable values when specifying the semantics of Wren,

$$SV = int(\mathrm{Integer}) + bool(\mathrm{Boolean}),$$

which can be thought of as an abbreviation of $\{<i,int> \mid i \in \mathrm{Integer}\} \cup \{<b,bool> \mid b \in \mathrm{Boolean}\} \cup \{\bot\}$.

Again it is not difficult to show that $\subseteq_{A+B}$ is a partial order on A+B, and the proof is left as an exercise.

Theorem: $\subseteq_{A+B}$ is a complete partial order on A+B.

Proof: $\bot_{A+B} \subseteq x$ for any $x \in A+B$ by definition. An ascending chain $x_1 \subseteq x_2 \subseteq x_3 \subseteq \ldots$ in A+B may repeat $\bot_{A+B}$ forever or eventually climb into either $A \times \{1\}$ or $B \times \{2\}$. In the first case, the least upper bound will be $\bot_{A+B}$, and in the other two cases the least upper bound will exist in A or B. ∎

Example 7 : The sum domain T+N (Boolean values and natural numbers) may be viewed as the structure portrayed in Figure 10.6, where the tags have been omitted to simplify the diagram. ∎

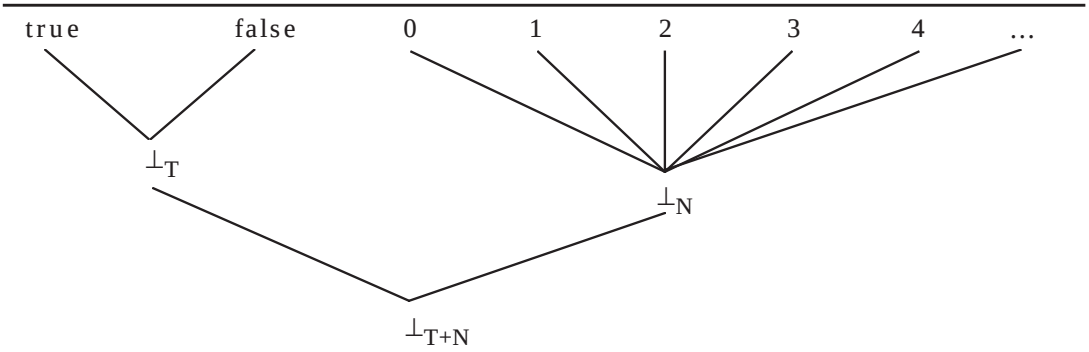


Figure 10.6: The sum domain T+N

A sum domain can be constructed with any finite set of domains in the same manner as with two domains. If D_1, D_2, ..., D_n are domains (sets with complete partial orders), then $D_1 + D_2 + \ldots + D_n = \{<d,i> \mid d \in D_i, 1 \le i \le n\} \cup \{\perp\}$ with the induced partial order is a domain.

Functions on sum domains include a constructor, a selector, and a testing function.

Definition : Let S = A+B, where A and B are two domains.

1. Injection (creation):

   inS : A→S is defined for a∈A as *inS* a = <a,1>∈S

   inS : B→S is defined for b∈B as *inS* b = <b,2>∈S

2. Projection (selection):

   outA : S→A is defined for s∈S as *outA* s = a∈A if s=<a,1>, and
   outA s = $\perp_A \in$ A if s=<b,2> or s=$\perp_S$.

   outB : S→B is defined for s∈S as *outB* s = b∈B if s=<b,2>, and
   outB s = $\perp_B \in$ B if s=<a,1> or s=$\perp_S$.

3. Inspection (testing): Recall that T = {true, false, $\perp_T$}.

   isA : S→T is defined for s∈S as
   (*isA* s) if and only if there exists a∈A with s=<a,1>.

   isB : S→T is defined for s∈S as
   (*isB* s) if and only if there exists b∈B with s=<b,2>.

   In both cases, $\perp_S$ is mapped to $\perp_T$. ∎

Example 8 : In the semantic domain of storable values for Wren shown in Figure 9.10, the identifiers *int* and *bool* act as the tags to specify the separate sets of integers and Boolean values. The notation

SV = *int*(Integer) + *bool*(Boolean)
= {*int*(n) | n∈Integer} ∪ {*bool*(b) | b∈Boolean} ∪ {$\perp_{SV}$}

represents the sum domain

SV = (Integer x {*int*}) ∪ (Boolean x {*bool*}) ∪ {$\perp_{SV}$}.

Then an injection function is defined by

inSV : Integer → SV where *inSV* n = *int*(n). ∎

Actually, the tags themselves can be thought of as constituting the injection function (or as constructors) with the syntax *int* : Integer → SV and *bool* : Boolean → SV, so that we can dispense with the special injection function *inSV*.

A projection function takes the form

outInteger : SV → Integer where *outInteger int*(n) = n
outInteger bool(b) = ⊥.

Inspection is handled by pattern matching, as in the semantic equation

execute ⟦**if** E **then** C⟧ sto = if p then *execute* ⟦C⟧ sto else sto
where *bool*(p) = *evaluate* ⟦E⟧ sto,

which stands for

execute ⟦**if** E **then** C⟧ sto =
if *isBoolean*(val)
then if *outBoolean*(val) then *execute* ⟦C⟧ sto else sto
else ⊥
where val = *evaluate* ⟦E⟧ sto.

Example 9 : In the sum domain Level + Gender shown in Figure 10.7, tags *lv* for level and *gd* for gender are attached to the elements from the two component domains. If a computation attempts to identify either the level or the gender of a particular student (but not both), it may utterly fail giving ⊥, it may be able to identify the level or the gender of the student, or as a middle ground it may know that the computation is working on the level value but may not be able to complete its work, thus producing the result $\perp_L$. ∎

An infinite sum domain may be defined in a similar way. If D_1, D_2, D_3, ... are domains, then $D_1 + D_2 + D_3 + \ldots$ contains elements of the form <d,i> where d∈D_i for i≥1 plus a new bottom element.

This infinite sum domain construction allows the definition of the domain of all finite sequences (lists) formed using elements from a domain D and denoted by D*.

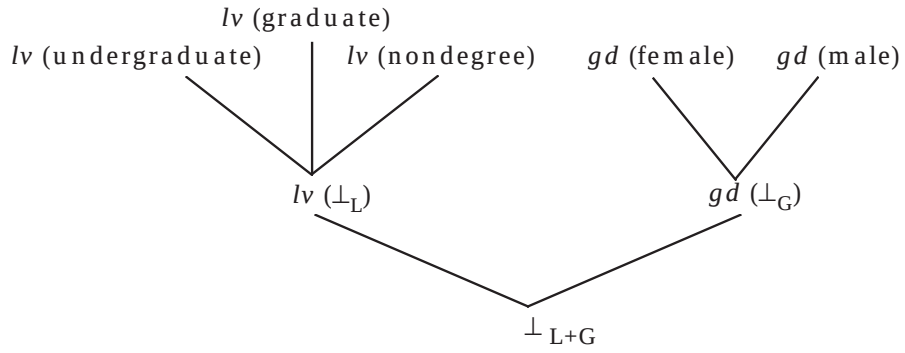


Figure 10.7: Level + Gender

$D^* = \{nil\}+D+D^2+D^3+D^4+\ldots$ where *nil* represents the empty sequence.

An element of D* is either the empty list *nil*, a finite ordered tuple from D^k for some $k\geq 1$, or $\perp$.

Special selector and constructor functions are defined on D*.

Definition : Let $L\in D^*$ and $e\in D$. Then $L=<d,k>$ for $d\in D^k$ for some $k\geq 0$ where $D^0 = \{nil\}$.

1. *head* : $D^*\rightarrow D$ where
   $head(L) = first(outD^k(L))$ if $k>0$ and $head(<nil,0>) = \perp$.

2. *tail* : $D^*\rightarrow D^*$ where
   $tail(L) = inD^*(<2nd(outD^k(L)),3rd(outD^k(L)),\ldots,kth(outD^k(L))>)$ if $k>0$ and $tail(<nil,0>) = \perp$.

3. *null* : $D^*\rightarrow T$ where
   $null(<nil,0>)$ = true and $null(L)$ = false if $L = <d,k>$ with $k>0$.
   Therefore $null(L) = isD^0(L)$.

4. *prefix* : $DxD^*\rightarrow D^*$ where
   $prefix(e,L) = inD^*(<e,1st(outD^k(L)),2nd(outD^k(L)),\ldots,kth(outD^k(L))>)$ and
   $prefix(e,<nil,0>)) = <<e>,1>$

5. *affix* : $D^*xD\rightarrow D^*$ where
   $affix(L,e) = inD^*(<1st(outD^k(L)),2nd(outD^k(L)),\ldots,kth(outD^k(L)),e>)$ and
   $affix(<nil,0>,e) = <<e>,1>$. ∎

Each of these five functions on lists maps bottom to bottom. The binary functions *prefix* and *affix* produce $\perp$ if either argument is bottom.

Function Domains

Definition : A function from a set A to a set B is **total** if $f(x) \in B$ is defined for every $x \in A$. If A with ordering $\subseteq_A$ and B with ordering $\subseteq_B$ are complete partial orders, define **Fun(A,B)** to be the set of all total functions from A to B. (This set of functions will be restricted later.) Define $\subseteq$ on Fun(A,B) as follows:

For $f,g \in$ Fun(A,B), $f \subseteq g$ if $f(x) \subseteq_B g(x)$ for all $x \in A$. ∎

Lemma: $\subseteq$ is a partial order on Fun(A,B).

Proof:

1. Reflexive: Since $\subseteq_B$ is reflexive, $f(x) \subseteq_B f(x)$ for all $x \in A$, so $f \subseteq f$ for any $f \in$ Fun(A,B).
2. Transitive: Suppose $f \subseteq g$ and $g \subseteq h$. Then $f(x) \subseteq_B g(x)$ and $g(x) \subseteq_B h(x)$ for all $x \in A$. Since $\subseteq_B$ is transitive, $f(x) \subseteq_B h(x)$ for all $x \in A$, and so $f \subseteq h$.
3. Antisymmetric: Suppose $f \subseteq g$ and $g \subseteq f$. Then $f(x) \subseteq_B g(x)$ and $g(x) \subseteq_B f(x)$ for all $x \in A$. Since $\subseteq_B$ is antisymmetric, $f(x) = g(x)$ for all $x \in A$, and so $f = g$. ∎

Theorem: $\subseteq$ is a complete partial order on Fun(A,B).

Proof: Define bottom for Fun(A,B) as the function $\perp(x) = \perp_B$ for all $x \in A$. Since $\perp(x) = \perp_B \subseteq_B f(x)$ for all $x \in A$ and $f \in$ Fun(A,B), $\perp \subseteq f$ for any $f \in$ Fun(A,B). Let $f_1 \subseteq f_2 \subseteq f_3 \subseteq \ldots$ be an ascending chain in Fun(A,B). Then for any $x \in A$, $f_1(x) \subseteq_B f_2(x) \subseteq_B f_3(x) \subseteq_B \ldots$ is a chain in B, which has a least upper bound, $y_x \in B$. Note that y_x is $lub\{f_i(x) \mid i \geq 1\}$. Define the function $F(x) = y_x$ for each $x \in A$. F serves as a least upper bound for the original chain. Set $lub\{f_i \mid i \geq 1\} = F$. ∎

The set Fun(A,B) of all total functions from A to B contains many functions with abnormal behavior that precludes calculating or even approximating them on a computer. For example, consider a function $H : (N \to N) \to (N \to N)$ defined by

for $g \in N \to N$, $H\ g = \lambda n$. if $g(n) = \perp$ then 0 else 1.

Certainly $H \in$ Fun$(N \to N, N \to N)$, but if we make this function acceptable in the domain theory that provides a foundation for denotational definitions, we have accepted a function that solves the halting problem—that is, whether an arbitrary function halts normally on given data. To exclude this and other abnormal functions, we place two restrictions on functions to ensure that they have agreeable behavior.

Definition : A function f in Fun(A,B) is **monotonic** if $x \subseteq_A y$ implies $f(x) \subseteq_B f(y)$ for all $x,y \in A$. ∎

Since we interpret $\subseteq$ to mean “approximates”, whenever y has at least as much information as x, it follows that f(y) has at least as much information

as f(x). We get more information out of a function by putting more information into it.

An ascending chain in a partially order set can be viewed as a subset of the partially ordered set on which the ordering is total (any two elements are comparable).

Definition : A function $f \in Fun(A,B)$ is **continuous** if it preserves least upper bounds—that is, if $x_1 \subseteq_A x_2 \subseteq_A x_3 \subseteq_A \ldots$ is an ascending chain in A, then $f(lub\{x_i \mid i \geq 1\}) = lub\{f(x_i) \mid i \geq 1\}$. ∎

Note that if f is also monotonic, then $f(x_1) \subseteq_B f(x_2) \subseteq_B f(x_3) \subseteq_B \ldots$ is an ascending chain. Intuitively, continuity means that there are no surprises when taking the least upper bounds (limits) of approximations. The diagram in Figure 10.8 shows the relation between the two chains.

$$\begin{array}{ccccccccc} x_1 & \subseteq_A & x_2 & \subseteq_A & x_3 & \subseteq_A & \ldots & \Rightarrow & lub\{x_i \mid i \geq 1\} \\ \downarrow & & \downarrow & & \downarrow & & & & \downarrow \\ f(x_1) & \subseteq_B & f(x_2) & \subseteq_B & f(x_3) & \subseteq_B & \ldots & \Rightarrow & lub\{f(x_i) \mid i \geq 1\} = f(lub\{x_i \mid i \geq 1\}) \end{array}$$

Figure 10.8: Continuity

A continuous function f has predictable behavior in the sense that if we know its value on the terms of an ascending chain $x_1 \subseteq x_2 \subseteq x_3 \subseteq \ldots$, we also know its value on the least upper bound of the chain since $f(lub\{x_i \mid i \geq 1\})$ is the least upper bound of the chain $f(x_1) \subseteq f(x_2) \subseteq f(x_3) \subseteq \ldots$. It is possible to predict the value of f on $lub\{x_i \mid i \geq 1\}$ by its behavior on each x_i.

Lemma: If $f \in Fun(A,B)$ is continuous, it is also monotonic.

Proof: Suppose f is continuous and $x \subseteq_A y$. Then $x \subseteq_A y \subseteq_A y \subseteq_A y \subseteq_A \ldots$ is an ascending chain in A, and so by the continuity of f,

$$f(x) \subseteq_B lub_B\{f(x),f(y)\} = f(lub_A\{x,y\}) = f(y).$$ ∎

Definition : Define $\mathbf{A \rightarrow B}$ to be the set of functions in Fun(A,B) that are (monotonic and) continuous. This set is ordered by the relation $\subseteq$ from Fun(A,B). ∎

Lemma: The relation $\subseteq$ restricted to $A \rightarrow B$ is a partial order.

Proof: The properties reflexive, transitive, and antisymmetric are inherited by a subset. ∎

The example function $H : (N \rightarrow N) \rightarrow (N \rightarrow N)$ defined by

$$\text{for } g \in N \rightarrow N,\ H\ g = \lambda n\ .\ \text{if } g(n) = \perp \text{ then } 0 \text{ else } 1$$

is neither monotonic nor continuous. It suffices to show that it is not monotonic by a counterexample.

Let $g_1 = \lambda n \,.\, \perp$ and $g_2 = \lambda n \,.\, 0$. Then $g_1 \subseteq g_2$. But $H(g_1) = \lambda n \,.\, 0$, $H(g_2) = \lambda n \,.\, 1$, and the functions $\lambda n \,.\, 0$ and $\lambda n \,.\, 1$ are not related by $\subseteq$ at all.

Two lemmas will be useful in proving the continuity of functions.

Lub Lemma : If $x_1 \subseteq x_2 \subseteq x_3 \subseteq \ldots$ is an ascending chain in a cpo A, and $x_i \subseteq d \in A$ for each $i \geq 1$, it follows that $lub\{x_i \mid i \geq 1\} \subseteq d$.

Proof: By the definition of least upper bound, if d is a bound for the chain, the least upper bound $lub\{x_i \mid i \geq 1\}$ must be no larger than d. ∎

Limit Lemma : If $x_1 \subseteq x_2 \subseteq x_3 \subseteq \ldots$ and $y_1 \subseteq y_2 \subseteq y_3 \subseteq \ldots$ are ascending chains in a cpo A, and $x_i \subseteq y_i$ for each $i \geq 1$, then $lub\{x_i \mid i \geq 1\} \subseteq lub\{y_i \mid i \geq 1\}$.

Proof: For each $i \geq 1$, $x_i \subseteq y_i \subseteq lub\{y_i \mid i \geq 1\}$. Therefore $lub\{x_i \mid i \geq 1\} \subseteq lub\{y_i \mid i \geq 1\}$ by the Lub lemma (take $d = lub\{y_i \mid i \geq 1\}$). ∎

Theorem: The relation $\subseteq$ on $A \rightarrow B$, the set of functions in Fun(A,B) that are monotonic and continuous, is a complete partial order.

Proof: Since $\subseteq$ is a partial order on $A \rightarrow B$, two properties need to be verified.

1. The bottom element in Fun(A,B) is also in $A \rightarrow B$, which can be proved by showing that the function $\perp(x) = \perp_B$ is monotonic and continuous.

2. For any ascending chain in $A \rightarrow B$, its least upper bound, which is an element of Fun(A,B), is also in $A \rightarrow B$, which means that it is monotonic and continuous.

Part 1 : If $x \subseteq_A y$ for some $x,y \in A$, then $\perp(x) = \perp_B = \perp(y)$, which means $\perp(x) \subseteq_B \perp(y)$, and so $\perp$ is a monotonic function. If $x_1 \subseteq_A x_2 \subseteq_A x_3 \subseteq_A \ldots$ is an ascending chain in A, then its image under the function $\perp$ will be the ascending chain $\perp_B \subseteq_B \perp_B \subseteq_B \perp_B \subseteq_B \ldots$, whose least upper bound is $\perp_B$. Therefore $\perp(lub\{x_i \mid i \geq 1\}) = \perp_B = lub\{\perp(x_i) \mid i \geq 1\}$, and $\perp$ is a continuous function.

Part 2 : Let $f_1 \subseteq f_2 \subseteq f_3 \subseteq \ldots$ be an ascending chain in $A \rightarrow B$, and let $F = lub\{f_i \mid i \geq 1\}$ be its least upper bound (in Fun(A,B)). Remember the definition of F, $F(x) = lub\{f_i(x) \mid i \geq 1\}$ for each $x \in A$. We need to show that F is monotonic and continuous so that we know F is a member of $A \rightarrow B$.

Monotonic : If $x \subseteq_A y$, then $f_i(x) \subseteq_B f_i(y) \subseteq_B lub\{f_i(y) \mid i \geq 1\}$ for any i since each f_i is monotonic. Therefore $F(y) = lub\{f_i(y) \mid i \geq 1\}$ is an upper bound for each $f_i(x)$, and so the least upper bound of all the $f_i(x)$ satisfies $F(x) = lub\{f_i(x) \mid i \geq 1\} \subseteq F(y)$, and F is monotonic. This result can also be proved using the Limit lemma. Since $f_i(x) \subseteq_B f_i(y)$ for each $i \geq 1$, $F(x) = lub\{f_i(x) \mid i \geq 1\} \subseteq lub\{f_i(y) \mid i \geq 1\} = F(y)$.

Continuous : Let $x_1 \subseteq_A x_2 \subseteq_A x_3 \subseteq_A \ldots$ be an ascending chain in A. We need to show that $F(lub\{x_j \mid j\geq 1\}) = lub\{F(x_j) \mid j\geq 1\}$ where $F(x) = lub\{f_i(x) \mid i\geq 1\}$ for each $x\in A$. Note that "i" is used to index the ascending chain of functions from $A\rightarrow B$ while "j" is used to index the ascending chains of elements in A and B. So F is continuous if $F(lub\{x_j \mid j\geq 1\}) = lub\{F(x_j) \mid j\geq 1\}$.

Recall these definitions and properties.

1. Each f_i is continuous: $f_i(lub\{x_j \mid j\geq 1\}) = lub\{f_i(x_j) \mid j\geq 1\}$ for each chain $\{x_j \mid j\geq 1\}$ in A.
2. Definition of F: $F(x) = lub\{f_i(x) \mid i\geq 1\}$ for each $x\in A$.

Thus
$$\begin{aligned} F(lub\{x_j \mid j\geq 1\}) &= lub\{f_i(lub\{x_j \mid j\geq 1\}) \mid i\geq 1\} && \text{by 2} \\ &= lub\{lub\{f_i(x_j) \mid j\geq 1\} \mid i\geq 1\} && \text{by 1} \\ &= lub\{lub\{f_i(x_j) \mid i\geq 1\} \mid j\geq 1\} && \ddagger \text{ needs to be shown} \\ &= lub\{F(x_j) \mid j\geq 1\} && \text{by 2.} \end{aligned}$$

The condition ‡ to be proved is illustrated in Figure 10.9.

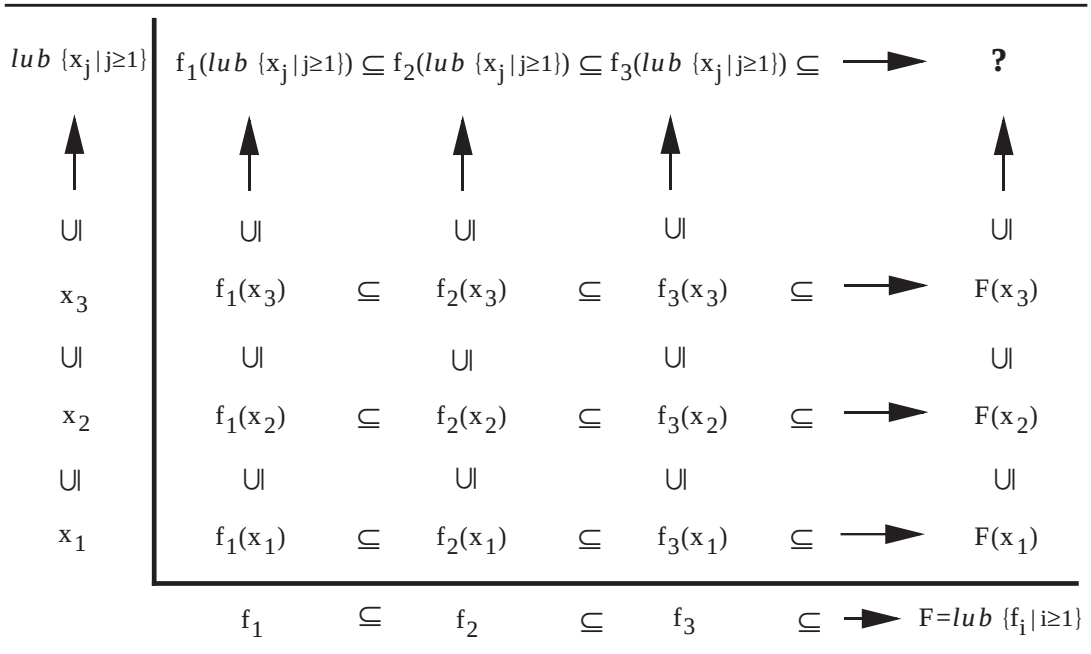


Figure 10.9: Continuity of $F = lub\{f_i \mid i\geq 1\}$

The rows in the diagram correspond to the definition of F as the least upper bound of the ascending chain of functions $lub\{f_i \mid i\geq 1\}$. The columns correspond to the continuity of each f_i—namely, that $lub\{f_i(x_j) \mid j\geq 1\} = f_i(lub\{x_j \mid j\geq 1\})$ for each i and each ascending chain $\{x_j \mid j\geq 1\}$ in A.

First Half : $lub\{lub\{f_i(x_j) \mid j \geq 1\} \mid i \geq 1\} \subseteq lub\{lub\{f_i(x_j) \mid i \geq 1\} \mid j \geq 1\}$

For all k and j, $f_k(x_j) \subseteq lub\{f_i(x_j) \mid i \geq 1\}$ by the definition of F (the rows of Figure 10.9). We have ascending chains $f_k(x_1) \subseteq f_k(x_2) \subseteq f_k(x_3) \subseteq \ldots$ for each k and $lub\{f_i(x_1) \mid i \geq 1\} \subseteq lub\{f_i(x_2) \mid i \geq 1\} \subseteq lub\{f_i(x_3) \mid i \geq 1\} \subseteq \ldots$. So for each k, $lub\{f_k(x_j) \mid j \geq 1\} \subseteq lub\{lub\{f_i(x_j) \mid i \geq 1\} \mid j \geq 1\}$ by the Limit lemma. This corresponds to the top row. Hence $lub\{lub\{f_k(x_j) \mid j \geq 1\} \mid k \geq 1\} \subseteq lub\{lub\{f_i(x_j) \mid i \geq 1\} \mid j \geq 1\}$ by the Lub lemma. Now change k to i.

Second Half : $lub\{lub\{f_i(x_j) \mid i \geq 1\} \mid j \geq 1\} \subseteq lub\{lub\{f_i(x_j) \mid j \geq 1\} \mid i \geq 1\}$

For all i and k, $f_i(x_k) \subseteq f_i(lub\{x_j \mid j \geq 1\}) = lub\{f_i(x_j) \mid j \geq 1\}$ by using the fact that each f_i is monotonic and continuous (the columns of Figure 10.9). So for each k, $lub\{f_i(x_k) \mid i \geq 1\} \subseteq lub\{lub\{f_i(x_j) \mid j \geq 1\} \mid i \geq 1\}$ by the Limit lemma. This corresponds to the rightmost column. Hence $lub\{lub\{f_i(x_k) \mid i \geq 1\} \mid k \geq 1\} \subseteq lub\{lub\{f_i(x_j) \mid j \geq 1\} \mid i \geq 1\}$ by the Lub lemma. Now change k to j.

Therefore F is continuous. ∎

Corollary : Let $f_1 \subseteq f_2 \subseteq f_3 \subseteq \ldots$ be an ascending chain of continuous functions in $A \to B$. Then $F = lub\{f_i \mid i \geq 1\}$ is a continuous function.

Proof: This corollary was proved in Part 2 of the proof of the previous theorem. ∎

In agreement with the notation for denotational semantics in Chapter 9, as a domain constructor, we treat $\to$ as a right associative operation. The domain $A \to B \to C$ means $A \to (B \to C)$, which is the set of continuous functions from A into the set of continuous functions from B to C. If $f \in A \to B \to C$, then for $a \in A$, $f(a) \in B \to C$. Generally, we write ":" to represent membership in a domain. So we write $g : A \to B$ for $g \in A \to B$.

Example 10 : Consider the functions from a small domain of students,

Student = {⊥, Autry, Bates}

to the domain of levels,

Level = {⊥, undergraduate, graduate, nondegree}.

We can think of the functions in Fun(Student,Level) as descriptions of our success in classifying two students, Autry and Bates. The set of all total functions, Fun(Student,Level), contains 64 (4^3) elements, but only 19 of these functions are monotonic and continuous. The structure of Student $\to$ Level is portrayed by the lattice-like structure in Figure 10.10, where the values in the domains are denoted by only their first letters. ∎

Since the domain Student of a function in Student$\to$Level is finite, it is enough to show that the function is monotonic as we show in the next theorem.

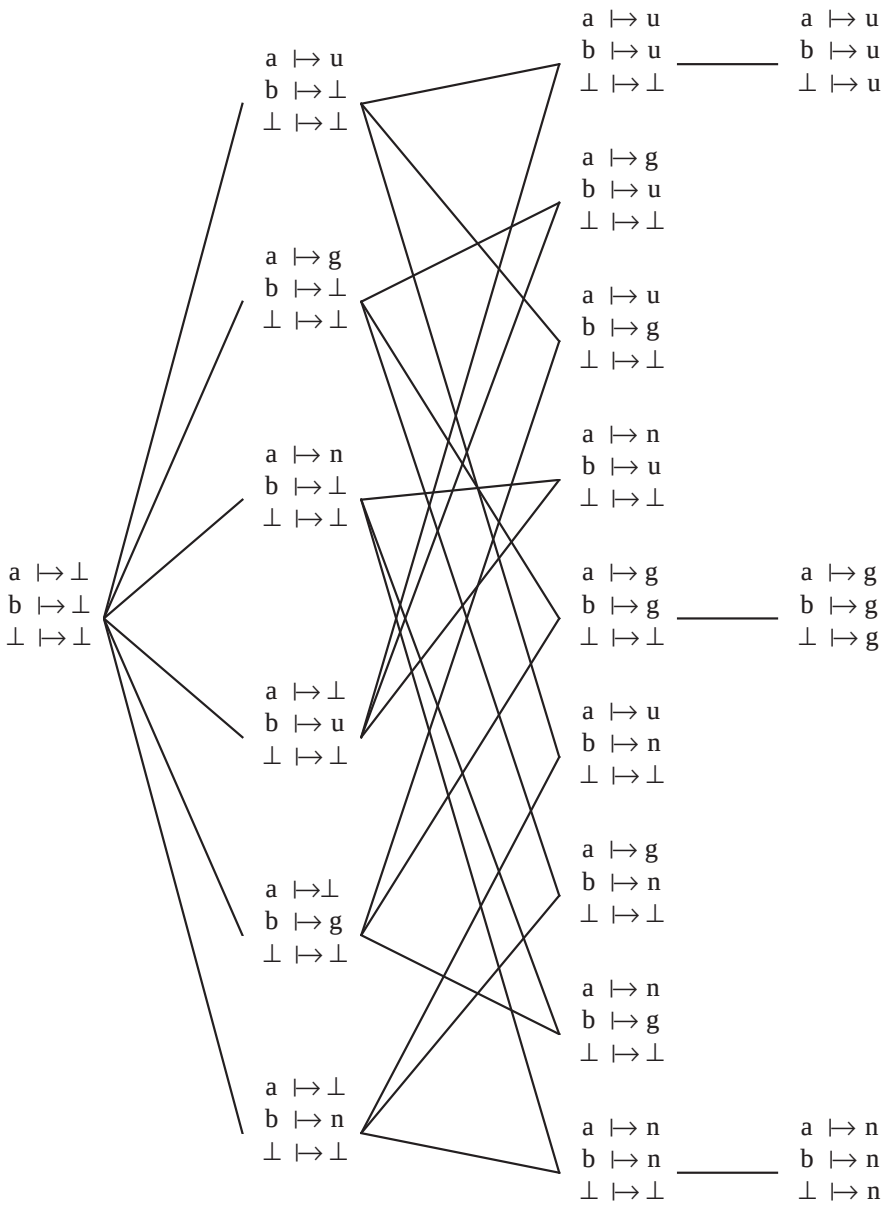


Figure 10.10: Function domain Student → Level

Theorem: If A and B are cpo's, A is a finite set, and f∈Fun(A,B) is monotonic, then f is also continuous.

Proof: Let $x_1 \subseteq_A x_2 \subseteq_A x_3 \subseteq_A \ldots$ be an ascending chain in A. Since A is finite, for some k, $x_k = x_{k+1} = x_{k+2} = \ldots$. So the chain is a finite set $\{x_1, x_2, x_3, \ldots, x_k\}$ whose least upper bound is x_k. Since f is monotonic, $f(x_1) \subseteq_B f(x_2) \subseteq_B f(x_3) \subseteq_B$

$\ldots \subseteq_B f(x_k) = f(x_{k+1}) = f(x_{k+2}) = \ldots$ is an ascending chain in B, which is also a finite set—namely, $\{f(x_1), f(x_2), f(x_3), \ldots, f(x_k)\}$ with $f(x_k)$ as its least upper bound. Therefore, $f(lub\{x_i \mid i \geq 1\}) = f(x_k) = lub\{f(x_i) \mid i \geq 1\}$, and f is continuous. ∎

Lemma: The function f : Student → Level defined by

f(⊥) = graduate, f(Autry) = nondegree, f(Bates) = graduate

is neither monotonic nor continuous.

Proof: Clearly, ⊥ ⊆ Autry. But f(⊥) = graduate and f(Autry) = nondegree are incomparable. Therefore, f is not monotonic. By the contrapositive of an earlier theorem, if f is not monotonic, it is also not continuous. ∎

Continuity of Functions on Domains

The notation used for the special functions defined on domains implied that they were continuous—for example, *first* : A×B→A. To justify this notation, a theorem is needed.

Theorem: The following functions on domains and their analogs are continuous:

1. *first* : A×B→A
2. *inS* : A→S where S = A+B
3. *outA* : A+B→A
4. *isA* : A+B→T

Proof:

1. Let $\langle a_1,b_1\rangle \subseteq \langle a_2,b_2\rangle \subseteq \langle a_3,b_3\rangle \subseteq \ldots$ be an ascending chain in A×B. Then $lub\{first\ \langle a_i,b_i\rangle \mid i \geq 1\} = lub\{a_i \mid i \geq 1\}$
   $= first\ \langle lub\{a_i \mid i \geq 1\}, lub\{b_i \mid i \geq 1\}\rangle = first(lub_{A\times B}\{\langle a_i,b_i\rangle \mid i \geq 1\})$.
2. An exercise.
3. An exercise.
4. An ascending chain $x_1 \subseteq x_2 \subseteq x_3 \subseteq \ldots$ in A+B may repeat $\perp_{A+B}$ forever or eventually climb into either A×{1} or B×{2}. In the first case, the least upper bound will be $\perp_{A+B}$, and in the other two cases the *lub* will be some a∈A or some b∈B.

   Case 1: $\{x_i \mid i \geq 1\} = \{\perp_{A+B}\}$. Then $isA(lub_{A+B}\{x_i \mid i \geq 1\}) = isA(\perp_{A+B}) = \perp_T$, and $lub_T\{isA(x_i) \mid i \geq 1\} = lub_T\{isA(\perp_{A+B})\} = lub_T\{\perp_T\} = \perp_T$.

   Case 2: $\{x_i \mid i \geq 1\} = \{\perp_{A+B}, \ldots, \perp_{A+B}, \langle a_1,1\rangle, \langle a_2,1\rangle, \ldots\}$ where $a_1 \subseteq_A a_2 \subseteq_A a_3 \subseteq_A \ldots$ is a chain in A. Suppose $a = lub\{a_i \mid i \geq 1\}$. Then $isA(lub_{A+B}\{x_i \mid i \geq 1\}) = isA(\langle a,1\rangle) = true$, and $lub_T\{isA(x_i) \mid i \geq 1\} = lub_T\{\perp, \ldots, \perp, true, true, \ldots\} = true$.

Case 3: $\{x_i \mid i\geq 1\} = \{\perp_{A+B},\ldots,\perp_{A+B},<b_1,2>,<b_2,2>,\ldots\}$ where $b_1 \subseteq_B b_2 \subseteq_B b_3 \subseteq_B \ldots$ is a chain in B. Suppose $b = lub\{b_i \mid i\geq 1\}$.Then $isA(lub_{A+B}\{x_i \mid i\geq 1\}) = isA(<b,2>)$ = false, and $lub_T\{isA(x_i) \mid i\geq 1\} = lub_T\{\perp,\ldots,\perp,\text{false},\text{false},\ldots\}$ = false. ∎

The functions defined on lists, such as *head* and *tail*, are mostly built from the selector functions for products and sums. The list functions can be shown to be continuous by proving that composition preserves the continuity of functions.

Theorem: The composition of continuous functions is continuous.

Proof: Suppose $f : A \rightarrow B$ and $g : B \rightarrow C$ are continuous functions. Let $a_1 \subseteq a_2 \subseteq a_3 \subseteq \ldots$ be an ascending chain in A. Then $f(a_1) \subseteq f(a_2) \subseteq f(a_3) \subseteq \ldots$ is an ascending chain in B with $f(lub\{a_i \mid i\geq 1\}) = lub\{f(a_i) \mid i\geq 1\}$ by the continuity of f. Since g is continuous, $g(f(a_1)) \subseteq g(f(a_2)) \subseteq g(f(a_3)) \subseteq \ldots$ is an ascending chain in C with $g(lub\{f(a_i) \mid i\geq 1\}) = lub\{g(f(a_i)) \mid i\geq 1\}$. Therefore $g(f(lub\{a_i \mid i\geq 1\})) = g(lub\{f(a_i) \mid i\geq 1\}) = lub\{g(f(a_i)) \mid i\geq 1\}$ and $g\circ f$ is continuous. ∎

To handle *tail*, *prefix*, and *affix* we need a generalization of this theorem to allow for tuples of continuous functions, a result that appears as an exercise.

Theorem: The following functions on lists are continuous:

1. $head : D^* \rightarrow D$
2. $tail : D^* \rightarrow D^*$
3. $null : D^* \rightarrow T$
4. $prefix : D \times D^* \rightarrow D^*$
5. $affix : D^* \times D \rightarrow D^*$

Proof: For 1, 2, 4, 5 use the continuity of the compositions of continuous functions and the previous theorems. A case analysis is needed to deal with ascending sequences that contain mostly *nil* values.

3. An ascending chain $x_1 \subseteq x_2 \subseteq x_3 \subseteq \ldots$ in D^* may repeat $\perp_{D^*}$ forever or eventually climb into D^k, where $k\geq 0$ and $D^0 = \{nil\}$.

   Case 1: $\{x_i \mid i\geq 1\} = \{\perp_{D^*}\}$. $null(lub_{D^*}\{x_i \mid i\geq 1\}) = null(\perp_{D^*}) = \perp_T$, and $lub_T\{null(x_i) \mid i\geq 1\} = lub_T\{\perp_T\} = \perp_T$.

   Case 2: $\{x_i \mid i\geq 1\} = \{\perp_{D^*},\perp_{D^*},\ldots,\perp_{D^*},<nil,0>,<nil,0>,\ldots\}$.
   $null(lub_{D^*}\{x_i \mid i\geq 1\}) = null(<nil,0>) = \text{true}$, and
   $lub_T\{null(x_i) \mid i\geq 1\} = lub_T\{\perp_T,\perp_T,\ldots,\perp_T,\text{true},\text{true},\ldots\} = \text{true}$.

   Case 3: $\{x_i \mid i\geq 1\} = \{\perp_{D^*},\ldots,\perp_{D^*},<d_1,k>,<d_2,k>,\ldots\}$ where $d_i \in D^k$ for some $k>0$ and $d_1 \subseteq_{D^k} d_2 \subseteq_{D^k} d_3 \subseteq_{D^k} \ldots$ is a chain in D^k. $null(lub_{D^*}\{x_i \mid i\geq 1\}) = null(<lub_{D^k}\{d_i \mid i\geq 1\},k>) = \text{false}$ since $(lub_{D^k}\{d_i \mid i\geq 1\}) \in D^k$, and $lub_D\{null(x_i) \mid i\geq 1\} = lub_D\{null<d_1,k>, null<d_2,k>, \ldots\} = lub_D\{\text{false}, \text{false}, \ldots\} = \text{false}$. ∎

Exercises

1. Determine which of the following ordered sets are complete partial orders:
   a) Divides ordering on {1,3,6,9,12,18}.
   b) Divides ordering on {2,3,6,12,18}.
   c) Divides ordering on {2,4,6,8,10,12}.
   d) Divides ordering on the set of positive integers.
   e) Divides ordering on the set P of prime numbers.
   f) Divides ordering on the set $P \cup \{1\}$.
   g) $\subseteq$ (subset) on the nonempty subsets of {a,b,c,d}.
   h) $\subseteq$ (subset) on the collection of all finite subsets of the natural numbers.
   i) $\subseteq$ (subset) on the collection of all subsets of the natural numbers whose complement is finite.
2. Which of the partially ordered sets in exercise 1 are also lattices?
3. Let S = {1,2,3,4,5,6,9,15,25,30} be ordered by the divides relation.
   a) Find all lower bounds for {6,30}.
   b) Find all lower bounds for {4,6,15}.
   c) Find all upper bounds for {1,2,3}.
   d) Does {4,9,25} have an upper bound?
4. Show that $\subseteq_{A \times B}$ is a partial order on A×B.
5. Show that $\subseteq_{A+B}$ is a partial order on A+B.
6. Prove that the least upper bound of an ascending chain $x_1 \subseteq x_2 \subseteq x_3 \subseteq \ldots$ in a domain D is unique.
7. Let Hair = {⊥,black,blond,brown} and Eyes = {⊥,blue,brown,gray} be two elementary domains (flat complete partially ordered sets).
   a) Sketch a Hasse diagram showing all the elements of Hair × Eyes and the relationships between its elements under $\subseteq$.
   b) Sketch a Hasse diagram showing all the elements of Hair+Eyes and the relationships between its elements under $\subseteq$.

8. Suppose that A = {⊥,a} and B = {⊥,b,c,d} are elementary domains.
   a) Sketch a Hasse diagram showing all seven elements of A→B and the relationships between its elements under ⊆.
   b) Give an example of one function in Fun(A,B) that is not monotonic.
   c) Sketch a Hasse diagram showing all the elements of A×B and the relationships between its elements under ⊆.

9. Suppose A = {⊥,a,b} and B = {⊥,c} are elementary domains.
   a) Sketch a Hasse diagram showing all the elements of (A→B)+(A×B) and their ordering under the induced partial order. Represent functions as sets of ordered pairs. Since A→B and A×B are disjoint, omit the tags on the elements, but provide subscripts for the bottom elements.
   b) Give one example of a function in Fun(A→B,A×B) that is continuous and one that is not monotonic.

10. Prove the following property:

    A function f in Fun(A,B) is continuous if and only if both of the following conditions hold.
    a) f is monotonic.
    b) For any ascending chain $x_1 \subseteq x_2 \subseteq x_3 \subseteq \ldots$ in A, $f(lub_A\{x_i \mid i \geq 1\}) \subseteq lub_B\{f(x_i) \mid i \geq 1\}$.

11. Let $A = \{\bot, a_1, a_2, \ldots, a_m\}$ and $B = \{\bot, b_1, b_2, \ldots, b_n\}$ be flat domains. Show that
    a) Fun(A,B) has $(n+1)^{m+1}$ elements.
    b) A→B has $n+(n+1)^m$ elements.

12. Prove that *inS* and *outA* are continuous functions.

13. Prove that *head* and *tail* are continuous functions.

14. Tell whether these functions F : (N→N)→(N→N) are monotonic and/or continuous.
    a) F g n = if *total*(g) then g(n) else ⊥, where *total*(g) is true if and only if g(n) is defined (not ⊥) for all proper n∈N.
    b) F g n = if g = (λn .0) then 1 else 0.
    c) F g n = if n∉ *dom*(g) then 0 else ⊥, where *dom*(g) ={n∈N | g(n)≠⊥} denotes the domain of g.

15. Let $N = \{\perp,0,1,2,3,\ldots\}$ be the elementary domain of natural numbers. A function $f : N \to N$ is called **strict** if $f(\perp) = \perp$. Consider the function $add1 : N \to N$ defined by $add1(n) = n+1$ for all $n \in N$ with $n \neq \perp$. Prove that if add1 is monotonic, it must also be strict.

16. Consider the function $F : (N \to N) \to (N \to N)$ defined by

    for $g \in N \to N$, $F\ g = \lambda n$. if $g(n) = \perp$ then 0 else 1

    Describe $F\ g_1$, $F\ g_2$, and $F\ g_3$ where the $g_k : N \to N$ are defined by

    $g_1(n) = n$

    $g_2(n) =$ if $n > 0$ then $n / 0$ else $\perp$

    $g_3(n) =$ if *even*(n) then $n+1$ else $\perp$

17. Prove that if $f \in Fun(A,B)$, where A and B are domains (cpo's), is a constant function (there is a $b \in B$ such that $f(a) = b$ for all $a \in A$), then f is continuous.

18. An ascending chain $x_1 \subseteq x_2 \subseteq x_3 \subseteq \ldots$ in a cpo A is called **stationary** if there is an $n \geq 1$ such that for all $i \geq n$, $x_i = x_n$. Carefully prove the following properties:

    a) If every ascending chain in A is stationary and $f \in Fun(A,B)$ is monotonic, then f must be continuous.

    b) If an ascending chain $x_1 \subseteq x_2 \subseteq x_3 \subseteq \ldots$ is not stationary, then for all $i \geq 1$, $x_i \neq lub\{x_j \mid j \geq 1\}$. *Hint*: Prove the contrapositive.

19. Prove the following lemma: If $a_1 \subseteq a_2 \subseteq a_3 \subseteq \ldots$ and $b_1 \subseteq b_2 \subseteq b_3 \subseteq \ldots$ are ascending chains with the property that for each $m \geq 1$ there exists an $n \geq 1$ such that $a_m \subseteq b_n$, it follows that $lub\{a_i \mid i \geq 1\} \subseteq lub\{b_i \mid i \geq 1\}$.

10.3 FIXED-POINT SEMANTICS

Functions, and in particular recursively defined functions, are central to computer science. Functions are used not only in programming but also in describing the semantics of programming languages as witnessed by the recursive definitions in denotational specifications. Recursion definitions entail a circularity that can make them suspect. Many of the paradoxes of logic and mathematics revolve about circular definitions—for example, the set of all sets. Considering the suspicious nature of circular definitions, how can we be certain that function definitions have a consistent model? The use of domains (complete partially ordered sets) and the associated fixed-point theory

to be developed below put recursive function definitions and denotational semantics on a firm, consistent foundation.

Our goal is to develop a coherent theory of functions that makes sense out of recursive definitions. In describing fixed-point semantics we restate some of the definitions from section 10.2 as we motivate the concepts. The discussion breaks into two parts: (1) interpreting partial functions so that they are total, and (2) giving meaning to a recursively defined function as an approximation of a sequence of "finite" functions.

First Step

We transform partial functions into analogous total functions.

Example 11 : Let f be a function on a few natural numbers with domain D = {0,1,2} and codomain C = {0,1,2} and with its rule given as

f(n) = 2 / n or as a set of ordered pairs: f = {<1,2>,<2,1>}.

Note that f(0) is undefined; therefore f is a partial function. Now extend f to make it a total function.

f = {<0,?>,<1,2>,<2,1>}.

Add an undefined element to the codomain, $C^+ = \{\perp_{C^+},0,1,2\}$, and for symmetry, do likewise with the domain, $D^+ = \{\perp_{D^+},0,1,2\}$.

Then define the **natural extension** of f by having $\perp_{D^+}$ map to $\perp_{C^+}$ under f.

$f^+ = \{<\perp,\perp>,<0,\perp>,<1,2>,<2,1>\}$.

From this point on, we drop the subscripts on $\perp$ unless they are needed to clarify an example. Finally, define a relationship that orders functions and domains according to how "defined" they are, putting a lattice-like structure on the elementary domains: For $x,y \in D^+$, $x \subseteq y$ if $x=\perp$ or $x=y$. It follows that $f \subseteq f^+$. ∎

This relation is read "f approximates f^+" or "f is less defined than or equal to f^+". D^+ and C^+ are examples of the flat domains of the previous section.

Consider the function g = {<⊥,⊥>,<0,0>,<1,2>,<2,1>}, which is an extension of f^+ that is slightly more defined. The relationship between the two functions is denoted by $f^+ \subseteq g$. Observe that the two functions agree where they are both defined (do not map to ⊥).

Theorem: Let f^+ be a natural extension of a function between two sets D and C so that f^+ is a total function from D^+ to C^+. Then f^+ is monotonic and continuous.

Proof: Let $x_1 \subseteq x_2 \subseteq x_3 \subseteq \ldots$ be an ascending chain in the domain $D^+ = D \cup \{\perp_{D^+}\}$. There are two possibilities for the behavior of the chain.

Case 1 : $x_i = \perp_{D^+}$ for all $i \geq 1$. Then $lub\{x_i \mid i \geq 1\} = \perp_{D^+}$, and $f^+(lub\{x_i \mid i \geq 1\}) = f^+(\perp_{D^+}) = \perp_{C^+} = lub\{\perp_{C^+}\} = lub\{f^+(x_i) \mid i \geq 1\}$.

Case 2 : $x_i = \perp_{D^+}$ for $1 \leq i \leq k$ and $\perp_{D^+} \neq x_{k+1} = x_{k+2} = x_{k+3} = \ldots$, since once the terms move above bottom, the sequence is constant in a flat domain. Then $lub\{x_i \mid i \geq 1\} = x_{k+1}$, and $f^+(lub\{x_i \mid i \geq 1\}) = f^+(x_{k+1}) = lub\{\perp_{C^+}, f^+(x_{k+1})\} = lub\{f^+(x_i) \mid i \geq 1\}$. If f^+ is continuous, it is also monotonic. ∎

Since many functions used in programming, such as "addition", "less than", and "or", are binary operations, their natural extensions need to be clarified.

Definition : The **natural extension** of a function whose domain is a Cartesian product—namely, $f : D_1^+ x D_2^+ x \ldots x D_n^+ \rightarrow C^+$—has the property that $f^+(x_1, x_2, \ldots, x_n) = \perp_C$ whenever at least one $x_i = \perp$. Any function that satisfies this property is known as a **strict** function. ∎

Theorem: If $f^+: D_1^+ x D_2^+ x \ldots x D_n^+ \rightarrow C^+$ is a natural extension where D_i^+, $1 \leq i \leq n$, and C^+ are elementary domains, then f^+ is monotonic and continuous.

Proof: Consider the case where n=2. We show f^+ is continuous.
Let $\langle x_1, y_1 \rangle \subseteq \langle x_2, y_2 \rangle \subseteq \langle x_3, y_3 \rangle \subseteq \ldots$ be an ascending chain in $D_1^+ x D_2^+$. Since D_1^+ and D_2^+ are elementary domains, the chains $\{x_i \mid i \geq 1\}$ and $\{y_i \mid i \geq 1\}$ must follow one of the two cases in the previous proof—namely, all $\perp$ or eventually constant proper values in D_1^+ and D_2^+, respectively.

Case 1 : $lub\{x_i \mid i \geq 1\} = \perp_{D_1^+}$ or $lub\{y_i \mid i \geq 1\} = \perp_{D_2^+}$ (or both). Then $f^+(lub\{\langle x_i, y_i \rangle \mid i \geq 1\}) = f^+(\langle lub\{x_i \mid i \geq 1\}, lub\{y_i \mid i \geq 1\} \rangle) = \perp_C$ because f^+ is a natural extension and one of its arguments is $\perp$; furthermore, $lub\{f^+(\langle x_i, y_i \rangle) \mid i \geq 1\} = lub\{\perp_{C^+}\} = \perp_{C^+}$, since at least one of the chains must be all $\perp$.

Case 2 : $lub\{x_i \mid i \geq 1\} = x \in D_1$ and $lub\{y_i \mid i \geq 1\} = y \in D_2$ (neither is $\perp$). Since D_1^+ and D_2^+ are both elementary domains, there is an integer k such that $x_i = x$ and $y_i = y$ for all $i \geq k$. So $f^+(lub\{\langle x_i, y_i \rangle \mid i \geq 1\}) = f^+(\langle lub\{x_i \mid i \geq 1\}, lub\{y_i \mid i \geq 1\} \rangle) = f^+(\langle x, y \rangle) \in C^+$ and $lub\{f^+(\langle x_i, y_i \rangle) \mid i \geq 1\} = lub\{\perp_{C^+}, f^+(\langle x, y \rangle)\} = f^+(\langle x, y \rangle)$. ∎

Example 12 : Consider the natural extension of the conditional expression operation (if a b c) = if a then b else c.

The natural extension unduly restricts the meaning of the conditional expression—for example, we prefer that the following expression returns 0 when m=1 and n=0 instead of causing a fatal error: if n>0 then m/n else 0.

But if we interpret the undefined operation 1/0 as $\perp$, when m=1 and n=0,

$$(\text{if}^+ \; n>0 \; m/n \; 0) = (\text{if}^+ \; \text{false} \; \perp \; 0) = \perp \text{ for a natural extension.}$$ ∎

As we continue with the development of fixed-point semantics, we drop the superscript plus sign ($^+$) on sets and functions since all sets will be assumed to be domains (cpo's) and all functions will be naturally extended unless otherwise specified.

Second Step

We now define the meaning of a recursive definition of a function defined on complete partially ordered sets (domains) as the limit of a sequence of approximations.

Example 13 : Consider a recursively defined function $f : N \to N$ where $N = \{\perp,0,1,2,3,\ldots\}$ and

$$f(n) = \text{if } n=0 \text{ then } 5 \text{ else if } n=1 \text{ then } f(n+2) \text{ else } f(n-2). \qquad (\dagger)$$

Two questions can be asked about a recursive definition of a function.

1. What function, if any, does this equation in f denote?
2. If the equation specifies more than one function, which one should be selected?

Define a **functional** F, a function on functions, by

$$F : (N \to N) \to (N \to N) \text{ where}$$
$$(F(f))\,(n) = \text{if } n=0 \text{ then } 5 \text{ else if } n=1 \text{ then } f(n+2) \text{ else } f(n-2). \qquad (\ddagger)$$

Assuming function application associates to the left, we usually omit the parentheses with multiple applications, writing F f n for (F(f)) (n). A function, $f : N \to N$, satisfies the original definition (†) if and only if it is a **fixed point** of the definition of F (‡)—namely, F f n = f(n) for all $n \in N$ or just F f = f. ∎

Just in case this equivalence has not been understood, we go through it once more carefully. Suppose $f : D \to C$ is a function defined recursively by $f(x) = \alpha(x,f)$ for each $x \in D$ where $\alpha(x,f)$ is some expression in x and f. Furthermore, let $F : (D \to C) \to (D \to C)$ be the functional defined by F f x = $\alpha(x,f)$. Then F(f) = f if and only if F f x = f x for all $x \in D$ if and only if $\alpha(x,f)$ = f x for all $x \in D$, which is the same as $f(x) = \alpha(x,f)$ for all $x \in D$. Observe that the symbol "f" plays different roles in (†) and (‡). In the recursive definition (†), "f" is the name of the function being defined, whereas in the functional definition (‡), "f" is a formal parameter to the (nonrecursive) functional F being defined.

The notation of the lambda calculus is frequently used to define these functionals.

$$F\,f = \lambda n \,.\, \text{if } n=0 \text{ then } 5 \text{ else if } n=1 \text{ then } f(n+2) \text{ else } f(n-2)$$

or

$$F = \lambda f \,.\, \lambda n \,.\, \text{if } n=0 \text{ then } 5 \text{ else if } n=1 \text{ then } f(n+2) \text{ else } f(n-2).$$

Fixed points occur frequently in mathematics. For instance, solving simple equations can be framed as fixed-point problems. Consider functions defined on the set of natural numbers N and consider fixed points of the functions—namely, $n \in N$ such that $g(n) = n$.

Function	Fixed points
$g(n) = n^2-6n$	0 and 7
$g(n) = n$	all $n \in N$
$g(n) = n+5$	none
$g(n) = 2$	2

For the first function, $g(0) = 0^2-6\cdot0 = 0$ and $g(7) = 7^2-6\cdot7 = 7$.

Certainly the function specified by a recursive definition must be a fixed point of the functional F. But that is not enough. The function $g = \lambda n \,.\, 5$ is a fixed point of F in example 13 as shown by the following calculation:

$$\begin{aligned} F\ g &= \lambda n \,.\, \text{if } n=0 \text{ then } 5 \text{ else if } n=1 \text{ then } g(n+2) \text{ else } g(n-2) \\ &= \lambda n \,.\, \text{if } n=0 \text{ then } 5 \text{ else if } n=1 \text{ then } 5 \text{ else } 5 \\ &= \lambda n \,.\, 5 = g. \end{aligned}$$

The only problem is that this fixed point does not agree with the operational view of the function definition. It appears that $f(1) = f(3) = f(1) = \ldots$ does not produce a value, whereas $g(1) = 5$. We need to find a fixed point for F that captures the entire operational behavior of the recursive definition.

When the functional corresponding to a recursive definition (equation) has more than one fixed point, we need to choose one of them as *the* function specified by the definition. It turns out that the fixed points of a suitable functional are partially ordered by $\subseteq$ in such a way that one of those functions is less defined than all of the other fixed points. Considering all the fixed points of a functional F, the least defined one makes sense as the function specified because of the following reasons:

1. Any fixed point of F embodies the information that can be deduced from F.
2. The least fixed point includes no more information than what *must* be deduced.

Define the meaning of a recursive definition of a function to be the least fixed point with respect to $\subseteq$ of the corresponding functional F. We show next that a unique least fixed point exists for a continuous functional. The following theorem proves the existence and provides a method for constructing the least fixed point.

Notation : We define f^k for each $k \geq 0$ inductively using the rules:
$f^0(x) = x$ is the identity function and
$f^{n+1}(x) = f(f^n(x))$ for $n \geq 0$. ∎

Fixed-Point Theorem: If D with $\subseteq$ is a complete partial order and $g : D \to D$ is any monotonic and continuous function on D, then g has a least fixed point in D with respect to $\subseteq$.

Proof:

Part 1: g has a fixed point. Since D is a cpo, $g^0(\bot) = \bot \subseteq g(\bot)$. Also, since g is monotonic, $g(\bot) \subseteq g(g(\bot)) = g^2(\bot)$. In general, since g is monotonic, $g^i(\bot) \subseteq g^{i+1}(\bot)$ implies $g^{i+1}(\bot) = g(g^i(\bot)) \subseteq g(g^{i+1}(\bot)) = g^{i+2}(\bot)$. So by induction, $\bot \subseteq g(\bot) \subseteq g^2(\bot) \subseteq g^3(\bot) \subseteq g^4(\bot) \subseteq \ldots$ is an ascending chain in D, which must have a least upper bound $u = lub\{g^i(\bot) \mid i \geq 0\} \in D$.

Then $g(u) = g(lub\{g^i(\bot) \mid i \geq 0\})$

$= lub\{g(g^i(\bot)) \mid i \geq 0\}$ because g is continuous

$= lub\{g^{i+1}(\bot) \mid i \geq 0\}$

$= lub\{g^i(\bot) \mid i > 0\} = u$.

That is, u is a fixed point for g. Note that $g^0(\bot) = \bot$ has no effect on the least upper bound of $\{g^i(\bot) \mid i \geq 0\}$.

Part 2: u is the least fixed point. Let $v \in D$ be another fixed point for g. Then $\bot \subseteq v$ and $g(\bot) \subseteq g(v) = v$, the basis step for induction. Suppose $g^i(\bot) \subseteq v$. Then since g is monotonic, $g^{i+1}(\bot) = g(g^i(\bot)) \subseteq g(v) = v$, the induction step. Therefore, by mathematical induction, $g^i(\bot) \subseteq v$ for all $i \geq 0$. So v is an upper bound for $\{g^i(\bot) \mid i \geq 0\}$. Hence $u \subseteq v$ by the Lub lemma, since u is the least upper bound for $\{g^i(\bot) \mid i \geq 0\}$. ∎

Corollary: Every continuous functional $F : (A \to B) \to (A \to B)$, where A and B are domains, has a least fixed point $F_{fp} : A \to B$, which can be taken as the meaning of the (recursive) definition corresponding to F.

Proof: This is an immediate application of the fixed-point theorem. ∎

Example 13 (revisited): Consider the functional $F : (N \to N) \to (N \to N)$ that we defined earlier corresponding to the recursive definition (†),

F f n = if n=0 then 5 else if n=1 then f(n+2) else f(n-2). (‡)

Construct the ascending sequence

$$\bot \subseteq F(\bot) \subseteq F^2(\bot) \subseteq F^3(\bot) \subseteq F^4(\bot) \subseteq \ldots$$

and its least upper bound following the proof of the fixed-point theorem.

Use the following abbreviations:

$f_0(n) = F^0 \bot n = \bot(n)$

$f_1(n) = F \bot n = F\ f_0\ n$

$f_2(n) = F\ (F \bot)\ n = F\ f_1\ n$

$f_{k+1}(n) = F^{k+1} \bot n = F\ f_k\ n$, in general.

Now calculate a few terms in the ascending chain

$$f_0 \subseteq f_1 \subseteq f_2 \subseteq f_3 \subseteq \ldots.$$

$f_0(n) = F^0\ \bot\ n = \bot(n) = \bot$ for $n \in N$, the everywhere undefined function.

$f_1(n) = F\ \bot\ n = F\ f_0\ n$
$= \text{if } n=0 \text{ then } 5 \text{ else if } n=1 \text{ then } f_0(n+2) \text{ else } f_0(n-2)$
$= \text{if } n=0 \text{ then } 5 \text{ else if } n=1 \text{ then } \bot(n+2) \text{ else } \bot(n-2)$
$= \text{if } n=0 \text{ then } 5 \text{ else } \bot$

$f_2(n) = F^2\ \bot\ n = F\ f_1\ n$
$= \text{if } n=0 \text{ then } 5 \text{ else if } n=1 \text{ then } f_1(n+2) \text{ else } f_1(n-2)$
$= \text{if } n=0 \text{ then } 5$
$\quad \text{else if } n=1 \text{ then } f_1(3)$
$\quad\quad \text{else (if } n-2=0 \text{ then } 5 \text{ else } \bot)$
$= \text{if } n=0 \text{ then } 5$
$\quad \text{else if } n=1 \text{ then } \bot$
$\quad\quad \text{else if } n=2 \text{ then } 5 \text{ else } \bot$

$f_3(n) = F^3\ \bot\ n = F\ f_2\ n$
$= \text{if } n=0 \text{ then } 5 \text{ else if } n=1 \text{ then } f_2(n+2) \text{ else } f_2(n-2)$
$= \text{if } n=0 \text{ then } 5$
$\quad \text{else if } n=1 \text{ then } f_2(3)$
$\quad\quad \text{else (if } n-2=0 \text{ then } 5$
$\quad\quad\quad \text{else if } n-2=1 \text{ then } \bot$
$\quad\quad\quad\quad \text{else if } n-2=2 \text{ then } 5 \text{ else } \bot)$
$= \text{if } n=0 \text{ then } 5$
$\quad \text{else if } n=1 \text{ then } \bot$
$\quad\quad \text{else if } n=2 \text{ then } 5$
$\quad\quad\quad \text{else if } n=3 \text{ then } \bot$
$\quad\quad\quad\quad \text{else if } n=4 \text{ then } 5 \text{ else } \bot$

$f_4(n) = F^4\ \bot\ n = F\ f_3\ n$
$= \text{if } n=0 \text{ then } 5 \text{ else if } n=1 \text{ then } f_3(n+2) \text{ else } f_3(n-2)$
$= \text{if } n=0 \text{ then } 5$
$\quad \text{else if } n=1 \text{ then } f_3(3)$
$\quad\quad \text{else (if } n-2=0 \text{ then } 5$
$\quad\quad\quad \text{else if } n-2=1 \text{ then } \bot$
$\quad\quad\quad\quad \text{else if } n-2=2 \text{ then } 5$
$\quad\quad\quad\quad\quad \text{else if } n-2=3 \text{ then } \bot$
$\quad\quad\quad\quad\quad\quad \text{else if } n-2=4 \text{ then } 5 \text{ else } \bot)$

= if n=0 then 5
else if n=1 then ⊥
else if n=2 then 5
else if n=3 then ⊥
else if n=4 then 5
else if n=5 then ⊥
else if n=6 then 5 else ⊥

A pattern seems to be developing.

Lemma: For all i≥0, $f_i(n)$ = if n<2i and *even*(n) then 5 else ⊥
= if n<2i then (if *even*(n) then 5 else ⊥) else ⊥.

Proof: The proof proceeds by induction on i.

1. By the previous computations, for i = 0 (also i= 1, 2, 3, and 4)
   $f_i(n)$ = if n<2i then (if *even*(n) then 5 else ⊥) else ⊥

2. As the induction hypothesis, assume that $f_i(n)$ = if n<2i then (if *even*(n) then 5 else ⊥) else ⊥, for some arbitrary i≥0.
   Then
   $f_{i+1}(n)$ = F f_i n
   = if n=0 then 5 else if n=1 then $f_i(n+2)$ else $f_i(n-2)$
   = if n=0 then 5
   else if n=1 then $f_i(3)$
   else (if n-2<2i then (if *even*(n–2) then 5 else ⊥) else ⊥)
   = if n=0 then 5
   else if n=1 then ⊥
   else (if n<2i+2 then (if *even*(n) then 5 else ⊥) else ⊥)
   = if n<2(i+1) then (if *even*(n) then 5 else ⊥) else ⊥.

Therefore our pattern for the f_i is correct. ∎

The least upper bound of the ascending chain $f_0 \subseteq f_1 \subseteq f_2 \subseteq f_3 \subseteq \ldots$, where $f_i(n)$ = if n<2i then (if *even*(n) then 5 else ⊥) else ⊥, must be defined (not ⊥) for any n where some f_i is defined, and must take the value 5 there. Hence the least upper bound is

$F_{fp}(n)$ = (*lub*{f_i | i≥0}) n
= (*lub*{F^i ⊥ | i≥0}) n
= if *even*(n) then 5 else ⊥, for all n∈N,

and this function can be taken as the meaning of the original recursive definition. Figure 10.11 shows the chain of approximating functions as sets of ordered pairs, omitting the undefined (⊥) values. Following this set theoretic viewpoint, the least upper bound of the ascending chain can be taken as the union of all these functions, *lub*{f_i | i≥0} = ∪{f_i | i≥0}, to get a function that is undefined for all odd values.

$f_0 = \varnothing$

$f_1 = \{ <0,5> \}$

$f_2 = \{ <0,5>,<2,5> \}$

$f_3 = \{ <0,5>,<2,5>,<4,5> \}$

$f_4 = \{ <0,5>,<2,5>,<4,5>,<6,5> \}$

$f_5 = \{ <0,5>,<2,5>,<4,5>,<6,5>,<8,5> \}$

$f_6 = \{ <0,5>,<2,5>,<4,5>,<6,5>,<8,5>,<10,5> \}$

$\vdots \qquad\qquad \vdots$

$f_k = \{ <0,5>,<2,5>,<4,5>,<6,5>,<8,5>,<10,5>,\ldots,<2 \bullet k-2,5> \}$

$\vdots \qquad\qquad \vdots$

Figure 10.11: Approximations to F_{fp}

Remember that the definition of the function $lub\{f_i \mid i \geq 0\}$ is given as the least upper bound of the f_i's on individual values of n,

$$(lub\{f_i \mid i \geq 0\})\ n = lub\{f_i(n) \mid i \geq 0\}.$$

The procedure for computing a least fixed point for a functional can be described as an operator on functions $F : D \rightarrow D$.

$$fix : (D \rightarrow D) \rightarrow D \text{ where}$$
$$fix = \lambda F\ .\ lub\{F^i(\perp) \mid i \geq 0\}.$$

The least fixed point of the functional

$$F = \lambda f\ .\ \lambda n\ .\ \text{if } n=0 \text{ then } 5 \text{ else if } n=1 \text{ then } f(n+2) \text{ else } f(n-2)$$

can then be expressed as $F_{fp} = fix\ F$ where $D = N \rightarrow N$.

For $F : (N \rightarrow N) \rightarrow (N \rightarrow N)$, *fix* has type $fix : ((N \rightarrow N) \rightarrow (N \rightarrow N)) \rightarrow (N \rightarrow N)$.

The fixed-point operator *fix* provides a fixed point for any continuous functional—namely, the least defined function with this fixed-point property.

Fixed-Point Identity : $F(fix\ F) = fix\ F$.

Summary : Recapping the fixed-point semantics of functions, we start with a recursive definition, say $fac : N \rightarrow N$, where

$$fac\ n = \text{if } n=0 \text{ then } 1 \text{ else } n \bullet fac(n-1) \text{ or}$$

$$fac = \lambda n\ .\ \text{if } n=0 \text{ then } 1 \text{ else } n \bullet fac(n-1)$$

Operationally, the meaning of fac on a value $n \in N$ results from unfolding the definition enough times, necessarily a finite number of times, until the basis case is reached. For example, we calculate fac(4) by the following process:

$$fac(4) = \text{if } 4=0 \text{ then } 1 \text{ else } 4 \bullet fac(3) = 4 \bullet fac(3)$$
$$= 4 \bullet (\text{if } 3=0 \text{ then } 1 \text{ else } 3 \bullet fac(2)) = 4 \bullet 3 \bullet fac(2)$$

$= 4 \bullet 3 \bullet (\text{if } 2=0 \text{ then } 1 \text{ else } 2 \bullet \text{fac}(1)) = 4 \bullet 3 \bullet 2 \bullet \text{fac}(1)$
$= 4 \bullet 3 \bullet 2 \bullet (\text{if } 1=0 \text{ then } 1 \text{ else } 1 \bullet \text{fac}(0)) = 4 \bullet 3 \bullet 2 \bullet 1 \bullet \text{fac}(0)$
$= 4 \bullet 3 \bullet 2 \bullet 1 \bullet 1 = 24$

The problem with providing a mathematical interpretation of this unfolding process is that we cannot predict ahead of time how many unfoldings of the definition are required. The idea of fixed-point semantics is to consider a corresponding (nonrecursive) functional

$\text{Fac} : (N \to N) \to (N \to N)$ where

$\text{Fac} = \lambda f \,.\, \lambda n \,.\, \text{if } n=0 \text{ then } 1 \text{ else } n \bullet f(n-1)$

and construct terms in the ascending chain

$$\bot \subseteq \text{Fac}(\bot) \subseteq \text{Fac}^2(\bot) \subseteq \text{Fac}^3(\bot) \subseteq \text{Fac}^4(\bot) \subseteq \ldots.$$

Using the abbreviations $\text{fac}_i = \text{Fac}^i(\bot)$ for $i \geq 0$, the chain can also be viewed as

$$\text{fac}_0 \subseteq \text{fac}_1 \subseteq \text{fac}_2 \subseteq \text{fac}_3 \subseteq \text{fac}_4 \subseteq \ldots.$$

A careful investigation of these "partial" functions $\text{fac}_i : N \to N$ reveals that

$$\text{fac}_0\ n = \bot$$

$$\begin{aligned} \text{fac}_i\ n &= \text{Fac}^i(\bot)\ n \\ &= \text{Fac}(\text{fac}_{i-1})\ n \\ &= \text{if } n<i \text{ then } n! \text{ else } \bot \quad \text{for } i \geq 1. \end{aligned}$$

The proof that this pattern is correct for the functions in the ascending chain is left as an exercise. It follows that any application of fac to a natural number can be handled by one of these nonrecursive approximating functions fac_i. For instance, $\text{fac}\ 4 = \text{fac}_5\ 4$, $\text{fac}\ 100 = \text{fac}_{101}\ 100$, and in general $\text{fac}\ m = \text{fac}_{m+1}\ m$.

The purpose of each approximating function $\text{fac}_i = \text{Fac}^i(\bot)$ is to embody any calculation of the factorial function that entails fewer than i unfoldings of the recursive definition. Fixed-point semantics gives the least upper bound of these approximating functions as the *meaning* of the original recursive definition of fac. The ascending chain, whose limit is the least upper bound, $lub\{\text{fac}_i \mid i \geq 0\} = lub\{\text{Fac}^i \bot \mid i \geq 0\}$, is made up of finite functions, each consistent with its predecessor in the chain, and having the property that any computation of fac can be obtained by one of the functions far enough out in the chain.

Continuous Functionals

To apply the theorem about the existence of a least fixed point to the functionals F as described in the previous examples, it must be established that these functionals are continuous.

Writing the conditional expression function if-then-else as a function if : TxNxN→N or alternatively taking a curried version if : T→N→N→N, these functionals take the form

F f n = if(n=0, 5, if(n=1, f(n+2), f(n–2))) uncurried if

Fac f n = (if n=0 1 n•f(n–1)) curried if

Since it has already been proved that the natural extension of an arbitrary function on elementary domains is continuous, parts of these definitions are known to be continuous—namely, the functions defined by the expressions "n=0", "n+2", and "n–1". Several lemmas will fill in the remaining properties needed to verify the continuity of these and other functionals.

Lemma: A constant function f : D→C, where f(x) = k for some fixed k∈C and for all x∈D, is continuous given either of the two extensions

1. The natural extension where $f(\perp_D) = \perp_C$.
2. The "unnatural" extension where $f(\perp_D) = k$.

Proof: Part 1 follows by a proof similar to the one for the earlier theorem about the continuity of natural extensions, and part 2 is left as an exercise at the end of this section. ∎

Lemma: An identity function f : D→D, where f(x) = x for all x in a domain D, is continuous.

Proof: If $x_1 \subseteq x_2 \subseteq x_3 \subseteq \ldots$ is an ascending chain in D, it follows that $f(lub\{x_i \mid i \geq 1\}) = lub\{x_i \mid i \geq 1\} = lub\{f(x_i) \mid i \geq 1\}$. ∎

In defining the meaning of the conditional expression function,

if(a,b,c) = if a then b else c.

where if : TxDxD→D for some domain D and T = {⊥,true,false},

the natural extension is considered too restrictive. The preferred approach is to define this function by

(if true then b else c) = b for any b,c∈D
(if false then b else c) = c for any b,c∈D
(if ⊥ then b else c) = $\perp_D$ for any b,c∈D

Note that this is not a natural extension. It allows an undefined or "erroneous" expression in one branch of a conditional as long as that branch is avoided when the expression is undefined. For example, h(*nil*) is defined for the function

h(L) = if L≠*nil* then *head*(L) else *nil*.

Lemma: The uncurried "if" function as defined above is continuous.

Proof: Let $<t_1,b_1,c_1> \subseteq <t_2,b_2,c_2> \subseteq <t_3,b_3,c_3> \subseteq \ldots$ be an ascending chain in TxDxD. Three cases need to be considered:

Case 1: $t_i = \perp_T$ for all $i \geq 1$.

Case 2: t_i = true for all $i \geq k$, for some fixed k.

Case 3: t_i = false for all $i \geq k$, for some fixed k.

The details of this proof are left as an exercise. ∎

Lemma: A generalized composition of continuous functions is continuous—namely, if $f : C_1 x C_2 x \ldots x C_n \rightarrow C$ is continuous and $g_i : D_i \rightarrow C_i$ is continuous for each i, $1 \leq i \leq n$, then $f \circ (g_1,g_2,\ldots,g_n) : D_1 x D_2 x \ldots x D_n \rightarrow C$, defined by $f \circ (g_1,g_2,\ldots,g_n) <x_1,x_2,\ldots,x_n> = f <g_1(x_1),g_2(x_2),\ldots,g_n(x_n)>$ is also continuous.

Proof: This is a straightforward application of the definition of continuity and is left as an exercise. ∎

The previous lemmas apply to functions on any domains. When considering the continuity of functionals, say

$F : (D \rightarrow D) \rightarrow (D \rightarrow D)$ for some domain D

where F is defined by a rule of the form

F f d = some expression in f and d,

a composition will probably involve the "independent" variable f—for example, in a functional such as

$F : (N \rightarrow N) \rightarrow (N \rightarrow N)$ where

F f n = n + (if n=0 then 0 else f(f(n-1))).

Lemma: If $F_1, F_2, \ldots, F_n$ are continuous functionals, say $F_i : (D^n \rightarrow D) \rightarrow (D^n \rightarrow D)$ for each i, $1 \leq i \leq n$, the functional $F : (D^n \rightarrow D) \rightarrow (D^n \rightarrow D)$ defined by $F\ f\ d = f <F_1\ f\ d, F_2\ f\ d, \ldots, F_n\ f\ d>$ for all $f \in D^n \rightarrow D$ and $d \in D^n$ is also continuous.

Proof: Consider the case where n=1.
So $F_1 : (D \rightarrow D) \rightarrow (D \rightarrow D)$, $F : (D \rightarrow D) \rightarrow (D \rightarrow D)$, and $F\ f\ d = f <F_1\ f\ d>$ for all $f \in D \rightarrow D$ and $d \in D$. Let $f_1 \subseteq f_2 \subseteq f_3 \subseteq \ldots$ be an ascending chain in $D \rightarrow D$. The proof shows that $lub\{F(f_i) \mid i \geq 1\} = F(lub\{f_i \mid i \geq 1\})$ in two parts.

Part 1: $lub\{F(f_i) \mid i \geq 1\} \subseteq F(lub\{f_i \mid i \geq 1\})$. For each $i \geq 1$, $f_i \subseteq lub\{f_i \mid i \geq 1\}$. Since F_1 is monotonic, $F_1(f_i) \subseteq F_1(lub\{f_i \mid i \geq 1\})$, which means that $F_1\ f_i\ d \subseteq F_1\ lub\{f_i \mid i \geq 1\}\ d$ for each $d \in D$.

Since f_i is monotonic, $f_i <F_1\ f_i\ d> \subseteq f_i <F_1\ lub\{f_i \mid i \geq 1\}\ d>$. But $F\ f_i\ d = f_i <F_1\ f_i\ d>$ and $f_i <F_1\ lub\{f_i \mid i \geq 1\}\ d> \subseteq lub\{f_i \mid i \geq 1\} <F_1\ lub\{f_i \mid i \geq 1\}\ d>$. Therefore, $F\ f_i\ d \subseteq lub\{f_i \mid i \geq 1\} <(F_1 lub\{f_i \mid i \geq 1\}\ d>$ for each $i \geq 1$ and $d \in D$. So by the Lub lemma, $lub\{F(f_i) \mid i \geq 1\}\ d = lub\{F\ f_i\ d \mid i \geq 1\} \subseteq lub\{f_i \mid i \geq 1\} <F_1\ lub\{f_i \mid i \geq 1\}\ d> = F\ lub\{f_i \mid i \geq 1\}\ d$ for each $d \in D$.

Part 2 : $F(lub\{f_i \mid i\geq 1\}) \subseteq lub\{F(f_i) \mid i\geq 1\}$.
For any $d\in D$,
$F\ lub\{f_i \mid i\geq 1\}\ d = lub\{f_i \mid i\geq 1\} <F_1\ lub\{f_j\}\ d>$ by the definition of F
$= lub\{f_i \mid i\geq 1\} <lub\{F_1(f_j)\}\ d>$ since F_1 is continuous
$= lub\{lub\{f_i \mid i\geq 1\} <\{F_1(f_j)\}\ d>\}$ since $lub\{f_i \mid i\geq 1\}$ is continuous
$= lub\{lub\{f_i <\{F_1(f_j)\}\ d> \mid i\geq 1\}\}$ by the definition of $lub\{f_i \mid i\geq 1\}$. †

If $j\leq i$, then $f_j \subseteq f_i$, $F_1\ f_j \subseteq F_1\ f_i$ since F_1 is monotonic, $F_1\ f_j\ d \subseteq F_1\ f_i\ d$ for each $d\in D$, and $f_i <F_1\ f_j\ d> \subseteq f_i <F_1\ f_i\ d>$ since f_i is monotonic.

If $i<j$, then $f_i \subseteq f_j$ and $f_i <F_1\ f_j\ d> \subseteq f_j <F_1\ f_j\ d>$ for each $d\in D$ by the meaning of $\subseteq$.

Therefore $f_i <F_1\ f_j\ d> \subseteq lub\{f_n <F_1\ f_n\ d> \mid n\geq 1\}$ for each $i,j\geq 1$.
But $lub\{f_n <F_1\ f_n\ d> \mid i\geq 1\} = lub\{F\ f_n\ d \mid i\geq 1\} = lub\{F(f_n) \mid i\geq 1\}\ d$ by the definition of F. So $f_i <F_1\ f_j\ d> \subseteq lub\{F(f_n) \mid n\geq 1\}\ d$ for each $i,j\geq 1$,
and $lub\{f_i <F_1\ f_j\ d> \mid i\geq 1\} \subseteq lub\{F(f_n) \mid n\geq 1\}\ d$ for each $j\geq 1$.
Hence $lub\{lub\{f_i <F_1\ f_j\ d> \mid i\geq 1\} \mid j\geq 1\} \subseteq lub\{F(f_n) \mid n\geq 1\}\ d$.
Combining with † gives $F(lub\{f_i \mid i\geq 1\})\ d \subseteq lub\{F(f_n) \mid n\geq 1\}\ d$. ∎

Continuity Theor em: Any functional H defined by the composition of naturally extended functions on elementary domains, constant functions, the identity function, the if-then-else conditional expression, and a function parameter f, is continuous.

Proof: The proof follows by structural induction on the form of the definition of the functional. The basis is handled by the continuity of natural extensions, constant functions, and the identity function, and the induction step relies on the previous lemmas, which state that the composition of continuous functions, possibly involving f, is continuous. The details are left as an exercise. ∎

Example 14 : Before proceeding, we work out the least fixed point of another functional by constructing approximating terms in the ascending chain.

$H : (N\to N)\to(N\to N)$ where

$H\ h\ n = n + (\text{if } n=0 \text{ then } 0 \text{ else } h(h(n-1)))$
$= \text{if } n=0 \text{ then } n \text{ else } n+h(h(n-1))$.

Consider the ascending chain $h_0 \subseteq h_1 \subseteq h_2 \subseteq h_3 \subseteq \ldots$ where $h_0\ n = H^0\ \perp\ n = \perp(n)$ and $h_i\ n = H^i\ \perp\ n = H\ h_{i-1}\ n$ for $i\geq 1$. Calculate terms of this sequence until a pattern becomes apparent.

$h_0(n) = \perp(n) = \perp$

$h_1(n) = H\ h_0\ n = H\ \perp\ n$
$= \text{if } n=0 \text{ then } n \text{ else } n+h_0(h_0(n-1))$
$= \text{if } n=0 \text{ then } n \text{ else } n+\perp(\perp(n-1))$
$= \text{if } n=0 \text{ then } 0 \text{ else } \perp$

Note that the natural extension of + is strict in ⊥.

$h_2(n) = H\ h_1\ n$
$= \text{if } n=0 \text{ then } 0 \text{ else } n+h_1(h_1(n-1))$
$= \text{if } n=0 \text{ then } 0 \text{ else } n+h_1(\text{if } n-1=0 \text{ then } 0 \text{ else } \bot)$
$= \text{if } n=0 \text{ then } 0 \text{ else } n+h_1(\text{if } n=1 \text{ then } 0 \text{ else } \bot)$
$= \text{if } n=0 \text{ then } 0 \text{ else } n+\text{if } n=1 \text{ then } h_1(0) \text{ else } h_1(\bot)$
$= \text{if } n=0 \text{ then } 0 \text{ else if } n=1 \text{ then } n+0 \text{ else } n+\bot$
$= \text{if } n=0 \text{ then } 0 \text{ else if } n=1 \text{ then } 1 \text{ else } \bot$

$h_3(n) = H\ h_2\ n$
$= \text{if } n=0 \text{ then } 0 \text{ else } n+h_2(h_2(n-1))$
$= \text{if } n=0 \text{ then } 0$
$\quad \text{else } n+h_2(\text{if } n-1=0 \text{ then } 0 \text{ else if } n-1=1 \text{ then } 1 \text{ else } \bot)$
$= \text{if } n=0 \text{ then } 0$
$\quad \text{else } n+h_2(\text{if } n=1 \text{ then } 0 \text{ else if } n=2 \text{ then } 1 \text{ else } \bot)$
$= \text{if } n=0 \text{ then } 0$
$\quad \text{else if } n=1 \text{ then } 1+h_2(0)$
$\quad\quad \text{else if } n=2 \text{ then } 2+h_2(1) \text{ else } n+h_2(\bot)$
$= \text{if } n=0 \text{ then } 0$
$\quad \text{else if } n=1 \text{ then } 1$
$\quad\quad \text{else if } n=2 \text{ then } 3 \text{ else } \bot$

$h_4(n) = H\ h_3\ n$
$= \text{if } n=0 \text{ then } 0 \text{ else } n+h_3(h_3(n-1))$
$= \text{if } n=0 \text{ then } 0$
$\quad \text{else } n+h_3(\text{if } n-1=0 \text{ then } 0$
$\quad\quad \text{else if } n-1=1 \text{ then } 1$
$\quad\quad\quad \text{else if } n-1=2 \text{ then } 3 \text{ else } \bot)$
$= \text{if } n=0 \text{ then } 0$
$\quad \text{else } n+h_3(\text{if } n=1 \text{ then } 0$
$\quad\quad \text{else if } n=2 \text{ then } 1$
$\quad\quad\quad \text{else if } n=3 \text{ then } 3 \text{ else } \bot)$
$= \text{if } n=0 \text{ then } 0$
$\quad \text{else if } n=1 \text{ then } 1+h_3(0)$
$\quad\quad \text{else if } n=2 \text{ then } 2+h_3(1)$
$\quad\quad\quad \text{else if } n=3 \text{ then } 3+h_3(3) \text{ else } n+h_3(\bot)$
$= \text{if } n=0 \text{ then } 0$
$\quad \text{else if } n=1 \text{ then } 1$
$\quad\quad \text{else if } n=2 \text{ then } 3$
$\quad\quad\quad \text{else if } n=3 \text{ then } \bot \text{ else } \bot$
$= h_3(n)$

Therefore $h_k(n) = h_3(n)$ for each $k \geq 3$, and the least fixed point is $lub\{h_k \mid k \geq 0\} = h_3$. Note that the last derivation shows that $H\ h_3 = h_3$. ∎

Fixed points for Nonrecursive Functions

Consider the function $g(n) = n^2 - 6n$ defined on the natural numbers N. The function g allows two interpretations in the context of fixed-point theory.

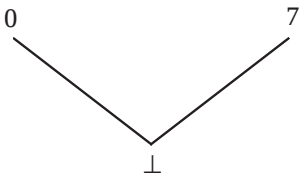
First Interpr etation: The natural extension $g^+: N^+ \to N^+$ of g is a continuous function on the elementary domain $N^+ = N \cup \{\bot\}$. Then the least fixed point of g^+, which will be an element of N^+, may be constructed as the least upper bound of the ascending sequence

$$\bot \subseteq g^+(\bot) \subseteq g^+(g^+(\bot)) \subseteq g^+(g^+(g^+(\bot))) \subseteq \ldots.$$

But $g^+(\bot) = \bot$, and if $(g^+)^{k-1}(\bot) = \bot$, then $(g^+)^k(\bot) = g^+((g^+)^{k-1}(\bot)) = g^+(\bot) = \bot$. So by induction $(g^+)^k(\bot) = \bot$ for any $k \geq 1$.

Therefore $lub\{(g^+)^k(\bot) \mid k \geq 0\} = lub\{\bot \mid k \geq 0\} = \bot$ is the least fixed point.

In fact, g^+ has three fixed points in $N \cup \{\bot\}$: $g^+(0) = 0$, $g^+(7) = 7$, and $g^+(\bot) = \bot$.



Second Interpr etation: Think of $g(n) = n^2 - 6n$ as a rule defining a "recursive" function that just has no actual recursive call of g.

The corresponding functional $G : (N \to N) \to (N \to N)$ is defined by the rule

$$G\ g\ n = n^2 - 6n.$$

A function g satisfies the definition $g(n) = n^2 - 6n$ if and only if it is a fixed point of G—that is, $G\ g = g$.

The fixed point construction proceeds as follows:

$$G^0\ \bot\ n = \bot(n) = \bot$$
$$G^1\ \bot\ n = n^2 - 6n$$
$$G^2\ \bot\ n = n^2 - 6n$$
$$\vdots$$
$$G^k\ \bot\ n = n^2 - 6n$$
$$\vdots$$

Therefore the least fixed point is $lub\{G^k(\bot) \mid k \geq 0\} = \lambda n\ .\ n^2 - 6n$, which follows the same definition rule as the original function g.

In the first interpretation we computed the least fixed point of the original function g, while in the second we obtained the least fixed point of a functional related to g. These two examples show that the least fixed point construction can be applied to any continuous function, although its importance comes from giving a consistent semantics to functions specified by actual recursive definitions.

Revisiting Denotational Semantics

In Chapter 9 we were tempted to define the meaning of a **while** command in Wren recursively with the semantic equation

execute ⟦**while** E **do** C⟧ sto =
    if *evaluate* ⟦E⟧ sto = *bool*(true)
        then *execute* ⟦**while** E **do** C⟧(*execute* ⟦C⟧ sto) else sto.

But this approach violates the principle of compositionality that states that the meaning of any syntactic phrase may be defined only in terms of the meanings of its proper subparts. This circular definition disobeys the principle, since the meaning of *execute* ⟦**while** E **do** C⟧ is defined in terms of itself.

Now we can solve this problem by using a fixed-point operator in the definition of the **while** command. The function *execute* ⟦**while** E **do** C⟧ satisfies the recursive definition above if and only if it is a fixed point of the functional

W = λf . λs . if *evaluate* ⟦E⟧ s = *bool*(true) then f(*execute* ⟦C⟧ s) else s
  = λf . λs . if *evaluate* ⟦E⟧ s = *bool*(true) then (f∘*execute* ⟦C⟧) s else s.

Therefore we obtain a nonrecursive and compositional definition of the meaning of a **while** command by means of

execute ⟦**while** E **do** C⟧ = *fix* W.

We gain insight into both the **while** command and fixed-point semantics by constructing a few terms in the ascending chain whose least upper bound is *fix* W,

$W^0 \perp \subseteq W^1 \perp \subseteq W^2 \perp \subseteq W^3 \perp \subseteq \ldots$ where *fix* W = *lub*$\{W^i(\perp) \mid i \geq 0\}$.

The fixed-point construction for W proceeds as follows:

$W^0(\perp)$ = λs . ⊥

$W^1(\perp)$ = W(W^0 ⊥)
  = λs . if *evaluate* ⟦E⟧ s = *bool*(true) then ⊥(*execute* ⟦C⟧ s) else s
  = λs . if *evaluate* ⟦E⟧ s = *bool*(true) then ⊥ else s

Let exC stand for the function *execute* ⟦C⟧ and continue the construction.

$W^2(\perp)$ = W(W^1 ⊥)
  = λs . if *evaluate* ⟦E⟧ s = *bool*(true) then W^1 ⊥ (exC s) else s
  = λs . if *evaluate* ⟦E⟧ s = *bool*(true)
        then (if *evaluate* ⟦E⟧ (exC s) = *bool*(true)
                then ⊥ else exC s)
        else s

$W^3(\perp) = W(W^2\ \perp)$

$= \lambda s$. if *evaluate* ⟦E⟧ s = *bool*(true) then $W^2\ \perp$ (exC s) else s

$= \lambda s$. if *evaluate* ⟦E⟧ s = *bool*(true)
then (if *evaluate* ⟦E⟧ (exC s) = *bool*(true)
then (if *evaluate* ⟦E⟧ (exC (exC s)) = *bool*(true)
then ⊥ else exC (exC s))
else (exC s))
else s

$= \lambda s$. if *evaluate* ⟦E⟧ s = *bool*(true)
then (if *evaluate* ⟦E⟧ (exC s) = *bool*(true)
then (if *evaluate* ⟦E⟧ (exC^2 s) = *bool*(true)
then ⊥ else (exC^2 s))
else (exC s))
else s

$W^4(\perp) = \lambda s$. if *evaluate* ⟦E⟧ s = *bool*(true)
then (if *evaluate* ⟦E⟧ (exC s) = *bool*(true)
then (if *evaluate* ⟦E⟧ (exC^2 s) = *bool*(true)
then (if *evaluate* ⟦E⟧ (exC^3 s) = *bool*(true)
then ⊥ else (exC^3 s))
else (exC^2 s))
else (exC s))
else s

In general,

$W^{k+1}(\perp) = W(W^k\ \perp)$

$= \lambda s$. if *evaluate* ⟦E⟧ s = *bool*(true)
then (if *evaluate* ⟦E⟧ (exC s) = *bool*(true)
then (if *evaluate* ⟦E⟧ (exC^2 s) = *bool*(true)
then (if *evaluate* ⟦E⟧ (exC^3 s) = *bool*(true)
:
then (if *evaluate* ⟦E⟧ (exC^k s) = *bool*(true)
then ⊥ else (exC^k s))
else (exC^{k-1} s))
:
else (exC^2 s))
else (exC s))
else s

The function $W^{k+1}(\perp)$ allows the body C of the **while** to be executed up to k times, which means that this approximation to the meaning of the **while** command can handle any instance of a **while** with at most k iterations of the body. Any application of a **while** command will have some finite number of iterations, say n. Therefore its meaning is subsumed in the approximation $W^{n+1}(\perp)$. The least upper bound of this ascending sequence provides seman-

tics for the **while** command: *execute* ⟦**while** E **do** C⟧ = *fix* W = *lub*{$W^i(\perp) \mid i \geq 0$}. Unlike previous examples of fixed-point constructions, we cannot derive a closed form representation of the least fixed point because of the complexity of the definition.

Another way to view the definition of *execute* ⟦**while** E **do** C⟧ is in terms of the fixed-point identity, W(*fix* W) = *fix* W, where

$$W = \lambda f \,.\, \lambda s \,.\, \text{if } \mathit{evaluate}\ [\![E]\!]\ s = \mathit{bool}(\text{true}) \text{ then } f(\mathit{execute}\ [\![C]\!]\ s) \text{ else } s.$$

In this context, *execute* ⟦**while** E **do** C⟧ = *fix* W. Now define loop = *fix* W.

Then

execute ⟦**while** E **do** C⟧ = loop
where loop s = (W loop) s
= loop
where loop s = if *evaluate* ⟦E⟧ s = *bool*(true)
then loop(*execute* ⟦C⟧ s) else s.

The local function "loop" is the least fixed point of W. Following this approach produces the compositional definition of *execute* ⟦**while** E **do** C⟧ that we used in our specification of Wren in Figure 9.11.

Fixed-Point Induction

Since recursively defined functions get their meaning from the least fixed-point construction, properties of these functions can be established by means of induction on the construction of the least fixed point *lub*{$F^i(\perp) \mid i \geq 0$}. For instance, alternate definitions and properties of "closed form" definitions can frequently be proved using fixed-point induction.

Let $\Phi(f)$ be a predicate that describes a property for an arbitrary function f defined recursively. To show Φ holds for the least fixed point F_{fp} of the functional F corresponding to a recursive definition of f, two conditions are needed.

Part 1 : Show by induction that Φ holds for each element in the ascending chain

$$\perp \subseteq F(\perp) \subseteq F^2(\perp) \subseteq F^3(\perp) \subseteq \ldots.$$

Part 2 : Show that Φ remains true when the least upper bound is taken.

Part 2 is handled by defining a class of predicates with the necessary property, the so-called admissible predicates.

Definition : A predicate is called **admissible** if it has the property that whenever the predicate holds for each term in an ascending chain of functions, it also must hold for the least upper bound of that chain. ∎

Theorem: Any finite conjunction of inequalities of the form $\alpha(F) \subseteq \beta(F)$, where α and β are continuous functionals, is an admissible predicate. This includes terms of the form $\alpha(F) = \beta(F)$.

Proof: The proof of this theorem is beyond the scope of this text. See the further readings at the end of the chapter. ∎

Mathematical induction is used to verify the condition in Part 1.

Given a functional $F : (D \to D) \to (D \to D)$ for some domain D and an admissible predicate $\Phi(f)$, show the following properties:

(a) $\Phi(\perp)$ holds where $\perp : D \to D$.

(b) For any $i \geq 0$, if $\Phi(F^i(\perp))$, then $\Phi(F^{i+1}(\perp))$.

An alternate version of condition (b) is

(b') For any $f : D \to D$, if $\Phi(f)$, then $\Phi(F(f))$.

Either formulation is sufficient to infer that the predicate Φ holds for every function in the ascending chain $\{F^i(\perp) \mid i \geq 0\}$.

We illustrate fixed-point induction with a simple example.

Example 15 : Let $f : N \to N$ be defined by f(n) = if n=0 then 1 else $3n^2$-n+f(n-1). Prove that $f \subseteq \lambda n \,.\, n^3+n^2$. The recursively defined function f corresponds to the functional $F : (N \to N) \to (N \to N)$ given by

F f n = if n=0 then 0 else $3n^2$-n+f(n-1).

Let $\Phi(f)$ be the predicate $f \subseteq \lambda n \,.\, n^3+n^2$.

(a) Since $\perp \subseteq \lambda n \,.\, n^3+n^2$, $\Phi(\perp)$ holds.

(b') Suppose $\Phi(f)$—that is, $f \subseteq \lambda n \,.\, n^3+n^2$.

Then F f n = if n=0 then 0 else $3n^2$-n+f(n-1)

$\subseteq$ if n=0 then 0 else $3n^2$-n+$(n-1)^3$+$(n-1)^2$

= if n=0 then 0 else $3n^2$-n+n^3-$3n^2$+3n-1+n^2-2n+1

= if n=0 then 0 else $n^3+n^2 = n^3+n^2$ for $n \geq 0$. ∎

A property proved by fixed-point induction may involve two functions, say $\Phi(f,g)$. Then satisfying the hypothesis (Part 1) for induction involves the following two steps:

(a) $\Phi(\perp,\perp)$.

(b') For any f and g given by functionals F and G, $\Phi(f,g)$ implies $\Phi(F(f),G(g))$.

Example 16 : A recursive definition of a function is called **tail recursive** if each recursive call of the function is the last computation performed by the function. For example, the factorial function can be defined recursively by

fac n = if n=0 then 1 else n•fac(n-1)

or it can be given a tail recursive definition using

tailfac (n,p) = if n=0 then p else tailfac(n-1,n•p)

where the factorial of n results from the call, tailfac(n,1).

The correctness of the tail-recursive approach can be verified by fixed-point induction. The functionals that correspond to these two recursive definitions have the form

F : (N→N)→(N→N), where F f n = if n=0 then 1 else n•f(n-1)

and

G : (NxN→N)→(NxN→N), where G g (n,p) = if n=0 then p else g(n-1,n•p).

We want to prove that F_{fp} n = G_{fp} (n,1) for all n∈N. The result follows from a stronger assertion—namely, that p•F_{fp}(n) = G_{fp} (n,p) for all n,p∈N.

Let Φ(f,g) be the predicate "p•f(n) = g(n,p) for all n,p∈N".

(a) Since f_0 n= ⊥ = g_0(n,p) for all n,p∈N, Φ(f_0,g_0) holds.

(b) Suppose Φ(f_i,g_i)—that is, p•f_i(n) = g_i(n,p) for all n,p∈N. Note that for some values of n, both sides of this equation are ⊥.

Then g_{i+1}(n,p) = G g_i (n,p)
= if n=0 then p else g_i(n-1,n•p)
= if n=0 then p else n•p•f_i(n-1) (induction hypothesis)
= p•(if n=0 then 1 else n•f_i(n-1))
= p•f_{i+1}(n).

Therefore by fixed-point induction Φ(F_{fp},G_{fp}) holds—that is,

p•F_{fp}(n) = G_{fp} (n,p) for all n∈N.

The verification of fac n = tailfac(n,1) follows taking p=1, since fac is F_{fp} and tailfac is G_{fp}. ∎

The property p•fac n = tailfac(n,p) can be verified using normal mathematical induction on n as well.

Exercises

1. Show that the converse of the theorem about natural extensions is not true—namely,

False Theor em: Let g be an extension of a function between two sets D and C so that g is a total function from D^+ to C^+. If g is monotonic and continuous, then g is the natural extension of f.

2. Use the construction of the functions h_i as in the example in this section to find the least fixed point for these functionals. State the recursive definitions that give rise to these functionals.

   a) H f n = if n=0 then 3 else f(n+1)

   b) H f n = if n=0 then 0 else (2n–1)+f(n–1)

3. Prove by induction that the approximating functions for the recursive definition

   fac n = if n=0 then 1 else n•fac(n-1)

   have the form

   fac_0 n = ⊥

   fac_i n = if n<i then n! else ⊥ for i≥1.

4. Prove that the "unnatural" extension of a constant function is continuous.

5. Complete the proof that if : T×N×N→N is continuous and also show that if : T→N→N→N is continuous.

6. Prove that a generalized composition of continuous functions is continuous.

7. Find a simple (nonrecursive) definition for each of these functions in N→N using a fixed-point construction.

   a) g(n) = if n>0 then 2+g(n–1) else 0

   b) h(n) = if n=0 then 0 else if n=1 then h(n+1)–1 else h(n–1)+1

   c) f(n) = if n=0 then 0 else if n=1 then f(n-1)+1 else n^2

   d) g(n) = if n=0 then 1 else 2n+g(n–1)

   e) h(n) = if n=0 then 1 else if n=1 then 2 else 4n–4+h(n–2)

   f) f(n) = if n=0 then f(n+1)+1 else 1

   g) f(n) = if n>100 then n–10 else f(f(n+11)) (McCarthy's 91 function)

8. Consider the following functional defined on functions over the natural numbers:

   G : (N→N)→(N→N)

   G = λg . λn . if n=0 then 2 else g(n)

a) Give and justify a recursive definition that corresponds to this functional—that is, an operational definition of a function that will be a fixed point of G.

b) Define four different functions, g_0, g_1, g_2, and g_3, that are fixed points of the functional G, including the least fixed point, g_0. Carefully prove that g_0 and g_1 are fixed points of G.

c) Draw a diagram showing the relationship "is less defined than or equal" between these four functions.

d) Informally describe the operational behavior of the recursive definition in part a). Which of the four fixed-point functions has the closest behavior to the operational view?

9. Let T = {⊥,true,false} be the elementary domain of Boolean values with the bottom element ⊥. The function *and* : T x T → T must agree with the following truth table:

and	true	false	⊥
true	true	false	?
false	false	false	?
⊥	?	?	?

Complete this truth-table in *two* ways to produce two different monotonic versions of the function *and* defined on T**.** Explain how these two *and* functions correspond to the possible interpretations of the predefined Boolean **and** function in a programming language such as Pascal.

10. Prove the Continuity Theorem:

Any functional H defined by the composition of naturally extended functions on elementary domains, constant functions, the identity function, the if-then-else conditional expression, and a function variable f, is continuous.

11. Use fixed-point induction to prove the equality of the following functions in N→N:

f(n) = if n>5 then n-5 else f(f(n+13))

g(n) = if n>5 then n-5 else g(n+8)

12. Use fixed point-induction to prove the equality of the following functions in NxN→N:

f(m,n) = if m=0 then n else f(2•m,n)+3

g(m,n) = if m=0 then n else g(2•m,n+3)

Hint: Let Φ(f,g) be ∀m>0∀n[f(m,n)=g(m,n) and g(m,n+3)=g(m,n)+3].

13. Let $f : N \to N$ be a function defined by a recursive rule of the form

$$f(n) = \text{if } p(n) \text{ then } n \text{ else } f(f(h(n))),$$

where $p:N \to T$ and $h:N \to N$ are two arbitrary functions.

Use fixed-point induction to show that $f \circ f = f$ (f is idempotent). *Hint*: Let $\Phi(g)$ be "$f(g(n)) = g(n)$ for all $n \in N$".

14. Let D be the set of natural numbers. Prove that the fixed-point operator

$$fix : (D \to D) \to D \text{ where}$$
$$fix = \lambda F \,.\, lub\{F^i(\perp) \mid i \geq 0\}$$

is monotonic and continuous.

15. Let N be the domain of natural numbers. The set of finite lists of natural numbers can be specified by the recursive definition $L = \{nil\} \cup (\mathbf{N} \times L)$, where *nil* is a special constant symbol. One way to give meaning to such a recursive definition is to take L to be the least fixed point of the function $F(X) = \{nil\} \cup (\mathbf{N} \times X)$—namely, $L = F(L)$.

a) Define and prove those properties that F must satisfy to guarantee the existence of a least fixed point.

b) Carefully describe the first four terms in the ascending chain that is used in constructing the least fixed point for F.

16. (Thanks to Art Fleck at the University of Iowa for this problem.) Context-free grammars can be viewed as systems of equations where the nonterminals are regarded as variables (or unknowns) over sets of strings; the solution for the start symbol yields the language to be defined. In general, such an equation system has solutions that are tuples of sets, one for each nonterminal. Such solutions can be regarded as fixed points in that when they are substituted in the right-hand side, the result is precisely the solution again. For example (using ε for the null string), the grammar

$$A ::= aAc \mid B$$
$$B ::= bB \mid C$$
$$C ::= \varepsilon \mid C$$

corresponds to the transformation on triples $\langle X,Y,Z \rangle$ of sets defined by

$$f(\langle X,Y,Z \rangle) = \langle \{a\} \bullet X \bullet \{c\} \cup Y,\ \{b\} \bullet Y \cup Z,\ \{\varepsilon\} \cup Z \rangle,$$

whose fixed point $\langle A, B, C \rangle$ then satisfies the set equations

$$A = \{a\} \bullet A \bullet \{c\} \cup B$$
$$B = \{b\} \bullet B \cup C$$
$$C = \{\varepsilon\} \cup C$$

for appropriate $A,B,C \subseteq \{a,b,c\}^*$. For instance, the equations above are satisfied by the sets $A = \{a^n\ b^*\ c^n \mid n \geq 0\}$, $B = b^*$, $C = \{\varepsilon\}$.

Show that the equation system corresponding to the grammar above has more than one possible solution so that simply seeking an arbitrary solution is insufficient for formal language purposes. However, the *least* fixed point solution provides exactly the language normally defined by the grammar. Illustrate how the first few steps of the ascending chain in the fixed-point construction lead to the desired language elements for the grammar above, and discuss the connection with derivations in the grammar.

Note: We have the natural partial order for tuples of sets where $\langle S_1, \ldots, S_k \rangle \subseteq \langle T_1, \ldots, T_k \rangle$ if $S_i \subseteq T_i$ for all i, $1 \leq i \leq k$.

17. Prove Park's Induction Principle: If $f : D \rightarrow D$ is a continuous function on a domain D and $d \in D$ such that $f\ d \subseteq d$, it follows that $\mathit{fix}\ f \subseteq d$.

18. Let A and B be two domains with functions $f : A \rightarrow B$ and $g : B \rightarrow A$. Prove that $\mathit{fix}\ (f \circ g) = f(\mathit{fix}\ (g \circ f))$.

10.4 LABORATORY: RECURSION IN THE LAMBDA CALCULUS

Before we implement a fixed-point finder to provide recursively defined functions in the lambda calculus evaluator presented in Chapter 5, we describe how a definition mechanism, like a macro system, can be added to the evaluator. An example showing the definition and use of symbols follows.

```
>>> Evaluating the Lambda Calculus <<<
Enter name of source file: cube
    define Thrice = (L f x (f (f (f x))))
    define Sqr = (L x (mul x x))
    define Cube = (L x (mul x (Sqr x)))
    (Thrice Cube 2)
Successful Scan
Successful Parse
Result =  134217728
yes
```

Without the capability of forming definitions, the lambda expressions that we want to evaluate get extremely large. Now the file submitted to the evaluator will contain zero or more definitions followed by one lambda expression to be evaluated. Symbols defined in earlier lines may be used in later definitions.

The system maintains definitions of new symbols in a definition table Tab using predicates extendTab and applyTab in the same with way that environ-

ments are handled with the SECD machine in Chapter 8 and Pelican in Chapter 9. Processing the definitions decomposes into two parts: (1) elaboration and (2) expansion. As the list of definitions is processed, the right side of each definition must be expanded and a new binding added to the table.

```
elaborate(Tab,[def(X,E)|Defns],NewTab) :- expand(E,Tab,[ ],NewE),
                                          extendTab(Tab,X,NewE,TempTab),
                                          elaborate(TempTab,Defns,NewTab).
elaborate(Tab,[ ],Tab).
```

The expansion mechanism keeps track of the variable occurrences that have been bound since only free occurrences of symbols are replaced. Moving inside of an abstraction appends the lambda variable to the set of bound variables BV.

```
expand(var(X),Tab,BV,var(X)) :- member(X,BV).      % X is bound
expand(var(X),Tab,BV,E) :- applyTab(Tab,X,E).      % X is free and defined
expand(var(X),Tab,BV,var(X)).                      % X is a free variable
expand(con(C),Tab,BV,con(C)).                      % C is a constant
expand(comb(Rator,Rand),Tab,BV,comb(NewRator,NewRand)) :-
    expand(Rator,Tab,BV,NewRator), expand(Rand,Tab,BV,NewRand).
expand(lamb(X,E),Tab,BV,lamb(X,NewE)) :-
    concat(BV,[X],NewBV), expand(E,Tab,NewBV,NewE).
```

The definition table is manipulated by two predicates. We add a binding Ide $\mapsto$Exp to the definition table Tab using extendTab.

```
extendTab(Tab,Ide,Exp,tab(Ide,Exp,Tab)).
```

We look up an identifier Ide in the definition table Tab using applyTab, which fails if the identifier is not found.

```
applyTab(tab(Ide,Exp,Tab),Ide,Exp).
applyTab(tab(Ide1,Exp1,Tab),Ide,Exp) :- applyTab(Tab,Ide,Exp).
```

The scanner must be altered to recognize the reserved word define and the equal symbol. The parser then produces a list of definitions of the form def(X,E) together with the lambda expression to be evaluated. The definitions are elaborated starting with an empty table nil, the lambda expression is expanded, and then the new expression can be evaluated.

```
go  :- nl,write('>>> Evaluating the Lambda Calculus <<<'), nl, nl,
       write('Enter name of source file:  '), nl, readfile(File), nl,
       see(File), scan(Tokens), nl, write('Successful Scan'), nl, !,
       seen, program(prog(D,E),Tokens,[eop]), write('Successful Parse'), nl, !,
       elaborate(nil,D,Tab), expand(E,Tab,[ ],Expr), !,
       evaluate(Expr,Result), nl, write('Result =  '), pp(Result),nl.
```

Conditional Expressions

Recursive definitions require some way of choosing between the basis case and the recursive case. An expression-oriented language such as the lambda calculus (or a functional programming language) uses a conditional expression

(if e_1 e_2 e_3) = if e_1 then e_2 else e_3.

Recall that function application is left associative so that the abstract syntax tree for this expression takes the form

comb(comb(comb(con(if),e1),e2),e3)

where if has been added as another constant to the applied lambda calculus. To see how to evaluate comb(con(if),e1), consider the behavior that we expect when the value of e1 is true or false.

Case 1: e1 evaluates to true. We want the value of comb(con(if),e1) to be a selector function that takes the next value e2 and ignores the value e3 after it. Therefore take the value of comb(con(if),e1) to be the parsed lambda expression lamb(x,lamb(y, var(x)). Then

comb(comb(comb(con(if),true),e2),e3)
⇒ comb(comb(lamb(x,lamb(y,var(x)),e2),e3)
⇒ comb(lamb(y,e2),e3)
⇒ e2

Case 2: e1 evaluates to false. Now we want the value of comb(con(if),e1) to select the second value, and so we take its value to be lamb(x,lamb(y,var(y)). The expression comb(comb(comb(con(if),false),e2),e3) is left for the reader to reduce.

The Prolog code to carry out the evaluation of "if" is shown below.

```
compute(if, true, lamb(x,(lamb(y,var(x))))).
compute(if, false, lamb(x,(lamb(y,var(y))))).
```

Now we can express a functional corresponding to a recursive definition in the applied lambda calculus.

define Fac = (L f n (if (zerop n) 1 (mul n (f (sub n 1))))).

Notice here the use of a predicate "zerop" that tests whether its argument is equal to zero or not.

Paradoxical Combinator

Given a mechanism (conditional expressions) for describing the functionals corresponding to recursive definitions of functions, the next step is to provide an implementation of the fixed-point operator *fix*. The (untyped) lambda calculus contains expressions that can replicate parts of themselves and

thereby act as fixed-point finders satisfying the fixed-point identity. The best known such expression, called the **paradoxical combinator** , is given by

define **Y** = λf . (λx . f (x x)) (λx . f (x x))

or for the lambda calculus evaluator

define Y = (L f ((L x (f (x x))) (L x (f (x x))))).

A reduction proves that **Y** satisfies the fixed-point identity.

Y E = (λf . (λx . f (x x)) (λx . f (x x))) E
⇒ (λx . E (x x)) (λx . E (x x))
⇒ E ((λx . E (x x)) (λx . E (x x)))
⇒ E (λh . (λx . h (x x)) (λx . h (x x)) E)
⇒ E (**Y** E).

The careful reader will have noticed that this calculation follows normal order reduction, a necessary prerequisite for having the **Y** combinator satisfy the fixed-point identity. Following an applicative order strategy leads to a nonterminating reduction.

Y E = (λf . (λx . f (x x)) (λx . f (x x))) E
⇒ (λf . f (λx . f (x x)) (λx . f (x x)))) E
⇒ (λf . f (f (λx . f (x x)) (λx . f (x x))))) E
⇒ ....

As motivation for the definition of **Y**, consider a lambda expression W with a free variable f

define W = λx . f (x x),

and notice what happens when it is applied to itself.

W W = (λx . f (x x)) (λx . f (x x)) ⇒ f((λx . f (x x)) (λx . f (x x)))
= f(W W) ⇒ f(f((λx . f (x x)) (λx . f (x x)))
= f(f(W W)) ⇒ f(f(f((λx . f (x x)) (λx . f (x x)))))
= f(f(f(W W))) ⇒ f(f(f(f(W W)))) ⇒ ....

By continuing this reduction, as many copies of f can be created as are needed. The fixed-point operator (W W) for f replicates the function f any number of times. The fixed-point operator **Y** can then be defined for an arbitrary function f by

Y f = W W

or abstracting the f

Y = λf . W W.

Actually, the lambda calculus has an infinite number of expressions that can act as fixed-point operators. Three of these are given in the exercises.

Using the paradoxical combinator, we can execute a function defined recursively as shown by the following transcript of a computation with the factorial function.

```
>>> Evaluating the Lambda Calculus <<<
Enter name of source file: fact8
  define Y = (L f ((L x (f (x x))) (L x (f (x x)))))
  define Fac = (L f n (if (zerop n) 1 (mul n (f (sub n 1)))))
  define Factorial = (Y Fac)
  (Factorial 8)
Successful Scan
Successful Parse
Result =  40320
yes
```

Without the mechanism for defining symbols, the expression must be written in its expanded form,

```
   ((L f ((L x (f (x x))) (L x (f (x x)))))
      (L f n (if (zerop n) 1 (mul n (f (sub n 1)))))
         8),
```

but the results obtained from the lambda calculus evaluator are the same.

Fixed-Point Identity

A second approach to providing a fixed-point operator in the evaluator is to code *fix* in the evaluator as a constant satisfying the fixed-point identity

$$F(fix\ F) = fix\ F.$$

All we have to do is add a reduction rule that carries out the effect of the fixed-point identity from right to left so as to replicate the functional F—namely, $fix\ F \Rightarrow F(fix\ F)$. In the Prolog code for the evaluator, insert the following clause just ahead of the clause for reducing other constants.

```
reduce(comb(con(fix),E),comb(E,comb(con(fix),E))).      % Fixed Point Operator
```

Also the constant "fix" must be added to the scanner and parser. A sample execution follows.

```
Enter name of source file: fixfact8
  define Fac = (L f n (if (zerop n) 1 (mul n (f (sub n 1)))))
  (fix Fac 8)
Successful Scan
Successful Parse
Result =  40320
yes
```

To provide a better understanding of the effect of following the fixed-point identity, consider the definition of factorial with its functional again.

fac n = if n=0 then 1 else n•fac(n-1) and

Fac = λf . λn . if n=0 then 1 else n•f(n-1).

The least fixed point of Fac, (*fix* Fac), serves as the definition of the factorial function. The function (*fix* Fac) is not recursive and can be "reduced" using the fixed-point identity

fix Fac ⇒ Fac(*fix* Fac).

The replication of the function encoded in the *fix* operator enables a reduction to create as many copies of the original function as it needs.

(*fix* Fac) 4 ⇒ (Fac (*fix* Fac)) 4
⇒ (λf . λn . if n=0 then 1 else n•f(n-1)) (*fix* Fac) 4
⇒ (λn . if n=0 then 1 else n•(*fix* Fac)(n-1)) 4
⇒ if 4=0 then 1 else 4•(*fix* Fac)(4-1)
⇒ 4•((*fix* Fac) 3) ⇒ 4•((Fac (*fix* Fac)) 3)
⇒ 4•((λf . λn . if n=0 then 1 else n•f(n-1)) (*fix* Fac) 3)
⇒ 4•((λn . if n=0 then 1 else n•((*fix* Fac)(n-1)) 3)
⇒ 4•(if 3=0 then 1 else 3•(*fix* Fac)(3-1))
⇒ 4•3•((*fix* Fac) 2) ⇒ 4•3•(Fac (*fix* Fac)) 2)
⇒ 4•3•((λf . λn . if n=0 then 1 else n•f(n-1)) (*fix* Fac) 2)
⇒ 4•3•((λn . if n=0 then 1 else n•(*fix* Fac)(n-1)) 2)
⇒ 4•3•(if 2=0 then 1 else 2•(*fix* Fac)(2-1))
⇒ 4•3•2•((*fix* Fac) 1) ⇒ 4•3•2•((Fac (*fix* Fac)) 1)
⇒ 4•3•2•((λf . λn . if n=0 then 1 else n•f(n-1)) (*fix* Fac) 1)
⇒ 4•3•2•((λn . if n=0 then 1 else n•(*fix* Fac)(n-1)) 1)
⇒ 4•3•2•(if 1=0 then 1 else 1•(*fix* Fac)(1-1))
⇒ 4•3•2•1•((*fix* Fac) 0) ⇒ 4•3•2•1•((Fac (*fix* Fac)) 0)
⇒ 4•3•2•1•((λf . λn . if n=0 then 1 else n•f(n-1)) (*fix* Fac) 0)
⇒ 4•3•2•1•((λn . if n=0 then 1 else n•(*fix* Fac)(n-1)) 0)
⇒ 4•3•2•1•(if 0=0 then 1 else 0•(*fix* Fac)(0-1))
⇒ 4•3•2•1•1 = 24

Exercises

1. Add the definition mechanism to the lambda calculus evaluator.

2. Extend the lambda calculus evaluator to recognize and interpret the conditional expression (if). Remember to add if to the list of reserved words in the scanner.

3. Show that each of the following expressions is a fixed-point operator in the lambda calculus:

   $\mathbf{Y_r}$ = λh . (λg . λx . h (g g) x) (λg . λx . h (g g) x)

   $\mathbf{Y_f}$ = λh . (λx . h (λy . x x y)) (λx . h (λy . x x y))

   $\mathbf{Y_g}$ = (λx . λy . y (x x y)) (λx . λy . y (x x y))

4. Using the following definitions, calculate fib 4 by applying the Fixed-Point Identity.

   G = λg . λn . if n=0 then 0 else if n=1 then 1 else g(n-1) + g(n-2)

   fib = *fix* G = *fix* (λg . λn . if n<2 then n else g(n-1) + g(n-2)).

5. Add several relational operators, such as = and <, to the lambda calculus evaluator and use them to test other recursive definitions.

10.5 FURTHER READING

Many of the books that contain material on denotational semantics also treat domain theory. In particular, see [Allison86], [Schmidt88], [Stoy77], and [Watt91]. David Schmidt's book has a chapter on recursively defined domains, including the inverse limit construction that justifies their existence. [Paulson87] also contains material on domain theory. For a more advanced treatment of domain theory, see [Mosses90] and [Gunter90]. Dana Scott's description of domains and models for the lambda calculus may be found in [Scott76], [Scott80], and [Scott82].

The early papers on fixed-point semantics [Manna72] and [Manna73] are a good source of examples, although the notation shows its age. Much of this material is summarized in [Manna74]. This book contains a proof of the theorem about admissible predicates for fixed-point induction. [Bird76] also contains considerable material on fixed-point semantics.

Most books on functional programming or the lambda calculus contain discussions of the paradoxical combinator and the fixed-point identity. Good examples include [Field88], [Peyton Jones87], and [Reade89]. For an advanced presentation of recursion in the lambda calculus, see [Barendregt84].