Chapter 6

Domain Theory II: Recursively Defined Functions

The examples in Chapter 5 provide strong evidence that denotational semantics is an expressive and convenient tool for language definition. Yet a few gaps remain to be filled. In Figure 5.3, the *copyout* function, which concatenates two lists, is not given. We can specify *copyout* using an iterative or recursive specification, but at this point neither is allowed in the function notation.

A similar situation arises with the semantics of a Pascal-like **while**-loop:

B: Boolean-expression

C: Command

C ::= $\cdots$ | **while** B **do** C | $\cdots$

Here is a recursive definition of its semantics: for **B**: Boolean-expression $\rightarrow Store \rightarrow Tr$ and **C**: Command $\rightarrow Store_{\perp} \rightarrow Store_{\perp}$:

$$\mathbf{C}[\![\mathbf{while}\ \mathrm{B}\ \mathbf{do}\ \mathrm{C}]\!] = \underline{\lambda} s.\ \mathbf{B}[\![\mathrm{B}]\!]s \rightarrow \mathbf{C}[\![\mathbf{while}\ \mathrm{B}\ \mathbf{do}\ \mathrm{C}]\!]\,(\mathbf{C}[\![\mathrm{C}]\!]s)\ [\!]\ s$$

Unfortunately, the clause violates a rule of Chapter 3: the meaning of a syntax phrase may be defined only in terms of the meanings of its proper subparts. We avoid this problem by stating:

$$\mathbf{C}[\![\mathbf{while}\ \mathrm{B}\ \mathbf{do}\ \mathrm{C}]\!] = w$$
$$\text{where } w : Store_{\perp} \rightarrow Store_{\perp} \text{ is } \quad w = \underline{\lambda} s.\ \mathbf{B}[\![\mathrm{B}]\!]s \rightarrow w(\mathbf{C}[\![\mathrm{C}]\!]s)\ [\!]\ s$$

But the recursion remains, for the new version exchanges the recursion in the syntax for recursion in the function notation.

We have steadfastly avoided recursion in function definitions because Section 3.3 showed that a recursive definition might not define a unique function. Recall the recursive specification of function q: $Nat \rightarrow Nat_{\perp}$ from Section 3.3:

$$q = \lambda n.\ n\ equals\ zero \rightarrow one\ [\!]\ q(n\ plus\ one)$$

Whatever q stands for, it must map a *zero* argument to *one*. Its behavior for other arguments, however, is not so clear. All the specification requires is that the answer for a nonzero argument n be the same as that for *n plus one*, its successor. A large number of functions satisfy this criterion. One choice is the function that maps *zero* to *one* and all other arguments to $\perp$. We write this function's graph as $\{\,(zero, one)\,\}$ (rather than $\{\,(zero, one), (one, \perp), (two, \perp), \cdots\}$, treating the $(n, \perp)$ pairs as ''ghost members''). This choice is a natural one for programming, for it corresponds to what happens when the definition is run as a routine on a machine. But it is not the only choice. The graph $\{\,(zero, one), (one, four), (two, four), (three, four), \cdots\}$ denotes a function that also has the behavior specified by q— *zero* maps to *one* and all other arguments map to the same answer as their successors. In general, any function whose graph is of the form $\{ (zero, one), (one, k), (two, k), \cdots \}$, for some $k \in Nat_{\perp}$, satisfies the specification. For a programmer, the last graph is an unnatural choice for the meaning of *q*, but a mathematician might like a function with the largest possible graph instead, the claim being that a ''fuller'' function gives more insight. In any case, a problem exists: a recursive specification may not define a unique function, so which one should be selected as *the* meaning of the specification? Since programming languages are implemented on machines, we wish to choose the function that suits operational intuitions. Fortunately, a well-developed theory known as *least fixed point semantics* establishes the meaning of recursive specifications. The theory:

1. Guarantees that the specification has at least one function satisfying it.
2. Provides a means for choosing a ''best'' function out of the set of all functions satisfying the specification.
3. Ensures that the function selected has a graph corresponding to the conventional operational treatment of recursion: the function maps an argument *a* to a defined answer *b* iff the operational evaluation of the specification with the representation of argument *a* produces the representation of *b* in a finite number of recursive invocations.

Two simple examples of recursive specifications will introduce all the important concepts in the theory. The theory itself is formalized in Sections 6.2 through 6.5. If you are willing to accept that recursive specifications do indeed denote unique functions, you may wish to read only Sections 6.1 and 6.6.

6.1 SOME RECURSIVELY DEFINED FUNCTIONS

Perhaps the best known example of a recursive function specification is the factorial function. Since it is so well understood, it makes an excellent testing ground for the theory. Its specification is *fac*: $Nat \rightarrow Nat_{\perp}$ such that:

$$fac(n) = n\ equals\ zero \rightarrow one\ []\ n\ times\ (fac(n\ minus\ one))$$

This specification differs from *q*'s because only one function satisfies the specification: the factorial function, whose graph is $\{ (zero, one), (one, one), (two, two), (three, six), \cdots, (i, i!), \cdots \}$. The graph will be the key to understanding the meaning of the specification. Note that it is an infinite set. It is often difficult for people to understand infinite objects; we tend to learn about them by considering their finite subparts and building up, step by step, toward the object. We can study the factorial function in this way by evaluating sample arguments with *fac*. Since the function underlying the recursive specification is not formally defined at the moment, an arrow ($\Rightarrow$) will be used when a recursive unfolding is made.

Here is an evaluation using *fac*:

$$fac(three) \Rightarrow three\ equals\ zero \rightarrow one\ []\ three\ times\ fac(three\ minus\ one)$$
$$= three\ times\ fac(three\ minus\ one)$$

$= three\ times\ fac(two)$
$\Rightarrow three\ times\ (two\ equals\ zero \rightarrow one\ [\!]\ two\ times\ fac(two\ minus\ one))$
$= three\ times\ (two\ times\ fac(one))$
$\Rightarrow three\ times\ (two\ times\ (one\ equals\ zero \rightarrow one$
$\qquad [\!]\ one\ times\ fac(one\ minus\ one)))$
$= three\ times\ (two\ times\ (one\ times\ fac(zero)))$
$\Rightarrow three\ times\ (two\ times\ (one\ times\ (zero\ equals\ zero \rightarrow one$
$\qquad [\!]\ zero\ times\ fac(zero\ minus\ one))))$
$= three\ times\ (two\ times\ (one\ times\ one))$
$= six$

The recursive unfoldings and simplifications correspond to the conventional operational treatment of a recursive routine. Four unfoldings of the recursive definition were needed to produce the answer. To make the *fac* specification produce an answer *b* from an argument *a*, at most a *finite* number of unfoldings are needed. If an *infinite* number of unfoldings are needed to produce an answer, the evaluation will never be complete. These ideas apply to the specification *q* as well. Only argument *zero* ever produces an answer in a finite number of unfoldings of *q*.

Rather than randomly supplying arguments to *fac*, we use a more systematic method to understand its workings. Our approach is to place a limit on the number of unfoldings of *fac* and see which arguments can produce answers. Here is a summary of *fac*'s behavior broken down in this fashion:

1. *Zero unfoldings*: no argument $n \in Nat$ can produce an answer, for no form $fac(n)$ can simplify to an answer without the initial unfolding. The corresponding function graph is $\{\,\}$.
2. *One unfolding*: this allows *fac* to be replaced by its body only once. Thus, $fac(zero) \Rightarrow zero\ equals\ zero \rightarrow one\ [\!]\ \cdots = one$, but all other nonzero arguments require further unfoldings to simplify to answers. The graph produced is $\{\,(zero, one)\,\}$.
3. *Two unfoldings*: since only one unfolding is needed for mapping argument *zero* to *one*, $(zero, one)$ appears in the graph. The extra unfolding allows argument *one* to evaluate to *one*, for $fac(one) \Rightarrow one\ equals\ zero \rightarrow one\ [\!]\ one\ times\ (fac(one\ minus\ one)) = one\ times\ fac(zero) \Rightarrow one\ times\ (zero\ equals\ zero \rightarrow one\ [\!]\ \cdots) = one\ times\ one = one$. All other arguments require further unfoldings and do not produce answers at this stage. The graph is $\{\,(zero, one), (one, one)\,\}$.
4. $(i+1)$ *unfoldings, for* $i \geq 0$: all arguments with values of *i* or less will simplify to answers $i!$, giving the graph $\{\,(zero, one), (one, one), (two, two), (three, six), \cdots, (i, i!)\,\}$.

The graph produced at each stage defines a function. In the above example, let fac_i denote the function defined at stage *i*. For example, $graph(fac_3) = \{\,(zero, one), (one, one), (two, two)\,\}$. Some interesting properties appear: for all $i \geq 0$, $graph(fac_i) \subseteq graph(fac_{i+1})$. This says that the partial functions produced at each stage are consistent with one another in their answer production. This isn't a startling fact, but it will prove to be important later. Further, for all $i \geq 0$, $graph(fac_i) \subseteq graph(factorial)$, which says that each fac_i exhibits behavior consistent with the ultimate solution to the specification, the factorial function. This implies:

$$\bigcup_{i=0}^{\infty} graph(fac_i) \subseteq graph(factorial)$$

Conversely, if some pair (a,b) is in $graph(factorial)$, then there must be some finite $i > 0$ such that (a,b) is in $graph(fac_i)$ also, as answers are produced in a finite number of unfoldings. (This property holds for *factorial* and its specification, but it may not hold in general. In Section 6.3, we show how to make it hold.) Thus:

$$graph(factorial) \subseteq \bigcup_{i=0}^{\infty} graph(fac_i)$$

and we have just shown:

$$graph(factorial) = \bigcup_{i=0}^{\infty} graph(fac_i)$$

The equality suits our operational intuitions and states that the factorial function can be totally understood in terms of the finite subfunctions $\{ fac_i \mid i \geq 0 \}$.

This example demonstrates the fundamental principle of least fixed point semantics: the meaning of any recursively defined function is exactly the union of the meanings of its finite subfunctions. It is easy to produce a nonrecursive representation of each subfunction. Define each fac_i: $Nat \rightarrow Nat_\perp$, for $i \geq 0$, as:

$$fac_0 = \lambda n. \perp$$
$$fac_{i+1} = \lambda n.n\ equals\ zero \rightarrow one\ [\!]\ n\ times\ fac_i(n\ minus\ one), \quad \text{for all } i \geq 0$$

The graph of each fac_i is the one produced at stage i of the *fac* unfolding. The importance is twofold: first, each fac_i is a nonrecursive definition, which suggests that a recursive specification can be understood in terms of a family of nonrecursive ones; and, second, a format common to all the fac_i's can be extracted. Let:

$$F = \lambda f.\lambda n.\ n\ equals\ zero \rightarrow one\ [\!]\ n\ times\ (f(n\ minus\ one))$$

Each $fac_{i+1} = F(fac_i)$. The nonrecursive F: $(Nat \rightarrow Nat_\perp) \rightarrow (Nat \rightarrow Nat_\perp)$ is called a *functional*, because it takes a function as an argument and produces one as a result. Thus:

$$graph(factorial) = \bigcup_{i=0}^{\infty} graph(F^i(\emptyset))$$

where $F^i = F \circ F \circ \cdots \circ F$, i times, and $\emptyset = (\lambda n. \perp)$. Another important fact is that $graph(F(factorial)) = graph(factorial)$, which implies $F(factorial) = factorial$, by the extensionality principle. The factorial function is a *fixed point* of F, as the answer F produces from argument *factorial* is exactly *factorial* again.

We can apply the ideas just discussed to the q specification. We use the associated functional Q: $(Nat \rightarrow Nat_\perp) \rightarrow (Nat \rightarrow Nat_\perp)$, which is:

$$Q = \lambda g.\lambda n.\ n\ equals\ zero \rightarrow one\ [\!]\ g\,(n\ plus\ one)$$

Then:

$$Q^0(\emptyset) = (\lambda n. \perp)$$
$$graph(Q^0(\emptyset)) = \{\ \}$$
$$Q^1(\emptyset) = \lambda n.\ n\ equals\ zero \rightarrow one\ [\!]\ (\lambda n. \perp)\,(n\ plus\ one)$$
$$= \lambda n.\ n\ equals\ zero \rightarrow one\ [\!]\ \perp$$
$$graph(Q^1(\emptyset)) = \{ (zero, one) \}$$

$Q^2(\emptyset) = Q(Q(\emptyset))$
$= \lambda n.\ n\ equals\ zero \rightarrow one\ []\ ((n\ plus\ one)\ equals\ zero \rightarrow one\ []\ \perp)$
$graph(Q^2(\emptyset)) = \{\ (zero, one)\ \}$

At this point a convergence has occurred: for all $i \geq 1$, $graph(Q^i(\emptyset)) = \{\ (zero, one)\ \}$. It follows that:

$$\bigcup_{i=0}^{\infty} graph(Q^i(\emptyset)) = \{\ (zero, one)\ \}$$

Let *qlimit* denote the function that has this graph. It is easy to show that $Q(qlimit) = qlimit$, that is, *qlimit* is a fixed point of Q.

Unlike the specification *fac*, *q* has many possible solutions. Recall that each one must have a graph of the form $\{\ (zero, one), (one, k), \cdots, (i, k), \cdots\ \}$ for some $k \in Nat_{\perp}$. Let *qk* be one of these solutions. We can show that:

1. qk is a fixed point of Q, that is, $Q(qk) = qk$.
2. $graph(qlimit) \subseteq graph(qk)$.

Fact 1 says that the act of satisfying a specification is formalized by the fixed point property—only fixed points of the associated functional are possible meanings of the specification. Fact 2 states that the solution obtained using the stages of unfolding method is the *smallest* of all the possible solutions. For this reason, we call it the *least fixed point* of the functional.

Now the method for providing a meaning for a recursive specification is complete. Let a recursive specification $f = F(f)$ denote the least fixed point of functional F, that is, the function associated with $\bigcup_{i=0}^{\infty} graph(F^i(\emptyset))$. The three desired properties follow: a solution to the specification exists; the criterion of leastness is used to select from the possible solutions; and, since the method for constructing the function exactly follows the usual operational treatment of recursive definitions, the solution corresponds to the one determined computationally.

The following sections formalize the method.

6.2 PARTIAL ORDERINGS

The theory is formalized smoothly if the subset relation used with the function graphs is generalized to a *partial ordering*. Then elements of semantic domains can be directly involved in the set theoretical reasoning. For a domain D, a binary relation $r \subseteq D \times D$ (or $r : D \times D \rightarrow \mathbb{B}$) is represented by the infix symbol $\sqsubseteq_D$, or just $\sqsubseteq$ if the domain of usage is clear. For $a, b \in D$, we read $a \sqsubseteq b$ as saying ''a is less defined than b.''

6.1 Definition:

A relation $\sqsubseteq : D \times D \rightarrow \mathbb{B}$ *is a partial ordering upon D iff* $\sqsubseteq$ *is:*

1. *reflexive: for all* $a \in D$, $a \sqsubseteq a$;
2. *antisymmetric: for all* $a, b \in D$, $a \sqsubseteq b$ *and* $b \sqsubseteq a$ *imply* $a = b$.
3. *transitive: for all* $a, b, c \in D$, $a \sqsubseteq b$ *and* $b \sqsubseteq c$ *imply* $a \sqsubseteq c$.

A partial ordering $\sqsubseteq$ on a domain D treats the members of D as if they were sets. In fact, given a set E, we can use $\subseteq$ as a partial ordering upon the members of $\mathbb{P}(E)$. A minimum partial order structure on a domain D is the *discrete* partial ordering, which makes each $d \in D$ less defined than itself and relates no other elements; that is, for all $d, e \in D$, $d \sqsubseteq e$ iff $d = e$.

A partially ordered set of elements can be represented by an acyclic graph, where $x \sqsubseteq y$ when there is an arc from element x to element y and x is beneath y on the page. For example, given $\mathbb{P}(\{\, one, two, three \,\})$, partially ordered by subset inclusion, we draw the graph:

$P0 =$ $\{\, one, two, three \,\}$

$\{\, one, two \,\}$ $\{\, one, three \,\}$ $\{\, two, three \,\}$

$\{\, one \,\}$ $\{\, two \,\}$ $\{\, three \,\}$

$\{\ \}$

to represent the partial ordering. Taking into account reflexivity, antisymmetry, and transitivity, the graph completely describes the partial ordering. For example, $\{\, two \,\} \sqsubseteq \{\, one, two \,\}$, $\{\, one \,\} \sqsubseteq \{\, one \,\}$, and $\{\ \} \sqsubseteq \{\, two, three \,\}$.

For the set $\{\, a, b, c, d \,\}$, the graph:

$P1 =$ a c

b

d

also defines a partial ordering, as does:

$P2 =$ a b c d

(the discrete ordering). There is a special symbol to denote the element in a partially ordered domain that corresponds to the empty set.

6.2 Definition:

For partial ordering $\sqsubseteq$ on D, if there exists an element $c \in D$ such that for all $d \in D$, $c \sqsubseteq d$, then c is the least defined element in D and is denoted by the symbol $\perp$ (read ‘‘bottom’’).

Partial orderings *P0* and *P1* have bottom elements, but *P2* does not. We also introduce an operation analogous to set union.

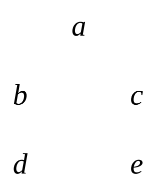
6.3 Definition:

For a partial ordering $\sqsubseteq$ on D, for all a, $b \in D$, the expression $a \sqcup b$ denotes the element in D (if it exists) such that:

1. *$a \sqsubseteq a \sqcup b$ and $b \sqsubseteq a \sqcup b$.*
2. *for all $d \in D$, $a \sqsubseteq d$ and $b \sqsubseteq d$ imply $a \sqcup b \sqsubseteq d$.*

The element $a \sqcup b$ is the *join* of *a* and *b*. The join operation produces the smallest element that is larger than both of its arguments. A partial ordering might not have joins for all of its pairs. For example, partial ordering *P2* has joins defined only for pairs (i, i)— $i \sqcup i = i$ for any $i \in P2$. Here are some other examples: for partial ordering *P1*, $c \sqcup d = c$ and $a \sqcup d = a$, but $a \sqcup c$ is not defined. Partial ordering *P0* has joins defined for all pairs. Conversely, for:

P3 =



$d \sqcup e$ is *not* defined. Even though all of *b, c,* and *a* are better defined than *d* and *e*, no one of the three is minimal; since both $b \not\sqsubseteq c$ and $c \not\sqsubseteq b$, neither of the two can be chosen as least.

An intersection-like operation called *meet* is definable in a similar fashion. We write $x \sqcap y$ to denote the best-defined element that is smaller than both *x* and *y*. *P0* has meets defined for all pairs of elements. In *P3*, $b \sqcap c$ is not defined for reasons similar to those given for the undefinedness of $d \sqcup e$; $d \sqcap e$ has no value either.

6.4 Definition:

A set D, partially ordered by $\sqsubseteq$, is a lattice iff for all a, $b \in D$, both $a \sqcup b$ and $a \sqcap b$ exist.

P0 is a lattice, but *P1*-*P3* are not. For any set *E*, $\mathbb{P}(E)$ is a lattice under the usual subset ordering: join is set union and meet is set intersection.

The concepts of join and meet are usefully generalized to operate over a (possibly infinite) set of arguments rather than just two.

6.5 Definition:

For a set D partially ordered by $\sqsubseteq$ and a subset X of D, $\bigsqcup X$ denotes the element of D (if it exists) such that:

1. *for all $x \in X$, $x \sqsubseteq \bigsqcup X$.*
2. *for all $d \in D$, if for all $x \in X$, $x \sqsubseteq d$, then $\bigsqcup X \sqsubseteq d$.*

The element $\bigsqcup X$ is called the *least upper bound* (*lub*) of *X*. The definition for *greatest lower bound* (*glb*) of *X* is similar and is written $\bigsqcap X$.

6.6 Definition:

A set D partially ordered by $\sqsubseteq$ is a complete lattice iff for all subsets X of D, both $\bigsqcup X$

and $\sqcap X$ exist.

The standard example of a complete lattice is the powerset lattice $\mathbb{P}(E)$. Any lattice with a finite number of elements must be a complete lattice. Not all lattices are complete, however; consider the set $F = \{ x \mid x \textit{ is a finite subset of } \mathbb{N} \}$ partially ordered by subset inclusion. F is clearly a lattice, as joins and meets of finite sets yield finite sets, but F is not complete, for the lub of the set $S = \{ \{zero\}, \{zero, one\}, \{zero, one, two\}, \cdots \}$ is exactly $\mathbb{N}$, which is not in F.

A complete lattice D must always have a bottom element, for $\sqcap D = \bot$. Dually, $\sqcup D$ denotes an element represented by $\top$ (*top*).

The definitions of lattice and complete lattice are standard ones and are included for completeness' sake. The theory for least fixed point semantics doesn't require domains to be lattices. The only property needed from partially ordered sets is that lubs exist for those sets representing the subfunction families. This motivates two important definitions.

6.7 Definition:

For a partially ordered set D, a subset X of D is a chain iff X is nonempty and for all $a, b \in X$, $a \sqsubseteq b$ or $b \sqsubseteq a$.

A *chain* represents a family of elements that contain information consistent with one another. (Recall the family of functions developed in the stages of unfolding of *fac* in Section 6.1.) Chains can be finite or infinite; in partial order *P1*, $\{ d, b, a \}$ forms a chain, as does $\{ c \}$, but $\{ a, c \}$ does not. The lub of a finite chain is always the largest element in the chain. This does not hold for infinite chains: consider again the set S defined above, which is a chain in both lattice F and complete lattice $\mathbb{P}(\mathbb{N})$.

Since chains abstract the consistent subfunction families, it is important to ensure that such chains always have lubs. A lattice may not have lubs for infinite chains, so ordinary lattices are not suitable. On the other hand, requiring a domain to be a complete lattice is too strong. The compromise settled upon is called a *complete partial ordering.*

6.8 Definition:

1. *A partially ordered set D is a complete partial ordering (cpo) iff every chain in D has a least upper bound in D.*
2. *A partially ordered set D is a pointed complete partial ordering (pointed cpo) iff it is a complete partial ordering and it has a least element.*

The partial orderings *P0* through *P3* are cpos, but only *P0* and *P1* are pointed cpos. The examples suggest that the requirements for being a cpo are quite weak. The solutions for recursive function specifications are found as elements within pointed cpos.

6.3 CONTINUOUS FUNCTIONS

When the partial order relation $\sqsubseteq$ was introduced, $\sqsubseteq$ was read as ''is less defined than.'' For functions, we judged definedness in terms of their graphs: a function f is less defined than a function g if f's graph is a subset of g's. The totally undefined function $(\lambda n. \bot)$ contains no answer-producing information at all, and it is appropriate that its graph is the empty set and that $(\lambda n. \bot) \sqsubseteq f$ for all functions f. The graph representation of a function provides a way to ''look inside'' the function and judge its information content. In general, any element d of an arbitrary domain D might be thought of as a set-like object, containing ''atoms'' of information. Then, $\sqsubseteq_D$ can still be thought of as a subset relation, and the least upper bound operator can still be thought of as set union.

Now consider what a function does to its arguments. A function $f: A \rightarrow B$ is a transformation agent, converting an element $x \in A$ to some $f(x) \in B$. How should this be accomplished? If x is thought of as a set of ''atoms,'' then f transforms x by mapping x's atoms into the B domain and recombining them there. If $x \sqsubseteq_A y$, an application of f should produce the analogous situation in B. A function $f: A \rightarrow B$ is *monotonic* iff for all $x, y \in A$, $x \sqsubseteq_A y$ implies $f(x) \sqsubseteq_B f(y)$. The condition is justified by practical issues in computing: a procedure for transforming a data structure such as an array performs the transformation by altering the array's subparts one at a time, combining them to form a new value. The procedure denotes a monotonic function. Nonmonotonic functions tend to be nasty entities, often impossible to implemement. A famous example of a nonmonotonic function is *program-halts* : $Nat_\bot \rightarrow \mathbb{B}$,

$$program\text{-}halts(x) = \begin{cases} true & \text{if } x \neq \bot \\ false & \text{if } x = \bot \end{cases}$$

program-halts is nonmonotonic. Consider a proper number n. It is always the case that $\bot \sqsubseteq n$, but it is normally *not* the case that $program\text{-}halts(\bot) \sqsubseteq program\text{-}halts(n)$, that is, $false \sqsubseteq true$, as *false* is not less defined than *true*— their information contents are disjoint.

For the moment, pretend that an implementation of *program-halts* exists and call its coded procedure PGM-HALTS. PGM-HALTS could see if a program P terminates with its input A, for P represents a function in domain $Nat \rightarrow Nat_\bot$, A represents a natural number, so PGM-HALTS(P(A)) returns TRUE or FALSE based upon whether or not P halts with A. It is easy to see how PGM-HALTS might return a TRUE value (it runs P(A) and once P(A) stops and outputs a numeral, it returns TRUE), but how can PGM-HALTS ever tell that P(A) will run forever (that is, be undefined) so that it can return FALSE? There is no obvious solution; in fact, there is no solution at all— this is the famous ''halting problem'' of computability theory: *program-halts* cannot be implemented on any computing machine.

We require that all the functions used in denotational semantics be monotonic. The requirement guarantees that any family of subfunctions $\bot$, $F(\bot)$, $F(F(\bot))$, $\cdots$, $F^i(\bot)$, $\cdots$ generated by a monotonic functional F is a chain.

A condition stronger than monotonicity is needed to develop least fixed point semantics. Just as the binary join operation was generalized to the lub operation, the monotonicity condition is generalized to *continuity*.

6.9 Definition:

For cpos A and B, a monotonic function $f: A \to B$ is continuous iff for any chain $X \subseteq A$, $f(\bigsqcup X) = \bigsqcup \{ f(x) \mid x \in X \}$.

A continuous function preserves limits of chains. That is, $f(\bigsqcup X)$ contains exactly the same information as that obtained by mapping all the x's to $f(x)$'s and joining the results. The reason that we use continuous functions is that we require the property $graph(\bigsqcup \{ F^i(\emptyset) \mid i \geq 0 \}) \subseteq \bigcup_{i=0}^{\infty} \{ graph(F^i(\emptyset)) \mid i \geq 0 \}$, which was noted in Section 6.1 in the *fac* example.

Continuous functions suggest a strategy for effectively processing objects of infinite size (information content). For some infinite object Y, it may not be possible to place a representation of Y in the computer store, so $f(Y)$ simply isn't computable in the conventional way. But if Y is the least upper bound of a chain $\{ y_i \mid i \in \mathbb{N} \}$ where each y_i has finite size, and f is a continuous function, then each y_i can be successively stored and f applied to each. The needed value $f(Y)$ is built up piece by piece as $f(y_0) \sqcup f(y_1) \sqcup f(y_2) \sqcup \cdots$. Since $\{ f(y_i) \mid i \in \mathbb{N} \}$ constitutes a chain and f is continuous, $f(Y) = \bigsqcup \{ f(y_i) \mid i \in \mathbb{N} \}$. (This was the same process used for determining the graph of the factorial function.) Of course, to completely compute the answer will take an infinite amount of time, but it is reassuring that every piece of the answer $f(Y)$ will appear within a finite amount of time.

Continuity is such an important principle that virtually all of the classical areas of mathematics utilize it in some form.

6.4 LEAST FIXED POINTS

A *functional* is a continuous function $f: D \to D$; usually D is a domain of form $A \to B$, but it doesn't have to be.

6.10 Definition:

For a functional $F: D \to D$ and an element $d \in D$, d is a fixed point of F iff $F(d) = d$. Further, d is the least fixed point of F if, for all $e \in D$, $F(e) = e$ implies $d \sqsubseteq e$.

6.11 Theorem:

If the domain D is a pointed cpo, then the least fixed point of a continuous functional $F: D \to D$ exists and is defined to be $fix\,F = \bigsqcup \{ F^i(\bot) \mid i \geq 0 \}$, where $F^i = F \circ F \circ \cdots \circ F$, i times.

Proof: First, $fix\,F$ is a fixed point of F, as

$$\begin{aligned} F(fix\,F) &= F(\bigsqcup \{ F^i(\bot) \mid i \geq 0 \}) \\ &= \bigsqcup \{ F(F^i(\bot)) \mid i \geq 0 \} \text{ by continuity of } F \\ &= \bigsqcup \{ F^i(\bot) \mid i \geq 1 \} \\ &= \bigsqcup \{ F^i(\bot) \mid i \geq 0 \} \text{ as } F^0(\bot) = \bot \text{ and } \bot \sqsubseteq F(\bot) \end{aligned}$$

$= \mathit{fix}\, F$

To show $\mathit{fix}\, F$ is least, let $e \in D$ be a fixed point of F. Now, $\bot \sqsubseteq e$, and by the monotonicity of F, $F^i(\bot) \sqsubseteq F^i(e) = e$, for all $i > 0$, since e is a fixed point. This implies $\mathit{fix}\, F = \bigsqcup\{ F^i(\bot) \mid i \geq 0 \} \sqsubseteq e$, showing $\mathit{fix}\, F$ is least. □

This produces the primary result of the chapter:

6.12 Definition:

The meaning of a recursive specification $f = F(f)$ is taken to be $\mathit{fix}\, F$, the least fixed point of the functional denoted by F.

Examples of recursive specifications and their least fixed point semantics are given in Section 6.6. First, a series of proofs are needed to show that semantic domains are cpos and that their operations are continuous.

6.5 DOMAINS ARE CPOS

The partial orderings defined in Section 6.3 introduce internal structure into domains. In this section we define the partial orderings for primitive domains and the compound domain constructions. The orderings make the domains into cpos. Our reason for using cpos (rather than pointed cpos) is that we want ordinary sets (which are discretely ordered cpos) to be domains. The lifting construction is the tool we use to make a cpo into a pointed cpo.

What partial ordering should be placed on a primitive domain? Each of a primitive domain's elements is a distinct, atomic answer value. For example, in $\mathbb{N}$, both *zero* and *one* are answer elements, and by no means does *one* contain more information than *zero* (or vice versa). The information is equal in ‘‘quantity’’ but disjoint in value. This suggests that $\mathbb{N}$ has the discrete ordering: $a \sqsubseteq b$ iff $a = b$. We always place the discrete partial ordering upon a primitive domain.

6.13 Proposition:

Any set of elements D with the discrete partial ordering is a cpo, and any operation $f: D \rightarrow E$ is continuous.

Proof: Trivial, as the only chains in D are sets of one element. □

The partial orderings of compound domains are based upon the orderings of their component domains. Due to its importance in converting cpos into pointed cpos, the partial ordering for the lifting construction is defined first.

6.14 Definition:

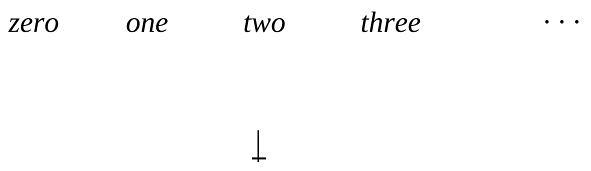
For a partially ordered set A, its lifting $A_\bot$ is the set $A \cup \{ \bot \}$, partially ordered by the relation $d \sqsubseteq_{A_\bot} d'$ iff $d = \bot$, or $d, d' \in A$ and $d \sqsubseteq_A d'$.

6.15 Proposition:

If A is a cpo, then $A_\perp$ is a pointed cpo. Further, $(\underline{\lambda} x.e) : A_\perp \to B_\perp$ is continuous when $(\lambda x.e) : A \to B_\perp$ is.

Proof: Left as an exercise. □

For the cpo $\mathbb{N}$, the pointed cpo $\mathbb{N}_\perp$ is drawn as:



6.16 Definition:

For partially ordered sets A and B, their product $A \times B$ is the set $\{ (a, b) \mid a \in A \text{ and } b \in B \}$ partially ordered by the relation $(a, b) \sqsubseteq_{A \times B} (a', b')$ iff $a \sqsubseteq_A a'$ and $b \sqsubseteq_B b'$.

6.17 Proposition:

If A and B are (pointed) cpos, then $A \times B$ is a (pointed) cpo, and its associated operation builders are continuous.

Proof: Any chain C in $A \times B$ has the form $C = \{ (a_i, b_i) \mid a_i \in A, b_i \in B, i \in I \}$, for some index set I. By Definition 6.16, both $M = \{ a_i \mid i \in I \}$ and $N = \{ b_i \mid i \in I \}$ are chains in A and B respectively, with lubs $\bigsqcup M$ and $\bigsqcup N$. That $\bigsqcup C = (\bigsqcup M, \bigsqcup N)$ follows from the definition of lub. Thus $A \times B$ is a cpo, and the pairing operation is continuous. (If A and B are both pointed, $(\perp, \perp)$ is the least element in $A \times B$. This element is distinct from the $\perp$ introduced by lifting.) To show that *fst* is continuous, consider the chain C above:

$$
\begin{aligned}
&\bigsqcup \{ fst(a_i, b_i) \mid i \in I \} = \bigsqcup \{ a_i \mid i \in I \} \\
&= fst(\bigsqcup \{ a_i \mid i \in I \}, \bigsqcup \{ b_i \mid i \in I \}) \\
&= fst(\bigsqcup \{ (a_i, b_i) \mid i \in I \}).
\end{aligned}
$$

The proof for *snd* is similar. □

It is straightforward to generalize Definition 6.16 and Proposition 6.17 to products of arbitrary size.

Here is a sample product construction: $\mathbb{B}_\perp \times \mathbb{B}_\perp =$

$(true, true)$ $(false, true)$ $(true, false)$ $(false, false)$

$(\bot, true)$ $(true, \bot)$ $(false, \bot)$ $(\bot, false)$

$(\bot, \bot)$

Now that the partial ordering upon products has been established, we can state an important result about the continuity of operations that take arguments from more than one domain.

6.18 Proposition:

A function $f: D_1 \times D_2 \times \cdots \times D_n \rightarrow E$ is continuous iff it is continuous in each of its individual arguments; that is, f is continuous in $D_1 \times D_2 \times \cdots \times D_n$ iff it is continuous in every D_i, for $1 \leq i \leq n$, where the other arguments are held constant.

Proof: The proof is given for $f: D_1 \times D_2 \rightarrow E$; a proof for general products is an extension of this one.

Only if: Let f be continuous. To show f continuous in D_1, set its second argument to some $x \in D_2$. For chain $\{ d_i \mid d_i \in D_1, i \in I \}$, $\bigsqcup \{ f(d_i, x) \mid i \in I \} = f(\bigsqcup \{ (d_i, x) \mid i \in I \})$, as f is continuous; this equals $f(\bigsqcup \{ d_i \mid i \in I \}, \bigsqcup \{ x \}) = f(\bigsqcup \{ d_i \mid i \in I \}, x)$. The proof for D_2 is similar.

If: Let f be continuous in its first and second arguments separately, and let $\{ (a_i, b_i) \mid a_i \in D_1, b_i \in D_2, i \in I \}$ be a chain. Then $f(\bigsqcup \{ (a_i, b_i) \mid i \in I \})$ $= f(\bigsqcup \{ a_i \mid i \in I \}, \bigsqcup \{ b_j \mid j \in I \}), = \bigsqcup \{ f(a_i, \bigsqcup \{ b_j \mid j \in I \}) \mid i \in I \}$ by f's continuity on its first argument; this equals $\bigsqcup \{ \bigsqcup \{ f(a_i, b_j) \mid j \in I \} \mid i \in I \}$ $= \bigsqcup \{ f(a_i, b_j) \mid i \in I \text{ and } j \in I \}$ as taking lubs is associative. Now for each pair (a_i, b_j) take $k = \max(i,j)$. For example, if $i \leq j$, k is j, and then $(a_i, b_j) \sqsubseteq (a_k, b_k) = (a_j, b_j)$, as $a_i \sqsubseteq a_j$ since $\{ a_i \mid i \in I \}$ is a chain. This implies $f(a_i, b_j) \sqsubseteq f(a_k, b_k)$. Similar reasoning holds when $j \leq i$. This means $f(a_i, b_j) \sqsubseteq f(a_k, b_k) \sqsubseteq \bigsqcup \{ f(a_k, b_k) \mid k \in I \}$, implying $\bigsqcup \{ f(a_i, b_j) \mid i \in I, j \in I \} \sqsubseteq \bigsqcup \{ f(a_k, b_k) \mid k \in I \}$. But for all $k \in I$, $f(a_k, b_k)$ is in $\{ f(a_i, b_j) \mid i \in I, j \in I \}$, implying $\bigsqcup \{ f(a_k, b_k) \mid k \in I \} \sqsubseteq \bigsqcup \{ f(a_i, b_j) \mid i \in I, j \in I \}$, and so the lubs are equal. This concludes the proof, since $\bigsqcup \{ f(a_i, b_j) \mid i \in I, j \in I \}$ $= \bigsqcup \{ f(a_k, b_k) \mid k \in I \} = \bigsqcup \{ f(a_i, b_i) \mid i \in I \}$. □

6.19 Definition:

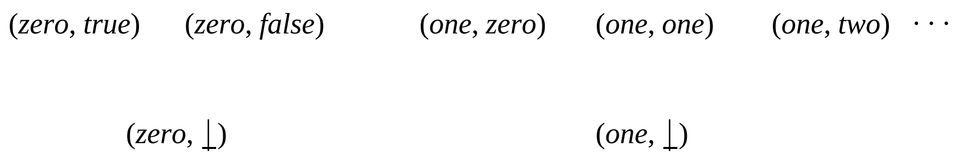
For partial orders A and B, their disjoint union $A + B$ is the set $\{ (zero, a) \mid a \in A \} \cup \{ (one, b) \mid b \in B \}$ partially ordered by the relation $d \sqsubseteq_{A+B} d'$ iff ($d = (zero, a)$, $d' = (zero, a')$, and $a \sqsubseteq_A a'$) or ($d = (one, b)$, $d' = (one, b')$, and $b \sqsubseteq_B b'$).

6.20 Proposition:

If A and B are cpos, then $A+B$ is a cpo, and its associated operation builders are continuous.

Proof: Left as an exercise. $\square$

As an example, $\mathbb{B}_\perp + \mathbb{N}_\perp$ is:



6.21 Definition:

For partially ordered sets A and B, their function space $A \rightarrow B$ is the set of all continuous functions with domain A and codomain B, partially ordered by the relation $f \sqsubseteq_{A \rightarrow B} g$ iff for all $a \in A$, $f(a) \sqsubseteq_B g(a)$.

6.22 Proposition:

If A and B are cpos, then $A \rightarrow B$ is a cpo, and its associated operation builders are continuous.

Proof: For the chain $C = \{ f_i : A \rightarrow B \mid i \in I \}$, for each $a_j \in A$, the set $A_j = \{ f_i(a_j) \mid i \in I \}$ is a chain in B, due to the partial ordering. Since B is a cpo, $\bigsqcup A_j$ exists for all such j. Use these lubs to define the function $g : A \rightarrow B$ such that $g(a_j) = \bigsqcup A_j$ for $a_j \in A$. We first show that g is continuous: for a chain $X = \{ a_j \in A \mid j \in J \}$, $\bigsqcup \{ g(a_j) \mid j \in J \}$ equals $\bigsqcup \{ \bigsqcup \{ f_i(a_j) \mid i \in I \} \mid j \in J \}$ by the definition of g; this value equals $\bigsqcup \{ \bigsqcup \{ f_i(a_j) \mid j \in J \} \mid i \in I \}$ as lubs are associative; this value equals $\bigsqcup \{ f_i(\bigsqcup \{ a_j \mid j \in J \}) \mid i \in I \}$ as each f_i is continuous; and this equals $\bigsqcup \{ f_i(\bigsqcup X) \mid i \in I \} = g(\bigsqcup X)$. That g is the lub of the chain follows by definition of the partial ordering.

To show that function application is continuous, let $g = \bigsqcup \{ f_i : A \rightarrow B \mid i \in I \}$; then $\bigsqcup \{ f_i(a) \mid i \in I \} = g(a) = (\bigsqcup \{ f_i \mid i \in I \})(a)$, taking care of the first argument; and $\bigsqcup \{ g(a_j) \mid j \in J \} = g(\bigsqcup \{ a_j \mid j \in J \})$ by the continuity of g, handling the second argument.

Finally, for abstraction, let $\{ e_i \mid i \in I \}$ be a set of expressions such that for any $a \in A$, $\{ [a/x]e_i \mid i \in I \}$ is a chain in B. Then $\{ \lambda x.e_i \mid i \in I \}$ is a chain in $A \rightarrow B$ by definition of the partial ordering. Earlier in the proof we saw that $\bigsqcup \{ \lambda x.e_i \mid i \in I \}$ is the function g such that $g(a) = \bigsqcup \{ (\lambda x.e_i)(a) \mid i \in I \} = \bigsqcup \{ [a/x]e_i \mid i \in I \}$. Thus, $g =$

$\lambda a. \bigsqcup\{\, [a/x]e_i \mid i \in I \,\} = \lambda x. \bigsqcup\{\, [x/x]e_i \mid i \in I \,\}$ by renaming the argument identifier; this equals $\lambda x. \bigsqcup\{\, e_i \mid i \in I \,\}$ by definition of substitution. Hence abstraction is continuous. □

6.23 Corollary:

If A is a cpo and B is a pointed cpo, then $A \rightarrow B$ is a pointed cpo.

Proof: The least defined element in $A \rightarrow B$ is the function $f\colon A \rightarrow B$, defined as $f(a) = b_0$, where b_0 is the least defined element in B. □

The partial ordering on $A \rightarrow B$ is known as the *pointwise* ordering, for $f \sqsubseteq g$ iff f produces less-defined answers than g at all argument points. The pointwise ordering formalizes the method of ordering functions based on the subset inclusion of their graphs. As an example, if we represent a function by its graph and not include any argument, answer pairs of the form $(t, \perp)$, the domain $\mathbb{B}_\perp \rightarrow \mathbb{B}_\perp$ appears:

$\{\, (true, true), (false, true), (\perp, true) \,\}$ $\{\, (true, false), (false, false), (\perp, false) \,\}$

$\{\, (true, true), (false, true) \,\}$ $\{\, (true, true), (false, false) \,\}$ $\{\, (true, false), (false, true) \,\}$ $\{\, (true, false), (false, false) \,\}$

$\{\, (true, true) \,\}$ $\{\, (false, true) \,\}$ $\{\, (false, false) \,\}$ $\{\, (true, false) \,\}$

$\{\,\}$

Since $\mathbb{B}_\perp$ is a pointed cpo, so is $\mathbb{B}_\perp \rightarrow \mathbb{B}_\perp$. The least element is not the $\perp$ introduced by lifting, but is the proper function $(\lambda t. \perp)$.

The preceding results taken together imply the following.

6.24 Theorem:

Any operation built using function notation is a continuous function.

The least fixed point semantics method operates on pointed cpos. The least element of a domain gives a starting point for building a solution. This starting point need not be the $\perp$ element added by lifting: Proposition 6.17 and Corollary 6.23 show that least elements can naturally result from a compound domain construction. In any case, lifting is an easy way of creating the starting point. In the examples in the next section, the symbol $\perp$ will be used to stand for the least member of a pointed cpo, regardless of whether this element is due to a lifting or not.

We will treat *fix* as an operation builder. (See Theorem 6.11.) For any pointed cpo D and continuous function $F{:}D \rightarrow D$, $\mathit{fix}\,F$ denotes the lub of the chain induced from F. From here on, any recursive specification $f = F(f)$ is taken as an abbreviation for $f = \mathit{fix}\,F$.

Since it is important to know if a cpo is pointed, the following rules are handy:

$\mathit{ispointed}(P) = \mathit{false}$, where P is a primitive domain
$\mathit{ispointed}(A \times B) = \mathit{ispointed}(A)$ and $\mathit{ispointed}(B)$
$\mathit{ispointed}(A + B) = \mathit{false}$
$\mathit{ispointed}(A \rightarrow B) = \mathit{ispointed}(B)$
$\mathit{ispointed}(A_\perp) = \mathit{true}$

6.6 EXAMPLES

Now that the theory is developed, we present a number of old and new recursive function specifications and determine their least fixed point semantics.

6.6.1 Factorial Function

We examine the factorial function and its specification once more. Recall that $\mathit{fac} : \mathit{Nat} \rightarrow \mathit{Nat}_\perp$ is defined as:

$$\mathit{fac} = \lambda n.\ n\ \mathit{equals}\ \mathit{zero} \rightarrow \mathit{one}\ [\!]\ n\ \mathit{times}\ \mathit{fac}(n\ \mathit{minus}\ \mathit{one})$$

which is an acceptable definition, since $\mathit{Nat}_\perp$ is a pointed cpo, implying that $\mathit{Nat} \rightarrow \mathit{Nat}_\perp$ is also. (Here is one small, technical point: the specification should actually read:

$$\mathit{fac} = \lambda n.\ n\ \mathit{equals}\ \mathit{zero} \rightarrow \mathit{one}\ [\!]\ (\text{let } n' = \mathit{fac}(n\ \mathit{minus}\ \mathit{one}) \text{ in } n\ \mathit{times}\ n')$$

because *times* uses arguments from *Nat* and not from $\mathit{Nat}_\perp$. We gloss over this point and say that *times* is strict on $\mathit{Nat}_\perp$ arguments.) The induced functional $F : (\mathit{Nat} \rightarrow \mathit{Nat}_\perp) \rightarrow (\mathit{Nat} \rightarrow \mathit{Nat}_\perp)$ is:

$$F = \lambda f.\lambda n.\ n\ \mathit{equals}\ \mathit{zero} \rightarrow \mathit{one}\ [\!]\ n\ \mathit{times}\ f(n\ \mathit{minus}\ \mathit{one})$$

The least fixed point semantics of *fac* is $\mathit{fix}\,F = \bigsqcup\{\, \mathit{fac}_i \mid i \geq 0 \,\}$, where:

$\mathit{fac}_0 = (\lambda n.\ \perp) = \perp \in \mathit{Nat} \rightarrow \mathit{Nat}_\perp$
$\mathit{fac}_{i+1} = F(\mathit{fac}_i)$ for $i \geq 0$

These facts have been mentioned before. What we examine now is the relationship between $\mathit{fix}\,F$ and the operational evaluation of *fac*. First, the fixed point property says that $\mathit{fix}\,F = F(\mathit{fix}\,F)$. This identity is a useful simplification rule. Consider the denotation of the phrase $(\mathit{fix}\,F)(\mathit{three})$. Why does this expression stand for *six*? We use equals-for-equals substitution to simplify the original form:

$(\mathit{fix}\,F)(\mathit{three})$
$= (F\,(\mathit{fix}\,F))(\mathit{three})$, by the fixed point property
$= ((\lambda f.\lambda n.\ n\ \mathit{equals}\ \mathit{zero} \rightarrow \mathit{one}\ [\!]\ n\ \mathit{times}\ f(n\ \mathit{minus}\ \mathit{one}))(\mathit{fix}\,F))(\mathit{three})$
$= (\lambda n.\ n\ \mathit{equals}\ \mathit{zero} \rightarrow \mathit{one}\ [\!]\ n\ \mathit{times}\ (\mathit{fix}\,F)(n\ \mathit{minus}\ \mathit{one}))(\mathit{three})$

We see that the fixed point property justifies the recursive unfolding rule that was informally used in Section 6.1.

Rather than expanding $(\mathit{fix}\,F)$ further, we bind *three* to *n*:

$= \mathit{three}\ \mathit{equals}\ \mathit{zero} \rightarrow \mathit{one}\ [\!]\ \mathit{three}\ \mathit{times}\ (\mathit{fix}\,F)(\mathit{three}\ \mathit{minus}\ \mathit{one})$
$= \mathit{three}\ \mathit{times}\ (\mathit{fix}\,F)(\mathit{two})$
$= \mathit{three}\ \mathit{times}\ (F(\mathit{fix}\,F))(\mathit{two})$
$= \mathit{three}\ \mathit{times}\ ((\lambda f.\lambda n.\ n\ \mathit{equals}\ \mathit{zero} \rightarrow \mathit{one}\ [\!]\ n\ \mathit{times}\ f(n\ \mathit{minus}\ \mathit{one}))(\mathit{fix}\,F))(\mathit{two})$
$= \mathit{three}\ \mathit{times}\ (\lambda n.n\ \mathit{equals}\ \mathit{zero} \rightarrow \mathit{one}\ [\!]\ n\ \mathit{times}\ (\mathit{fix}\,F)(n\ \mathit{minus}\ \mathit{one}))(\mathit{two})$

The expression $(\mathit{fix}\,F)$ can be expanded at will:

$= \mathit{three}\ \mathit{times}\ (\mathit{two}\ \mathit{times}\ (\mathit{fix}\,F)(\mathit{one}))$
. . .
$= \mathit{three}\ \mathit{times}\ (\mathit{two}\ \mathit{times}\ (\mathit{one}\ \mathit{times}\ (\mathit{fix}\,F)(\mathit{zero})))$
. . .
$= \mathit{six}$

6.6.2 Copyout Function

The interactive text editor in Figure 5.3 utilized a function called *copyout* for converting an internal representation of a file into an external form. The function's definition was not specified because it used recursion. Now we can alleviate the omission. The domains $\mathit{File} = \mathit{Record}^*$ and $\mathit{Openfile} = \mathit{Record}^* \times \mathit{Record}^*$ were used. Function *copyout* converts an open file into a file by appending the two record lists. A specification of $\mathit{copyout}$: $\mathit{Openfile} \rightarrow \mathit{File}_\perp$ is:

$$\begin{aligned} \mathit{copyout} = \lambda(\mathit{front}, \mathit{back}).\ & \mathit{null}\ \mathit{front} \rightarrow \mathit{back} \\ & [\!]\ \mathit{copyout}((\mathit{tl}\ \mathit{front}), ((\mathit{hd}\ \mathit{front})\ \mathit{cons}\ \mathit{back})) \end{aligned}$$

It is easy to construct the appropriate functional F; *copyout* has the meaning $(\mathit{fix}\,F)$. You should prove that the function $F^i(\perp)$ is capable of appending list pairs whose first component has length $i-1$ or less. This implies that the lub of the $F^i(\perp)$ functions, $(\mathit{fix}\,F)$, is capable of concatenating all pairs of lists whose first component has finite length.

Here is one more remark about *copyout*: its codomain was stated above as $\mathit{File}_\perp$, rather than just *File*, as originally given in Figure 5.3. The new codomain was used because least fixed point semantics requires that the codomain of any recursively defined function be pointed. If we desire a recursive version of *copyout* that uses *File* as its codomain, we must apply the results of Exercise 14 of Chapter 3 and use a primitive recursive-like definition. This is left as an important exercise. After you have studied all of the examples in this chapter, you should determine which of them can be handled *without* least fixed point semantics by using primitive recursion. The moral is that least fixed point semantics is a powerful and useful tool, but in many cases we can deal quite nicely without it.

6.6.3 Double Recursion

Least fixed point semantics is helpful for understanding recursive specifications that may be difficult to read. Consider this specification for $g : Nat \rightarrow Nat_{\perp}$:

$$g = \lambda n.\ n\ equals\ zero \rightarrow one\ []\ (g(n\ minus\ one)\ plus\ g(n\ minus\ one))\ minus\ one$$

The appropriate functional F should be obvious. We gain insight by constructing the graphs for the first few steps of the chain construction:

$$graph(F^0(\perp)) = \{\ \}$$
$$graph(F^1(\perp)) = \{\ (zero, one)\ \}$$
$$graph(F^2(\perp)) = \{\ (zero, one), (one, one)\ \}$$
$$graph(F^3(\perp)) = \{\ (zero, one), (one, one), (two, one)\ \}$$

You should be able to construct these graphs yourself and verify the results. For all $i \geq 0$, $graph(F^{i+1}(\perp)) = \{\ (zero, one), (one, one), \cdots, (i, one)\ \}$, implying $(fix\,F) = \lambda n.\ one$. The recursive specification disguised a very simple function.

6.6.4 Simultaneous Definitions

Some recursive specifications are presented as a collection of mutually recursive specifications. Here is a simple example of an $f: Nat \rightarrow Nat_{\perp}$ and a $g : Nat \rightarrow Nat_{\perp}$ such that each function depends on the other to produce an answer:

$$f = \lambda x.\ x\ equals\ zero \rightarrow g(zero)\ []\ f(g(x\ minus\ one))\ plus\ two$$
$$g = \lambda y.\ y\ equals\ zero \rightarrow zero\ []\ y\ times\ f(y\ minus\ one)$$

Identifier g is free in f's specification and f is free in g's. Attempting to solve f's or g's circularity separately creates a problem, for there is no guarantee that f is defined without a well-defined function for g, and no function for g can be produced without a function for f. This suggests that the least fixed points for f and g be formed simultaneously. We build a functional over *pairs*:

$$F : ((Nat \rightarrow Nat_{\perp}) \times (Nat \rightarrow Nat_{\perp})) \rightarrow ((Nat \rightarrow Nat_{\perp}) \times (Nat \rightarrow Nat_{\perp})),$$
$$\begin{aligned} F = \lambda(f,g).\ (\ &\lambda x.\ x\ equals\ zero \rightarrow g(zero)\ []\ f(g(x\ minus\ one))\ plus\ two, \\ &\lambda y.\ y\ equals\ zero \rightarrow zero\ []\ y\ times\ f(y\ minus\ one)\ ) \end{aligned}$$

A pair of functions (α, β) is sought such that $F(\alpha, \beta) = (\alpha, \beta)$.

We build a chain of function pairs. Here are the graphs that are produced by the chain construction (note that $\perp$ stands for the pair $((\lambda n. \perp), (\lambda n. \perp))$):

$$\begin{aligned}
F^0(\perp) &= (\,\{\,\},\ \{\,\}\,)\\
F^1(\perp) &= (\,\{\,\},\ \{\,(zero, zero)\,\}\,)\\
F^2(\perp) &= (\,\{\,(zero, zero)\,\},\ \{\,(zero, zero)\,\}\,)\\
F^3(\perp) &= (\,\{\,(zero, zero), (one, two)\,\},\ \{\,(zero, zero), (one, zero)\,\}\,)\\
F^4(\perp) &= (\,\{\,(zero, zero), (one, two), (two, two)\,\},\\
&\qquad \{\,(zero, zero), (one, zero), (two, four)\,\}\,)\\
F^5(\perp) &= (\,\{\,(zero, zero), (one, two), (two, two)\,\},\\
&\qquad \{\,(zero, zero), (one, zero), (two, four), (three, six)\,\}\,)
\end{aligned}$$

At this point, the sequence converges: for all $i > 5$, $F^i(\perp) = F^5(\perp)$. The solution to the mutually recursive pair of specifications is $\mathit{fix}\,F$, and $f = \mathit{fst}(\mathit{fix}\,F)$ and $g = \mathit{snd}(\mathit{fix}\,F)$.

Any finite set of mutually recursive function definitions can be handled in this manner. Thus, the least fixed point method is powerful enough to model the most general forms of computation, such as general recursive equation sets and flowcharts.

6.6.5 The While-Loop

Recall that a specification of the semantics of a **while**-loop is:

$$\mathbf{C}[\![\mathbf{while}\ \mathrm{B}\ \mathbf{do}\ \mathrm{C}]\!] = \underline{\lambda} s.\ \mathbf{B}[\![\mathrm{B}]\!]s \rightarrow \mathbf{C}[\![\mathbf{while}\ \mathrm{B}\ \mathbf{do}\ \mathrm{C}]\!]\,(\mathbf{C}[\![\mathrm{C}]\!]s)\ []\ s$$

The definition is restated in terms of the *fix* operation as:

$$\mathbf{C}[\![\mathbf{while}\ \mathrm{B}\ \mathbf{do}\ \mathrm{C}]\!] = \mathit{fix}(\lambda f.\underline{\lambda} s.\ \mathbf{B}[\![\mathrm{B}]\!]s \rightarrow f(\mathbf{C}[\![\mathrm{C}]\!]s)\ []\ s)$$

The functional used in this definition is interesting because it has functionality $\mathit{Store}_\perp \rightarrow \mathit{Store}_\perp$, where $\mathit{Store} = \mathit{Id} \rightarrow \mathit{Nat}$.

Let's consider the meaning of the sample loop command $\mathbf{C}[\![\mathbf{while}\ \mathrm{A}>0\ \mathbf{do}\ (\mathrm{A}{:=}\mathrm{A}-1;\ \mathrm{B}{:=}\mathrm{B}+1)]\!]$. We let $\mathit{test} = \mathbf{B}[\![\mathrm{A}>0]\!]$ and $\mathit{adjust} = \mathbf{C}[\![\mathrm{A}{:=}\mathrm{A}-1;\ \mathrm{B}{:=}\mathrm{B}+1]\!]$. The functional is:

$$F = \lambda f.\underline{\lambda} s.\ \mathit{test}\ s \rightarrow f(\mathit{adjust}\ s)\ []\ s$$

As is our custom, we work through the first few steps of the chain construction, showing the graphs produced at each step:

$$\begin{aligned}
\mathit{graph}(F^0(\perp)) &= \{\ \}\\
\mathit{graph}(F^1(\perp)) &= \{\ (\,\{\,([\![\mathrm{A}]\!], zero), ([\![\mathrm{B}]\!], zero), \cdots\,\},\\
&\qquad \{\,([\![\mathrm{A}]\!], zero), ([\![\mathrm{B}]\!], zero), \cdots\,\}\,), \cdots,\\
&\qquad (\,\{\,([\![\mathrm{A}]\!], zero), ([\![\mathrm{B}]\!], four), \cdots\,\},\\
&\qquad \{\,([\![\mathrm{A}]\!], zero), ([\![\mathrm{B}]\!], four), \cdots\,\}\,), \cdots\ \}.
\end{aligned}$$

The graph for $F^1(\perp)$ is worth studying carefully. Since the result is a member of $Store_\perp \rightarrow Store_\perp$, the graph contains pairs of function graphs. Each pair shows a store prior to its ‘‘loop entry’’ and the store after ‘‘loop exit.’’ The members shown in the graph at this step are those stores whose [[A]] value equals *zero*. Thus, those stores that already map [[A]] to *zero* fail the test upon loop entry and exit immediately. The store is left unchanged. Those stores that require loop processing are mapped to $\perp$:

$$
\begin{aligned}
& graph(F^2(\perp)) = \\
& \quad \{ (\{ ([\![A]\!], zero), ([\![B]\!], zero), \cdots \}, \{ ([\![A]\!], zero), ([\![B]\!], zero), \cdots \}), \cdots, \\
& \quad (\{ ([\![A]\!], zero), ([\![B]\!], four), \cdots \}, \{ ([\![A]\!], zero), ([\![B]\!], four)), \cdots \}), \cdots, \\
& \quad (\{ ([\![A]\!], one), ([\![B]\!], zero), \cdots \}, \{ ([\![A]\!], zero), ([\![B]\!], one), \cdots \}), \cdots, \\
& \quad (\{ ([\![A]\!], one), ([\![B]\!], four), \cdots \}, \{ ([\![A]\!], zero), ([\![B]\!], five), \cdots \}), \cdots \}.
\end{aligned}
$$

Those input stores that require one or fewer iterations to process appear in the graph. For example, the fourth illustrated pair denotes a store that has [[A]] set to *one* and [[B]] set to *four* upon loop entry. Only one iteration is needed to reduce [[A]] down to *zero*, the condition for loop exit. In the process [[B]] is incremented to *five*:

$$
\begin{aligned}
& graph(F^3(\perp)) = \\
& \quad \{ (\{ ([\![A]\!], zero), ([\![B]\!], zero), \cdots \}, \{ ([\![A]\!], zero), ([\![B]\!], zero), \cdots \}), \cdots, \\
& \quad (\{ ([\![A]\!], zero), ([\![B]\!], four), \cdots \}, \{ ([\![A]\!], zero), ([\![B]\!], four), \cdots \}), \cdots, \\
& \quad (\{ ([\![A]\!], one), ([\![B]\!], zero), \cdots \}, \{ ([\![A]\!], zero), ([\![B]\!], one), \cdots \}), \cdots, \\
& \quad (\{ ([\![A]\!], one), ([\![B]\!], four), \cdots \}, \{ ([\![A]\!], zero), ([\![B]\!], five), \cdots \}), \\
& \quad (\{ ([\![A]\!], two), ([\![B]\!], zero), \cdots \}, \{ ([\![A]\!], zero), ([\![B]\!], two), \cdots \}), \cdots, \\
& \quad (\{ ([\![A]\!], two), ([\![B]\!], four), \cdots \}, \{ ([\![A]\!], zero), ([\![B]\!], six), \cdots \}), \cdots \}.
\end{aligned}
$$

All stores that require two iterations or less for processing are included in the graph. The graph of $F^{i+1}(\perp)$ contains those pairs whose input stores finish processing in i iterations or less. The least fixed point of the functional contains mappings for those stores that conclude their loop processing in a finite number of iterations.

The **while**-loop’s semantics makes a good example for restating the important principle of least fixed point semantics: the meaning of a recursive specification is totally determined by the meanings of its finite subfunctions. Each subfunction can be represented nonrecursively in the function notation. In this case:

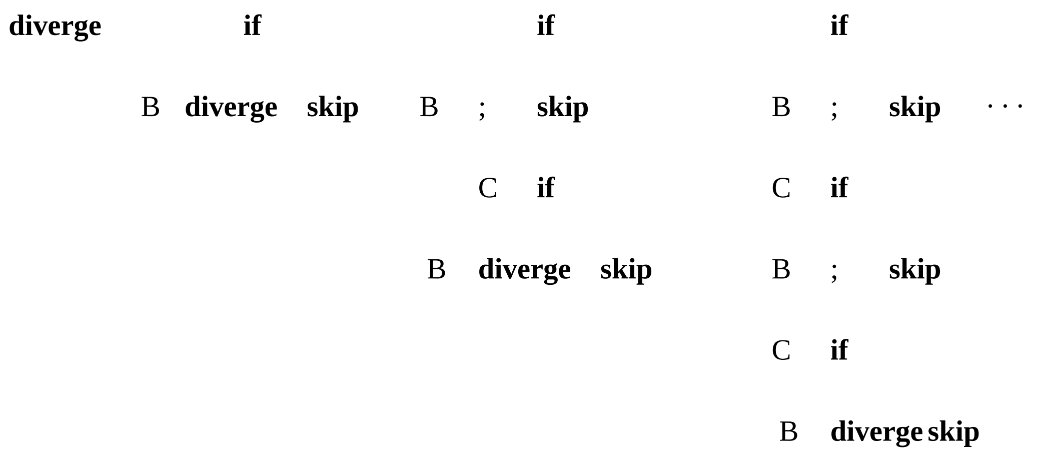
$$
\begin{aligned}
\mathbf{C}[\![\textbf{while}\ B\ \textbf{do}\ C]\!] = \bigsqcup \{ & \underline{\lambda} s. \perp, \\
& \underline{\lambda} s. \mathbf{B}[\![B]\!]s \rightarrow \perp [\!]\ s, \\
& \underline{\lambda} s. \mathbf{B}[\![B]\!]s \rightarrow (\mathbf{B}[\![B]\!](\mathbf{C}[\![C]\!]s) \rightarrow \perp [\!]\ \mathbf{C}[\![C]\!]s)\ [\!]\ s, \\
& \underline{\lambda} s. \mathbf{B}[\![B]\!]s \rightarrow (\mathbf{B}[\![B]\!](\mathbf{C}[\![C]\!]s) \rightarrow \\
& \qquad\qquad (\mathbf{B}[\![B]\!](\mathbf{C}[\![C]\!](\mathbf{C}[\![C]\!]s)) \rightarrow \perp [\!]\ \mathbf{C}[\![C]\!](\mathbf{C}[\![C]\!]s)) \\
& \qquad\qquad [\!]\ \mathbf{C}[\![C]\!]s) \\
& \qquad [\!]\ s, \quad \cdots \}
\end{aligned}
$$

The family of expressions makes apparent that iteration is an unwinding of a loop body;

this corresponds to the operational view. Can we restate this idea even more directly? Recall that **C**⟦**diverge**⟧ = $\underline{\lambda}s.\,\bot$. Substituting the commands into the set just constructed gives us:

C⟦**while** B **do** C⟧ = $\bigsqcup$ { **C**⟦**diverge**⟧,
  C⟦**if** B **then diverge else skip**⟧,
  C⟦**if** B **then** (C; **if** B **then diverge else skip**) **else skip**⟧,
  C⟦**if** B **then** (C; **if** B **then**
    (C; **if** B **then diverge else skip**)
   else skip) **else skip**⟧, $\cdots$ }

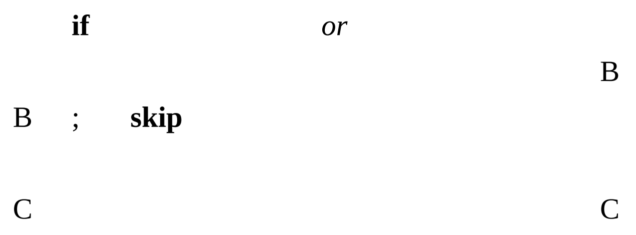
A family of finite noniterative programs represents the loop. It is easier to see what is happening by drawing the abstract syntax trees:



At each stage, the finite tree becomes larger and better defined. The obvious thing to do is to place a partial ordering upon the trees: for all commands C, **diverge** $\sqsubseteq$ C, and for commands C_1 and C_2, $C_1 \sqsubseteq C_2$ iff C_1 and C_2 are the same command type (have the same root node) and all subtrees in C_1 are less defined than the corresponding trees in C_2. This makes families of trees like the one above into chains. What is the lub of such a chain? It is the infinite tree corresponding to:

if B **then** (C; **if** B **then** (C; **if** B **then** (C; $\cdots$) **else skip**) **else skip**) **else skip**

Draw this tree, and define L = **if** B **then** (C; L) **else skip**. The **while**-loop example has led researchers to study languages that contain infinite programs that are represented by recursive definitions, such as L. The goal of such studies is to determine the semantics of recursive and iterative constructs by studying their circularity at the syntax level. The fundamental discovery of this research is that, whether the recursion is handled at the syntax level or at the semantics level, the result is the same: **C**⟦**while** B **do** C⟧ = **C**⟦L⟧. An introduction to this approach is found in Guessarian (1981). We stay with resolving circularity in the semantics, due to our large investment in the existing forms of abstract syntax, structural induction, and least fixed point semantics. Finally, the infinite tree L is abbreviated:



Every flowchart loop can be read as an abbreviation for an infinite program. This brings us back to representations of functions again, for the use of finite loops to represent infinite flowcharts parallels the use of finite function expressions to denote infinite objects— functions. The central issue of computability theory might be stated as the search for finite representations of infinite objects.

6.6.6 Soundness of Hoare's Logic

The facts that we uncovered in the previous example come to good use in a proof of soundness of Hoare's logic for a **while**-loop language. Hoare's logic is an axiomatic semantics, where axioms and inference rules specify the behavior of language constructs. In Hoare's logic, a behavioral property of a command $[\![C]\!]$ is expressed as a proposition $P\{C\}Q$. P and Q are Boolean expressions describing properties of the program variables used in $[\![C]\!]$. Informally interpreted, the proposition says ''if P holds true prior to the evaluation of C and if C terminates, then Q holds true after the evaluation of C.'' A formal interpretation of the proposition using denotational semantics is: ''$P\{C\}Q$ is valid iff for all $s \in Store$, $\mathbf{B}[\![P]\!]s = true$ and $\mathbf{C}[\![C]\!]s \neq \bot$ imply $\mathbf{B}[\![Q]\!](\mathbf{C}[\![C]\!]s) = true$.''

Figure 6.1 presents Hoare's logic for commands in Figure 5.2 augmented by the **while**-loop. Lack of space prevents us from specifying some example prop-ositions and performing their verification. But we will show that the axiom and rules in the figure are *sound*; that is, if the antecedent propositions to a rule are valid, then the consequent of the rule is also valid. The significance of the soundness proof is that the axiom and rules are more than empty definitions— they are theorems with respect to the language's denotational semantics. Thus, the axiomatic definition is complementary to the denotational one in the sense described in the Introduction.

Here is the proof of soundness:

1. *Axiom a*: For arbitrary Boolean expressions P and Q, identifier $[\![x]\!]$, and expression $[\![E]\!]$, we must show that $\mathbf{B}[\![[E/x]P]\!]s = true$ and $\mathbf{C}[\![x\text{:=}E]\!]s \neq \bot$ imply that $\mathbf{B}[\![P]\!][\,[\![x]\!] \mapsto \mathbf{E}[\![E]\!]s]s = true$. But this claim was proved in Exercise 5 of Chapter 5.
2. *Rule b*: We assume each of the three antecedents of the rule to be valid. To show the consequent, we are allowed to assume $\mathbf{B}[\![P]\!]s = true$ and $\mathbf{C}[\![C_1;C_2]\!]s \neq \bot$; we must prove that

Figure 6.1

a) $[\mathrm{E}/x]P\{\mathrm{x{:=}E}\}P$

b) $$\frac{P\{\mathrm{C}_1\}Q,\ \ Q\ implies\ R,\ \ R\{\mathrm{C}_2\}S}{P\{\mathrm{C}_1;\mathrm{C}_2\}S}$$

c) $$\frac{\mathrm{B}\ and\ P\{\mathrm{C}_1\}Q,\ \ (not\ \mathrm{B})\ and\ P\{\mathrm{C}_2\}Q}{P\{\textbf{if}\ \mathrm{B}\ \textbf{then}\ \mathrm{C}_1\ \textbf{else}\ \mathrm{C}_2\}Q}$$

d) $$\frac{P\ and\ \mathrm{B}\{\mathrm{C}\}P}{P\{\textbf{while}\ \mathrm{B}\ \textbf{do}\ \mathrm{C}\}(not\ \mathrm{B})\ and\ P}$$

$\mathbf{B}\llbracket S\rrbracket(\mathbf{C}\llbracket \mathrm{C}_1;\mathrm{C}_2\rrbracket s) = true$. First, note that $\mathbf{C}\llbracket \mathrm{C}_1;\mathrm{C}_2\rrbracket s \neq \bot$ implies that both $\mathbf{C}\llbracket \mathrm{C}_1\rrbracket s \neq \bot$ and $\mathbf{C}\llbracket \mathrm{C}_2\rrbracket(\mathbf{C}\llbracket \mathrm{C}_1\rrbracket s) \neq \bot$. By the validity of $P\{\mathrm{C}_1\}Q$ and $\mathbf{C}\llbracket \mathrm{C}_1\rrbracket s \neq \bot$, we have $\mathbf{B}\llbracket Q\rrbracket(\mathbf{C}\llbracket \mathrm{C}_1\rrbracket s) = true$. From $Q\ implies\ R$, we obtain $\mathbf{B}\llbracket R\rrbracket(\mathbf{C}\llbracket \mathrm{C}_1\rrbracket s) = true$. Finally, by the validity of $R\{\mathrm{C}_2\}S$ and $\mathbf{C}\llbracket \mathrm{C}_2\rrbracket(\mathbf{C}\llbracket \mathrm{C}_1\rrbracket s) \neq \bot$, we get $\mathbf{B}\llbracket S\rrbracket(\mathbf{C}\llbracket \mathrm{C}_2\rrbracket(\mathbf{C}\llbracket \mathrm{C}_1\rrbracket s)) = \mathbf{B}\llbracket S\rrbracket(\mathbf{C}\llbracket \mathrm{C}_1;\mathrm{C}_2\rrbracket s) = true$.

3. *Rule c*: The proof is straightforward and is left as an exercise.
4. *Rule d*: We assume the antecedent $\mathrm{B}\ and\ P\{\mathrm{C}\}P$ is valid. We also assume that $\mathbf{B}\llbracket P\rrbracket s = true$ and $\mathbf{C}\llbracket \textbf{while}\ \mathrm{B}\ \textbf{do}\ \mathrm{C}\rrbracket s \neq \bot$. We must show $\mathbf{B}\llbracket (not\ \mathrm{B})\ and\ P\rrbracket(\mathbf{C}\llbracket \textbf{while}\ \mathrm{B}\ \textbf{do}\ \mathrm{C}\rrbracket s) = true$. The following abbreviations will prove helpful to the proof:

$$F = \lambda f.\underline{\lambda}s.\ \mathbf{B}\llbracket \mathrm{B}\rrbracket s \rightarrow f(\mathbf{C}\llbracket \mathrm{C}\rrbracket s)\ []\ s$$
$$F_i = (F \circ F \circ \cdots \circ F)(\underline{\lambda}s.\ \bot),\ \ F \text{ repeated } i \text{ times}$$
$$\mathbf{C}\llbracket \mathrm{C}\rrbracket^i = \mathbf{C}\llbracket \mathrm{C}\rrbracket \circ \mathbf{C}\llbracket \mathrm{C}\rrbracket \circ \cdots \circ \mathbf{C}\llbracket \mathrm{C}\rrbracket,\ \ \mathbf{C}\llbracket \mathrm{C}\rrbracket \text{ repeated } i \text{ times}$$

Since $\mathbf{C}\llbracket \textbf{while}\ \mathrm{B}\ \textbf{do}\ \mathrm{C}\rrbracket s = \bigsqcup\{F_i(s) \mid i \geq 0\} \neq \bot$, there must exist some $k \geq 0$ such that $F_k(s) \neq \bot$. Pick the least such k that satisfies this property. (There must be a least one, for the F_i's form a chain.) Further, k is greater than *zero*, for F_0 is $(\underline{\lambda}s.\ \bot)$. Hence, $F_k(s) = \mathbf{B}\llbracket \mathrm{B}\rrbracket s \rightarrow F_{k-1}(\mathbf{C}\llbracket \mathrm{C}\rrbracket s)\ []\ s$. If $\mathbf{B}\llbracket \mathrm{B}\rrbracket s = false$, then k must be *one*; if $\mathbf{B}\llbracket \mathrm{B}\rrbracket s = true$, then $F_k(s) = F_{k-1}(\mathbf{C}\llbracket \mathrm{C}\rrbracket s)$. This line of reasoning generalizes to the claim that $F_k(s) = F_1(\mathbf{C}\llbracket \mathrm{C}\rrbracket^{k-1} s)$, which holds because F_k is the least member of the F_i's that maps s to a non-$\bot$ value.

Let $s^* = \mathbf{C}\llbracket \mathrm{C}\rrbracket^{k-1} s$. Now $F_1(s^*) = \mathbf{B}\llbracket \mathrm{B}\rrbracket s^* \rightarrow F_0(s^*)\ []\ s^*$. Clearly, $\mathbf{B}\llbracket \mathrm{B}\rrbracket s^*$ must be *false* and $F_1(s^*) = s^*$, else $F_k(s) = F_1(s^*) = F_0(\mathbf{C}\llbracket \mathrm{C}\rrbracket s^*) = (\underline{\lambda}s.\ \bot)(\mathbf{C}\llbracket \mathrm{C}\rrbracket^k s) = \bot$, which would be a contradiction. So we have that $\mathbf{B}\llbracket \mathrm{B}\rrbracket s^* = \mathbf{B}\llbracket \mathrm{B}\rrbracket(F_k\ s) = false$.

Now consider $\mathbf{B}\llbracket P\rrbracket s$: by assumption, it has value *true*. By using the assumption that $B\ and\ P\{\mathrm{C}\}P$ is valid, we can use reasoning like that taken in the previous paragraph to prove for $0 \le i \le k$ that $\mathbf{B}[\![P]\!](\mathbf{C}[\![\mathrm{C}]\!]^i\, s) = \mathit{true}$. Hence, $\mathbf{B}[\![(\mathit{not}\,\mathrm{B})\ \mathit{and}\, P]\!]s^* = \mathbf{B}[\![(\mathit{not}\,\mathrm{B})\ \mathit{and}\, P]\!](F_k\, s) = \mathit{true}$.

Now we complete the proof. We know that $F_k \sqsubseteq \mathit{fix}\, F = \mathbf{C}[\![\mathbf{while}\ \mathrm{B}\ \mathbf{do}\ \mathrm{C}]\!]$. Hence, $F_k(s) \sqsubseteq \mathbf{C}[\![\mathbf{while}\ \mathrm{B}\ \mathbf{do}\ \mathrm{C}]\!]s$, and $\mathit{true} = \mathbf{B}[\![(\mathit{not}\,\mathrm{B})\ \mathit{and}\, P]\!](F_k\, s) \sqsubseteq \mathbf{B}[\![(\mathit{not}\,\mathrm{B})\ \mathit{and}\, P]\!](\mathbf{C}[\![\mathbf{while}\ \mathrm{B}\ \mathbf{do}\ \mathrm{C}]\!]s)$ by the monotonicity of $\mathbf{B}[\![(\mathit{not}\,\mathrm{B})\ \mathit{and}\, P]\!]$. But this implies that $\mathbf{B}[\![(\mathit{not}\,\mathrm{B})\ \mathit{and}\, P]\!](\mathbf{C}[\![\mathbf{while}\ \mathrm{B}\ \mathbf{do}\ \mathrm{C}]\!]s) = \mathit{true}$, because $\mathbb{B}$ is discretely ordered.

6.7 REASONING ABOUT LEAST FIXED POINTS

We use mathematical induction to reason about numbers and structural induction to reason about derivation trees. An induction principle is useful for reasoning about recursively specified functions. Since the meaning of a recursively specified function is the limit of the meanings of its finite subfunctions, the *fixed point induction principle* proves a property about a recursively defined function by proving it for its finite subfunctions: if all the subfunctions have the property, then the least fixed point must have it as well.

We begin by formalizing the notion of ‘‘property’’ as an *inclusive predicate.* A predicate P is a (not necessarily continuous) function from a domain D to $\mathbb{B}$.

6.25 Definition:

A predicate $P: D \to \mathbb{B}$ *is inclusive iff for every chain* $C \subseteq D$, *if* $P(c) = \mathit{true}$ *for every* $c \in C$, *then* $P(\bigsqcup C) = \mathit{true}$ *also.*

We only work with inclusive predicates.

Say that for a recursively defined function $f: A \to B$, $f = F(f)$, we wish to show that $P(f)$ holds true. If P is an inclusive predicate, then we can use the fact that $f = \mathit{fix}\, F$ as follows:

1. Show that $P(\bot)$ holds, for $\bot \in A \to B$.
2. Assuming for arbitrary $i \ge 0$, that $P(F^i(\bot))$ holds, show that $P(F^{i+1}(\bot))$ holds.

Mathematical induction guarantees that for all $n \ge 0$, $P(F^n(\bot))$ holds. Since P is inclusive, we have that $P(\mathit{fix}\, F)$ holds.

A more elegant form of this reasoning is fixed point induction.

6.26 Definition:

The fixed point induction principle: For a pointed cpo D, a continuous functional $F: D \to D$, *and an inclusive predicate* $P: D \to \mathbb{B}$, *if:*

1. $P(\bot)$ *holds.*
2. *For arbitrary* $d \in D$, *when* $P(d)$ *holds, then* $P(F(d))$ *holds.*

then $P(\mathit{fix}\, F)$ *holds as well.*

The proof of the validity of the fixed point induction principle is left as an exercise.

Fixed point induction helps us to show that the g function defined in the example of Section 6.6.3 is indeed the constant function that maps its arguments to *one.*

6.27 Proposition:

For all $n \in Nat$, $g(n) \neq \bot$ implies $g(n) = one$.

Proof: We use fixed point induction on the predicate $P(f)$ = ''for all $n \in Nat$, $f(n) \neq \bot$ implies $f(n) = one$. '' For the basis step, we must show that $P(\lambda m. \bot)$ holds. For an arbitrary $n \in Nat$, $(\lambda m. \bot)(n) = \bot$, so the claim is vacuously satisfied. For the inductive step, we assume the inductive hypothesis $P(f)$ and show $P(F(f))$. For an arbitrary $n \in Nat$, we must consider the following two cases to determine the value of $F(f)(n)$:

1. n equals *zero*: then $F(f)(n) = one$, and this satisfies the claim.
2. n is greater than *zero*: then $F(f)(n) = (f(n\ minus\ one)\ plus\ f(n\ minus\ one))\ minus\ one$. Consider the value of $f(n\ minus\ one)$:
   a. If it is $\bot$, then since *plus* and *minus* are strict, $F(f)(n) = \bot$, and the claim is vacuously satisifed.
   b. If it is non-$\bot$, then by the inductive hypothesis $f(n\ minus\ one) = one$, and simple arithmetic shows that $F(f)(n) = one$. $\square$

How do we know that the predicate we just used is inclusive? It is difficult to see how the logical connectives ''for all,'' ''implies,'' ''and,'' ''$\neq$,'' etc., all fit together to make an inclusive predicate. The following criterion is helpful.

6.28 Proposition:

(Manna, Ness, & Vuillemin, 1972) Let f be a recursively defined function that we wish to reason about. A logical assertion P is an inclusive predicate if P has the form:

$$\text{for all } d_1 \in D_1, \cdots, d_m \in D_m,\ \mathop{\mathbf{AND}}_{j=1}^{n} (\mathop{\mathbf{OR}}_{k=1}^{p} Q_{jk})$$

for $m \geq 0$, $n \geq 0$, $p \geq 0$, where Q_{jk} can be either:

1. *A predicate using only $d_1, \cdots, d_m$ as free identifiers.*
2. *An expression of form $E_1 \sqsubseteq E_2$, where E_1 and E_2 are function expressions using only $f, d_1, \cdots, d_m$ as free identifiers.*

So an inclusive predicate can be a universally quantified conjunction of disjunctions. Here is an example: ''for all $n \in Nat$, $(\bot \sqsubseteq f(n))$ and $(n\ equals\ one$ or $f(n) \sqsubseteq zero)$.'' By using the following logical equivalences (among others) we can show that a large variety of predicates are inclusive:

$(E_1 \sqsubseteq E_2)$ and $(E_2 \sqsubseteq E_1)$ iff $(E_1 = E_2)$
$(P_1$ implies $P_2)$ iff $((\text{not}\ P_1)$ or $P_2)$
$(\text{not}(\text{not}\ P_1))$ iff P_1
$(P_1$ or $(P_2$ and $P_3))$ iff $((P_1$ or $P_2)$ and $(P_1$ or $P_3))$

For example, the predicate used in the proof of Proposition 6.27 is inclusive because we can use the logical equivalences just mentioned to place the predicate in the form specified in Proposition 6.28. The proof of Proposition 6.27 was conducted with the predicate in its original form, but it's important to verify that the predicate is convertible to an inclusive one.

Fixed point induction is primarily useful for showing equivalences of program constructs. Here is one example: we define the semantics of a **repeat**-loop to be:

$$\mathbf{C}[\![\textbf{repeat}\ \mathrm{C}\ \textbf{until}\ \mathrm{B}]\!] = \mathit{fix}(\lambda f.\underline{\lambda} s.\ \text{let}\ s' = \mathbf{C}[\![\mathrm{C}]\!]s\ \text{in}\ \ \mathbf{B}[\![\mathrm{B}]\!]s' \rightarrow s'\ [\!]\ (f\,s'))$$

6.29 Proposition:

For any command $[\![\mathrm{C}]\!]$ *and Boolean expression* $[\![\mathrm{B}]\!]$, $\mathbf{C}[\![\mathrm{C};\textbf{while}\ \neg\mathrm{B}\ \textbf{do}\ \mathrm{C}]\!] = \mathbf{C}[\![\textbf{repeat}\ \mathrm{C}\ \textbf{until}\ \mathrm{B}]\!]$.

Proof: The fixed point induction must be performed over the two recursive definitions simultaneously. The predicate we use is:

$$P(f, g) = \text{``for all } s \in \mathit{Store}_\perp,\ \ f(\mathbf{C}[\![\mathrm{C}]\!]s) = (g\,s)\text{''}$$

For the basis step, we must show that $P((\lambda s.\perp), (\lambda s.\perp))$ holds, but this is obvious. For the inductive step, the inductive hypothesis is $P(f, g)$, and we must show that $P(F(f), G(g))$ holds, where:

$$F = (\lambda f.\underline{\lambda} s.\ \mathbf{B}[\![\neg\mathrm{B}]\!]s \rightarrow f(\mathbf{C}[\![\mathrm{C}]\!]s)\ [\!]\ s)$$
$$G = (\lambda f.\underline{\lambda} s.\ \text{let}\ s' = \mathbf{C}[\![\mathrm{C}]\!]s\ \text{in}\ \mathbf{B}[\![\mathrm{B}]\!]s' \rightarrow s'\ [\!]\ \ (f\,s'))$$

For an arbitrary $s \in \mathit{Store}_\perp$, if $s = \perp$, then $F(f)(\mathbf{C}[\![\mathrm{C}]\!]\perp) = \perp = G(g)(\perp)$, because $\mathbf{C}[\![\mathrm{C}]\!]$, $F(f)$, and $G(g)$ are all strict. (The fact that $\mathbf{C}[\![\mathrm{C}]\!]$ is strict for any $\mathrm{C} \in \mathrm{Command}$ requires a small, separate proof and is left as an exercise.) On the other hand, if $s \neq \perp$, then consider the value of $\mathbf{C}[\![\mathrm{C}]\!]s$. If it is $\perp$, then again $F(f)(\perp) = \perp = (\text{let}\ s' = \perp\ \text{in}\ \mathbf{B}[\![\mathrm{B}]\!]s' \rightarrow s'\ [\!]\ (g\,s'))) = G(g)(\perp)$. So say that $\mathbf{C}[\![\mathrm{C}]\!]s$ is some defined store s_0. We have that:

$$F(f)(s_0) = \mathbf{B}[\![\neg\mathrm{B}]\!]s_0 \rightarrow f(\mathbf{C}[\![\mathrm{C}]\!]s_0)\ [\!]\ s_0\ \ = \mathbf{B}[\![\mathrm{B}]\!]s_0 \rightarrow s_0\ [\!]\ f(\mathbf{C}[\![\mathrm{C}]\!]s_0)$$

and

$$G(g)(s) = \mathbf{B}[\![\mathrm{B}]\!]s_0 \rightarrow s_0\ [\!]\ (g\,s_0)$$

By the inductive hypothesis, it follows that $f(\mathbf{C}[\![\mathrm{C}]\!]s_0) = (g\,s_0)$. This implies that $F(f)(s_0) = G(g)(s)$, which completes the proof. □

SUGGESTED READINGS

Least fixed points: Bird 1976; Guessarian 1981; Kleene, 1952; Park 1969; Manna 1974; Manna, Ness, & Vuillemin 1972; Manna & Vuillemin 1972; Rogers 1967

Complete partial orderings: Birkhoff 1967; Gierz, et al. 1980; Kamimura & Tang 1983, 1984a; Milner 1976; Plotkin 1982; Reynolds 1977; Scott 1976, 1980a, 1982a; Wadsworth 1978

EXERCISES

1. Simplify the following expressions to answer forms or explain at some point in the simplification why no final answer will ever be found:

   a. $f(three)$, for f defined in Section 6.6.4.
   b. $f(four)$, for f defined in Section 6.6.4.
   c. $\mathbf{C}[\![$**while** X> 0 **do** (Y:=X; X:=X−1)$]\!]s_0$, for $s_0 = [\,[\![X]\!] \mapsto two\,]newstore$, for the **while**-loop defined in Section 6.6.5.
   d. $\mathbf{C}[\![$**while** X> 0 **do** Y:=X$]\!]newstore$, for the **while**-loop defined in Section 6.6.5.

2. For each of the recursively defined functions that follow:

   i. Build the functional F associated with the definition.
   ii. Show the (graph) set representations of $F^0(\emptyset)$, $F^1(\emptyset)$, $F^2(\emptyset)$, and $F^i(\emptyset)$, for $\emptyset = \lambda n.\, \bot$.
   iii. Define $\bigcup_{i=0}^{\infty} graph(F^i(\emptyset))$.
   iv. Attempt to give a nonrecursively defined function whose denotation is the value in part iii.

   a. $f : Nat \times Nat \rightarrow Nat_\bot$, $\;f(m,n) = m\ equals\ zero \rightarrow n\ [\!]\ one\ plus\ f(m\ minus\ one,\ n)$
   b. $g : Nat_\bot \rightarrow Nat_\bot$, $\;g = \underline{\lambda} n.\ n\ plus\ (n\ equals\ zero \rightarrow zero\ [\!]\ g(g(n\ minus\ one)))$
   c. (Ackermann's function) $A : Nat \times Nat \rightarrow Nat_\bot$,
      $A(m,n) = m\ equals\ zero \rightarrow n$
      $\quad [\!]\ n\ equals\ zero \rightarrow A(m\ minus\ one,\ one)$
      $\quad [\!]\ A(m\ minus\ one,\ A(m,n))$
   d. $f : Nat \times Nat \rightarrow Nat_\bot$, $g : Nat \rightarrow Tr_\bot$,
      $f(m,n) = g(n) \rightarrow m\ [\!]\ f(m\ minus\ two,\ n\ minus\ two)$
      $g(m) = m\ equals\ zero \rightarrow true\ [\!]\ m\ equals\ one \rightarrow false\ [\!]\ g(m\ minus\ two)$

3. Recall that $newstore = (\lambda i.\ zero)$.

   a. Without using fixed point induction, prove that $\mathbf{C}[\![$**while** A> 0 **do** A:=A−1$]\!]s_0 = newstore$, where $s_0 = [[\![A]\!] \mapsto two\,]newstore$.
   b. Use fixed point induction to show that $\mathbf{C}[\![$**while**A=0 **do** B:=1$]\!]newstore = \bot$.

4. a. For the function g defined in Section 6.6.3, why can't we prove the property ‘‘for all $n \in Nat$, $g(n) = one$’’ using fixed point induction?
   b. Prove the above property using mathematical induction. (Hint: first prove ‘‘for all $n \in Nat$, $(F^{n+1}(\lambda n.\, \bot))(n) = one$.’’)
   c. Based on your experiences in part b, suggest a proof strategy for proving claims of the form: ‘‘for all $n \in Nat$, $f(n) \neq \bot$.’’ Will your strategy generalize to argument domains other than Nat?

5. Use the logical equivalences given in Section 5.4 to formally show that the following predicates are inclusive for $f: Nat \rightarrow Nat_{\perp}$

   a. $P(f) =$ ''for all $n \in Nat$, $f(n) \neq \perp$ implies $f(n) \sqsubseteq zero$''
   b. $P(f, g) =$ ''for all $n \in Nat$, $f(n) \neq \perp$ implies $f(n) = g(n)$''

   Give counterexamples that show that these predicates are *not* inclusive:

   c. $P(f) =$ ''there exists an $n \in Nat$, $f(n) = \perp$''
   d. $P(f) =$ ''$f \neq (\lambda n.n)$,'' for $f: Nat \rightarrow Nat_{\perp}$

6. For the recursively defined function:

   $$factoo(m,n) = m\ equals\ zero \rightarrow n\ [\!]\ factoo(m\ minus\ one,\ m\ times\ n)$$

   prove the property that for all $n \in Nat$, $fac(n) = factoo(n, one)$, where *fac* is defined in Section 6.6.1:

   a. Using fixed point induction.
   b. Using mathematical induction.

   When is it appropriate to use one form of reasoning over the other?

7. For the functional $F: (Nat \rightarrow Nat_{\perp}) \rightarrow (Nat \rightarrow Nat_{\perp})$,

   $$F = \lambda f.\lambda m.\ m\ equals\ zero \rightarrow one\ [\!]\ m\ equals\ one \rightarrow f(m\ plus\ two)\ [\!]\ f(m\ minus\ two)$$

   prove that all of these functions are fixed points of F:

   a. $\lambda n.\ one$
   b. $\lambda n.\ ((n\ mod\ two)\ equals\ zero) \rightarrow one\ [\!]\ two$
   c. $\lambda n.\ ((n\ mod\ two)\ equals\ zero) \rightarrow one\ [\!]\ \perp$

   Which of the three is the least fixed point (if any)?

8. Prove the following properties for all B∈ Boolean-expr and C∈ Command:

   a. **C**⟦**while** B **do** C⟧ = **C**⟦**if** B **then** C; **while** B **do** C **else skip**⟧
   b. **C**⟦**repeat** C **until** B⟧ = **C**⟦C; **if** B **then skip else repeat** C **until** B⟧
   c. **C**⟦**while** B **do** C⟧ = **C**⟦(**while** B **do** C); **if** B **then** C⟧
   d. **C**⟦**if** B **then repeat** C **until** ¬B⟧ = **C**⟦**while** B **do** C⟧

   where the **while**-loop is defined in Section 6.6.5, the **repeat**-loop is defined in Section 6.7, and **C**⟦**skip**⟧ = $\underline{\lambda} s.\ s$.

9. Formulate Hoare-style inference rules and prove their soundness for the following commands:

   a. ⟦**if** B **then** C⟧ from Figure 5.2.
   b. ⟦**diverge**⟧ from Figure 5.2.
   c. ⟦**repeat** C **until** B⟧ from Section 6.7.

10. A language designer has proposed a new control construct called **entangle.** It satifies this equality:

$$\mathbf{C}[\![\textbf{entangle}\ \mathrm{B}\ \textbf{in}\ \mathrm{C}]\!] = \mathbf{C}[\![\textbf{if}\ \mathrm{B}\ \textbf{then}\ (\mathrm{C};\ (\textbf{entangle}\ \mathrm{B}\ \textbf{in}\ \mathrm{C});\ \mathrm{C})\ \textbf{else}\ \mathrm{C}]\!]$$

    a. Define a semantics for **entangle.** Prove that the semantics satisfies the above property.
    b. Following the pattern shown in Section 6.6.5, draw out the family of approximate syntax trees for $[\![\textbf{entangle}\ \mathrm{B}\ \textbf{in}\ \mathrm{C}]\!]$.
    c. Comment why the **while-** and **repeat**-loop control constructs are easily implementable on a computer while **entangle** is not.

11. Redraw the semantic domains listed in exercise 4 of Chapter 3, showing the partial ordering on the elements.

12. For cpos D and E, show that for any $f, g : D \to E$, $f \sqsubseteq_{D\to E} g$ iff (for all $a_1, a_2 \in A$, $a_1 \sqsubseteq_A a_2$ implies $f(a_1) \sqsubseteq_B g(a_2)$).

13. Why is $fix\, F : D \to E$ continuous when $F : (D \to E) \to (D \to E)$ is?

14. Show that D^* is a cpo when D is and that its associated operations are continuous.

15. Show that the operation $(\_ \to \_ [\!]\ \_) : Tr \times D \times D \to D$ is continuous.

16. Show that the composition of continuous functions is a continuous function.

17. The cpos D and E are *order isomorphic* iff there exist continuous functions $f\!: D \to E$ and $g : E \to D$ such that $g \circ f = id_D$ and $f \circ g = id_E$. Thus, order isomorphic cpos are in 1-1, onto, order preserving correspondence. Prove that the following pairs of cpos are order isomorphic. (Let A, B, and C be arbitrary cpos.)

    a. $\mathbb{N}$ and $\mathbb{N} + \mathbb{N}$
    b. $\mathbb{N}$ and $\mathbb{N} \times \mathbb{N}$
    c. $A \times B$ and $B \times A$
    d. $A + B$ and $B + A$
    e. $Unit \to A$ and A
    f. $(Unit \times A) \to B$ and $A \to B$
    g. $(A \times B) \to C$ and $A \to (B \to C)$

18. Let A and B be cpos, and let $\prod_{a:A} B$ be the usual A-fold product of B elements. Define $apply : A \times (\prod_{a:A} B) \to B$ to be the indexing operation; that is, $apply(a,p) = p\!\downarrow\! a$.

    a. Show that the product is order isomorphic with the set of *all* functions with domain A and codomain B.
    b. Given that the semantic domain $A \to B$ contains just the *continuous* functions from A

to B, propose a domain construction for the infinite product above that is order isomorphic with $A \to B$.

19. Researchers in semantic domain theory often work with *bounded complete* ω-*algebraic cpos*. Here are the basic definitions and results:

    An element $d \in D$ is *finite* iff for all chains $C \subseteq D$, if $d \sqsubseteq \bigsqcup C$, then there exists some $c \in C$ such that $d \sqsubseteq c$. Let **fin**D be the set of finite elements in D.

    a. Describe the finite elements in:

       i. $\mathbb{N}$
       ii. $\mathbb{P}(\mathbb{N})$, the powerset of $\mathbb{N}$ partially ordered by $\subseteq$
       iii. $\mathbb{N} \to \mathbb{N}_\perp$ (hint: work with the graph representations of the functions)
       iv. $D + E$, where D and E are cpos
       v. $D \times E$, where D and E are cpos
       vi. $D_\perp$, where D is a cpo

    A cpo D is *algebraic* if, for every $d \in D$, there exists a chain $C \subseteq$ **fin**D such that $\bigsqcup C = d$. D is ω-*algebraic* if **fin**D has at most a countably infinite number of elements.

    b. Show that the cpos in parts i through vi of part a are ω-algebraic, when D and E represent ω-algebraic cpos.
    c. Show that $\mathbb{N} \to \mathbb{N}$ is algebraic but is not ω-algebraic.
    d. For algebraic cpos D and E, prove that a monotonic function from **fin**D to **fin**E can be uniquely extended to a continuous function in $D \to E$. Prove that $D \to E$ is order isomorphic to the partially ordered set of monotonic functions from **fin**D to **fin**E. Next, show that a function that maps from algebraic cpo D to algebraic cpo E and is continuous on **fin**D to **fin**E need not be continuous on D to E.

    A cpo D is *bounded complete* if, for all $a, b \in D$, if there exists a $c \in D$ such that $a \sqsubseteq c$ and $b \sqsubseteq c$, then $a \sqcup b$ exists in D.

    e. Show that the following cpos are bounded complete ω-algebraic:

       i. The cpos in parts i through vi in part b, where D and E represent bounded complete ω-algebraic cpos.
       ii. $D \to E$, where D and E are bounded complete ω-algebraic cpos and E is pointed. (Hint: first show that for any $d \in$ **fin**D and $e \in$ **fin**E that the ''step function'' $(\lambda a.\, (d \sqsubseteq a) \to e\ [\!]\ \perp)$ is a finite element in $D \to E$. Then show that any $f: D \to E$ is the lub of a chain whose elements are finite joins of step functions.)

    For a partially ordered set E, a nonempty set $A \subseteq E$ is an *ideal* if:

       i. For all $a, b \in E$, if $a \in A$ and $b \sqsubseteq a$, then $b \in A$.
       ii. For all $a, b \in A$, there exists some $c \in A$ such that $a \sqsubseteq c$ and $b \sqsubseteq c$.

    f. For an algebraic cpo D, let **id**D = *{ A | A*$\subseteq$ **fin**D *and A is an ideal }*. Show that the partially ordered set (**id**D, $\subseteq$) is order isomorphic with D.

20. We gain important insights by applying *topology* to domains. For a cpo D, say that a set

$U \subseteq D$ is a *Scott-open set* iff:

i. It is closed upwards: for all $a \in U$, $b \in D$, if $a \sqsubseteq b$, then $b \in U$.
ii. It contains a lub only if it contains a ''tail'' of the chain: for all chains $C \subseteq D$, if $\bigsqcup C \in U$, then there exists some $c \in C$ such that $c \in U$.

A collection of sets $S \subseteq \mathbb{P}(D)$ is a *topology on D* iff:

i. Both $\{\ \}$ and D belong to S.
ii. For any $R \subseteq S$, $\bigcup R \in S$, that is, S is closed under arbitrary unions.
iii. For any $U, V \in S$, $U \cap V \in S$, that is, S is closed under binary intersections.

a. Show that for any cpo D that the collection of Scott-open sets forms a topology on D.
b. Describe the Scott-open sets for these cpos: Nat, $Tr_\bot$, $Tr_\bot \times Tr_\bot$, $Nat \rightarrow Nat$, $Nat \rightarrow Nat_\bot$.
c. Show that every element $d \in D$ is characterized by a unique collection of Scott-open sets, that is, prove that for all distinct $d, e \in D$, there exists some Scott-open set $U \subseteq D$ such that ($d \in U$ *and* $e \notin U$) or ($e \in U$ *and* $d \notin U$). (Hint: first show that the set $D - \{ e \mid e \sqsubseteq d \}$, for any $d \in D$, is Scott-open; use this result in your proof.)

Part c says that the Scott-open sets have the T_0-*separation property*. Because of this result, we treat the Scott-open sets as ''detectable properties'' of D-elements. Each element is identified by the properties that it satisfies.

d. Show that for all $d, e \in D$, $d \sqsubseteq e$ iff for all open sets $U \subseteq D$, $d \in U$ implies $e \in U$.

A function $f: D \rightarrow E$ is *topologically continuous* iff for all open sets $U \subseteq E$, $f^{-1}(U)$ is open in D. That is, a set of answer values share a property only because their corresponding arguments also share a property.

e. Prove that a function $f: D \rightarrow E$ is chain-continuous (see Definition 6.9) iff it is topologically continuous (with respect to the Scott-topologies on D and E).

21. Section 6.1 is a presentation of Kleene's *first recursion theorem*, which states that the meaning of a recursive specification $f = F(f)$ that maps arguments in $\mathbb{N}$ to answers in $\mathbb{N}$ (or nontermination) is exactly the union of the $F^i(\emptyset)$ approximating functions. The proof of the theorem (see Kleene 1952, or Rogers 1967) doesn't mention continuity or pointed cpos. Why do we require these concepts for Chapter 6?