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, \( Bintree = (Data + (Data \times Bintree \times Bintree))_1 \), 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 \( Alist = (Nil + (A \times Alist))_1 \), 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 \( \text{cons}, \text{hd}, \text{tl}, \text{and null} \) 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 \( Proc = Param \rightarrow Store \rightarrow Store_1 \), where \( Param = \text{Int} + \text{Real} + \cdots + Proc \), 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 \( (fa) \) 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 \( \perp \). \( D_0 \) 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 \( Alist = (\text{Nil} + (A \times Alist))_1 \) as an example. \( \text{Nil} = \text{Unit} \) represents the empty list, and a nonempty list of \( A \)-elements has the structure \( A \times Alist \). An \( Alist \) can also be undefined. Domain \( Alist_0 = \{ \perp \} \), and for each \( i > 0 \), \( Alist_{i+1} = (\text{Nil} + (A \times Alist_i))_1 \). To get started, we draw \( Alist_1 = (\text{Nil} + (A \times Alist_0))_1 \) as:

\[
\begin{align*}
\text{Nil} & \quad A \times \{ \perp \} \\
\perp & 
\end{align*}
\]

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

Next, \( Alist_2 = (\text{Nil} + (A \times Alist_1))_1 \). The product \( A \times Alist_1 \) can be visualized as a union of three distinct sets of elements: \( \{ [a, \perp] \mid a \in A \} \), the set of partial lists of one element; \( \{ [a, \text{nil}] \mid a \in A \} \), the set of proper lists of one element, and \( \{ [a_1, a_2, \perp] \mid a_1, a_2 \in A \} \), the set of partial lists of two elements. Drawing \( A \times Alist_1 \) with these sets, we obtain:

\[
\begin{align*}
A \times \text{Nil} & \quad A \times A \times \{ \perp \} \\
\text{Nil} & \quad A \times \{ \perp \} \\
\perp & 
\end{align*}
\]

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 \( \perp; \text{nil} \); 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 \( \perp \) serves double duty: it represents both a nontermination situation and a “don’t know yet” situation. That is, a list \( [a_0, a_1, \ldots] \) 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 \( Alist_{\text{fin}} = \bigcup_{i=0}^{\infty} Alist_i \), partially ordered to be consistent with the
\( A\text{list}_I \)'s.

(That is, \( \mathbb{F}_\text{list}_I \) \( x \) iff there exists some \( j \geq 0 \) such that \( \mathbb{F}_\text{list}_I \) \( x \).) Domain \( \text{Alist}_\text{fun} \) contains \( \bot, \text{nil} \), 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: \( \text{Alist}_\text{fun} \) is not a cpo, for the chain \( \bot, [a_0, \cdot \cdot \cdot], [a_0, a_1, \cdot \cdot \cdot], \cdot \cdot \cdot \) does not have a least upper bound in \( \text{Alist}_\text{fun} \).

The obvious remedy to the problem is to add the needed upper bounds to the domain. The lub of the aforementioned chain is the \textit{infinite} list \( [a_0, a_1, \cdot \cdot \cdot, a_i, \cdot \cdot \cdot] \). It is easy to see where the infinite lists would be added to the domain. The result, called \( \text{Alist}_\infty \), is:

\[
\prod_{i=0}^{\infty} A = A \times A \times A \times \cdots
\]

\[
A \times A \times \text{Nil} \quad A \times A \times \{ \bot \}
\]

\[
A \times \text{Nil} \quad A \times \{ \bot \}
\]

\[
\text{Nil} \quad A \times \{ \bot \}
\]

\[
\bot
\]

The infinite elements are a boon; realistic computing situations involving infinite data structures are now expressible and understandable. Consider the list \( l \) specified by \( l = (a \ \text{cons} \ l) \), for \( a \in A \). The functional \((\lambda l. a \ \text{cons} \ l): \text{Alist}_\infty \to \text{Alist}_\infty \) has as its least fixed point \( [a, a, a, \cdot \cdot \cdot] \), the infinite list of \( a \)'s. We see that \( l \) satisfies the properties \((\text{hd} l) = a, (tl l) = l, \) and \((\text{null} l) = \text{false} \). The term \textit{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 \( \text{hd} \)).

\( \text{Alist}_\infty \) 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 \( \phi_i: \text{Alist}_I \to \text{Alist}_{i+1} \) for \( i \geq 0 \). Each \( \phi_i \) embeds \( \text{Alist}_I \) into \( \text{Alist}_{i+1} \). By continuity, the partial ordering and lubs in \( \text{Alist}_I \) are preserved in \( \text{Alist}_{i+1} \). However, it is easy to find \( \phi \) functions that do an "embedding" that is unnatural (e.g., \( \phi = \lambda x. \bot \)). To guarantee that the function properly embeds \( \text{Alist}_I \) into \( \text{Alist}_{i+1} \), we also define a family of continuous functions \( \psi_i: \text{Alist}_{i+1} \to \text{Alist}_I \) that map the elements in \( \text{Alist}_{i+1} \) to those in \( \text{Alist}_I \) that best approximate them. \( \text{Alist}_I \) is a subdomain of \( \text{Alist}_{i+1} \) when \( \psi_i \circ \phi_i = \text{id}_{\text{Alist}_I} \) holds; that is, every element in \( \text{Alist}_I \) can be embedded by \( \phi_i \) and recovered by \( \psi_i \). To force the embedding of \( \text{Alist}_I \) into the "lower portion" of \( \text{Alist}_{i+1} \), we also require that \( \phi_i \circ \psi_i = \text{id}_{\text{Alist}_I} \). This makes it clear that the new elements in \( \text{Alist}_{i+1} \) not in \( \text{Alist}_I \) "grow out" of \( \text{Alist}_I \).
The function pairs \((\phi_i, \psi_i)\), \(i \geq 0\), are generated from the recursive specification. To get started, we define \(\phi_0 : \text{Alist}_0 \rightarrow \text{Alist}_1 = \lambda x. \text{Alist}_0(x)\) and \(\psi_0 : \text{Alist}_1 \rightarrow \text{Alist}_0\) as \((\lambda x. \text{Alist}_i(x))\). It is easy to show that the \((\phi_0, \psi_0)\) pair satisfies the two properties mentioned above. For every \(i > 0\):

\[
\phi_i : \text{Alist}_i \rightarrow \text{Alist}_{i+1} = \lambda x. \text{cases } x \text{ of } \begin{cases} 
\text{isNil()} & \rightarrow \text{inNil()} \\
\lbrack \rbrack & \rightarrow \text{Alist}_{i-1}(a, l) \rightarrow \text{inAlist}_{i-1}(a, \phi_{i-1}(l)) \end{cases}
\]

The embedding is based on the structure of the argument from \(\text{Alist}_i\). The structures of undefined and empty lists are preserved, and a list with head element \(a \in \text{Alist}\) and tail \(l \in \text{Alist}_{i-1}\) is mapped into a pair \((a, \phi_{i-1}(l))\). To get an intuition about the \(\text{Alist}_i\) by analyzing its structure and using \(\psi_{i-1}\) where needed. A mathematical induction proof shows that each pair \((\phi_i, \psi_i)\) satisfies the required properties.

A chain-like sequence has been created:

\[
\begin{array}{cccccc}
\psi_0 & \psi_1 & \psi_2 & \cdots & \psi_i & \psi_{i+1} & \cdots \\
\text{Alist}_0 & \text{Alist}_1 & \text{Alist}_2 & \cdots & \text{Alist}_i & \text{Alist}_{i+1} & \cdots \\
\phi_0 & \phi_1 & \phi_2 & \cdots & \phi_i & \end{array}
\]

What is the “lub” of this chain? (It will be \(\text{Alist}_\omega\).) To give us some intuition about the lub, we represent the elements of an \(\text{Alist}\) domain as tuples. An element \(x \in \text{Alist}\) appears as an \((i+1)\)-tuple of the form \((x_0, x_1, \ldots, x_{i-1}, x_i)\), where \(x_i = x, x_{i-1} = \psi_{i-1}(x_i), \ldots, x_1 = \psi_1(x_2)\), and \(x_0 = \psi_0(x_1)\). For example, \([a_0, a_1, \text{nil}]\) has tuple form \((\lbrack, [a_0, \ldots], [a_0, a_1, \ldots], [a_0, a_1, \text{nil}]\)\); \([a_0, a_1, a_2, \ldots]\) has tuple form \((\lbrack, [a_0, \ldots], [a_0, a_1, \ldots], [a_0, a_1, a_2, \ldots])\); \([a_0, \text{nil}]\) has tuple form \((\lbrack, [a_0, \ldots], [a_0, \text{nil}], [a_0, \text{nil}]\)\); and \((\lbrack, [a_0, \ldots])\) has tuple form \((\lbrack, [a, \ldots])\). The tuples trace the incrementation of information in an element until the information is complete. They suggest that the limit domain of the chain, \(\text{Alist}_\omega\), has elements whose tuple representations have infinite length. A finite list \(x\) with \(i \in \text{Alist}_\omega\) and has tuple representation \((x_0, x_1, \ldots, x_{i-1}, x, x, \ldots)\) — it stabilizes. The infinite lists have tuple representations that never stabilize: for example, an infinite list of \(a\)‘s has the representation \((\lbrack, [a, \ldots], [a, a, \ldots], [a, a, a, \ldots], [a, a, a, a, \ldots])\). 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 \(\text{Alist}_\omega\) to be the set of infinite tuples induced from the \(\text{Alist}_i\)‘s and the \(\psi_i\)‘s:
\( \text{Alist}_\infty = \{ (x_0, x_1, \ldots, x_i, \ldots) \mid \text{for all } n \geq 0, x_n \in \text{Alist}_n \text{ and } x_n = \psi_n(x_{n+1}) \} \)

partially ordered by, for all \( x, y \in \text{Alist}_\infty \), \( x \leq y \) iff for all \( n \geq 0 \), \( x \downarrow n \leq y \downarrow n \). \( \text{Alist}_\infty \) contains only those tuples with information consistent with the \( \text{Alist}_i \)'s. The partial ordering is the natural one for a subdomain of a product domain.

Now we must show that \( \text{Alist}_\infty \) satisfies the recursive specification; that is, \( \text{Alist}_\infty = (\text{Nil} + (A \times \text{Alist}_\infty)) \downarrow \). 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 \( \Phi : \text{Alist}_\infty \to (\text{Nil} + (A \times \text{Alist}_\infty)) \downarrow \) and \( \Psi : (\text{Nil} + (A \times \text{Alist}_\infty)) \downarrow \to \text{Alist}_\infty \) such that \( \Psi \circ \Phi = \text{id}_{\text{Alist}_\infty} \) and \( \Phi \circ \Psi = \text{id}_{(\text{Nil} + (A \times \text{Alist}_\infty)) \downarrow} \). The \( \Phi \) function exposes the list structure inherent in an \( \text{Alist}_\infty \) element, and the \( \Psi \) map gives the tuple representation of list structured objects.

The isomorphism property is strong enough that \( \text{Alist}_\infty \) may be considered a solution of the specification. The \( \Phi \) and \( \Psi \) maps are used in the definitions of operations on the domain. For example, \( \text{head} : \text{Alist}_\infty \to A \downarrow \) is defined as:

\[
\text{head} = \lambda x. \text{cases } \Phi(x) \text{ of isNil() } \Rightarrow [] \text{ is } A \times \text{Alist}_\infty(a, l) \to a \text{ end}
\]
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 $D_\infty$ that is isomorphic to $F(D_\infty)$. $D_\infty$ 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 $\phi_i$ and $\psi_i$ maps.

11.1 Definition:

For pointed cpos $D$ and $D_\infty$, a pair of continuous functions $(f : D \to D_\infty, g : D_\infty \to D)$ is a retraction pair iff:

1. $g \circ f = \text{id}_D$
2. $f \circ g = \text{id}_{D_\infty}$

$f$ 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 $(f_1 : D \to D_\infty, g_1 : D_\infty \to D)$ and $(f_2 : D_\infty \to D_\infty, g_2 : D_\infty \to D_\infty)$ is itself a retraction pair.

Proof: $(g_1 \circ g_2) \circ (f_2 \circ f_1) = g_1 \circ (g_2 \circ f_2) \circ f_1 = g_1 \circ \text{id}_{D_\infty} \circ f_1 = g_1 \circ f_1 = \text{id}_D$. The proof that $(f_2 \circ f_1) \circ (g_1 \circ g_2) \circ \text{id}_{D_\infty}$ 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, \(f \circ g_1 = \text{id}_D\), which implies \(g_2 \circ f \circ g_2 \circ \text{id}_D\) by the monotonicity of \(g_2\). But \(g_2 \circ f \circ g_1 = (g_2 \circ f) \circ g_1 = \text{id}_D \circ g_1 = g_1\), implying \(g_1 = g_2\). Repeating the above derivation with \(g_1\) and \(g_2\) swapped gives \(g_2 = g_1\), 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 \((f, g) : D \leftrightarrow D\). The operations on \(r\)-pairs are:

1. Composition: for \((f_1, g_1) : D \leftrightarrow D'\) and \((f_2, g_2) : D' \leftrightarrow D'\),
   \[(f_2 \circ f_1, g_1 \circ g_2) : D \leftrightarrow D'\] is defined as \((f_2 \circ f_1, g_1 \circ g_2)\).

2. Reversal: for \((f, g) : D \leftrightarrow D\), \((f, g)^R : D \leftrightarrow D\) is defined as \((g, f)\).

The reversal of a retraction pair might not be a retraction pair. The identity \(r\)-pair for the domain \(D \leftrightarrow D\) is \(\text{id}_D \leftrightarrow D = (\text{id}_D, \text{id}_D)\). It is easy to show that the composition and reversal operations upon \(r\)-pairs are continuous. We use the letters \(r, s, t, \ldots\) 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 \((\phi_0, \psi_0)\). 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 } \text{isC}(c) \rightarrow \text{inE}(f(c)) \mid \text{isC}(c) \rightarrow \text{inE}(f(c)) \text{ end},
   
   (\lambda x. \text{cases } y \text{ of } \text{isE}(e) \rightarrow \text{inC}(g(e)) \mid \text{isE}(e) \rightarrow \text{inC}(g(e)) \text{ end}))
   
   : C + C' \leftrightarrow E + E',
   \]
3. \( r \rightarrow s \) denote:
   \[
   ((\lambda x. f_1 \cdot x \cdot g), (\lambda y. g_1 \cdot y \cdot f)) : (C \rightarrow C_1) \leftrightarrow (E \rightarrow E_1)
   \]
4. \( (r)_i \) denote:
   \[
   ((\lambda x. f x), (\lambda y. g y)) : C_i \leftrightarrow E_i
   \]

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 \( \text{Nlist} = (\text{Nil} + (\text{Nat} \times \text{Nlist})) \) gives a construction \( F(D) = ((\text{Nil} + (\text{Nat} \times D)) \) such that, for any cpo \( A \), \( (\text{Nil} + (\text{Nat} \times A)) \) is also a cpo, and for any r-pair \( r \), \( (\text{Nil} + (\text{Nat} \times r)) \) is an r-pair. The r-pair is constructed using Definition 11.7; the r-pairs corresponding to \( \text{Nil} \) and \( \text{Nat} \) in the example are the identity r-pairs (\( \text{id}_\text{Nil} \), \( \text{id}_\text{Nat} \)) and (\( \text{id}_\text{Nat}, \text{id}_\text{Nat} \)), 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. \( F(\text{id}_E) = \text{id}_{F(E)} \leftrightarrow F(E) \)
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 \( \{(D_i \mid i \geq 0), \{r_i : D_i \leftrightarrow D_{i+1} \mid i \geq 0 \}\} \) 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 \( t_{mn} : D_m \leftrightarrow D_n \) be defined as:

\[
\begin{align*}
t_{mn} &= \begin{cases} 
  r_{n-1} \circ \cdots \circ r_m & \text{if } m < n \\
  \text{id}_{D_m} \leftrightarrow D_n & \text{if } m = n \\
  r_n^R \circ \cdots \circ r_{m-1}^R & \text{if } m > n
\end{cases}
\end{align*}
\]
To make this clear, let each $r_i$ be the r-pair $(\varphi_i: D_i \rightarrow D_{i+1}, \psi_i: D_{i+1} \rightarrow D_i)$ and each $t_{mn}$ be the r-pair $(\theta_{mn}: D_m \rightarrow D_n, \theta_{nm}: D_n \rightarrow D_m)$. Then for $m < n$, $t_{mn} = (\theta_{mn}, \theta_{nm}) = (\theta_{n-1} \circ \cdots \circ \theta_{m+1} \circ \theta_m, 
abla_m \circ \psi_{m+1} \circ \cdots \circ \psi_{n-1})$, which is drawn as:

$$
\begin{array}{cccccc}
\varphi_m & \varphi_{m+1} & \cdots & \varphi_{n-1} & \\
D_m & D_{m+1} & D_{m+2} & \cdots & D_{n-1} & D_n \\
\psi_m & \psi_{m+1} & \cdots & \psi_{n-1} & \\
\end{array}
$$

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 $\theta_{mn}$’s is consistent.

11.10 Proposition:

For any retraction sequence and $m,n,k \geq 0$:

1. $t_{mn} \circ t_{kn} = t_{mn} \circ t_{lm} = t_{km}$
2. $t_{mn} \circ t_{nm} = t_{kn}$, when $m \geq k$ or $m \geq n$
3. $t_{mn}$ is a retraction pair when $m \leq n$

Proof: Left as an exercise.

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 $\psi_i$ embeddings.

11.11 Definition:

The inverse limit of a retraction sequence:

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

is the set:
\[ D_\infty = \{ (x_0, x_1, \cdots, x_i, \cdots) \mid \text{for all } n > 0, \ x_n \in D_n \text{ and } x_n = \psi_n(x_{n+1}) \} \]
partially ordered by the relation: for all \( x, y \in D_\infty \), \( x \preceq y \) if for all \( n \geq 0 \), \( x_m \preceq y_m \) for all \( m \).

**11.12 Theorem:**

\( D_\infty \) is a pointed cpo.

**Proof:** Recall that each \( D_i \) in the retraction sequence is a pointed cpo. First, \( \downarrow D_\infty = (\downarrow D_0, \downarrow D_1, \cdots, \downarrow D_n, \cdots) \in D_\infty \), since every \( \psi_i(\downarrow D_\infty) = \downarrow D_i \), by Proposition 11.4.

Second, for any chain \( C = \{ c_i \mid i \in I \} \) in \( D_\infty \), the definition of the partial ordering on \( D_\infty \) makes \( C_n = \{ c_i \downarrow n \mid i \in I \} \) a chain in \( D_n \) with a lub of \( \bigcup C_n, n \geq 0 \). Now \( \psi_n(\bigcup C_{n+1}) = \bigcup \{ \psi_n(c_i \downarrow (n+1)) \mid i \in I \} = \bigcup \{ c_i \downarrow n \mid i \in I \} = \bigcup C_n \).

Hence \( (\bigcup C_0, \bigcup C_1, \cdots, \bigcup C_i, \cdots) \) belongs to \( D_\infty \). 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:

\[
(\{ D_i \mid D_0 = \{ \bot \}, D_{i+1} = F(D_i), \ for \ i \geq 0 \},
(\phi_i, \psi_i) : D_i \leftrightarrow D_{i+1} \mid \phi_0 = (\lambda x. \downarrow D_0), \ \psi_0 = (\lambda x. \downarrow D_0),
(\phi_{i+1}, \psi_{i+1}) = F(\phi_i, \ psi_i), \ for \ i \geq 0) \]

is a retraction sequence.

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

Thus, the inverse limit \( D_\infty \) exists for the retraction sequence generated by \( F \). The final task is to show that \( D_\infty \) is isomorphic to \( F(D_\infty) \) by defining functions \( \Phi : D_\infty \rightarrow F(D_\infty) \) and \( \Psi : F(D_\infty) \rightarrow D_\infty \) such that \( \Psi \circ \Phi = \text{id}_{D_\infty} \) and \( \Phi \circ \Psi = \text{id}_{F(D_\infty)} \). Just as the elements of \( D_\infty \) were built from elements of the \( D_i \)'s, the maps \( \Phi \) and \( \Psi \) are built from the retraction pairs \((\phi_i, \psi_i)\).

For \( m \geq 0 \):

\[
t_{m, m} : D_m \leftrightarrow D_m \text{ is:}
(\theta_{m, m}, \theta_{m, m}) = (\lambda x. (\theta_{m, 0}(x), \theta_{m, 1}(x), \cdots, \theta_{m, m}(x), \cdots)), (\lambda x. \downarrow m) \)
\]

\[
t_{m, \infty} : D_m \rightarrow D_\infty \text{ is:}
(\theta_{m, \infty}, \theta_{m, \infty}) = t^R_{m, \infty}
\]

\[
t_{\infty, \infty} : D_\infty \rightarrow D_\infty \text{ is:}
(\theta_{\infty, \infty}, \theta_{\infty, \infty}) = (\text{id}_{D_\infty}, \text{id}_{D_\infty})
\]

You are given the exercises of showing that \( \theta_{m, \infty} : D_m \rightarrow D_\infty \) is well defined and proving the following proposition.
11.14 Proposition:

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

Since each $t_{mn}$ is a retraction pair, the value $\theta_m(\theta_n(x))$ 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 $\infty$, the approximations approach identity.

11.15 Lemma:

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

Proof: For every $m \geq 0$, $t_{m\infty}$ is a retraction pair, so $\theta_m \circ \theta_m \circ id_{D_\infty}$. Because \{ $\theta_m \circ \theta_m$ | $m \geq 0$ \} is a chain in $D_\infty \rightarrow D_\infty$, $\bigsqcup_{m=0}^{\infty} \theta_m \circ \theta_m \circ id_{D_\infty}$ holds. Next, for any $x = (x_0, x_1, \ldots, x_i, \ldots) \in D_\infty$ and any $i \geq 0$, $\theta_m(\theta_m(x)) = (\theta_m(x), \theta_m(x), \ldots, \theta_m(x), \ldots)$. Since $\theta_m(x) = (x, x, \ldots)$, each $m$th component of tuple $x$ will appear as the $m$th component in $\theta_m(\theta_m(x))$, for all $m \geq 0$. So $x \Rightarrow \bigsqcup_{m=0}^{\infty} (\theta_m \circ \theta_m)(x) = (\bigsqcup_{m=0}^{\infty} \theta_m \circ \theta_m)(x)$. By extensionality, $id_{D_\infty} = \bigsqcup_{m=0}^{\infty} \theta_m \circ \theta_m$, which implies the result. \qed

11.16 Corollary:

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

11.17 Corollary:

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

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

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

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

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

The isomorphism maps are defined as a retraction pair ($\Phi$, $\Psi$) 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 $D_\infty \leftrightarrow F(D_\infty)$.
(\Phi, \Psi) : D_\infty \leftrightarrow F(D_\infty) = \bigcap_{n=0}^{\infty} F(t_{m=1}) \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} \leftrightarrow D_\infty \text{ and } (\Phi, \Psi) \circ (\Phi, \Psi)^R = id_{F(D_\infty)} \leftrightarrow F(D_\infty). \text{ The proofs require the following technical lemmas.}

11.18 Lemma:

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

Proof: \( F(t_{m=1}) \circ (\Phi, \Psi) \)

\[ = F(t_{m=1}) \circ \bigcap_{n=0}^{\infty} F(t_{m=1}) \circ t_{\infty(n+1)} \]

\[ = \bigcap_{n=0}^{\infty} F(t_{m=1}) \circ F(t_{m=1}) \circ t_{\infty(n+1)}, \text{ by continuity} \]

\[ = \bigcap_{n=0}^{\infty} F(t_{m=1} \circ t_{m=1}) \circ t_{\infty(n+1)}, \text{ by Lemma 11.8, part 2} \]

\[ = \bigcap_{n=0}^{\infty} F(t_{m=1}) \circ t_{\infty(n+1)}, \text{ by Proposition 11.14} \]

\[ = \bigcap_{n=0}^{\infty} t_{\infty(n+1)} \circ t_{\infty(n+1)} \]

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

11.19 Lemma:

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

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} \leftrightarrow D_\infty

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

\[ = \bigcap_{n=0}^{\infty} (F(t_{m=1}) \circ t_{\infty(m+1)})^R \circ (\Phi, \Psi), \text{ by continuity of } R \]

\[ = \bigcap_{n=0}^{\infty} t_{\infty(m+1)}^R \circ F(t_{m=1}) \circ (\Phi, \Psi), \text{ by Proposition 11.6} \]

\[ = \bigcap_{n=0}^{\infty} t_{\infty(m+1)^{\infty}} \circ F(t_{m=1}^{\infty}) \circ (\Phi, \Psi), \text{ by Lemma 11.8, part 3} \]

\[ = \bigcap_{n=0}^{\infty} t_{m=1}^{\infty} \circ t_{\infty(m+1)} \circ (\Phi, \Psi), \text{ by continuity} \]

\[ = \bigcap_{n=0}^{\infty} t_{m=1}^{\infty} \circ t_{\infty(m+1)} \circ (\Phi, \Psi) \text{, by Lemma 11.18} \]

\[ = \bigcap_{n=0}^{\infty} F(t_{m=1}) \circ t_{\infty(m+1)} \circ t_{m=1} \]
= \text{id}_{D_\infty} \leftrightarrow D_\infty, \text{ by Corollary 11.16} \square

11.21 Theorem:

$$(\Phi, \Psi) \circ (\Phi, \Psi)^R = \text{id}_{F(D_\infty) \leftrightarrow F(D_\infty)}$$

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 $D_\infty$ is a “fixed point” of the “chain” generated by a “functional” $F$. To complete the list of parallels, we can show that $D_\infty$ 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 $(\Phi, \Psi)$ proves that $D_\infty$ is isomorphic to $F(D_\infty)$. Then define the following r-pairs:

$t_{(m+1)\infty} : D_{m+1} \leftrightarrow D_\infty$ as $(\Psi', \Phi_1) \circ F(t_m\infty)$

Each $t_m\infty$ is a retraction pair, and $\{ t_m\infty \circ t_{m+1}\infty \mid m \geq 0 \}$ is a chain in $D_\infty \leftrightarrow D_\infty$. Next, define $(\alpha, \beta) : D_\infty \leftrightarrow D_\infty$ to be $\bigcup_{m \geq 0} t_m\infty \circ t_{m+1}\infty$. We can show that $(\alpha, \beta)$ is a retraction pair; that is, $D_\infty$ embeds into $D_\infty$. Since $D_\infty$ is arbitrary, $D_\infty$ must be the least pointed cpo solution to the retraction sequence.

We gain insight into the structure of the isomorphism maps $\Phi$ and $\Psi$ 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 $(\Phi, \Psi)$, a retraction pair, can be made much clearer:

$$(\Phi, \Psi) = \bigcup_{m=0}^{\infty} F(t_m\infty) \circ t_{m+1}\infty$$

$$= \bigcup_{m=0}^{\infty} F(\theta_m\infty, \theta_m\infty) \circ (\theta_{m+1}\infty, \theta_{m+1}\infty)$$
= \bigcup_{m \geq 0} (F\Theta_{m \infty} \circ \Theta_{m+1} \circ F\Theta_{m \infty})
= \bigcup_{m \geq 0} (F\Theta_{m \infty} \circ \Theta_{m+1} \circ F\Theta_{m \infty})
= (\bigcup_{m \geq 0} F\Theta_{m \infty} \circ \Theta_{m+1} \circ \bigcup_{m \geq 0} F\Theta_{m \infty})

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 $D_{\infty}$. 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 $D_{\infty}$ 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 $Alist = (Nil + (A \times Alist)) \downarrow$, the retraction sequence is $(\{ D_n \mid n \geq 0 \}, \{ (\phi_n, \psi_n) \mid n \geq 0 \})$, where:

\[ D_0 = \{ \bot \} \]
\[ D_{i+1} = (Nil + (A \times D_i)) \downarrow \]

and

\[ \phi_0 : D_0 \rightarrow D_1 = (\lambda x. \bot) \]
\[ \psi_0 : D_1 \rightarrow D_0 = (\lambda x. \bot) \]
\[ \phi_i : D_i \rightarrow D_{i+1} = (id_{Nil} + (id_A \times \phi_{i-1})) \downarrow \]
\[ = \lambda x. \text{cases } x \text{ of} \]
\[ \text{isNil()} \rightarrow \text{inNil()} \]
\[ [] \rightarrow \text{inA} \rightarrow D_{i+1}(a, \phi_{i-1}(d)) \text{ end} \]
\[ \psi_l : D_{i+1} \rightarrow D_l = (id_{\text{Nil}} + (id_A \times \psi_{l-1})) \]

\[ = \lambda x. \text{cases } x \text{ of } \]

\[ \text{isNil()} \rightarrow \text{inNil()} \]

\[ [] \text{isAxD}_l(a,d) \rightarrow \text{inAxD}_{l-1}(a, \psi_{l-1}(d)) \text{ end} \]

An element in \( D_n \) is a list with \( n \) or less \( A \)-elements. The map \( \theta_{mm} : D_m \rightarrow D_n \) 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 \( \bot \). If \( m \leq n \), the list is embedded intact into \( D_n \). An \( A\text{list}_m \) element is a tuple \( x = (x_0, x_1, \ldots, x_i, \ldots) \), 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 \( x_i = \psi_l(x_{i+1}) \). The map \( \theta_{mm} : A\text{list}_m \rightarrow D_m \) projects an infinite tuple into a list of \( m \) elements \( \theta_{mm} (x) = x_i \), and \( \theta_{mm} : D_m \rightarrow A\text{list}_m \) creates the tuple corresponding to an \( m \)-element list \( \theta_{mm} (l) = (\theta_{m0} (l), \theta_{m1} (l), \ldots, \theta_{mn} (l), \theta_{m(m+1)} (l), \ldots) \), where \( \theta_{mk} (l) = l \) for \( k \geq m \). It is easy to see that any finite list has a unique representation in \( A\text{list}_m \), and Lemma 11.15 clearly holds. But why can we treat \( A\text{list}_m \) as if it were a domain of lists? And where are the infinite lists? The answers to both these questions lie with \( \Phi : A\text{list}_\infty \rightarrow F(A\text{list}_\infty) \). It is defined as:

\[ \Phi = \bigcup_{m=0}^{\infty} \bigcup_{i=0}^{m} (id_{\text{Nil}} + (id_A \times \theta_{m+1})) \]

\[ = \bigcup_{m=0}^{\infty} \bigcup_{i=0}^{m} \lambda x. \text{cases } x \text{ of } \]

\[ \text{isNil()} \rightarrow \text{inNil()} \]

\[ [] \text{isAxD}_m(a,d) \rightarrow \text{inAxD}_{\text{list}_\infty}(a, \theta_{m+1}(d)) \text{ end} \circ \theta_{m+1} \]

\( \Phi \) reveals the list structure in an \( A\text{list}_\infty \) tuple. A tuple \( x \in A\text{list}_\infty \) represents:

1. The undefined list when \( \Phi(x) = \bot \) (then \( x = ([], \bot, \bot, \ldots) \)).
2. The \( \text{nil} \) list when \( \Phi(x) = \text{inNil}() \) (then \( x = ([], \text{nil}, \text{nil}, \ldots) \)).
3. A list whose head element is \( a \) and tail component is \( d \) when \( \Phi(x) = \text{inAxD}_{\text{list}_\infty}(a, d) \) (then \( x = ([a, \theta_{m0}(d)], [a, \theta_{m1}(d)], \ldots, [a, \theta_{m(i-1)}(d)], \ldots) \)). \( \Phi(d) \) 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 \( k \)th element of the infinite list that \( x \) represents is embedded in those \( x_j \) such that \( j \geq k \). \( \Phi \) finds the \( k \)th element: it is \( a_k \), where \( \Phi(x) = \text{inAxD}_{\text{list}_\infty}(a_1, d^2) \), and \( \Phi(d^i) = \text{inAxD}_{\text{list}_\infty}(a_i, d^{i+1}) \), for \( i > 1 \).

The inverse map to \( \Phi \) is \( \Psi : F(A\text{list}_\infty) \rightarrow A\text{list}_\infty \). It embeds a list into \( A\text{list}_\infty \) so that operations like \( \text{cons} : A\times A\text{list}_\infty \rightarrow A\text{list}_\infty \) have well-formed definitions. For \( a \in A \) and \( x \in A\text{list}_\infty \), \( \Psi(a, d) = ([a, \theta_{m0}(x)], [a, \theta_{m1}(x)], \ldots, [a, \theta_{m(i-1)}(x)], \ldots) \). The isomorphism properties of \( \Phi \) and \( \Psi \) 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 $\text{Proc} = \text{Proc} \to A_1$. The family of pointed cpos that results begins with:

$$D_0 = \{ \bot \}$$

$$D_1 = D_0 \to A_1$$

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 $a \in A_1$.

$$D_2 = D_1 \to A_1$$

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

$$D_{i+1} = D_i \to A_1$$

In general, a $D_{i+1}$-level procedure accepts $D_i$-level arguments. (Note that $D_{i-1}$, $D_{i-2}$, \ldots 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 $(p_0, p_1, \ldots, p_i, \ldots) \in \text{Proc}_\infty$. 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 $p_{k+1}(q_k)$. The tuple is intended to operate upon arguments in $\text{Proc}_\infty$, and these elements no longer have “levels.” The solution is simple: take the argument $q \in \text{Proc}_\infty$ and map it down to level $D_0$ (that is, $\theta_0(q)$) and apply $p_1$ to it; map it down to level $D_1$ (that is, $\theta_1(q)$) and apply $p_{i+1}$ to it; \ldots; and lub the results! This is precisely what $\Phi : \text{Proc}_\infty \to F(\text{Proc}_\infty)$ does:

$$\Phi = \bigcup_{m=0}^{\infty} (\theta_m \to id_{A_1}) \circ \theta_{m+1}$$

$$= \bigcup_{m=0}^{\infty} (\lambda x. id_{A_1} \circ x \circ \theta_m) \circ \theta_{m+1}$$

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

$$\Phi(p) = \bigcup_{m=0}^{\infty} (\lambda x. id_{A_1} \circ x \circ \theta_m)(\theta_{m+1}(p))$$

$$= \bigcup_{m=0}^{\infty} (\theta_m(p)) \circ \theta_m$$

$$= \bigcup_{m=0}^{\infty} (p \downarrow (m+1)) \circ \theta_m$$

$$= \bigcup_{m=0}^{\infty} p_{m+1} \circ \theta_m$$

Thus:
268  Domain Theory III: Recursive Domain Specifications

\[
(\Phi(p))(q) = \left( \bigcup_{m=0}^{\infty} p_{m+1} \circ \Theta_m \right)(q)
\]

\[
= \bigcup_{m=0}^{\infty} p_{m+1}(\Theta_m(q))
\]

\[
= \bigcup_{m=0}^{\infty} p_{m+1}(q_m)
\]

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:

\textit{Record} = \textit{Id} \rightarrow \textit{Denotable-value}

\textit{Denotable-value} = (\textit{Record} + \textit{Nat} + \cdots)_1

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:

\[
R_0 = \text{Unit}
\]

\[
D_0 = \text{Unit}
\]

\[
R_{i+1} = \text{Id} \rightarrow D_i
\]

\[
D_{i+1} = (R_i + \text{Nat} + \cdots)_1, \text{ for } i \geq 0
\]

and

\[
R\Phi_0 : R_0 \rightarrow R_1 = (\lambda x. \downarrow R_i)
\]

\[
R\Psi_0 : R_1 \rightarrow R_0 = (\lambda x. \downarrow R_i)
\]

\[
D\Phi_0 : D_0 \rightarrow D_1 = (\lambda x. \downarrow D_i)
\]

\[
D\Psi_0 : D_1 \rightarrow D_0 = (\lambda x. \downarrow D_i)
\]

\[
R\Phi_i : R_i \rightarrow R_{i+1} = (\lambda x. D\Phi_{i-1} \circ x \circ id_{\text{Id}})
\]

\[
R\Psi_i : R_{i+1} \rightarrow R_i = (\lambda x. D\Psi_{i-1} \circ x \circ id_{\text{Id}})
\]

\[
D\Phi_i : D_i \rightarrow D_{i+1} = (R\Phi_{i-1} + id_{\text{Nat}} + \cdots)_1
\]

\[
D\Psi_i : D_{i+1} \rightarrow D_i = (R\Psi_{i-1} + id_{\text{Nat}} + \cdots)_1, \text{ for } i > 0
\]

The inverse limits are \(\text{Record}_\infty\) and \(\text{Denotable-value}_\infty\). Two pairs of isomorphism maps result: \((R\Phi, R\Psi)\) and \((D\Phi, D\Psi)\). Elements of \(\text{Record}_\infty\) represent record structures that map identifiers to values in \(\text{Denotable-value}_\infty\). \(\text{Denotable-value}_\infty\) contains \(\text{Record}_\infty\) as a component, hence any \(r \in \text{Record}_\infty\) exists as the denotable value \(\text{inRecord}_\infty(r)\). Actually, the previous sentence is a bit imprecise— \((\text{Record}_\infty + \text{Nat} + \cdots)_1\) contains \(\text{Record}_\infty\) as a component, and an
\( r \in \text{Record}_\infty \) is embedded in \( \text{Denotable-value}_\infty \) by writing \( D\Psi(\text{inRecord}_\infty(r)) \). Like all the other inverse limit domains, \( \text{Denotable-value}_\infty \) and \( \text{Record}_\infty \) 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:

\[
\lambda i. (\text{zero} \rightarrow \text{Nat}) \rightarrow (\text{inRecord}_\infty(r) \rightarrow \text{inRecord}_\infty(r))
\]

Record \( r \) contains an infinite number of copies of itself. Any indexing sequence \( (r \|B\|B\ldots |B|) \) produces \( r \) again. Since \( \text{Record}_\infty \) is a pointed cpo, the recursive definition of \( r \) has a least fixed point solution, which is a tuple in \( \text{Record}_\infty \). You should consider how the least fixed point solution is calculated in \( \text{Record}_\infty \) 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


**EXERCISES**

1. Construct the approximating domains \( D_0, D_1, D_2, \ldots, D_{i+1} \) for each of the following:
   a. \( N = (\text{Unit} + N)_1 \)
   b. \( \text{Nlist} = \text{N}_1 \times \text{Nlist} \)
   c. \( \text{Mlist} = (\text{N} \times \text{Mlist})_1 \)
   d. \( P = P \rightarrow B_1 \)
   e. \( Q = (Q \rightarrow B)_1 \)

   Describe the structure of \( D_\infty \) for each of the above.

2. Define the domain \( D = D \rightarrow (D + \text{Unit})_1 \). What \( D_\infty \) element is the denotation of each of the following?
   a. \( (\lambda d. \bot) \)
   b. \( (\lambda d. \text{inD}(d)) \)
   c. \( f = (\lambda d. \text{inD}(f)) \)
3. Let $\text{Nlist} = (\text{Nat}^\ast)_\downarrow$ and $\text{Natlist} = (\text{Unit} + (\text{Nat} \times \text{Natlist}))_\downarrow$.

   a. What lists does $\text{Natlist}$ have that $\text{Nlist}$ does not?
   b. Define $\text{cons} : \text{Nat} \times \text{Nlist} \rightarrow \text{Nlist}$ for $\text{Nlist}$. Is your version strict in its second argument? Is it possible to define a version of $\text{cons}$ that is nonstrict in its second argument? Define a $\text{cons}$ operation for $\text{Natlist}$ that is nonstrict in its second argument.
   c. Determine the denotation of $l \in \text{Nlist}$ in $l = \text{zero cons } l$ and of $l \in \text{Natlist}$ in $l = \text{zero cons } l$.
   d. A lazy list is an element of $\text{Natlist}$ that is built with the nonstrict version of $\text{cons}$. Consider the list processing language in Figure 7.5. Make the Atom domain be $\text{Nat}$, 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 $\text{Natlist} = (\text{Unit} + (\text{Nat} \times \text{Natlist}))_\downarrow$ 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 $\text{Natlist}$. Redefine the command continuations $\text{finish} : \text{Cmdcont}$ to be $\text{finish} = (\lambda s. \text{in Unit})$ and $\text{error} : \text{String} \rightarrow \text{Cmdcont}$ to be $\text{error} = (\lambda t. \lambda s. \downarrow)$. Add this command to the language:

   $$\text{C}[[\text{while } 1 \text{ do print } 0]] e c s$$

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

   $$\text{strict}(f)(\downarrow) = \downarrow_B, \quad \text{that is, the least element in } B$$
   $$\text{strict}(f)(a) = f(a), \quad \text{for a proper value } a \in A$$

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

   $$g \ast f = \lambda s. (\lambda l, p). (\lambda l, p). (l \text{ append } l, p)(\text{strict}(g)(p))(fs)$$

   where $\text{append} : \text{Natlist} \times \text{Natlist} \rightarrow \text{Natlist}$ is the list concatenation operation and is nonstrict in its second argument.) Prove that $\text{C}_D[[\text{while } 1 \text{ do print } 0]] e s$ is an infinite list of zeros.
   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 \( \{ \bot \} \)? Say that a retraction sequence had \( D_0 = \text{Unit} \). Does an inverse limit still result? State the conditions under which \( B_1 \) 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. \( D = D \to D \)
   ii. \( D = D_1 \to D_1 \)
   iii. \( D = (D \to D) \)
   b. Recall the lambda calculus system that was introduced in exercise 11 of Chapter 3. Once again, its syntax is:
      \[
      E ::= (E_1 \ E_2) \ | \ (\lambda I. E) \ | \ I
      \]
      Say that the meaning of a lambda-expression \( (\lambda I. E) \) is a function. Making use of the domain \( \text{Environment} = \text{Identifier} \to D \) with the usual operations, define a valuation function \( E : \text{Lambda-expression} \to \text{Environment} \to D \) 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 \( \llbracket E \rrbracket \), \( E[\llbracket (E_1 \ E_2) \rrbracket] = E[\llbracket (E_2 I)E_1 \rrbracket] \)) implies \( E[\llbracket E_1 \rrbracket] = E[\llbracket E_2 \rrbracket] \)” hold? (Warning: this is a nontrivial problem.)
   c. For each of the three versions of \( E \) that you defined in part b, prove that the \( \beta \)-rule is sound, that is, prove:
      \[
      E[\llbracket (\lambda I. E_1) E_2 \rrbracket] = E[\llbracket (E_2 I)E_1 \rrbracket]
      \]
   d. Augment the syntax of the lambda-calculus with the abstraction form \( (\lambda \text{val} I. E) \). Add the following reduction rule:
      \[
      \beta_{\text{val}}\text{-rule}: \ (\lambda \text{val} I. E_1) E_2 \triangleright [E_2 I]E_1
      \]
      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 \( \beta_{\text{val}} \)-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 \( D_0 = \text{Unit} \) and that \( D_\infty \) 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:

\[ C = (\text{Skip} + \text{Assign} + \text{Comp} + \text{Cond}) \]

where \( \text{Skip} = \text{Unit} \) represents the \text{skip} command

\( \text{Assign} \) is a set of primitive assignment commands

\( \text{Comp} = C \times C \) represents command composition

\( \text{Cond} = \text{Bool} \times C \times C \) represents conditional

\( \text{Bool} \) 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 \( \text{wh}(b, c) \) be the \( C \)-value \( \text{wh}(b, c) = \text{in}\text{Cond}(b, \text{in}\text{Comp}(c, \text{wh}(b, c)), \text{in}\text{Skip}()) \), for \( c \in C \) and \( b \in \text{Bool} \). Using the methods outlined in Section 6.6.5, draw a tree-like picture of the denotation of \( \text{wh}(b_0, c_0) \). Next, write the tuple representation of the value as it appears in \( C_\infty \).
c. Define a function \( \text{sem} : C \to \text{Store} \to \text{Store} \) that maps a member of \( C \) to a store transformation function. Define a congruence between the domains \( \text{Bool} \) and \( \text{Boolean-expr} \) and between \( \text{Assign} \) and the collection of trees of the form \( \text{[I:=E]} \).
Prove for all \( b \in \text{Bool} \) and its corresponding \( \text{[B]} \) and for \( c \in C \) and its corresponding \( \text{[C]} \) that \( \text{sem}(\text{wh}(b, c)) = \text{C[while B do C]} \).

10. The domain \( I = (\text{Dec} \times I)_1 \), where \( \text{Dec} = \{0, 1, \cdots, 9\} \), 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?
11. Just as the inverse limit $D_\infty$ is determined by its approximating domains $D_i$, a function $g : D_\infty \rightarrow C$ is determined by a family of approximating functions. Let $G = \{ g_i : D_i \rightarrow C \mid i \geq 0 \}$ be a family of functions such that, for all $i \geq 0$, $g_{i+1} \circ \theta_i = g_i$. (That is, the maps always agree on elements in common.)

a. Prove that for all $i \geq 0$, $g_i \circ \psi_{\mathbb{I}, i+1}$.

b. Prove that there exists a unique $g : D_\infty \rightarrow C$ such that for all $i \geq 0$, $g \circ \theta_i = g_i$. Call $g$ the mediating morphism for $G$, and write $g = \text{med} G$.

c. For a continuous function $h : D_\infty \rightarrow C$, define the family of functions $H = \{ h_i : D_i \rightarrow C \mid i \geq 0, h_i = h \circ \theta_i \}$.

i. Show that $h_{i+1} \circ \theta_i = h_i$ for each $i \geq 0$.

ii. Prove that $h = \text{med} H$ and that $H = \{ (\text{med} H) \circ \theta_i \mid i \geq 0 \}$.

Thus, the approximating function families are in 1-1, onto correspondence with the continuous functions in $D_\infty \rightarrow C$.

d. Define the approximating function family for the map $hd : \text{Alist} \rightarrow \text{Alist}$, $\text{Alist} = (\text{Unit} + (A \times \text{Alist}))_1$, $hd = \lambda d. \text{cases} \Phi(d) \text{of isUnit}(d) \rightarrow_1 \text{isA} \times \text{Alist}(a, l) \rightarrow \text{a}$ end. Describe the graph of each $hd_i$.

e. Let $p_i : D_i \rightarrow \mathbb{B}$ be a family of continuous predicates. Prove that for all $i \geq 0$, $p_i$ holds for $d_i \in D_i$, (that is, $p_i(d_i) = \text{true}$) iff $\text{med} \{ p_i \mid i \geq 0 \}(d) = \text{true}$, where $d = (d_0, d_1, \cdots, d_i, \cdots) : D_\infty$. Conversely, let $P : D_\infty \rightarrow \mathbb{B}$ be a continuous predicate. Prove that $P$ holds for a $d \in D_\infty$ (that is, $P(d) = \text{true}$) iff for all $i \geq 0$, $P \circ \theta_i(d) = \text{true}$.

f. Results similar to those in parts a through c hold for function families $\{ f_i : C \rightarrow D_i \mid i \geq 0 \}$ such that for all $i \geq 0$, $f_i = \psi_i \circ f_{i+1}$. Prove that there exists a unique $f : C \rightarrow D_\infty$ such that for all $i \geq 0$, $f_i = \theta_i \circ f$. 