Chapter 11

Domain Theory III: Recursive Domain Specifications

Several times we have made use of recursively defined domains (also called *reflexive domains*) of the form $D = F(D)$. In this chapter, we study recursively defined domains in detail, because:

1. Recursive definitions are natural descriptions for certain data structures. For example, the definition of binary trees, $Bintree = (Data + (Data \times Bintree \times Bintree))_{\bot}$, clearly states that a binary tree is a leaf of data or two trees joined by a root node of data. Another example is the definition of linear lists of A-elements $Alist = (Nil + (A \times Alist))_{\bot}$, where $Nil = Unit$. The definition describes the internal structure of the lists better than the A^* domain does. *Alist*'s definition also clearly shows why the operations *cons*, *hd*, *tl*, and *null* are essential for assembling and disassembling lists.
2. Recursive definitions are absolutely necessary to model certain programming language features. For example, procedures in ALGOL60 may receive procedures as actual parameters. The domain definition must read $Proc = Param \rightarrow Store \rightarrow Store_{\bot}$, where $Param = Int + Real + \cdots + Proc$, to properly express the range of parameters.

Like the recursively defined functions in Chapter 6, recursively defined domains require special construction. Section 11.1 introduces the construction through an example, Section 11.2 develops the technical machinery, and Section 11.3 presents examples of reflexive domains.

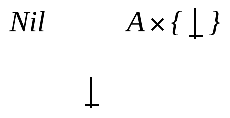
11.1 REFLEXIVE DOMAINS HAVE INFINITE ELEMENTS

We motivated the least fixed point construction in Chapter 6 by treating a recursively defined function f as an operational definition— f's application to an argument a was calculated by recursively unfolding f's definition as needed. If the combination $(f\,a)$ simplified to an answer b in a finite number of unfoldings, the function satisfying the recursive specification mapped a to b as well. We used this idea to develop a sequence of functions that approximated the solution; a sequence member f_i resulted from unfolding f's specification i times. The f_i's formed a chain whose least upper bound was the function satisfying the recursive specification. The key to finding the solution was building the sequence of approximations. A suitable way of combining these approximations was found and the problem was solved.

Similarly, we build a solution to a recursive domain definition by building a sequence of approximating domains. The elements in each approximating domain will be present in the solution domain, and each approximating domain D_i will be a *subdomain* of approximating domain D_{i+1}; that is, the elements and partial ordering structure of D_i are preserved in D_{i+1}.

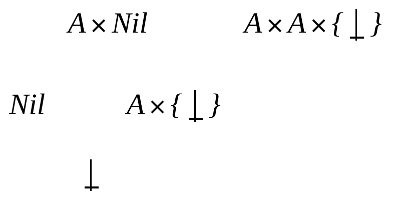
Since semantic domains are nonempty collections, we take domain D_0 to be $\{\,\bot\,\}$. D_0 is a pointed cpo, and in order to preserve the subdomain property, each approximating domain D_i will be a pointed cpo as well. Domain D_{i+1} is built from D_i and the recursive definition.

Let's apply these ideas to $Alist = (Nil + (A \times Alist))_\bot$ as an example. $Nil = Unit$ represents the empty list, and a nonempty list of A-elements has the structure $A \times Alist$. An $Alist$ can also be undefined. Domain $Alist_0 = \{\,\bot\,\}$, and for each $i > 0$, $Alist_{i+1} = (Nil + (A \times Alist_i))_\bot$. To get started, we draw $Alist_1 = (Nil + (A \times Alist_0))_\bot$ as:



because $A \times Alist_0 = A \times \{\,\bot\,\} = \{\,(a, \bot) \mid a \in A\,\}$. (For readability, from here on we will represent a k-element list as $[a_0, a_1, \cdots, a_k]$, omitting the injection tags. Hence, $Alist_1 = \{\,\bot, [nil]\,\} \cup \{\,[a, \bot] \mid a \in A\,\}$.) $Alist_0$ is a subdomain of $Alist_1$, as $\bot \in Alist_0$ embeds to $\bot \in Alist_1$.

Next, $Alist_2 = (Nil + (A \times Alist_1))_\bot$. The product $A \times Alist_1 = A \times (Nil + (A \times Alist_0))_\bot$ can be visualized as a union of three distinct sets of elements: $\{\,[a, \bot] \mid a \in A\,\}$, the set of *partial* lists of one element; $\{\,[a, nil] \mid a \in A\,\}$, the set of *proper* lists of one element, and $\{\,[a_1, a_2, \bot] \mid a_1, a_2 \in A\,\}$, the set of partial lists of two elements. Drawing $A \times Alist_1$ with these sets, we obtain:



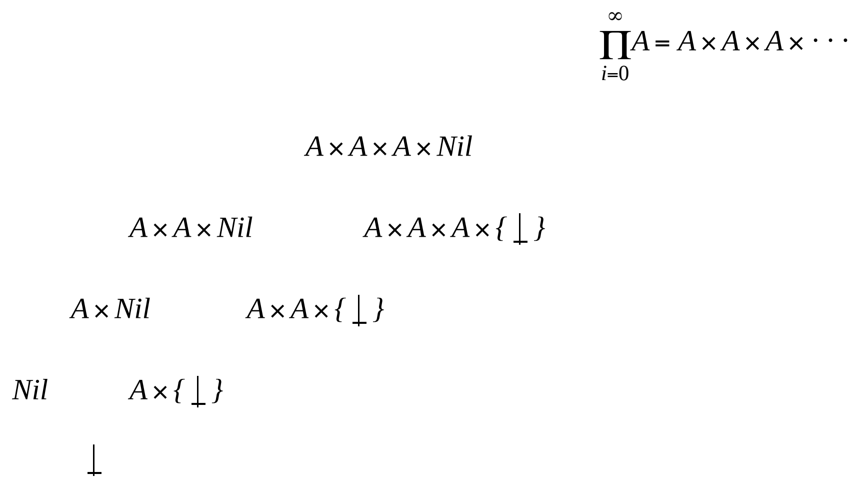
It is easy to see where $Alist_1$ embeds into $Alist_2$— into the lower portion. $Alist_2$ contains elements with more information than those in $Alist_1$.

A pattern is emerging: $Alist_i$ contains $\bot$; nil; proper lists of $(i-1)$ or less A-elements; and partial lists of i A-elements, which are capable of expanding to lists of greater length in the later, larger domains. The element $\bot$ serves double duty: it represents both a nontermination situation and a ''don't know yet'' situation. That is, a list $[a_0, a_1, \bot]$ may be read as the result of a program that generated two output elements and then ''hung up,'' or it may be read as an approximation of a list of length greater than two, where information as to what follows a_1 is not currently available.

What is the limit of the family of domains $Alist_i$? Using the least fixed point construction as inspiration, we might take $Alist_{fin} = \bigcup_{i=0}^{\infty} Alist_i$, partially ordered to be consistent with the $Alist_i$'s. (That is, $x \sqsubseteq_{Alist_{fin}} x'$ iff there exists some $j \geq 0$ such that $x \sqsubseteq_{Alist_j} x'$.) Domain $Alist_{fin}$ contains $\bot$, nil, and all proper lists of finite length. But it also contains all the partial lists! To discard the partial lists would be foolhardy, for partial lists have real semantic value. But they present a problem: $Alist_{fin}$ is not a cpo, for the chain $\bot$, $[a_0, \bot]$, $[a_0, a_1, \bot]$, $\cdots$,

$[a_0, a_1, \cdots, a_i, \perp], \cdots$ does not have a least upper bound in $Alist_{fin}$.

The obvious remedy to the problem is to add the needed upper bounds to the domain. The lub of the aforementioned chain is the *infinite* list $[a_0, a_1, \cdots, a_i, a_{i+1}, \cdots]$. It is easy to see where the infinite lists would be added to the domain. The result, called $Alist_\infty$, is:



$\prod_{i=0}^{\infty} A = A \times A \times A \times \cdots$
$A \times A \times A \times Nil$
$A \times A \times Nil$
$A \times A \times A \times \{\perp\}$
$A \times Nil$
$A \times A \times \{\perp\}$
Nil
$A \times \{\perp\}$
$\perp$

The infinite elements are a boon; realistic computing situations involving infinite data structures are now expressible and understandable. Consider the list l specified by $l = (a\ cons\ l)$, for $a \in A$. The functional $(\lambda l.\ a\ cons\ l): Alist_\infty \rightarrow Alist_\infty$ has as its least fixed point $[a, a, a, \cdots]$, the infinite list of a's. We see that l satisfies the properties $(hd\ l) = a$, $(tl\ l) = l$, and $(null\ l) = false$. The term *lazy list* has been coined for recursively specified lists like l, for when one is used in computation, no attempt is ever made to completely evaluate it to its full length. Instead it is ''lazy''— it produces its next element only when asked (by the disassembly operation *hd*).

$Alist_\infty$ appears to be the solution to the recursive specification. But a formal construction is still needed. The first step is formalizing the notion of subdomain. We introduce a family of continuous functions $\phi_i : Alist_i \rightarrow Alist_{i+1}$ for $i \geq 0$. Each ϕ_i embeds $Alist_i$ into $Alist_{i+1}$. By continuity, the partial ordering and lubs in $Alist_i$ are preserved in $Alist_{i+1}$. However, it is easy to find ϕ_i functions that do an ''embedding'' that is unnatural (e.g., $\phi_i = \lambda x.\ \perp_{Alist_{i+1}}$). To guarantee that the function properly embeds $Alist_i$ into $Alist_{i+1}$, we also define a family of continuous functions $\psi_i : Alist_{i+1} \rightarrow Alist_i$ that map the elements in $Alist_{i+1}$ to those in $Alist_i$ that best approximate them. $Alist_i$ is a subdomain of $Alist_{i+1}$ when $\psi_i \circ \phi_i = id_{Alist_i}$ holds; that is, every element in $Alist_i$ can be embedded by ϕ_i and recovered by ψ_i. To force the embedding of $Alist_i$ into the ''lower portion'' of $Alist_{i+1}$, we also require that $\phi_i \circ \psi_i \sqsubseteq id_{Alist_{i+1}}$. This makes it clear that the new elements in $Alist_{i+1}$ not in $Alist_i$ ''grow out'' of $Alist_i$.

The function pairs (ϕ_i, ψ_i), $i \geq 0$, are generated from the recursive specification. To get started, we define $\phi_0 : Alist_0 \rightarrow Alist_1$ as $(\lambda x.\ \perp_{Alist_1})$ and $\psi_0 : Alist_1 \rightarrow Alist_0$ as $(\lambda x.\ \perp_{Alist_0})$. It is easy to show that the (ϕ_0, ψ_0) pair satisfies the two properties mentioned above. For every $i > 0$:

```
$\phi_i : Alist_i \rightarrow Alist_{i+1} = \underline{\lambda} x.$ cases $x$ of
        is$Nil() \rightarrow$ in$Nil()$
        [] is$A \times Alist_{i-1}(a, l) \rightarrow$ in$A \times Alist_i(a, \phi_{i-1}(l))$ end
```

The embedding is based on the structure of the argument from $Alist_i$. The structures of undefined and empty lists are preserved, and a list with head element $a \in A$ and tail $l \in Alist_{i-1}$ is mapped into a pair $(a, \phi_{i-1}(l)) \in A \times Alist_i$, courtesy of $\phi_{i-1} : Alist_{i-1} \rightarrow Alist_i$. Similarly:

```
$\psi_i : Alist_{i+1} \rightarrow Alist_i = \underline{\lambda} x.$ cases $x$ of
        is$Nil() \rightarrow$ in$Nil()$
        [] is$A \times Alist_i(a, l) \rightarrow$ in$A \times Alist_{i-1}(a, \psi_{i-1}(l))$ end
```

The function converts its argument to its best approximation in $Alist_i$ by analyzing its structure and using ψ_{i-1} where needed. A mathematical induction proof shows that each pair (ϕ_i, ψ_i) satisifies the required properties.

A chain-like sequence has been created:

$$\begin{array}{ccccccccccc} & \psi_0 & & \psi_1 & & & & \psi_i & & \\ Alist_0 & & Alist_1 & & Alist_2 & \cdots & Alist_i & & Alist_{i+1} & \cdots \\ & \phi_0 & & \phi_1 & & & & \phi_i & & \end{array}$$

What is the ''lub'' of this chain? (It will be $Alist_\infty$.) To give us some intuition about the lub, we represent the elements of an $Alist_i$ domain as tuples. An element $x \in Alist_i$ appears as an $(i{+}1)$-tuple of the form $(x_0, x_1, \cdots, x_{i-1}, x_i)$, where $x_i = x$, $x_{i-1} = \psi_{i-1}(x_i)$, $\cdots$, $x_1 = \psi_1(x_2)$, and $x_0 = \psi_0(x_1)$. For example, $[a_0, a_1, nil] \in Alist_3$ has tuple form $(\perp, [a_0, \perp], [a_0, a_1, \perp], [a_0, a_1, nil])$; $[a_0, a_1, a_2, \perp] \in Alist_3$ has form $(\perp, [a_0, \perp], [a_0, a_1, \perp], [a_0, a_1, a_2, \perp])$; $[a_0, nil] \in Alist_3$ has form $(\perp, [a_0, \perp], [a_0, nil], [a_0, nil])$; and $\perp \in Alist_3$ has form $(\perp, \perp, \perp, \perp)$. The tuples trace the incrementation of information in an element until the information is complete. They suggest that the limit domain of the chain, $Alist_\infty$, has elements whose tuple representations have *infinite* length. A finite list x with i A-elements belongs to $Alist_\infty$ and has tuple representation $(x_0, x_1, \cdots, x_{i-1}, x, x, \cdots)$— it stabilizes. The infinite lists have tuple representations that never stabilize: for example, an infinite list of a's has the representation $(\perp, [a, \perp], [a, a, \perp], [a, a, a, \perp], \cdots, [a, a, a, \cdots, a, \perp], \cdots)$. The tuple shows that the infinite list has information content that sums all the finite partial lists that approximate it.

Since there is no real difference between an element and its tuple representation (like functions and their graphs), we take the definition of the limit domain $Alist_\infty$ to be the set of infinite tuples induced from the $Alist_i$'s and the ψ_i's:

$$Alist_\infty = \{\, (x_0, x_1, \cdots, x_i, \cdots) \mid \textit{for all } n \geq 0,\ x_n \in Alist_n \text{ and } x_n = \psi_n(x_{n+1}) \,\}$$

partially ordered by, for all $x, y \in Alist_\infty$, $x \sqsubseteq y$ iff for all $n \geq 0$, $x \downarrow n \sqsubseteq_{Alist_n} y \downarrow n$. $Alist_\infty$ contains only those tuples with information consistent with the $Alist_i$'s. The partial ordering is the natural one for a subdomain of a product domain.

Now we must show that $Alist_\infty$ satisfies the recursive specification; that is,

$Alist_\infty = (Nil + (A \times Alist_\infty))_\perp$.

Unfortunately this equality doesn't hold! The problem is that the domain on the right-hand side uses the one on the left-hand side as a component— the left-hand side domain is a set of tuples but the right-hand side one is a lifted disjoint union. The situation isn't hopeless, however, as the two domains have the same size (cardinality) and possess the same partial ordering structure. The two domains are *order isomorphic*. The isomorphism is proved by functions $\Phi : Alist_\infty \rightarrow (Nil + (A \times Alist_\infty))_\perp$ and $\Psi : (Nil + (A \times Alist_\infty))_\perp \rightarrow Alist_\infty$ such that $\Psi \circ \Phi = id_{Alist_\infty}$ and $\Phi \circ \Psi = id_{Nil+(A\times Alist_\infty))_\perp}$. The Φ function exposes the list structure inherent in an $Alist_\infty$ element, and the Ψ map gives the tuple representation of list structured objects.

The isomorphism property is strong enough that $Alist_\infty$ may be considered a solution of the specification. The Φ and Ψ maps are used in the definitions of operations on the domain. For example, $head : Alist_\infty \rightarrow A_\perp$ is defined as:

$$head = \underline{\lambda}x.\ \text{cases}\ \Phi(x)\ \text{of is}Nil() \rightarrow \perp\ []\ \text{is}A{\times}Alist_\infty(a,l) \rightarrow a\ \text{end}$$

$tail : Alist_\infty \rightarrow Alist_\infty$ is similar. The map $construct : A \times Alist_\infty \rightarrow Alist_\infty$ is $construct(a, x) = \Psi(\text{in}A{\times}Alist_\infty(a, x))$. These conversions of structure from $Alist_\infty$ to list form and back are straightforward and weren't mentioned in the examples in the previous chapters. The isomorphism maps can always be inserted when needed.

The Φ and Ψ maps are built from the (ϕ_i, ψ_i) pairs. A complete description is presented in the next section.

11.2 THE INVERSE LIMIT CONSTRUCTION

The method just described is the *inverse limit construction*. It was developed by Scott as a justification of Strachey's original development of denotational semantics. The formal details of the construction are presented in this section. The main result is that, for any recursive domain specification of form $D = F(D)$ (where F is an expression built with the constructors of Chapter 3 such that $F(E)$ is a pointed cpo when E is), there is a domain D_∞ that is isomorphic to $F(D_\infty)$. D_∞ is the *least* such pointed cpo that satisfies the specification. If you take faith in the above claims, you may wish to skim this section and proceed to the examples in Section 11.3.

Our presentation of the inverse limit construction is based on an account by Reynolds (1972) of Scott's results. We begin by formalizing the relationship between the ϕ_i and ψ_i maps.

11.1 Definition:

For pointed cpos D and D′, a pair of continuous functions $(f : D \rightarrow D',\ g : D' \rightarrow D)$ *is a retraction pair iff:*

1. $g \circ f = id_D$
2. $f \circ g \sqsubseteq id_{D'}$

f is called an embedding *and g is called a* projection.

11.2 Proposition:

The composition $(f_2 \circ f_1,\ g_1 \circ g_2)$ *of retraction pairs* $(f_1 : D \to D',\ g_1 : D' \to D)$ *and* $(f_2 : D' \to D'',\ g_2 : D'' \to D')$ *is itself a retraction pair.*

Proof: $(g_1 \circ g_2) \circ (f_2 \circ f_1) = g_1 \circ (g_2 \circ f_2) \circ f_1) = g_1 \circ id_{D'} \circ f_1 = g_1 \circ f_1 = id_D$. The proof that $(f_2 \circ f_1) \circ (g_1 \circ g_2) \sqsubseteq id_{D''}$ is similar. □

11.3 Proposition:

An embedding (projection) has a unique corresponding projection (embedding).

Proof: Let (f, g_1) and (f, g_2) both be retraction pairs. We must show that $g_1 = g_2$. First, $f \circ g_1 \sqsubseteq id_{D'}$ which implies $g_2 \circ f \circ g_1 \sqsubseteq g_2 \circ id_{D'}$ by the monotonicity of g_2. But $g_2 \circ f \circ g_1 = (g_2 \circ f) \circ g_1 = id_D \circ g_1 = g_1$, implying $g_1 \sqsubseteq g_2$. Repeating the above derivation with g_1 and g_2 swapped gives $g_1 \sqsubseteq g_2$, implying that $g_1 = g_2$. The uniqueness of an embedding f to a projection g is left as an exercise. □

11.4 Proposition:

The components of a retraction pair are strict functions.

Proof: Left as an exercise. □

Retraction pairs are special cases of function pairs $(f : D \to D',\ g : D' \to D)$ for cpos D and D'. Since we will have use for function pairs that may not be retraction pairs on pointed cpos, we assign the name *r-pair* to a function pair like the one just seen.

11.5 Definition:

For cpos D and D', a continuous pair of functions $(f : D \to D',\ g : D' \to D)$ *is called an* r-pair *and is written* $(f,g) : D \longleftrightarrow D'$. *The operations on r-pairs are:*

1. *Composition: for* $(f_1, g_1) : D \longleftrightarrow D'$ *and* $(f_2, g_2) : D' \longleftrightarrow D''$, $(f_2, g_2) \circ (f_1, g_1) : D \longleftrightarrow D''$ *is defined as* $(f_2 \circ f_1,\ g_1 \circ g_2)$.
2. *Reversal: for* $(f,g) : D \longleftrightarrow D'$, $(f,g)^R : D' \longleftrightarrow D$ *is defined as* (g,f).

The reversal of a retraction pair might not be a retraction pair. The identity r-pair for the domain $D \longleftrightarrow D$ is $id_{D \longleftrightarrow D} = (id_D,\ id_D)$. It is easy to show that the composition and reversal operations upon r-pairs are continuous. We use the letters $r, s, t, \cdots$ to denote r-pairs.

11.6 Proposition:

For r-pairs $r : D \leftrightarrow D'$ *and* $s : D' \leftrightarrow D''$:

1. $(r \circ s)^R = s^R \circ r^R$
2. $(r^R)^R = r$.

Proof: Left as an exercise. □

When we build a solution to $D = F(D)$, we build the approximating domains $\{ D_i \mid i \geq 0 \}$ from an initial domain D_0 by systematically applying the domain construction F. We use a similar procedure to generate the r-pairs $(\phi_i, \psi_i) : D_i \leftrightarrow D_{i+1}$ from a starting pair (ϕ_0, ψ_0). First, the domain builders defined in Chapter 3 are extended to build r-pairs.

11.7 Definition:

For r-pairs $r = (f, g) : C \leftrightarrow E$ *and* $s = (f', g') : C' \leftrightarrow E'$, *let:*

1. $r \times s$ *denote:*
   $((\lambda(x,y).\,(f(x), f'(y))),\ (\lambda(x,y).\,(g(x), g'(y)))) : C \times C' \leftrightarrow E \times E'$
2. $r + s$ *denote:*
   $((\lambda x.\ \text{cases } x \text{ of is}C(c) \rightarrow \text{in}E(f(c))\ [\!]\ \text{is}C'(c) \rightarrow \text{in}E'(f'(c)) \text{ end},$
   $(\lambda x.\ \text{cases } y \text{ of is}E(e) \rightarrow \text{in}C(g(e))\ [\!]\ \text{is}E'(e) \rightarrow \text{in}C'(g'(e)) \text{ end}))$
   $: C + C' \leftrightarrow E + E'$
3. $r \rightarrow s$ *denote:* $((\lambda x.\, f' \circ x \circ g),\ (\lambda y.\, g' \circ y \circ f)) : (C \rightarrow C') \leftrightarrow (E \rightarrow E')$
4. $(r)_\perp$ *denote:* $((\underline{\lambda} x.\, f x),\ (\underline{\lambda} y.\, g y)) : C_\perp \leftrightarrow E_\perp$

For $D = F(D)$, the domain expression F determines a construction for building a new domain $F(A)$ from an argument domain A and a construction for building a new r-pair $F(r)$ from an argument r-pair r. For example, the recursive specification $Nlist = (Nil + (Nat \times Nlist))_\perp$ gives a construction $F(D) = ((Nil + (Nat \times D))_\perp$ such that, for any cpo A, $(Nil + (Nat \times A))_\perp$ is also a cpo, and for any r-pair r, $(Nil + (Nat \times r))_\perp$ is an r-pair. The r-pair is constructed using Definition 11.7; the r-pairs corresponding to Nil and Nat in the example are the identity r-pairs (id_{Nil}, id_{Nil}) and (id_{Nat}, id_{Nat}), respectively. You are left with the exercise of formalizing what a ‘‘domain expression’’ is. Once you have done so, produce a structural induction proof of the following important lemma.

11.8 Lemma:

For any domain expression F and r-pairs $r : D \leftrightarrow D'$ *and* $s : D' \leftrightarrow D''$:

1. $F(id_{E \leftrightarrow E}) = id_{F(E) \leftrightarrow F(E)}$
2. $F(s) \circ F(r) = F(s \circ r)$
3. $(F(r))^R = F(r^R)$
4. *if r is a retraction pair, then so is* $F(r)$

The lemma holds for the domain expressions built with the domain calculus of Chapter 3.

Now that r-pairs and their fundamental properties have been stated, we formulate the inverse limit domain.

11.9 Definition:

A retraction sequence *is a pair* $(\{ D_i \mid i \geq 0 \}, \{ r_i : D_i \longleftrightarrow D_{i+1} \mid i \geq 0 \})$ *such that for all* $i \geq 0$, D_i *is a pointed cpo, and each r-pair* r_i *is a retraction pair.*

We often compose retraction pairs from a retraction sequence. Let $t_{mn} : D_m \longleftrightarrow D_n$ be defined as:

$$t_{mn} = \begin{cases} r_{n-1} \circ \cdots \circ r_m & \text{if } m < n \\ id_{D_m \longleftrightarrow D_m} & \text{if } m = n \\ r_n^R \circ \cdots \circ r_{m-1}^R & \text{if } m > n \end{cases}$$

To make this clear, let each r_i be the r-pair $(\phi_i : D_i \to D_{i+1}, \psi_i : D_{i+1} \to D_i)$ and each t_{mn} be the r-pair $(\theta_{mn} : D_m \to D_n, \theta_{nm} : D_n \to D_m)$. Then for $m < n$, $t_{mn} = (\theta_{mn}, \theta_{nm}) = (\phi_{n-1}, \psi_{n-1}) \circ \cdots \circ (\phi_{m+1}, \psi_{m+1}) \circ (\phi_m, \psi_m) = (\phi_{n-1} \circ \cdots \circ \phi_{m+1} \circ \phi_m, \psi_m \circ \psi_{m+1} \circ \cdots \circ \psi_{n-1})$, which is drawn as:

$$\begin{array}{ccccccccccc} & \psi_m & & \psi_{m+1} & & & & & & \psi_{n-1} & \\ D_m & & D_{m+1} & & & D_{m+2} & & \cdots & D_{n-1} & & D_n \\ & \phi_m & & \phi_{m+1} & & & & & & \phi_{n-1} & \end{array}$$

Drawing a similar diagram for the case when $m > n$ makes it clear that $t_{mn} = (\theta_{mn}, \theta_{nm}) = (\theta_{nm}, \theta_{mn})^R = t_{nm}^R$, so the use of the θ_{mn}'s is consistent.

11.10 Proposition:

For any retraction sequence and $m, n, k \geq 0$:
1. $t_{mn} \circ t_{km} \sqsubseteq t_{kn}$
2. $t_{mn} \circ t_{km} = t_{kn}$, *when* $m \geq k$ *or* $m \geq n$
3. t_{mn} *is a retraction pair when* $m \leq n$

Proof: Left as an exercise. $\square$

As the example in Section 11.1 pointed out, the limit of a retraction sequence is built from the members of the D_i domains and the ψ_i embeddings.

11.11 Definition:

The inverse limit of a retraction sequence:

$$(\{ D_i \mid i \geq 0 \}, \{ (\phi_i, \psi_i) : D_i \longleftrightarrow D_{i+1} \mid i \geq 0 \})$$

is the set:

$$D_\infty = \{ (x_0, x_1, \cdots, x_i, \cdots) \mid \textit{for all } n > 0,\ x_n \in D_n \textit{ and } x_n = \psi_n(x_{n+1}) \}$$

partially ordered by the relation: for all $x,y \in D_\infty$, $x \sqsubseteq y$ *iff for all* $n \geq 0$, $x{\downarrow}n \sqsubseteq_{D_n} y{\downarrow}n$.

11.12 Theorem:

D_∞ *is a pointed cpo.*

Proof: Recall that each D_i in the retraction sequence is a pointed cpo. First, $\perp_{D_\infty} = (\perp_{D_0}, \perp_{D_1}, \cdots, \perp_{D_i}, \cdots) \in D_\infty$, since every $\psi_i(\perp_{D_{i+1}}) = \perp_{D_i}$, by Proposition 11.4. Second, for any chain $C = \{ c_i \mid i \in I \}$ in D_∞, the definition of the partial ordering on D_∞ makes $C_n = \{ c_i{\downarrow}n \mid i \in I \}$ a chain in D_n with a lub of $\bigsqcup C_n$, $n \geq 0$. Now $\psi_n(\bigsqcup C_{n+1})$ $= \bigsqcup\{ \psi_n(c_i{\downarrow}(n+1)) \mid i \in I \}$ $= \bigsqcup\{ c_i{\downarrow}n \mid i \in I \}$ $= \bigsqcup C_n$. Hence $(\bigsqcup C_0, \bigsqcup C_1, \cdots, \bigsqcup C_i, \cdots)$ belongs to D_∞. It is clearly the lub of C. □

Next, we show how a domain expression generates a retraction sequence.

11.13 Proposition:

If domain expression F maps a pointed cpo E to a pointed cpo F(E), then the pair:

$$\begin{aligned}
&(\{ D_i \mid D_0 = \{ \perp \},\ D_{i+1} = F(D_i),\ \textit{for } i \geq 0 \}, \\
&\quad \{ (\phi_i, \psi_i) : D_i \leftrightarrow D_{i+1} \mid \phi_0 = (\lambda x. \perp_{D_1}),\ \psi_0 = (\lambda x. \perp_{D_0}), \\
&\qquad\qquad (\phi_{i+1}, \psi_{i+1}) = F(\phi_i, \psi_i),\ \textit{for } i \geq 0) \}
\end{aligned}$$

is a retraction sequence.

Proof: From Lemma 11.8, part 4, and mathematical induction. □

Thus, the inverse limit D_∞ exists for the retraction sequence generated by F. The final task is to show that D_∞ is isomorphic to $F(D_\infty)$ by defining functions $\Phi : D_\infty \to F(D_\infty)$ and $\Psi : F(D_\infty) \to D_\infty$ such that $\Psi \circ \Phi = id_{D_\infty}$ and $\Phi \circ \Psi = id_{F(D_\infty)}$. Just as the elements of D_∞ were built from elements of the D_i's, the maps Φ and Ψ are built from the retraction pairs (ϕ_i, ψ_i). For $m \geq 0$:

$$\begin{aligned}
&t_{m\infty} : D_m \leftrightarrow D_\infty \text{ is:} \\
&\qquad (\theta_{m\infty}, \theta_{\infty m}) = ((\lambda x. (\theta_{m0}(x), \theta_{m1}(x), \cdots, \theta_{mi}(x), \cdots)),\ (\lambda x. x{\downarrow}m)) \\
&t_{\infty m} : D_\infty \leftrightarrow D_m \text{ is: } (\theta_{\infty m}, \theta_{m\infty}) = t^R_{m\infty} \\
&t_{\infty\infty} : D_\infty \leftrightarrow D_\infty \text{ is: } (\theta_{\infty\infty}, \theta_{\infty\infty}) = (id_{D_\infty}, id_{D_\infty})
\end{aligned}$$

You are given the exercises of showing that $\theta_{m\infty} : D_m \to D_\infty$ is well defined and proving the following proposition.

11.14 Proposition:

Proposition 11.10 holds when ∞ subscripts are used in place of m and n in the t_{mn} *pairs.*

Since each $t_{m\infty}$ is a retraction pair, the value $\theta_{m\infty}(\theta_{\infty m}(x))$ is less defined than $x \in D_\infty$. As m increases, the approximations to x become better. A pleasing and important result is that as m tends toward ∞, the approximations approach identity.

11.15 Lemma:

$$id_{D_\infty} = \bigsqcup_{m=0}^{\infty} \theta_{m\infty} \circ \theta_{\infty m}.$$

Proof: For every $m \geq 0$, $t_{m\infty}$ is a retraction pair, so $\theta_{m\infty} \circ \theta_{\infty m} \sqsubseteq id_{D_\infty}$. Because $\{\, \theta_{m\infty} \circ \theta_{\infty m} \mid m \geq 0 \,\}$ is a chain in $D_\infty \to D_\infty$, $\bigsqcup_{m=0}^{\infty} \theta_{m\infty} \circ \theta_{\infty m} \sqsubseteq id_{D_\infty}$ holds. Next, for any $x = (x_0, x_1, \cdots, x_i, \cdots) \in D_\infty$ and any $i \geq 0$, $\theta_{i\infty} \circ \theta_{\infty i}(x) = \theta_{i\infty}(\theta_{\infty i}(x)) = (\theta_{i0}(x{\downarrow}i), \theta_{i1}(x{\downarrow}i), ..., \theta_{ii}(x{\downarrow}i), \cdots)$. Since $\theta_{ii}(x{\downarrow}i) = (x{\downarrow}i) = x_i$, each mth component of tuple x will appear as the mth component in $\theta_{m\infty}(\theta_{\infty m}(x))$, for all $m \geq 0$. So $x \sqsubseteq \bigsqcup_{m=0}^{\infty}(\theta_{m\infty} \circ \theta_{\infty m})(x) = (\bigsqcup_{m=0}^{\infty} \theta_{m\infty} \circ \theta_{\infty m})(x)$. By extensionality, $id_{D_\infty} \sqsubseteq \bigsqcup_{m=0}^{\infty} \theta_{m\infty} \circ \theta_{\infty m}$, which implies the result. $\square$

11.16 Corollary:

$$id_{D_\infty \longleftrightarrow D_\infty} = \bigsqcup_{m=0}^{\infty} t_{m\infty} \circ t_{\infty m}$$

11.17 Corollary:

$$id_{F(D_\infty) \longleftrightarrow F(D_\infty)} = \bigsqcup_{m=0}^{\infty} F(t_{m\infty}) \circ F(t_{\infty m})$$

Proof: $id_{F(D_\infty) \longleftrightarrow F(D_\infty)} = F(id_{D_\infty \longleftrightarrow D_\infty})$, by Lemma 11.8, part 1

$= F(\bigsqcup_{m=0}^{\infty} t_{m\infty} \circ t_{\infty m})$, by Corollary 11.16

$= \bigsqcup_{m=0}^{\infty} F(t_{m\infty} \circ t_{\infty m})$, by continuity

$= \bigsqcup_{m=0}^{\infty} F(t_{m\infty}) \circ F(t_{\infty m})$, by Lemma 11.8, part 2 $\square$

The isomorphism maps are defined as a retraction pair (Φ, Ψ) in a fashion similar to the r-pairs in Corollaries 11.16 and 11.17. The strategy is to combine the two r-pairs into one on $D_\infty \longleftrightarrow F(D_\infty)$:

$$(\Phi, \Psi) : D_\infty \longleftrightarrow F(D_\infty) = \bigsqcup_{m=0}^{\infty} F(t_{m\infty}) \circ t_{\infty(m+1)}$$

The r-pair structure motivates us to write the isomorphism requirements in the form $(\Phi, \Psi)^R \circ (\Phi, \Psi) = id_{D_\infty \longleftrightarrow D_\infty}$ and $(\Phi, \Psi) \circ (\Phi, \Psi)^R = id_{F(D_\infty) \longleftrightarrow F(D_\infty)}$. The proofs require the following technical lemmas.

11.18 Lemma:

For any $m \geq 0$, $F(t_{\infty m}) \circ (\Phi, \Psi) = t_{\infty(m+1)}$

Proof: $F(t_{\infty m}) \circ (\Phi, \Psi)$

$= F(t_{\infty m}) \circ \bigsqcup_{n=0}^{\infty} F(t_{n\infty}) \circ t_{\infty(n+1)}$

$= \bigsqcup_{n=0}^{\infty} F(t_{\infty m}) \circ F(t_{n\infty}) \circ t_{\infty(n+1)}$, by continuity

$= \bigsqcup_{n=0}^{\infty} F(t_{\infty m} \circ t_{n\infty}) \circ t_{\infty(n+1)}$, by Lemma 11.8, part 2

$= \bigsqcup_{n=0}^{\infty} F(t_{nm}) \circ t_{\infty(n+1)}$, by Proposition 11.14

$= \bigsqcup_{n=0}^{\infty} t_{(n+1)(m+1)} \circ t_{\infty(n+1)}$

By Proposition 11.14, $t_{(n+1)(m+1)} \circ t_{\infty(n+1)} = t_{\infty(m+1)}$, for $n \geq m$. Thus, the least upper bound of the chain is $t_{\infty(m+1)}$. $\square$

11.19 Lemma:

For any $m \geq 0$, $(\Phi, \Psi) \circ t_{(m+1)\infty} = F(t_{m\infty})$

Proof: Similar to the proof of Lemma 11.18 and left as an exercise. $\square$

11.20 Theorem:

$(\Phi, \Psi)^R \circ (\Phi, \Psi) = id_{D_\infty \longleftrightarrow D_\infty}$

Proof: $(\Phi, \Psi)^R \circ (\Phi, \Psi) = (\bigsqcup_{m=0}^{\infty} F(t_{m\infty}) \circ t_{\infty(m+1)})^R \circ (\Phi, \Psi)$

$= (\bigsqcup_{m=0}^{\infty} (F(t_{m\infty}) \circ t_{\infty(m+1)})^R) \circ (\Phi, \Psi)$, by continuity of R

$= (\bigsqcup_{m=0}^{\infty} t^R_{\infty(m+1)} \circ F(t_{m\infty})^R) \circ (\Phi, \Psi)$, by Proposition 11.6

$= (\bigsqcup_{m=0}^{\infty} t_{(m+1)\infty} \circ F(t_{\infty m})) \circ (\Phi, \Psi)$, by Lemma 11.8, part 3

$= \bigsqcup_{m=0}^{\infty} t_{(m+1)\infty} \circ F(t_{\infty m}) \circ (\Phi, \Psi)$, by continuity

$= \bigsqcup_{m=0}^{\infty} t_{(m+1)\infty} \circ t_{\infty(m+1)}$, by Lemma 11.18

$= \bigsqcup_{m=0}^{\infty} t_{m\infty} \circ t_{\infty m}$, as $t_{0\infty} \circ t_{\infty 0} \sqsubseteq t_{1\infty} \circ t_{\infty 1}$

$= id_{D_\infty \longleftrightarrow D_\infty}$, by Corollary 11.16 $\square$

11.21 Theorem:

$(\Phi, \Psi) \circ (\Phi, \Psi)^R = id_{F(D_\infty) \longleftrightarrow F(D_\infty)}$

Proof: Similar to the proof of Theorem 11.20 and left as an exercise. $\square$

Analogies of the inverse limit method to the least fixed point construction are strong. So far, we have shown that D_∞ is a ‘‘fixed point’’ of the ‘‘chain’’ generated by a ‘‘functional’’ F. To complete the list of parallels, we can show that D_∞ is the ‘‘least upper bound’’ of the retraction sequence. For the retraction sequence $(\{ D_i \mid i \geq 0 \}, \{ (\phi_i, \psi_i) \mid i \geq 0 \})$ generated by F, assume that there exists a pointed cpo D' and retraction pair $(\Phi', \Psi') : D' \leftrightarrow F(D')$ such that (Φ', Ψ') proves that D' is isomorphic to $F(D')$. Then define the following r-pairs:

$$t'_{0\infty} : D_0 \leftrightarrow D' \text{ as } ((\lambda x.\, \bot_{D'}), (\lambda x.\, \bot_{D_0}))$$
$$t'_{(m+1)\infty} : D_{m+1} \leftrightarrow D' \text{ as } (\Psi', \Phi') \circ F(t'_{m\infty})$$

Each $t'_{m\infty}$ is a retraction pair, and $\{ t'_{m\infty} \circ t_{\infty m} \mid m \geq 0 \}$ is a chain in $D_\infty \leftrightarrow D'$. Next, define $(\alpha, \beta) : D_\infty \leftrightarrow D'$ to be $\bigsqcup_{m=0}^{\infty} t'_{m\infty} \circ t_{\infty m}$. We can show that (α, β) is a retraction pair; that is, D_∞ embeds into D'. Since D' is arbitrary, D_∞ must be the least pointed cpo solution to the retraction sequence.

We gain insight into the structure of the isomorphism maps Φ and Ψ by slightly abusing our notation. Recall that a domain expression F is interpreted as a map on r-pairs. F is required to work upon r-pairs because it must ‘‘invert’’ a function’s domain and codomain to construct a map upon function spaces (see Definition 11.7, part 3). The inversion is done with the function’s r-pair mate. But for retraction components, the choice of mate is unique (by Proposition 11.3). So, if r-pair $r = (f, g)$ is a retraction pair, let $F(f, g)$ be alternatively written as (Ff, Fg), with the understanding that any ‘‘inversions’’ of f or g are fulfilled by the function’s retraction mate. Now the definition of (Φ, Ψ), a retraction pair, can be made much clearer:

$$\begin{aligned}
(\Phi, \Psi) &= \bigsqcup_{m=0}^{\infty} F(t_{m\infty}) \circ t_{\infty(m+1)} \\
&= \bigsqcup_{m=0}^{\infty} F(\theta_{m\infty}, \theta_{\infty m}) \circ (\theta_{\infty(m+1)}, \theta_{(m+1)\infty}) \\
&= \bigsqcup_{m=0}^{\infty} (F\theta_{m\infty}, F\theta_{\infty m}) \circ (\theta_{\infty(m+1)}, \theta_{(m+1)\infty}) \\
&= \bigsqcup_{m=0}^{\infty} (F\theta_{m\infty} \circ \theta_{\infty(m+1)}, \theta_{(m+1)\infty} \circ F\theta_{\infty m}) \\
&= (\bigsqcup_{m=0}^{\infty} F\theta_{m\infty} \circ \theta_{\infty(m+1)}, \bigsqcup_{m=0}^{\infty} \theta_{(m+1)\infty} \circ F\theta_{\infty m})
\end{aligned}$$

We see that $\Phi : D_\infty \rightarrow F(D_\infty)$ maps an $x \in D_\infty$ to an element $x_{(m+1)} \in D_{m+1}$ and then maps x_{m+1} to an F-structured element whose components come from D_∞. The steps are performed for each $m > 0$, and the results are joined. The actions of $\Psi : F(D_\infty) \rightarrow D_\infty$ are similarly interpreted. This roundabout method of getting from D_∞ to $F(D_\infty)$ and back has the advantage of being entirely representable in terms of the elements of the retraction sequence. We exploit this transparency in the examples in the next section.

11.3 APPLICATIONS

We now examine three recursive domain specifications and their inverse limit solutions. The tuple structure of the elements of the limit domain and the isomorphism maps give us deep insights into the nature and uses of recursively defined domains.

11.3.1 Linear Lists

This was the example in Section 11.1. For the recursive definition $Alist = (Nil + (A \times Alist))_\perp$, the retraction sequence is $(\{ D_n \mid n \geq 0 \}, \{ (\phi_n, \psi_n) \mid n \geq 0 \})$, where:

$$D_0 = \{ \perp \}$$
$$D_{i+1} = (Nil + (A \times D_i))_\perp$$

and

$$\phi_0 : D_0 \to D_1 = (\lambda x. \perp_{D_1})$$
$$\psi_0 : D_1 \to D_0 = (\lambda x. \perp_{D_0})$$

```
φ_i : D_i → D_{i+1} = (id_{Nil} + (id_A × φ_{i−1}))_⊥
                    = λ̲x. cases x of
                            isNil() → inNil()
                            [] isA×D_{i+1}(a,d) → inA×D_i(a, φ_{i−1}(d)) end

ψ_i : D_{i+1} → D_i = (id_{Nil} + (id_A × ψ_{i−1}))_⊥
                    = λ̲x. cases x of
                            isNil() → inNil()
                            [] isA×D_i(a,d) → inA×D_{i−1}(a, ψ_{i−1}(d)) end
```

An element in D_n is a list with n or less A-elements. The map $\theta_{mn} : D_m \to D_n$ converts a list of m (or less) A-elements to one of n (or less) A-elements. If $m > n$, the last $m-n$ elements are truncated and replaced by $\perp$. If $m \leq n$, the list is embedded intact into D_n. An $Alist_\infty$ element is a tuple $x = (x_0, x_1, \cdots, x_i, \cdots)$, where each x_i is a list from D_i and x_i and x_{i+1} agree on their first i A-elements, because $x_i = \psi_i(x_{i+1})$. The map $\theta_{\infty m} : Alist_\infty \to D_m$ projects an infinite tuple into a list of m elements $\theta_{\infty m}(x) = x_i$, and $\theta_{m\infty} : D_m \to Alist_\infty$ creates the tuple corresponding to an m-element list $\theta_{m\infty}(l) = (\theta_{m0}(l), \theta_{m1}(l), \cdots, \theta_{mm}(l), \theta_{m(m+1)}, \cdots)$, where $\theta_{mk}(l) = l$ for $k \geq m$. It is easy to see that any finite list has a unique representation in $Alist_\infty$, and Lemma 11.15 clearly holds. But why can we treat $Alist_\infty$ as if it were a domain of lists? And where are the infinite lists? The answers to both these questions lie with $\Phi : Alist_\infty \to F(Alist_\infty)$. It is defined as:

$$\Phi = \bigsqcup_{m=0}^{\infty} (id_{Nil} + (id_A \times \theta_{m\infty}))_\perp \circ \theta_{\infty(m+1)}$$

```
  = ⨆_{m=0}^{∞} (λ̲x. cases x of
                    isNil() → inNil()
                    [] isA×D_m(a,d) → inA×Alist_∞(a, θ_{m∞}(d))
                    end) ∘ θ_{∞(m+1)}
```

Φ reveals the list structure in an $Alist_\infty$ tuple. A tuple $x \in Alist_\infty$ represents:

1. The undefined list when $\Phi(x) = \bot$ (then x is $(\bot, \bot, \bot, \cdots)$).
2. The *nil* list when $\Phi(x) = \text{in}Nil()$ (then x is $(\bot, [nil], [nil], \cdots)$).
3. A list whose head element is a and tail component is d when $\Phi(x) = \text{in}A{\times}Alist_\infty(a, d)$ (then x is $(\bot, [a, \theta_{\infty 0}(d)], [a, \theta_{\infty 1}(d)], \cdots, [a, \theta_{\infty (i-1)}(d)], \cdots)$). $\Phi(d)$ shows the list structure in the tail.

As described in Section 11.1, an infinite list is represented by a tuple x such that for all $i \geq 0$ $x_i \neq x_{i+1}$. Each $x_i \in D_i$ is a list with i (or less) A-elements; hence the kth element of the infinite list that x represents is embedded in those x_j such that $j \geq k$. Φ finds the kth element: it is a_k, where $\Phi(x) = \text{in}A{\times}Alist_\infty(a_1, d^2)$, and $\Phi(d^i) = \text{in}A{\times}Alist_\infty(a_i, d^{i+1})$, for $i > 1$.

The inverse map to Φ is $\Psi : F(Alist_\infty) \rightarrow Alist_\infty$. It embeds a list into $Alist_\infty$ so that operations like $cons : A{\times}Alist_\infty \rightarrow Alist_\infty$ have well-formed definitions. For $a \in A$ and $x \in Alist_\infty$, $\Psi(a, d) = (\bot, [a, \theta_{\infty 0}(x)], [a, \theta_{\infty 1}(x)], \cdots, [a, \theta_{\infty (i-1)}(x)], \cdots)$. The isomorphism properties of Φ and Ψ assure us that this method of unpacking and packing tuples is sound and useful.

11.3.2 Self-Applicative Procedures

Procedures in ALGOL60 can take other procedures as arguments, even to the point of self-application. A simplified version of this situation is $Proc = Proc \rightarrow A_\bot$. The family of pointed cpos that results begins with:

$$D_0 = \{ \bot \}$$
$$D_1 = D_0 \rightarrow A_\bot$$

The argument domain to D_1-level procedures is just the one-element domain, and the members of D_1 are those functions with graphs of form $\{ (\bot, a) \}$, for $a \in A_\bot$.

$$D_2 = D_1 \rightarrow A_\bot$$

A D_2-level procedure accepts D_1-level procedures as arguments.

$$D_{i+1} = D_i \rightarrow A_\bot$$

In general, a D_{i+1}-level procedure accepts D_i-level arguments. (Note that $D_{i-1}, D_{i-2}, \cdots$ are all embedded in D_i.) If we sum the domains, the result, $\sum_{i=0}^{\infty} D_i$, resembles a Pascal-like hierarchy of procedures. But we want a procedure to accept arguments from a level equal to or greater than the procedure's own. The inverse limit's elements do just that.

Consider an element $(p_0, p_1, \cdots, p_i, \cdots) \in Proc_\infty$. It has the capability of handling a procedure argument at any level. For example, an argument $q_k : D_k$ is properly handled by p_{k+1}, and the result is $p_{k+1}(q_k)$. But the tuple is intended to operate upon arguments in $Proc_\infty$, and these elements no longer have ''levels.'' The solution is simple: take the argument $q \in Proc_\infty$ and map it down to level D_0 (that is, $\theta_{\infty 0}(q)$) and apply p_1 to it; map it down to level D_1 (that is, $\theta_{\infty 1}(q)$) and apply p_2 to it; . . .; map it down to level D_i (that is, $\theta_{\infty i}(q)$) and apply p_{i+1} to it; . . . ; and lub the results! This is *precisely* what $\Phi : Proc_\infty \rightarrow F(Proc_\infty)$ does:

$$\Phi = \bigsqcup_{m=0}^{\infty} (\theta_{m\infty} \rightarrow id_{A_\bot}) \circ \theta_{\infty (m+1)}$$

$$= \bigsqcup_{m=0}^{\infty} (\lambda x.\, id_{A_\perp} \circ x \circ \theta_{\infty m}) \circ \theta_{\infty (m+1)}$$

The application $p(q)$ is actually $(\Phi(p))(q)$. Consider $\Phi(p)$; it has value:

$$\Phi(p) = \bigsqcup_{m=0}^{\infty} (\lambda x.\, id_{A_\perp} \circ x \circ \theta_{\infty m})(\theta_{\infty (m+1)}(p))$$
$$= \bigsqcup_{m=0}^{\infty} (\theta_{\infty (m+1)}(p)) \circ \theta_{\infty m}$$
$$= \bigsqcup_{m=0}^{\infty} (p \downarrow (m+1)) \circ \theta_{\infty m}$$
$$= \bigsqcup_{m=0}^{\infty} p_{m+1} \circ \theta_{\infty m}$$

Thus:

$$(\Phi(p))(q) = (\bigsqcup_{m=0}^{\infty} p_{m+1} \circ \theta_{\infty m})(q)$$
$$= \bigsqcup_{m=0}^{\infty} p_{m+1}(\theta_{\infty m}(q))$$
$$= \bigsqcup_{m=0}^{\infty} p_{m+1}(q \downarrow m)$$
$$= \bigsqcup_{m=0}^{\infty} p_{m+1}(q_m)$$

The scheme is general enough that even self-application is understandable.

11.3.3 Recursive Record Structures

Recall that the most general form of record structure used in Chapter 7 was:

$$Record = Id \rightarrow Denotable\text{-}value$$
$$Denotable\text{-}value = (Record + Nat + \cdots)_\perp$$

Mutually defined sets of equations like the one above can also be handled by the inverse limit technique. We introduce m-tuples of domain equations, approximation domains, and r-pairs. The inverse limit is an m-tuple of domains. In this example, m=2, so a pair of retraction sequences are generated. We have:

$$R_0 = Unit$$
$$D_0 = Unit$$
$$R_{i+1} = Id \rightarrow D_i$$
$$D_{i+1} = (R_i + Nat + \cdots)_\perp, \text{ for } i \geq 0$$

and

$$R\phi_0 : R_0 \rightarrow R_1 = (\lambda x.\, \perp_{R_1})$$
$$R\psi_0 : R_1 \rightarrow R_0 = (\lambda x.\, \perp_{R_0})$$
$$D\phi_0 : D_0 \rightarrow D_1 = (\lambda x.\, \perp_{D_1})$$

$D\psi_0 : D_1 \to D_0 = (\lambda x.\, \bot_{D_0})$
$R\phi_i : R_i \to R_{i+1} = (\lambda x.\, D\phi_{i-1} \circ x \circ id_{Id})$
$R\psi_i : R_{i+1} \to R_i = (\lambda x.\, D\psi_{i-1} \circ x \circ id_{Id})$
$D\phi_i : D_i \to D_{i+1} = (R\phi_{i-1} + id_{Nat} + \cdots)_\bot$
$D\psi_i : D_{i+1} \to D_i = (R\psi_{i-1} + id_{Nat} + \cdots)_\bot$, for $i > 0$

The inverse limits are $Record_\infty$ and $Denotable\text{-}value_\infty$. Two pairs of isomorphism maps result: $(R\Phi, R\Psi)$ and $(D\Phi, D\Psi)$. Elements of $Record_\infty$ represent record structures that map identifiers to values in $Denotable\text{-}value_\infty$. $Denotable\text{-}value_\infty$ contains $Record_\infty$ as a component, hence any $r \in Record_\infty$ exists as the denotable value in$Record_\infty(r)$. Actually, the previous sentence is a bit imprecise— $(Record_\infty + Nat + \cdots)_\bot$ contains $Record_\infty$ as a component, and an $r \in Record_\infty$ is embedded in $Denotable\text{-}value_\infty$ by writing $D\Psi(\text{in}Record_\infty(r))$. Like all the other inverse limit domains, $Denotable\text{-}value_\infty$ and $Record_\infty$ are domains of infinite tuples, and the isomorphism maps are necessary for unpacking and packing the denotable values and records.

Consider the recursively defined record:

$$r = [\,[\![A]\!] \mapsto \text{in}Nat(zero)\,]\,[\,[\![B]\!] \mapsto \text{in}Record_\infty(r)\,]\,(\lambda i.\, \bot)$$

Record r contains an infinite number of copies of itself. Any indexing sequence $(r\,[\![B]\!]\,[\![B]\!] \cdots [\![B]\!])$ produces r again. Since $Record_\infty$ is a pointed cpo, the recursive definition of r has a least fixed point solution, which is a tuple in $Record_\infty$. You should consider how the least fixed point solution is calculated in $Record_\infty$ and why the structure of r is more complex than that of a recursively defined record from a nonrecursive domain.

SUGGESTED READINGS

Inverse limit construction: Plotkin 1982; Reynolds 1972; Scott 1970, 1971, 1972; Scott & Strachey 1971

Generalizations & alternative approaches: Adamek & Koubek 1979; Barendregt 1977, 1981; Gunter 1985a, 1985b, 1985c; Kamimura & Tang 1984a; Kanda 1979; Lehman & Smyth 1981; Milner 1977; Scott 1976, 1983; Smyth & Plotkin 1982; Stoy 1977; Wand 1979

EXERCISES

1. Construct the approximating domains $D_0, D_1, D_2, \ldots, D_{i+1}$ for each of the following:

   a. $N = (Unit + N)_\bot$
   b. $Nlist = \mathbb{N}_\bot \times Nlist$
   c. $Mlist = (\mathbb{N} \times Mlist)_\bot$
   d. $P = P \to \mathbb{B}_\bot$
   e. $Q = (Q \to \mathbb{B})_\bot$

Describe the structure of D_∞ for each of the above.

2. Define the domain $D = D \rightarrow (D + Unit)_\perp$. What D_∞ element is the denotation of each of the following?

   a. $(\lambda d.\, \perp)$
   b. $(\lambda d.\, \mathrm{in}D(d))$
   c. $f = (\lambda d.\, \mathrm{in}D(f))$

3. Let $Nlist = (Nat^*)_\perp$ and $Natlist = (Unit + (Nat \times Natlist))_\perp$.

   a. What lists does *Natlist* have that *Nlist* does not?
   b. Define $cons : Nat \times Nlist \rightarrow Nlist$ for *Nlist*. Is your version strict in its second argument? Is it possible to define a version of *cons* that is nonstrict in its second argument? Define a *cons* operation for *Natlist* that is nonstrict in its second argument.
   c. Determine the denotation of $l \in Nlist$ in $l = zero\ cons\ l$ and of $l \in Natlist$ in $l = zero\ cons\ l$.
   d. A *lazy list* is an element of *Natlist* that is built with the nonstrict version of *cons*. Consider the list processing language in Figure 7.5. Make the *Atom* domain be *Nat*, and make the *List* domain be a domain of lazy lists of denotable values. Redefine the semantics. Write an expression in the language whose denotation is the list of all the positive odd numbers.

4. One useful application of the domain $Natlist = (Unit + (Nat \times Natlist))_\perp$ is to the semantics of programs that produce infinite streams of output.

   a. Consider the language of Figure 9.5. Let its domain *Answer* be *Natlist*. Redefine the command continuations $finish : Cmdcont$ to be $finish = (\lambda s.\, \mathrm{in}Unit())$ and $error : String \rightarrow Cmdcont$ to be $error = (\lambda t.\lambda s.\, \perp)$. Add this command to the language:

   $$\mathbf{C}[\![\mathbf{print}\ \mathrm{E}]\!] = \lambda e.\lambda c.\, \mathbf{E}[\![\mathrm{E}]\!](\lambda n.\lambda s.\ \mathrm{in}Nat{\times}Natlist(n,\ (c\, s)))$$

   Prove for $e \in Environment$, $c \in Cmdcont$, and $s \in Store$ that $\mathbf{C}[\![\mathbf{while}\ 1\ \mathbf{do}\ \mathbf{print}\ 0]\!]e\, c\, s$ is an infinite list of *zero*s.
   b. Construct a programming language with a direct semantics that can also generate streams. The primary valuation functions are $\mathbf{P}_D$: Program $\rightarrow Store \rightarrow Natlist$ and $\mathbf{C}_D$: Command $\rightarrow Environment \rightarrow Store \rightarrow Poststore$, where $Poststore = Natlist \times Store_\perp$. (Hint: make use of an operation $strict : (A \rightarrow B) \rightarrow (A_\perp \rightarrow B)$, where B is a pointed cpo, such that:

   $strict(f)(\perp) = \perp_B$, that is, the least element in B
   $strict(f)(a) = f(a)$, for a proper value $a \in A$

   Then define the composition of command denotations $f, g \in Store \rightarrow Poststore$ as:

   $$g{*}f = \lambda s.\, (\lambda(l, p).\, (\lambda(l', p').\, (l\ append\ l', p'))(strict(g)(p)))(f\, s)$$

   where $append : Natlist \times Natlist \rightarrow Natlist$ is the list concatenation operation and is nonstrict in its second argument.) Prove that $\mathbf{C}_D[\![\mathbf{while}\ 1\ \mathbf{do}\ \mathbf{print}\ 0]\!]e\, s$ is an infinite list of *zero*s.

c. Define a programming language whose programs map an infinite stream of values and a store to an infinite stream of values. Define the language using both direct and continuation styles. Attempt to show a congruence between the two definitions.

5. a. The specification of record r in Section 11.3 is incomplete because the isomorphism maps are omitted. Insert them in their proper places.

   b. How can recursively defined records be used in a programming language? What pragmatic disadvantages result?

6. a. Why does the inverse limit method require that a domain expression f in $D = F(D)$ map pointed cpos to pointed cpos?

   b. Why must we work with r-pairs when an inverse limit domain is always built from a sequence of retraction pairs?

   c. Can a retraction sequence have a domain D_0 that is not $\{\perp\}$? Say that a retraction sequence had $D_0 = Unit_\perp$. Does an inverse limit still result? State the conditions under which $\mathbb{B}_\perp$ could be used as D_0 in a retraction sequence generated from a domain expression F in $D = F(D)$. Does the inverse limit satisfy the isomorphism? Is it the least such domain that does so?

7. a. Show the approximating domains $D_0, D_1, \ldots, D_i$ for each of the following:

   i. $D = D \rightarrow D$
   ii. $D = D_\perp \rightarrow D_\perp$
   iii. $D = (D \rightarrow D)_\perp$

   b. Recall the lambda calculus system that was introduced in exercise 11 of Chapter 3. Once again, its syntax is:

   $$\text{E} ::= (E_1\, E_2) \mid (\lambda I.\, E) \mid I$$

   Say that the meaning of a lambda-expression $(\lambda I.E)$ is a function. Making use of the domain $Environment = Identifier \rightarrow D$ with the usual operations, define a valuation function $\mathbf{E}$: Lambda-expression $\rightarrow Environment \rightarrow D$ for each of the three versions of D defined in part a. Which of the three semantics that you defined is extensional; that is, in which of the three does the property ''(for all $[\![E]\!]$, $\mathbf{E}[\![(E_1\, E)]\!] = \mathbf{E}[\![(E_2\, E)]\!]$) implies $\mathbf{E}[\![E_1]\!] = \mathbf{E}[\![E_2]\!]$'' hold? (Warning: this is a nontrivial problem.)

   c. For each of the three versions of $\mathbf{E}$ that you defined in part b, prove that the β-rule is sound, that is, prove:

   $$\mathbf{E}[\![(\lambda I.\, E_1)E_2]\!] = \mathbf{E}[\![[E_2/I]E_1]\!]$$

   d. Augment the syntax of the lambda-calculus with the abstraction form (λ**val** $I.E$). Add the following reduction rule:

   β**val**-rule: $(\lambda \mathbf{val}\, I.\, E_1)E_2 \;\Rightarrow\; [E_2/I]E_1$
   where E_2 is not a combination $(E_1{'}\, E_2{'})$

Define a semantics for the new version of abstraction for each of the three versions of valuation function in part b and show that the β**val**-rule is sound with respect to each.

8. For the definition $D = D \to D$, show that an inverse limit can be generated starting from $D_0 = Unit_\perp$ and that D_∞ is a nontrivial domain. Prove that this inverse limit is the smallest nontrivial domain that satisfies the definition.

9. Scott proposed the following domain for modelling flowcharts:

   $C = (Skip + Assign + Comp + Cond)_\perp$
   where $Skip = Unit$ represents the **skip** command
   $Assign$ is a set of primitive assigment commands
   $Comp = C \times C$ represents command composition
   $Cond = Bool \times C \times C$ represents conditional
   $Bool$ is a set of primitive Boolean expressions

   a. What relationship does domain C have to the set of derivation trees of a simple imperative language? What ‘‘trees’’ are lacking? Are there any extra ones?
   b. Let $wh(b, c)$ be the C-value $wh(b, c)$ = in$Cond$(b, in$Comp$(c, $wh(b, c)$), in$Skip$()), for $c \in C$ and $b \in Bool$. Using the methods outlined in Section 6.6.5, draw a tree-like picture of the denotation of $wh(b_0, c_0)$. Next, write the tuple representation of the value as it appears in C_∞.
   c. Define a function $sem : C \to Store_\perp \to Store_\perp$ that maps a member of C to a store transformation function. Define a congruence between the domains $Bool$ and Boolean-expr and between $Assign$ and the collection of trees of the form [[I:=E]]. Prove for all $b \in Bool$ and its corresponding [[B]] and for $c \in C$ and its corresponding [[C]] that $sem(wh(b, c))$ = **C**[[**while** B **do** C]].

10. The domain $I = (Dec \times I)_\perp$, where $Dec = \{0, 1, \cdots, 9\}$, defines a domain that contains infinite lists of decimal digits. Consider the interval [0,1], that is, all the real numbers between 0 and 1 inclusive.

    a. Show that every value in [0,1] has a representation in I. (Hint: consider the decimal representation of a value in the interval.) Are the representations unique? What do the partial lists in I represent?
    b. Recall that a number in [0,1] is *rational* if it is represented by a value m/n, for $m, n \in \mathbb{N}$. Say that an I-value is *recursive* if it is representable by a (possibly recursive) function expression $f = \alpha$. Is every rational number recursive? Is every recursive value rational? Are there any nonrecursive values in I?
    c. Call a domain value *transcendental* if it does not have a function expression representation. State whether or not there are any transcendental values in the following domains. ($\mathbb{N}$ has none.)

       i. $\mathbb{N} \times \mathbb{N}$
       ii. $\mathbb{N} \to \mathbb{N}$
       iii. $Nlist = (\mathbb{N} \times Nlist)_\perp$

iv. $N = (Unit + N)_\perp$

11. Just as the inverse limit D_∞ is determined by its approximating domains D_i, a function $g : D_\infty \to C$ is determined by a family of approximating functions. Let $G = \{ g_i : D_i \to C \mid i \geq 0 \}$ be a family of functions such that, for all $i \geq 0$, $g_{i+1} \circ \phi_i = g_i$. (That is, the maps always agree on elements in common.)

   a. Prove that for all $i \geq 0$, $g_i \circ \psi_i \sqsubseteq g_{i+1}$.
   b. Prove that there exists a unique $g : D_\infty \to C$ such that for all $i \geq 0$, $g \circ \theta_{i\infty} = g_i$. Call g the *mediating morphism for G*, and write $g = med\, G$.
   c. For a continuous function $h : D_\infty \to C$, define the family of functions $H = \{ h_i : D_i \to C \mid i \geq 0,\ h_i = h \circ \theta_{i\infty} \}$.

      i. Show that $h_{i+1} \circ \phi_i = h_i$ for each $i \geq 0$.
      ii. Prove that $h = med\, H$ and that $H = \{ (med\, H) \circ \theta_{i\infty} \mid i \geq 0 \}$.

      Thus, the approximating function families are in 1-1, onto correspondence with the continuous functions in $D_\infty \to C$.
   d. Define the approximating function family for the map $hd : Alist \to A_\perp$, $Alist$ = $(Unit + (A \times Alist))_\perp$, hd = $\lambda l.$ cases $\Phi(l)$ of is$Unit() \to \perp$ [] is$A{\times}Alist(a, l) \to a$ end. Describe the graph of each hd_i.
   e. Let $p_i : D_i \to \mathbb{B}$ be a family of continuous predicates. Prove that for all $i \geq 0$, p_i holds for $d_i \in D_i$, (that is, $p_i(d_i) = true$) iff $med\{ p_i \mid i \geq 0 \}(d) = true$, where $d = (d_0, d_1, \cdots, d_i, \cdots) : D_\infty$. Conversely, let $P : D_\infty \to \mathbb{B}$ be a continuous predicate. Prove that P holds for a $d \in D_\infty$ (that is, $P(d) = true$) iff for all $i \geq 0$, $P \circ \theta_{i\infty}(d_i) = true$.
   f. Results similar to those in parts a through c hold for function families $\{ f_i : C \to D_i \mid i \geq 0 \}$ such that for all $i \geq 0$, $f_i = \psi_i \circ f_{i+1}$. Prove that there exists a unique $f : C \to D_\infty$ such that for all $i \geq 0$, $f_i = \theta_{\infty i} \circ f$.