Palo Alto Research Center

The Semantics of Lazy (And Industrious) Evaluation

Robert Cartwright
James Donahue

XEROX

The Semantics of Lazy (And Industrious) Evaluation

Robert Cartwright
Mathematical Sciences Department
Rice University
Houston, Texas 77251

James Donahue
Xerox Corporation
Palo Alto Research Center
Palo Alto, California 94304

CSL-83-9 April 1984 [P83-00010]

Abstract: Lazy evaluation has gained widespread acceptance among language theoreticians—particularly among the advocates of "functional programming." The implementation of lazy evaluation is easy to describe, but its semantic consequences are deceptively complex. This paper develops a comprehensive semantic theory of lazy evaluation as a change in the value space over which computation is performed. It also explores several approaches to formalizing the theory of lazy evaluation within a programming logic.

A version of this paper appeared in the Conference Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, pp. 253-264.

CR Categories and Subject Descriptors: F (Theory of Computation), F.1.2 (Modes of Computation), F.3.1 (Specifying and Verifying and Reasoning about Programs), F.4.1 (Mathematical logic), D.1.1 (Applicative Programming).

Additional Keywords and Phrases: lazy evaluation, formal semantics, programming logics, algebraic specification.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

1. Introduction

Since the publication of two influential papers on lazy evaluation in 1976 [Hend76, Frie76], the idea has gained widespread acceptance among language theoreticians—particularly among the advocates of "functional programming" [Hend80, Back78]. There are two basic reasons for the popularity of lazy evaluation. First, by making some of the data constructors in a functional language non-strict, it supports programs that manipulate "infinite objects" such as recursively enumerable sequences; this may make some applications easier to program. Second, by delaying evaluation of arguments until they are actually needed, it may speed up computations involving ordinary finite objects.

Despite the popularity of lazy evaluation, its semantics are deceptively complex. Although the implementation of lazy evaluation is easy to describe, its semantic consequences are not. In lazy domains, the existence of infinite objects nullifies the usual principle of structural induction for program data. Replacing conventional data constructors by their lazy counterparts profoundly changes the structure of the data domain. As a result, reasoning about programs defined over lazy spaces is a subtle, counterintuitive endeavor. Many simple theorems about ordinary data objects do not hold in the context of lazy evaluation. For example, although the function $reverse \circ reverse$ is the identity function on ordinary linear lists, it does not equal the identity function in the context of lazy evaluation; applying reverse to an infinite list yields the undefined object $\perp$. In response to these issues, this paper develops a comprehensive semantic theory of lazy evaluation and explores several approaches to formalizing that theory within a programming logic. The paper includes four new interesting results.

First, there are several semantically distinct definitions of lazy evaluation that plausibly capture the intuitive notion. In contrast to usual implementation-oriented approaches in the literature, we define lazy evaluation as a change in the value space over which computation is performed. We use a small collection of domain constructors from denotational semantics [Scot76, Scot81, Scot83] to build abstract value spaces that correspond to the meanings of computations using various lazy constructors. Our abstract approach to defining lazy domains accommodates several distinct interpretations of the informal concept of lazy lists developed in the literature [Frie76, Hend76]. Apparently trivial programs produce radically different results under the different interpretations.

Second, non-trivial lazy spaces are similar in structure (under the approximation ordering) to universal domains (as defined by Scott [Scot76]) such as the $\mathbf{P}\omega$ model for the untyped lambda calculus. Specifically, we show that $\mathbf{P}\omega$ (with the standard primitive operations $\mathbf{0}$, succ, pred, cond, K, S, and apply) is isomorphic to the simple lazy space $\mathbf{Trivseq} = \mathbf{Triv} \times \mathbf{Trivseq}$ (with corresponding primitive operations) where **Triv** is the trivial data domain consisting of two objects $\{\perp, \mathbf{true}\}$ and $\times$ denotes the standard cartesian product of two sets. The corresponding primitive operations on **Trivseq** are recursively definable (using first order recursion equations) in terms of the constants **true** and $\perp$, the constructor and selector functions for forming and tearing apart objects in **Trivseq**, and the logical operations **and** and **por** (parallel or) on **Triv**. Hence, lazy trivial sequences (as defined above) provide an elegant model of the (untyped) lambda calculus that is intuitively familiar to most computer scientists.

Third, we prove that neither initial algebra specifications [ADJ76,77] nor final algebra specifications [Kami80] have the power to define lazy spaces. This result, which is surprisingly easy to prove, establishes a fundamental limitation on the power of equational theories as data type specifications.

Fourth, although lazy spaces have the same "higher-order" structure as **P**ω, they nevertheless have an elegant, natural characterization within first order logic. In this paper, we develop a simple, yet comprehensive, first order theory of lazy spaces relying on three axiom schemes asserting:

- the principle of structural induction for finite objects;
- the existence of least upper bounds for directed sets; and
- the continuity of functions.

To demonstrate the deductive power of the system, we show that there is a simple, natural translation of the higher-order logic LCF [Gord77] into our first order system. In addition, we derive a generalized induction rule (analogous to fixed point induction in LCF) for admissible predicates called *lazy induction* that extends conventional structural induction to lazy spaces, greatly simplifying the proof of many theorems. An instance of this generalized rule reduces to ordinary fixed point induction.

The remainder of the paper is divided into eight sections. Section 2 provides a brief overview of Scott's theory of data domains [Scot76, Scot81, Scot83]. Section 3 develops the specific machinery required to define the abstract semantics of lazy data domains. Using this machinery, Section 4 presents a taxonomy of lazy lists, demonstrating that there are many semantically distinct data domains that capture the intuitive notion of lazy evaluation. Section 5 explores various approaches to formalizing our semantics definition of lazy domains within a logical theory. In Section 6, we prove that algebraic specification is too weak to accomplish the task and that lazy spaces have the same rich "higher-order" structure as **P**ω. In Section 7, we present a simple first order theory for lazy data domains and demonstrate that it is at least as powerful as the corresponding theory formulated in the higher-order logic LCF. Section 8 gives some sample program proofs using the first order theory developed in the preceding section. Finally, Section 9 assesses the intuitive significance of our results and speculates about promising directions for future research.

2. Background

2.1 Mathematical Foundations

The following group of definitions rigorously describes our concept of data domain, which is an adaptation and distillation of several different expositions by Scott [Scot 76, 81, 83].

Definition: A *partial order* S is a pair $\langle|S|, \sqsubseteq\rangle$ consisting of a set $|S|$ of objects and a binary relation $\sqsubseteq$ over $|S|$ such that:

(i) $\sqsubseteq$ is *reflexive*: $\forall x \in |S|\ x \sqsubseteq x$.

(ii) $\sqsubseteq$ is *antisymmetric*: $\forall\ x, y \in |S|\ x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$.

(iii) $\sqsubseteq$ is *transitive*: $\forall x, y, z \in |S|\ x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$.

A subset $R \subseteq |S|$ *is consistent* iff there exists $u \in |S|$ such that $\forall r \in R\ r \sqsubseteq u$; u is called an *upper bound* of R. A subset $R \subseteq |S|$ is *directed* iff for every finite subset $E \subseteq R$ has an upper bound in R.

Notation: Given a partial order S, we will use the symbol S as an abbreviation for the more cumbersome notation $|S|$ whenever no confusion is possible. Hence $x \in S$ and $R \subseteq S$ abbreviate $x \in |S|$ and $R \subseteq |S|$, respectively.

Definition: A (*data*) *space* S is a partial order with the following two properties:

(i) Every directed subset $R \in S$ (including the empty set) has a least upper bound in S (denoted $\mathbf{lub}_S\ R$). The least upper bound of the empty set is denoted by the special symbol $\perp_S$ (pronounced "bottom").

(ii) S has a countable subset $B = \{b_i \in S \mid i \in N\}$ called the *basis elements* of S (the elements in the enumeration $b_1, b_2, \ldots$ are not necessarily distinct; hence, B can be finite), such that:

(a) B is closed under the least upper bound operation on finite consistent subsets.

(b) Every element $x \in S$ is the least upper bound of the subset of B that approximates it, i.e.,

$$\forall\ x \in S\ \ x = \mathbf{lub}_S\ \{y \in B \mid y \sqsubseteq x\}.$$

(c) Every basis element $x \in B$ is *finite*: for every directed subset C of B, $x \sqsubseteq \mathbf{lub}_S\ C$ implies that $\exists y \in S$ such that $x \sqsubseteq y$.

Theorem: A data space S has a unique basis; it consists of the finite elements of S.

Proof: By property (c) above, every basis element must be finite. To show that every finite element must be a basis element, let e be an arbitrary finite element. Let E be the set $\{b \sqsubseteq e \mid b \in B\}$. Since e is finite, there exists a finite subset $E' \subseteq E$ such that $\mathbf{lub}_S\ E' = e$. But $\mathbf{lub}_S\ E'$ must be a basis element, because the basis is closed under least upper bounds on finite sets. □

Notation: When no confusion is possible, we will frequently omit the subscripts (identifying a space) on the symbols **lub** and $\perp$.

Definition: An element s of a data space S is *finitely-founded* iff the set $\{y \in S \mid y \sqsubseteq s\}$ is finite. A data space S is *finitely-founded* iff the finitely-founded elements of S form a basis for S. A data space S is *flat* (*industrious*) iff for every element $y \in S$, $x \sqsubseteq y \Leftrightarrow (x = y \vee x = \bot)$. A finitely-founded data space S is *lazy* iff it is not flat.

Although all lazy spaces are finitely-founded, many "higher-order" spaces (such as mappings from one lazy space to another) are not finitely-founded.

Definition: An *ideal* over B is a set F such that:

(i) $\forall x, y \in \mathrm{F}\ \mathbf{lub}\{x, y\} \in \mathrm{F}$, and

(ii) $\forall x \in \mathrm{F}, y \in \mathrm{B}\ y \sqsubseteq x \Rightarrow y \in \mathrm{F}$.

Definition: Two spaces S_1, S_2 with bases B_1, B_2 are *isomorphic* iff there exists a bijective (one-to-one and onto) function $\mathbf{h}:S_1 \rightarrow S_2$ such that:

(i) $\mathbf{h}(B_1) = B_2$ and $\mathbf{h}(\bot_1) = \bot_2$.

(ii) For any finite set $S \subseteq S_1$, S is consistent iff $\mathbf{h}(S)$ is consistent.

(iii) For any consistent set $S \leq S_1$, $\mathbf{h}(\mathbf{lub}\ S) = \mathbf{lub}\ \mathbf{h}(S)$.

Theorem: A data space S with basis B is isomorphic to the space $I(B)$ consisting of the set of ideals over B under the partial ordering defined by the subset relation on ideals (which are simply sets of basis elements).

Proof: The function $\mathbf{h}:I(B) \rightarrow S$ defined by $\mathbf{h}(F) = \mathbf{lub}_S\ F$ maps $I(B)$ *onto* S and clearly preserves the approximation ordering on $I(B)$:

$$F_1 \sqsubseteq F_2 \Leftrightarrow \mathbf{lub}_S\ F_1 \sqsubseteq \mathbf{lub}_S\ F_2$$

Similarly the function $\mathbf{h}':S \rightarrow I(B)$ defined by:

$$\mathbf{h}'(x) = \{y \in B \mid y \sqsubseteq_S x\}$$

maps S *into* $I(B)$ and preserves the approximation ordering on S. Moreover, it is obvious (from the definition of a basis) that for all $x \in S$ $h(h'(x)) = x$.

To complete the proof, we must show that h' maps *S onto* $I(B)$, i.e., that for each $x \in S$, there is a unique ideal F_x in $I(B)$ such that $\mathbf{lub}_S\ F_x = x$. Assume that two distinct ideals F and G have the same least upper bound in S. Without loss of generality, we can assume that F-G is non-empty. Let $w \in F\text{-}G$. Since w is a basis element, it is finite, implying that G (a directed set approximating $w \sqsubseteq x$) contains an element v such that $w \sqsubseteq v$. Since G is an ideal, G must contain w, which is an obvious contradiction.

Remarks: The preceding theorem shows that the structure of space S is completely determined by the structure of B. In the *neighborhood system* formulation of domain theory [Scot 81], the elements of a space are *filters* rather than ideals because each element of the universe is identified with a filter of sets (called *neighborhoods*) that "contain" ($\supseteq$) rather than "approximate" ($\sqsubseteq$) the element.

Definition: Let S be an arbitrary data space with basis B. A function $\mathbf{f}:S^{\#f} \rightarrow S$ is *approximable* iff:

$$\forall\ x_1, ..., x_{\#f} \in S\ \mathbf{f}(x_1, ..., x_{\#f}) = \mathbf{lub}\ \{\mathbf{f}(b_1, ..., b_{\#f}) \mid b_1, ..., b_{\#f} \in B\ \wedge_{1 \leq i \leq \#f}\ b_i \sqsubseteq x_i\ \}.$$

An approximable function $\mathbf{f}:S^{\#f} \rightarrow S$ is *strict* iff the image of every argument list containing $\perp$ is $\perp$, i.e.,

$$\forall\ x_1, ..., x_{\#f} \in S\ x_1 = \perp \vee \ldots \vee x_{\#f} = \perp \Rightarrow \mathbf{f}(x_1, ..., x_{\#f}) = \perp.$$

Definition: A space R is a *subspace* of the space S iff:

(i) $|R| \subseteq |S|$, $\sqsubseteq_R \subseteq \sqsubseteq_S$, and $\perp_R = \perp_S$.

(ii) $|R| \cap B$ forms a basis for R.

(iii) For all directed subsets $R' \subseteq R$, $\mathbf{lub}_R\ R' = \mathbf{lub}_S\ R'$.

Remark: Some formulations of domain theory use a weaker definition of subspace. In particular, they omit condition (ii) and replace condition (iii) by a stipulation that the consistency relation in the subspace R agree with consistency relation in the parent space S. In Section 2.5, we discuss some of the implications of this alternative.

Definition: Let G be a countable set of symbols. A *domain* **D** *with signature* G is a pair <D, **G**> consisting of a space D (called the *universe*) and an *interpretation function* **G** mapping each symbol $g \in G$ into an approximable function **g** (called an *operation*) over D.

Definition: Two domains $\mathbf{D}_1$, $\mathbf{D}_2$ with signature G are *isomorphic* iff the spaces D_1 and D_2 are isomorphic under a function $\mathbf{h}:D_1 \rightarrow D_2$ and for each operation symbol $g \in G$,

$$\forall x_1, ..., x_{\#g} \in D_1\ \mathbf{h}(\mathbf{g}_1(x_1, ..., x_{\#g})) = \mathbf{g}_2(x_1, ..., x_{\#g})).$$

where $\mathbf{g}_1$ and $\mathbf{g}_2$ denote the interpretations of g in $\mathbf{D}_1$ and $\mathbf{D}_2$, respectively.

Definition: A domain **E** with signature H is a *subdomain* of the domain **D** with signature G iff:

(i) E is a subspace of D.

(ii) $H \subseteq G$ and for each operation symbol $h \in H$, $\mathbf{G}_D(h)$ (the interpretation of h in **D**) restricted to E is $\mathbf{G}_E(h)$ (the interpretation of h in **E**).

The obvious difference between a space and a domain is that a domain identifies a collection of primitive operations—in addition to a universe of values—that form a set of building blocks for defining new functions over the universe. In contrast, a space leaves the possible operations on data unspecified.

Notation: Given a domain **D** with signature G, we will frequently write **G** instead of **G**(G) to denote the set of functions over **D** interpreting the operation symbols G.

2.2 *Sample Spaces*

Many common data spaces such as the natural numbers and ordinary (industrious) lists are degenerate in the sense that they contain no limit points; in these spaces, every element is a basis element. For example, let Nat be the natural numbers **N** under the partial ordering $\sqsubseteq_{\text{Nat}}$ defined by:

$$x \sqsubseteq_{\text{Nat}} y \;\Leftrightarrow\; x = y \;\vee\; x = \bot.$$

Nat is a space with basis Nat. Similarly, let Bool, the space of Boolean truth values, be defined as the set:

$$\{\bot, \mathbf{true}, \mathbf{false}\}$$

under the partial ordering $\sqsubseteq_{\text{Bool}}$ defined by:

$$x \sqsubseteq_{\text{Bool}} y \;\Leftrightarrow\; x = y \vee x = \bot.$$

An example of a more interesting space is $\mathbf{P}\omega$, the power set of the natural numbers under the partial ordering $\sqsubseteq$ determined by set inclusion. The finite (basis) elements of $\mathbf{P}\omega$ are precisely the finite sets of natural numbers.

2.3 *Space Constructions*

In specifying data spaces, it is often convenient to construct composite spaces from simpler ones. There are two fundamental mechanisms for constructing composite spaces: the Cartesian product construction and the approximable function construction. We will discuss several other constructions later in the paper, but they are all based on these two mechanisms.

We will define the two constructions without proving that the constructed spaces are well-formed. The interested reader is encouraged to verify that the constructions actually build legitimate data spaces.

Definition: Given data spaces S_1, S_2 with bases B_1, B_2 and approximation orderings $\sqsubseteq_1$, $\sqsubseteq_2$, the *Cartesian product space* $S_1 \times S_2$ is the data space determined by the basis set:

$$\{(x, y) \mid x \in B_1, y \in B_2\}$$

under the relation $\sqsubseteq$ defined by:

$$(x_1, y_1) \sqsubseteq (x_2, y_2) \Leftrightarrow x_1 \sqsubseteq_1 x_2 \wedge y_1 \sqsubseteq_2 y_2.$$

The bottom element of $S_1 \times S_2$ is $(\bot_1, \bot_2)$ where $\bot_1$ and $\bot_2$ denote the least elements of S_1 and S_2.

Notation: In informal mathematics, no distinction is typically made between a unary function **f** defined on the Cartesian product S×S and the corresponding binary function **f'** over S. Since we will be dealing with spaces S that contain S$\times$S as a subspace, we cannot ignore the difference between the two. Consequently, we will employ the following conventions. First, unless we explicitly state otherwise, the expression R$\times$S always denotes the Cartesian product space formed from R and S. Second, the "exponentiated" expression S^k denotes the domain of a k-ary function over the universe S. To avoid unneccesary confusion, we will confine our attention to unary functions when it is feasible.

The second fundamental space construction is the formation of the space of approximable mappings from one data space into another. An approximable mapping is a data object that denotes a function.

Definition: Assume that we are given data spaces S_1, S_2 with bases B_1, B_2. A binary relation $f \subseteq B_1 \times B_2$ is an *approximable mapping from* S_1 to S_2 iff:

(i) f is *consistent*: for all $x \in B_1$, the set

$\{ y \in B_2 \mid \exists x' \in B_1 [x' \sqsubseteq x \land x' f y] \}$ is consistent.

(ii) f is *directed-closed*: for all $\langle x', y'\rangle \in B_1 \times B_2$ $(\exists \langle x, y\rangle \in f [x \sqsubseteq x' \land y' \sqsubseteq y] \Rightarrow \langle x', y'\rangle \in f)$.

Definition: Given an approximable mapping f from S_1 to S_2, the *function determined by f* is the function $\mathbf{f}: S_1 \rightarrow S_2$ defined by:

$$\mathbf{f}(x) = \mathbf{lub}\{ y \in B_2 \mid \exists x' \in B_1 [x' \sqsubseteq x \land x' f y] \}.$$

Observation: If f is an approximable mapping from S to S, then the function **f** over S determined by f is approximable.

Definition: Given the spaces S_1, S_2 with corresponding bases B_1, B_2 and approximation orderings $\sqsubseteq_1$, $\sqsubseteq_2$, the space of approximable mappings $S_1 \Rightarrow S_2$ is the space determined by the basis:

$$\{f \mid \exists \text{ finite consistent } f' \subseteq B_1 \times B_2 \text{ such that } f \text{ is the directed closure of } f'\}$$

under the partial ordering $\sqsubseteq$ defined by:

$$f_1 \sqsubseteq f_2 \Leftrightarrow \forall x \in B_1\, f_1(x) \sqsubseteq_2 f_2(x).$$

The least element of $S_1 \Rightarrow S_2$ is the relation $\{(b_1, \bot_2) \mid b_1 \in B_1\}$ which is the directed closure of the empty relation; it determines the everywhere "undefined" function Ω defined by $\lambda x.\bot_2$.

Theorem: For any data space S that contains a subspace isomorphic to S$\Rightarrow$S, there is an approximable function **Apply** over S such that for every approximable mapping $f \in S \Rightarrow S$ and corresponding function $\mathbf{f}: S \rightarrow S$ such that:

$$\forall x \in S_1\ \mathbf{Apply}(f, x) = \mathbf{f}(x).$$

Proof: Let **Apply** be defined by the equation $\mathbf{Apply}(f, x) = \text{lub}\ \{b \in B \mid \exists\ (u, b) \in f\ u \sqsubseteq x\}$. The theorem follows immediately from the definition of the function f determined by f. □

Although we have only defined the notion of approximable mappings corresponding to unary functions, there is a standard transformation (usually called *currying*) that converts a multiple argument function $f{:}S^{\#f} \rightarrow S$ to an equivalent unary function f': $S \rightarrow [S \rightarrow \ldots \rightarrow [S \rightarrow S] \ldots]$ defined by the lambda expression:

$$\lambda x_1 . \ldots . \lambda x_{\#f} . f(x_1, \ldots, x_{\#f}).$$

2.4 Computability

In order to formalize the idea of computable mappings (functions) on a data space, we must identify a concrete representation for the elements of the space.

Definition: An *effective presentation* of a data space S is an enumeration $B = \langle b_i \mid i \in N \rangle$ of the basis of S (the elements in the enumeration are not necessarily distinct) such that:

(i) The binary relation **CON** defined by **CON**(i, j) $\Leftrightarrow$ ($\exists k\ b_i \sqsubseteq b_k \ \wedge\ b_j \sqsubseteq b_k$) is recursive.

(ii) The ternary relation **LUB** defined by **LUB**(i, j, k) $\Leftrightarrow$ ($b_k = \textbf{lub}\{b_i, b_j\}$) is recursive.

The enumeration *B* is called an *effective presentation* of S.

Theorem: Given effective presentations B_1, B_2 for the spaces S_1, S_2, we can construct effective presentations for $S_1 \times S_2$ and $S_1 \Rightarrow S_2$.

Proof: Omitted.

A subspace S of an effectively presented space S (with presentation $B = \langle b_i \mid i \in N \rangle$) is *effective* iff the index set for the basis of S $\{i \mid b_i \in S\}$ is recursively enumerable.

Notation: We will use italicized identifiers *A*, *B*, ... to denote effective presentations and the matching Roman identifiers A, B, ... to denote the corresponding sets of basis elements.

In an abstract implementation of an effectively presented space S, each element x of the universe is represented by a natural number x^R encoding the index set $In(x) = \{i_1, i_2, \ldots\}$ of the set of basis elements $\{b_{i1}, b_{i2}, \ldots\}$ approximating x. More precisely, there is a binary total recursive function β such that for all $x \in S$, $\lambda k . \beta(x^R, k)$ has range In(x). In this context, a computable function f over S is implemented by a #f-ary partial recursive function f^R such that for all $x_1, \ldots, x_{\#f} \in S$, the function: $\lambda k . \beta(f^R(x_1^R, \ldots, x_{\#f}^R), k))$ has range $In(f^R(x_1, \ldots, x_{\#f}))$.

Given the preceding motivation, we formalize the notions of computable function and computable mapping as follows:

Definition: An approximable mapping *f* is *computable* iff it is recursively enumerable, using the indices given in the enumerations B_1, B_2 to name elements in B_1 and B_2. The function f determined by an approximable mapping *f* from S_1 into S_2 is *computable* iff *f* is computable.

A computable function **f**: $S_1 \rightarrow S_2$ is "computable" in the sense that given an arbitrary element $x \in S_1$ (represented by the code x^R), we can enumerate the set of basis elements that approximate the image element $\mathbf{f}(x) \in S_2$.

Definition: A data domain $\mathbf{D} = \langle D, G \rangle$ is *computable* iff there exists an effective presentation *B* for D such that every operation $g \in G$ is computable. An element $d \in D$ is *accessible* iff the index set of the ideal of basis elements approximating d is recursively enumerable. An element $d \in D$ is *definable* in D iff there is a variable-free term ρ_d constructed from the operation symbols in G such that denotes d. A function f: $D^n \rightarrow D$ is *recursively definable* in D iff there is a term τ_f composed solely from the free variables $x_1, ..., x_n$ and the operations G such that f is the least function (using the approximation ordering on the corresponding mappings in $D^n \Rightarrow D$) satisfying the equation (called a *recursive program* for f):

(*) $f(x_1, ..., x_n) = \tau_f$.

The domain D is *expressive* iff every accessible element of D is definable in D. The domain D is *computationally complete* iff every computable function f: $D^n \rightarrow D$ ($n \geq 0$) is recursively definable in D. D is *reflexively complete* iff the following three properties hold:

(i) $D \Rightarrow D$ is isomorphic to a subspace Map_D of D.

(ii) Every accessible element of Map_D is definable in D.

(iii) The function **Apply**: $D \rightarrow D$ defined in the previous section is recursively definable in D.

Remarks: By Kleene's recursion theorem, the least function f satisfying the equation (*) must exist since it is simply the least fixed-point of the approximable function F: $[D^n \rightarrow D] \rightarrow [D^n \rightarrow D]$ denoted by the lambda expression:

$\lambda f \,.\, \lambda\, [x_1, ..., x_n] \,.\, \tau_f$.

Observation: If a domain **D** is reflexively complete, then it is computationally complete. A particularly appealing property of Scott's theory of data domains is that the set of approximable mappings between effectively presented spaces is an effectively presentable space in its own right. Moreover, the set of computable mappings within this space are precisely the accessible elements of the space. We will discuss this issue in more detail below. In this paper, we will be exclusively concerned with computable spaces and domains.

2.5 Retractions on the Universal Domain

A fairly rich collection of spaces can be constructed by starting with a few very simple primitive spaces (such as Nat and Bool) and constructing more complex spaces by composing the Cartesian product and approximable mapping space constructions. However, it is easy to devise spaces such as infinite cartesian products of primitive spaces that are beyond the scope of this simple scheme. Scott has developed a much more comprehensive approach to the problem of constructing spaces based on the concept of a *universal space.*

Definition: A *universal space* U is a computable space with effective presentation B such that every data space D is isomorphic to a subspace S of U. Moreover, if D is effectively presented, then S must be an effective subspace of U.

Since every space D has an isomorphic image S within the universal space, the problem of defining an arbitrary space can be reduced to defining an arbitrary subspace of a particular universal space. A simple, elegant way to identify an arbitrary (computable) subspace S of a universal space is to define a (computable) *retraction* characterizing S.

Definition: A *retraction* on U is a strict approximable function $\mathbf{a}$: U $\Rightarrow$ U such that $\mathbf{a} \circ \mathbf{a} = \mathbf{a}$. A retraction a is *finitary* iff the image $\mathbf{a}(\mathrm{U})$ is a subspace of U. A retraction is a *projection* iff it preserves basis elements and least upper bounds. In other words, **a** must satisfy the following two properties:

(i) $\forall \mathrm{b} \in \mathrm{B}\ \mathrm{a(b)} \in \mathrm{B}$.

(ii) $\forall$ consistent $u, v \in B\ \mathrm{a}(\mathrm{lub}\{u, v\}) = \mathrm{lub}\{\mathrm{a}(u), \mathrm{a}(v)\}$.

The range of a (finitary) retraction a is called the (finitary) *retract* of a.

Remark: A projection is clearly a special form of finitary retraction.

Theorem: For every subspace S of a universal space U, there is a projection **a** with retract S.

Proof: The projection **a** is defined by $\mathbf{a}(x) = \{b \in \mathrm{B} \mid b \in \mathrm{S} \wedge b \sqsubseteq x\}$. It is easy to verify that $\mathbf{a}(\mathrm{U}) = \mathrm{S}$. □

Remark: The reader should be aware that we are using a very strong definition of subspace, which imposes severe restrictions on the structure of a universal space (e.g., it cannot be finitely-founded). In fact, by our definition of subspace, the well known "universal" space T^{ω} is not universal. If we weaken the definition of subspace as discussed in Section 2.1, then T^{ω} is universal and the preceding theorem no longer holds. In this case, the basis elements of a subspace $\mathrm{S} \subseteq \mathrm{U}$ may be infinite in U (even though they must be finite in S). Moreover, there is no suitable notion of a canonical retraction (analogous to a projection) characterizing an arbitrary subspace. For this reason, we prefer the strong definition of subspace.

Definition: A *universal domain* **U** is a reflexively complete domain <U, **G**> such that the universe U is a universal space.

Remark: Given a universal space U, we can construct a universal domain by identifying a finite set of functions **G** over U such that:

(i) the **Apply** operation is recursively definable in <U, **G**>, and

(ii) every recursively enumerable element of U is denoted by some variable-free term formed from G.

Moreover, since U⇒U is isomorphic to a subspace of U and U is reflexively complete, there is a term ρ_g (composed from G) for each operation **g** that is recursively definable in <U, G>, such that:

$$\forall\ x_1, \ldots, x_{\#g} \in U\ \ \mathbf{Apply}(\ (\ \ldots\ \mathbf{Apply}(\rho_g, x_1), \ldots), x_{\#g}) = g(x_1, \ldots, x_{\#g}).$$

Notation: To simplify the syntax of expressions over a universal domain **U**, we will adopt the following conventions. First, since there is an element ρ_f within U corresponding to every recursively definable operation **f**, we will use the mapping ρ_f in place of each operation **f** other than constants and the special operation **Apply**. Hence, instead of the expression **f**(x, y) we will write **Apply(Apply(**ρ_f, x), y). Second, we will abbreviate every application of the form **Apply**(u, v) by (u v). Third, we will elide parentheses by making application left associative; hence $u\ v\ w$ abbreviates (($u\ v$) w). Finally, we will abbreviate applications of the form $f\,(g\ x)$ by $f{\circ}g\ x$. This notation is consistent with the conventions usually employed in the untyped lambda calculus [Bare77].

Although there are many different possible formulations of the universal domain, the particular choice is unimportant. Given an arbitrary universal domain **U** with basis **B**, we can recursively define (in terms of the primitive operations **G** on the universal domain) the basic set of operations $\mathbf{O}_{\mathbf{lazy}}$ that we need to construct lazy spaces. $\mathbf{O}_{\mathrm{lazy}}$ consists of the projection mappings R_{Bool}, $R_{\times}$, and $R_{\Rightarrow}$ identifying the subspaces Bool ({**true**, **false**, ⊥}), U×U, and U⇒U, and the mappings:

true, **false**: Bool
δ: U ⇒ Bool
if-then-else: Bool ⇒ (U ⇒ (U ⇒ U))
and: Bool ⇒ (Bool ⇒ Bool)
or: Bool ⇒ (Bool ⇒ Bool)
por: Bool ⇒ (Bool ⇒ Bool)
not: Bool ⇒ Bool
pair: U ⇒ (U ⇒ U × U))
left: U × U ⇒ U
right: U × U ⇒ U
S: (U⇒U) ⇒ ((U⇒U ⇒ U⇒U))
K: U ⇒ (U⇒U)

satisfying the axioms:

$\delta \perp = \perp$
$x \neq \perp \;\Rightarrow\; (\delta\ x) = \mathbf{true}$
$\textit{if-then-else}\ \mathbf{true}\ \ x\ \ y\ = x$
$\textit{if-then-else}\ \mathbf{false}\ \ x\ \ y\ = y$
$\textit{if-then-else}\ \perp\ x\ y\ = \perp$
$\textit{and}\ x\ y\ = \textit{if-then-else}\ x\ y\ \mathbf{false}$
$\textit{or}\ x\ y\ = \textit{if-then-else}\ x\ \mathbf{true}\ y$
$x \neq \mathbf{true} \wedge y \neq \mathbf{true} \;\Rightarrow\; (\mathrm{por}\ x\ y) = (\mathrm{or}\ x\ y)$
$x = \mathbf{true} \vee y = \mathbf{true} \;\Rightarrow\; (\mathrm{por}\ x\ y) = \mathbf{true}$
$\textit{not}\ x = \textit{if-then-else}\ x\ \mathbf{false}\ \mathbf{true}$
$R_{\times}\ x = x \;\Rightarrow\; \textit{pair}\ (\textit{left}\ x)\ (\textit{right}\ x) = x$
$\textit{left}\ (\textit{pair}\ x\ y) = x$
$\textit{right}\ (\textit{pair}\ x\ y) = y$
$R_{\Rightarrow}\ f = \lambda x\,.\ \mathbf{lub}\{y \in \mathrm{B} \mid \exists\ \mathrm{u} \in \mathrm{B}\ \mathrm{u} \sqsubseteq x \wedge \mathrm{u}\ \mathrm{f}\ y\}$
$S\ x\ y\ z = \ x\ z\ (y\ z)$
$K\ x\ y = x.$

The notation $f{:}\mathrm{S}_1 \Rightarrow \mathrm{S}_2$ means that f is a mapping in $\mathrm{U} \Rightarrow \mathrm{U}$ such that $\forall x \in \mathrm{S}_1\ f\ x \in \mathrm{S}_2$. The behavior of f on points outside of the space S_1 is not specified.

With the exception of *por*, *S, and K*, these mappings are generalizations of familiar operations from lazy LISP (where *left*, *right*, and *pair* correspond to *car*, *cdr*, and *cons*). The declared domain for each mapping is its intended domain of usage. Each mapping is actually defined over the entire universal space U; space declarations are enforced by projecting argument values outside the declared domain onto the declared domain D (using the projection mapping R_{D}).

Since $\mathbf{O}_{\mathrm{lazy}}$ includes the **Apply** operation and the *S* and *K* mappings, we can form a variable-free term that denotes the mapping corresponding to any function that is recursively definable in terms of the operations $\mathbf{O}_{\mathrm{lazy}}$. It is well known [Bare77] that any closed term (no free variables) in the (untyped) lambda calculus can be expressed as a composition of the operations *S and K*. Moreover, the least fixed point operator $\mathbf{Y}$: $(\mathrm{U} \rightarrow \mathrm{U}) \rightarrow \mathrm{U}$ that maps an approximable function into its least fixed point is defined by the lambda expression:

$$\lambda f\ .\ (\lambda x.\ f\,(x\ x))\ (\lambda x.\ f\,(x\ x)).$$

The corresponding mapping Y is defined by:

$Y = S\ \alpha\ \alpha$
$I = S\ K\ K$
$\alpha = (S\ (S\ (K\ S)\ (S\ (K\ K)\ I))\ (S\ (K\ S)\ (K\ I))\ (K\ I)).$

Consequently, the mapping corresponding to an arbitrary recursive definition:

$$\mathrm{f}(x_1, \ldots, x_{\#\mathrm{f}}) = \tau_{\mathrm{f}}$$

is simply:

$$Y\,(\lambda^* x_1 \,.\, \ldots \,.\, \lambda^* x_{\#\mathrm{f}} \,.\, \tau_{\mathrm{f}})$$

where $\lambda^* \, x \,.\, \alpha$ denotes the term (formed using S and K) signifying the mapping corresponding to the function $\lambda x \,.\, \alpha$.

Notation: As a notational convenience, we will use lambda expressions (without the * exponent) to denote mappings instead of compositions of S and K; they are much easier to read. On a formal level, these lambda expressions simply abbreviate the corresponding compositions of S and K. Similarly, we will elide applications of the Y operator by using the equation:

$$f = \tau[f]$$

to abbreviate the recursive definition:

$$f = Y\,(\lambda f \,.\, \tau[f]).$$

We will also use the standard infix abbreviations for applications of Boolean mappings:

$$\text{if } x \text{ then } y \text{ else } z \;\equiv\; \textit{if-then-else}\; x\; y\; z$$
$$x \text{ and } y \;\equiv\; \textit{and}\; x\; y$$
$$x \text{ or } y \;\equiv\; \textit{or}\; x\; y$$
$$x \text{ por } y \;\equiv\; \textit{por}\; x\; y.$$

3. The Construction of Lazy Spaces

In constructing a composite space (such as a Cartesian product or discriminated union) from component spaces, we must decide how to form the bottom element of the composite space, i.e., determine which constructed objects are identified with the undefined composite object. This decision implicitly determines whether the composite space corresponds to lazy or industrious computation.

Let D_1 and D_2 be arbitrary computable subspaces of our universal space U characterized by the projection mappings R_1 and R_2 in $\mathrm{U} \Rightarrow \mathrm{U}$. Using the Cartesian mapping *pair*: $\mathrm{U} \Rightarrow (\mathrm{U} \Rightarrow \mathrm{U} \times \mathrm{U})$, we can form a surprisingly wide variety of simple composite space using the following space constructions.

3.1 Ordinary product

$$\mathrm{D}_1 \times \mathrm{D}_2 = \{\langle x, y \rangle \mid x \in \mathrm{D}_1, y \in \mathrm{D}_2\}.$$

The corresponding basic mappings are:

$P_\times$: $D_1 \Rightarrow (D_2 \Rightarrow D_1 \times D_2) = \lambda x.\ \lambda y.\ \mathit{pair}\ x\ y$
$\mathit{fst}_\times$: $D_1 \times D_2 \Rightarrow D_1 = \lambda z.\ \mathit{left}\ z$
$\mathit{snd}_\times$: $D_1 \times D_2 \Rightarrow D_2 = \lambda z.\ \mathit{right}\ z$
$R_\times$: $U \Rightarrow D_1 \times D_2 = \lambda x.\ P_\times\ (R_1 \circ \mathit{fst}_\times\ x)\ (R_2 \circ \mathit{snd}_\times\ x)$.

3.2 Coalesced product

$D_1 \otimes D_2 = \{\langle x, y\rangle \mid x \in D_1, y \in D_2, x \neq \bot, y \neq \bot\} \cup \{\bot\}$.

The corresponding basic mappings are:

$P_\otimes$: $D_1 \Rightarrow (D_2 \Rightarrow D_1 \otimes D_2) = \lambda x.\ \lambda y.\ \mathit{if}\ \delta x$ **and** $\delta y\ \mathit{then\ pair}\ x\ y\ \mathit{else}\ \bot$
$\mathit{fst}_\otimes$: $D_1 \otimes D_2 \Rightarrow D_1 = \lambda z.\ \mathit{left}\ z$
$\mathit{snd}_\otimes$: $D_1 \otimes D_2 \Rightarrow D_2 = \lambda z.\ \mathit{right}\ z$
$R_\otimes$: $U \Rightarrow D_1 \otimes D_2 = \lambda x.\ \mathit{if}\ \delta x\ \mathit{then}\ P_\otimes\ (R_1 \circ \mathit{fst}_\otimes\ x)\ (R_2 \circ \mathit{snd}_\otimes\ x)\ \mathit{else}\ \bot$.

3.3 Separated product

$D_1 \boxtimes D_2 = \{\langle \mathbf{true}, \langle x, y\rangle\rangle \mid x \in D_1, y \in D_2\}$.

The corresponding basic mappings are:

$P_\boxtimes$: $D_1 \Rightarrow (D_2 \Rightarrow D_1 \boxtimes D_2) = \lambda x.\ \lambda y.\ \mathit{pair}\ \mathbf{true}\ (\mathit{pair}\ x\ y)$
$\mathit{fst}_\boxtimes$: $D_1 \boxtimes D_2 \Rightarrow D_1 = \lambda z.\ \mathit{left} \circ \mathit{right}\ z$
$\mathit{snd}_\boxtimes$: $D_1 \boxtimes D_2 \Rightarrow D_2 = \lambda z.\ \mathit{right} \circ \mathit{right}\ z$
$R_\boxtimes$: $U \Rightarrow D_1 \boxtimes D_2 = \lambda x.\ P_\boxtimes\ (R_1 \circ \mathit{fst}_\boxtimes\ x)\ (R_2 \circ \mathit{snd}_\boxtimes\ x)$.

3.4 Coalesced sum

$D_1 \oplus D_2 = \{\langle \mathbf{true}, x\rangle \mid x \in D_1, x \neq \bot\} \cup \{\langle \mathbf{false}, y\rangle \mid y \in D_2, y \neq \bot\} \cup \{\bot\}$.

The corresponding basic mappings are:

$\mathit{inL}_\oplus$: $D_1 \Rightarrow D_1 \oplus D_2 = \lambda x.$ if δx then $\mathit{pair}\ \mathbf{true}\ x\ \mathit{else}\ \bot$
$\mathit{inR}_\oplus$: $D_2 \Rightarrow D_1 \oplus D_2 = \lambda x.$ if δx then $\mathit{pair}\ \mathbf{false}\ x\ \mathit{else}\ \bot$
$\mathit{outL}_\oplus$: $D_1 \oplus D_2 \Rightarrow D_1 = \lambda z.\ \mathit{right}\ z$
$\mathit{outR}_\oplus$: $D_1 \oplus D_2 \Rightarrow D_2 = \lambda z.\ \mathit{right}\ z$
$\mathit{isL}_\oplus$: $D_1 \oplus D_2 \Rightarrow \mathrm{Bool} = \lambda z.\ \mathit{left}\ z$
$\mathit{isR}_\oplus$: $D_1 \oplus D_2 \Rightarrow \mathrm{Bool} = \lambda z.\ \mathit{not} \circ \mathit{left}\ z$
$R_\oplus$: $U \Rightarrow D_1 \oplus D_2 = \lambda x.$ if $\mathit{isL}_\oplus\ x$ then $\mathit{inL}_\oplus \circ R_1 \circ \mathit{outL}_\oplus\ x$ else $\mathit{inR}_\oplus \circ R_2 \circ \mathit{outR}_\oplus\ x$.

3.5 Separated sum

$D_1 + D_2 = \{\langle \mathbf{true}, x\rangle \mid x \in D_1\} \cup \{\langle \mathbf{false}, y\rangle \mid y \in D_2\} \cup \{\bot\}$.

The corresponding basic mappings are:

inL_+: $D_1 \Rightarrow D_1 + D_2 \;=\; \lambda x.\ pair\ \mathbf{true}\ x$
inR_+: $D_2 \Rightarrow D_1 + D_2 \;=\; \lambda y.\ pair\ \mathbf{false}\ y$
outL$_+$: $D_1 + D_2 \Rightarrow D_1 \;=\; \lambda z.\ right\ z$
$outR_+$: $D_1 + D_2 \Rightarrow D_2 \;=\; \lambda z.\ right\ z$
isL_+: $D_1 + D_2 \Rightarrow \mathrm{Bool} \;=\; \lambda z.\ left\ z$
isR_+: $D_1 + D_2 \Rightarrow \mathrm{Bool} \;=\; \lambda z.\ not \circ left\ z$
R_+: $U \Rightarrow D_1 + D_2 \;=\; \lambda x.$ if $isL_+\ x$ then $inL_+ \circ R_1 \circ outL_+\ x$ else $inR_+ \circ R_2 \circ outR_+\ x.$

3.6 Lifted space

$$D^\dagger = \{\langle \mathbf{true}, x\rangle \mid x \in D\} \cup \{\bot\}.$$

Let R_D be the projection mapping corresponding to D. The basic mappings corresponding to $D^\dagger$ are:

delay: $D \Rightarrow D^\dagger \;=\; \lambda x.\ pair\ \mathbf{true}\ x$
force: $D^\dagger \Rightarrow D \;=\; \lambda z.\ right\ z$
$R_\dagger$: $U \Rightarrow U \;=\; \lambda x.\ delay \circ R_1 \circ force\ x$

In constructing products and unions, there are three plausible symmetric ways to handle composite objects containing an undefined component:

1. A composite object (e.g., an ordered pair) containing an undefined component is identified with the undefined object in the constructed space. Coalesced products (⊗) and sums (⊕) obey this convention.

2. A constructed object containing at least one defined component is distinguished from the bottom element of the composite space. In this case, two such objects are equal only if all of their corresponding components are equal. Ordinary Cartesian products (×) obey this convention.

3. A composite object is always distinguished from the bottom element of the constructed space. In this case, the bottom element is outside the range of the constructor function corresponding to the composite space. Separated products (⊠), separated sums (+), and lifted spaces (†) all obey this convention.

Each of these three different approaches to constructing composite data objects corresponds to a different evaluation protocol (sometimes called a "computation rule" [Manna 74]) for evaluating applications of constructor functions to argument expressions. The first scheme corresponds to conventional "call-by-value" computation: evaluate all argument expressions before forming the composite object. The second scheme corresponds to dovetailing the evaluation of all argument expressions until one of them converges, and forming a composite lazy object (where the arguments other than the one that converged remain unevaluated as closures [Hend80]). The third scheme corresponds to forming a composite lazy object without evaluating any of the argument expressions.

In a lazy composite object, unevaluated arguments are evaluated only when the corresponding selector function (e.g., car and cdr in lazy LISP) is applied to the composite object. If such an application does not occur in the course of executing a program, the corresponding argument is never evaluated.

The lifting operator † provides an explicit mechanism for constructing a space of "suspended" or "unevaluated" elements corresponding to a given space D. Note that the composition of the lifted space construction with the coalesced product construction is identical to the separated product construction, i.e.,

$$D_1 \boxtimes D_2 \;=\; D_1{}^{\dagger} \otimes D_2{}^{\dagger}.$$

Similarly, the separated sum construction can be defined in terms of the appropriate composition of the lifting operator with the coalesced sum construction:

$$D_1 + D_2 \;=\; D_1{}^{\dagger} \oplus D_2{}^{\dagger}.$$

Consequently, without loss of generality, we can confine our attention (when it is convenient) to the four space constructors: $\times$ (ordinary product), $\otimes$ (coalesced product), $\oplus$ (coalesced sum), and † (lifting operator).

4. A Taxonomy of Lists

The variety of mechanisms available for constructing lazy spaces suggests that there may be several different lazy spaces that correspond to an ordinary (industrious) recursive data space (such as lists)—each with subtly different properties. In fact, the number of semantically distinct possibilities is surprisingly large. We will illustrate this phenomenon by studying list spaces in detail. In particular, we are interested in determining and classifying the possible lazy variations on the domain consisting of the retract **List**:

$$(0)\quad \text{List} \;=\; \text{Atom} \oplus (\text{List} \otimes \text{List}),$$

and the set of operations $\mathbf{O}_{\mathbf{List}}$:

$\bot$: List
$\bot_{\mathbf{At}}$: **List**
$\bot_{\mathbf{Pa}}$: **List**
t, **f**, A_1, A_2, ... : List
cons: $\text{List}^2 \rightarrow \text{List}$
car: List → List
cdr: List → List

cond: $\text{List}^3 \rightarrow \text{List}$
isAtom: $\text{List} \rightarrow \text{List}$
isPair: $\text{List} \rightarrow \text{List}$

where **t**, **f**, A_1, A_2, ... are constants denoting Lists that are atoms. We presume that *Atom* is an unspecified flat, expressive subdomain of U including the elements **true** and **false** and a set of objects Nat isomorphic to the natural numbers.

The space List defined in equation (0) is the retract characterized by the projection mapping:

$$\begin{aligned} R_{\text{List}} &= \lambda u.\ \text{if } isL\ u \text{ then } inL_{\oplus} \circ R_{\text{Atom}} \circ outL_{\oplus}\ u \\ &\quad \text{else } inR_{\oplus} \circ P_{\otimes}\ (R \circ fst_{\oplus} \circ outR_{\oplus}\ u)\ (R \circ snd_{\oplus} \circ outR_{\oplus}\ u)) \end{aligned}$$

where R_{Atom} is the retraction for Atom. In accordance with the conventions we adopted in Section 2.5, we will define the mappings in U determining the operations $\mathbf{O}_{\mathbf{List}}$. The elements (mappings) of U denoting the operations in $\mathbf{O}_{\mathbf{List}}$ are defined by:

$$\begin{aligned} \bot &= \bot \\ \bot_{\text{At}} &= inL_{\oplus}\ \bot \\ \bot_{\mathbf{Pa}} &= cons\ \bot\ \bot \\ \mathbf{t} &= inL_{\oplus}\ \mathbf{true} \\ \mathbf{f} &= inL_{\oplus}\ \mathbf{false} \\ A_i &= inL_{\oplus}(\alpha_i) \end{aligned}$$

where α_i denotes the appropriate element of Atom.

$$\begin{aligned} cons &= \lambda x.\ \lambda y.\ inR_{\oplus} \circ P_{\otimes}\ x\ y \\ car &= \lambda x.\ fst_{\otimes} \circ outR_{\oplus}\ x \\ cdr &= \lambda x.\ snd_{\otimes} \circ outR_{\oplus}\ x \\ cond &= \lambda x.\ \lambda y.\ \lambda z.\ \text{if } isL\ x \text{ then } y \text{ else } z \\ isAtom &= \lambda x.\ \text{if } isL\ x \text{ then } \mathbf{t} \text{ else } \mathbf{f} \\ isPair &= \lambda x.\ \text{if } isR\ x \text{ then } \mathbf{t} \text{ else } \mathbf{f} \\ isPair &= \lambda x.\ isR\ x. \end{aligned}$$

In the process of classifying lazy variations on the domain **List**, we will identify which one corresponds to the implementation-oriented semantics for Lazy LISP presented in the literature [Hend76, Frie76]. Our investigation will demonstrate that apparently innocuous variations in the definition of recursive data spaces have profound semantic consequences.

The obvious syntactic variations on the industrious List space defined above replace $\oplus$ by $+$, or $\otimes$ by $\times$ or $\boxtimes$. The variant spaces are:

$$\begin{array}{lll}
(1)\ \text{List} & = & \text{Atom} + (\text{List} \times \text{List}) \\
(2)\ \text{List} & = & \text{Atom} + (\text{List} \otimes \text{List}), \\
(3)\ \text{List} & = & \text{Atom} \oplus (\text{List} \times \text{List}) \\
(4)\ \text{List} & = & \text{Atom} \oplus (\text{List} \boxtimes \text{List}) \\
(5)\ \text{List} & = & \text{Atom} + (\text{List} \boxtimes \text{List})
\end{array}$$

In each variant domain, the primitive operations $\mathbf{O}_{\mathbf{List}}$ are defined in the obvious way analogous to their definition in domain (0). For example, in variation (1), the functions **cons**, **car**, **cdr** are determined by the following mappings:

$$\begin{array}{lll}
cons & = & \lambda x.\ \lambda y.\ inR_{+} \circ P_{\times}\ \ x\ y \\
car & = & \lambda x.\ fst_{\times} \circ outR_{+}\ \ x \\
cdr & = & \lambda x.\ snd_{\times} \circ outR_{+}\ \ x
\end{array}$$

We will subsequently consider other possible variations that involve the explicit use of the † operator.

As a gross categorization, we can classify list spaces on the basis of whether they accommodate infinite lists. The ordinary industrious space (0) does not, but all of the lazy variants (1)-(5) do. For example, the list **zeros** defined by the equation:

$$\mathbf{zeros} = cons\ \mathbf{0}\ \mathbf{zeros}$$

denotes the undefined element $\perp$ of the industrious space (0) while it denotes a linear list of 0's in each of the other spaces (1)-(5).

Within the class of spaces that support infinite objects, there are significant differences in the kinds of infinite and undefined objects that can appear within infinite and partial objects. By applying this form of analysis, we can demonstrate that the first four spaces (1)-(4) have fundamentally different internal structure. We can also show that space (5) is distinct from the other spaces, but the difference between it and space (1) is not significant because the two spaces (and corresponding domains) are isomorphic.

In space (1), lists can contain undefined atoms (the element <**true**, $\perp$>), undefined pairs (the element <**false**, $\perp$>), and undefined lists ($\perp$). In space (2), lists can contain undefined atoms and the undefined pair but not undefined lists. In space (3), lists can contain undefined lists but not undefined atoms and undefined pairs. In space (4), lists can contain undefined lists and undefined pairs, but not undefined atoms. In space (5), as in space (1), lists can contain undefined atoms, undefined pairs, and undefined lists. However, space (5) contains a different form of undefined pair (<**true**, <**true**, $\perp$>>) than spaces (1), (2), and (4). By inspecting a few simple examples, we can easily prove that the first four lazy domains are distinct (non-isomorphic); corresponding computations yield different answers. In domain (1), we can define:

(a) the infinite list containing no atoms;

(b) the infinite sequence containing undefined lists ($\perp$) alternating with zeros; and

(c) the list consisting of undefined atoms

by the expressions:

(a) **BigTree** = *cons* **BigTree BigTree**,

(b) **AltSeq** = *cons* $\perp$ (*cons* **0 AltSeq**), and

(c) $\perp_{At}$.

However, in the other three domains (2)-(4), at least one of the corresponding lists does not exist. In space (2), **AltSeq** denotes the undefined pair $\perp_{Pa}$; lists may not contain undefined lists. In space (3), both **BigTree** and $\perp_{At}$ denote the undefined list $\perp$; every defined list must contain a defined atom. In space (4), $\perp_{At}$ denotes the undefined list $\perp$; lists cannot contain undefined atoms. Hence, domains (1), (2), (3), and (4) are structurally distinct (nonisomorphic); the set of finite elements is fundamentally different in each case.

Although each pair (created by a **cons** operation) in domain (5) contains a redundant level of lifting, domain (5) is isomorphic to domain (1) under the function **h**:U → U determined by the mapping:

$$h = \lambda x.\ \text{if } isL\ x \text{ then } x \text{ else } pair\ \mathbf{true}\ \mathbf{h}(right \circ right\ x).$$

The function **h** simply strips one level of lifting from the representation of every List pair. The interested reader should confirm that all of the operations in O_{List} (restricted to their respective domains) are preserved by **h**.

With the aid of the † operator, we can define an even wider class of lazy list domains. First, we can define three more basic variations on lazy lists (spaces (6), (7), and (8) below) completing an enumeration of the eight possible ways (spaces (0)-(8) excluding (5)) to include or exclude undefined atoms, undefined pairs, and undefined lists. Second, we can define pairing operators that are lazy in only one argument (unlike $P_{\times}$, $P_{\boxtimes}$). Finally, we can add redundant levels of delayed evaluation in the formation of either atomic lists or paired lists analogous to the extra level that appears in paired lists in space (5). Since every domain in the final class (involving redundant levels of lifting) is isomorphic to a space outside the class, we will not discuss this class any further.

To facilitate classifying the extra spaces, we rewrite the definitions of the five basic lazy list spaces (1)-(5) in terms of the operators $\times$, $\otimes$, $+$, $\oplus$, and $\dagger$:

(1) $\text{List} = \text{Atom}^{\dagger} \oplus (\text{List} \times \text{List})^{\dagger}$
(2) $\text{List} = \text{Atom}^{\dagger} \oplus (\text{List} \otimes \text{List})^{\dagger}$
(3) $\text{List} = \text{Atom} \oplus (\text{List} \times \text{List})$
(4) $\text{List} = \text{Atom} \oplus (\text{List} \times \text{List})^{\dagger}$
(5) $\text{List} = \text{Atom}^{\dagger} \oplus [(\text{List} \times \text{List})^{\dagger}]^{\dagger}$.

In this standardized form, the close relationship between space (5) and space (1) is evident.

The remaining interesting variations on lazy lists are:

(6) $\text{List} = \text{Atom}^{\dagger} \oplus (\text{List} \boxtimes \text{List})$
(7) $\text{List} = \text{Atom} \oplus (\text{List} \otimes \text{List})^{\dagger}$
(8) $\text{List} = \text{Atom}^{\dagger} \oplus (\text{List} \otimes \text{List})$
(9) $\text{List} = \text{Atom} \oplus (\text{List}^{\dagger} \otimes \text{List})$
(10) $\text{List} = \text{Atom} \oplus (\text{List} \otimes \text{List}^{\dagger})$
(11) $\text{List} = \text{Atom}^{\dagger} \oplus (\text{List}^{\dagger} \otimes \text{List})$
(12) $\text{List} = \text{Atom}^{\dagger} \oplus (\text{List} \otimes \text{List}^{\dagger})$.

Variation (6) accommodates undefined atoms and undefined lists, but not undefined pairs. Variation (7) does exactly the opposite: it accommodates undefined pairs, but not undefined atoms or lists. Variation (8) is only marginally lazy: within lists it accommodates undefined atoms, but not undefined lists or undefined pairs. Variations (9), (10), (11), (12) all delay the evaluation of only one argument of a paired list. As a result, spaces (9) and (11) allow infinitely deep lists but not infinitely long ones while spaces (10) and (12) do the opposite. Spaces (9) and (10) prohibit undefined atoms while spaces (11) and (12) accommodate them.

At this point, the question arises: Which denotational definition of lazy lists corresponds to the standard implementation-oriented definition given in the literature [Frie76]? The answer is (4), because their space accommodates undefined lists and undefined pairs but not undefined atoms.

The situation is somewhat more complicated in the case of the semantics presented in [Hend76]. Their semantic definition describes a space isomorphic to (1), but the definable data points are contained within a subdomain isomorphic to (4), because the operations in their domain cannot generate undefined atoms.

5. Axiomatizing Lazy Data Domains

Since there are significant differences between various formulations of lazy data domains, it is important to develop clear, comprehensive axiomatic definitions for the alternatives. Naively, we might attempt to specify a lazy space like:

(1) List = Atom + List $\times$ List

(given an axiomatization for Atom) by devising a list of equations such as those presented in Section 3 and designating the lazy space as the corresponding initial algebra [ADJ76, 77] (or alternatively the corresponding final algebra [Kami80]). From our previous discussion, it seems reasonable to conjecture that this task will be deceptively difficult given the variety of lazy spaces available. In fact, it is impossible. No recursively enumerable set of equations can specify a non-trivial lazy space as either the initial or final algebra corresponding to the specification. We will formally prove this fact after we establish a few important properties of lazy spaces.

Unlike ordinary data domains, lazy spaces have infinite strictly ascending chains of objects $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$ (where $\sqsubseteq$ denotes the approximation relation introduced in Section 2) where each object d_i is constructed in exactly the same way as d_{i+1} except that d_i uses $\perp$ to approximate substructures of d_{i+1}. In ordinary industrious data domains (such as LISP Lists), the undefined object $\perp$ cannot be embedded inside constructed objects, which precludes the existence of infinite ascending chains of successively more complete approximations.

This apparently small change in the definition of data constructors (e.g., the LISP **cons** operation) profoundly changes the structure of the data domain. Ordinary structural induction, for example, no longer holds, because lazy spaces contain the limit elements of infinite ascending chains—which cannot be constructed from primitive constants (e.g., atoms) in a finite number of steps. For example, in the space of industrious lists, $\text{List}_{(0)}$, let the operation **leafcount** be recursively defined by the equation:

$$\textbf{leafcount}(x) = \textbf{if isAtom}(x) \textbf{ then 1 else leafcount(car}(x)\textbf{))} + \textbf{leafcount(cdr}(x)\textbf{))},$$

where **if** α **then** β **else** γ abbreviates **cond(**α, β, γ) and the addition operation (+) is defined on integer atoms in the usual way. Then the following theorem is easily proved by structural induction on x:

$$\forall x\ x \neq \perp \Rightarrow \textbf{leafcount}(x) > 0.$$

On the other hand, as soon as we extend the space $\text{List}_{(0)}$ to include limit points, the principle of structural induction fails. In a List space including the object BigTree (such as $\text{List}_{(1)}$), the preceding theorem is clearly false.

Since lazy spaces include limit points, they have a much more complex topological structure than their industrious counterparts. An important illustration of this phenomenon is the following observation. Let Triv denote the trivial subspace of U consisting of the objects **true** and $\perp$. Although the industrious space:

$$\text{Trivseq}_{\text{Ind}} = \text{Triv} \otimes \text{Trivseq}_{\text{Ind}}$$

is completely degenerate (it contains no elements other than $\perp$), the corresponding lazy space:

$$\text{Trivseq} = (\text{Triv} \times \text{Trivseq})$$

is isomorphic to Scott's **P**ω model for the untyped lambda calculus under the mapping α defined by:

$$\alpha(x) = \{ \mathrm{i} \mid x_{\mathrm{i}} = \mathbf{true} \}$$

where x_{i} denotes the ith element of $x = \langle x_0, x_1, ..., x_j, ... \rangle$.

Pω is the space consisting of all subsets of the natural numbers under the approximation ordering defined by the subset relation. If we strengthen the definition of a space by adding the requirement that every space must contain a maximum element $\top$ and we weaken the definition of subspace as discussed in Section 2.1, then **P**ω is a universal space. Hence, **P**ω contains a subspace D such that D is isomorphic to the space **P**$\omega \Rightarrow$**P**ω. Moreover, if we augment the space **P**ω by a very small set of operations $\mathbf{O}_{\mathbf{P}\omega}$, the resulting domain **P**ω is universal. $\mathbf{O}_{\mathbf{P}\omega}$ consists of the constant **0** denoting the singleton set {0}, the primitive binary operation **Apply**: $\mathbf{P}\omega^2 \rightarrow \mathbf{P}\omega$ (defined exactly as in Section 2.3), and the primitive mappings (which are constant operations):

$$\begin{aligned}
&succ\text{: } \mathbf{P}\omega \Rightarrow \mathbf{P}\omega \\
&pred\text{: } \mathbf{P}\omega \Rightarrow \mathbf{P}\omega \\
&cond\text{: } \mathbf{P}\omega \Rightarrow (\mathbf{P}\omega \Rightarrow (\mathbf{P}\omega \Rightarrow \mathbf{P}\omega)) \\
&K\text{: } \mathbf{P}\omega \Rightarrow (\mathbf{P}\omega \Rightarrow \mathbf{P}\omega) \\
&S\text{: } \mathbf{P}\omega \Rightarrow (\mathbf{P}\omega \Rightarrow (\mathbf{P}\omega \Rightarrow \mathbf{P}\omega))
\end{aligned}$$

defined by:

$$\begin{aligned}
&\mathbf{0} = \{0\} \\
&succ\ x = \{e+1 \mid e \in x) \\
&pred\ x = \{e \mid e+1 \in x) \\
&cond\ x\ y\ z = \{e \mid e \in y \land 0 \in x) \cup \{e \mid e \in z \land 1 \in y\} \\
&K\ x\ y = x \\
&S\ x\ y\ z = (x\ z)\ (y\ z).
\end{aligned}$$

Surprisingly, all of these operations are recursively definable in a domain containing the lazy subspaces Trivseq and Triv together with the obvious "structural" operations:

true, $\perp$: Triv
por, **and**: $\text{Triv}^2 \rightarrow \text{Triv}$
cons: Triv$\times$Trivseq $\rightarrow$ Trivseq
hd: Trivseq $\rightarrow$ Trivseq
tl: Trivseq $\rightarrow$ Trivseq.

Note that the Cartesian product symbol $\times$ immediately above does not conform to our normal usage of the notation: **cons** is a *binary* function —not a unary function on pairs. The recursive definitions of the operations $\mathbf{O}_{\mathbf{P}\omega}$ in Trivseq (which are a bit tedious) appear in the **Appendix**.

Since **P**ω together with the binary operation **Apply**: **P**$\omega^2 \rightarrow$ **P**ω and mapping constants S and K forms a model for the (untyped) lambda calculus (excluding η-reduction), the lazy space Trivseq together with the corresponding operations also constitutes a model for the untyped lambda calculus. Trivseq is a particularly attractive model for computer scientists, because it is based on widely understood concepts from applicative programming. Lazy spaces are the natural "higher order" generalization of familiar recursive data structures.

We have now developed sufficient machinery to prove the theorem establishing the inadequacy of algebraic specification as a formalism for specifying lazy spaces:

Theorem: Neither initial algebra specifications nor final algebra specifications (consisting of a recursively enumerable set of equations) can define non-trivial lazy spaces.

Proof: We will prove the theorem for the specific lazy space Trivseq, but it is clear that Trivseq can be implemented within any non-trivial lazy space D using an abstraction function (homomorphism) mapping D onto Trivseq.

The initial algebra corresponding to a recursively enumerable set of equations A is the set of equivalence classes of variable-free terms under the relation MustEqual, where MustEqual(a, b) is true iff the sentence $a=b$ is derivable from A by first order deduction. Hence the equality relation on variable-free terms is recursively enumerable. Yet the equality relation for a Trivseq is obviously not recursively enumerable; otherwise, we could recursively enumerate the set of all pairs of equivalent programs (using the untyped λ-calculus as our programming language)—a set which is obviously not recursively enumerable.

Similarly, the final algebra corresponding to a set of equations A (assuming the final algebra exists) is the set of equivalence classes under the complement of the relation CannotEqual where CannotEqual(a, b) is true iff the sentence $a \neq b$ is derivable from A $\cup$ {**true**$\neq$**false**} by first order deduction. Note that if A has no final algebra, then the complement of CannotEqual is not an equivalence relation. For a final algebra, the inequality relation is obviously recursively enumerable, but again the inequality relation for Trivseq clearly is not. Otherwise, we could recursively enumerate the set of all pairs of inequivalent programs (corresponding to unequal *partial* recursive functions), a set which is obviously not recursively enumerable. □

Since lazy spaces are so similar in structure to **P**ω, an obvious approach to formulating a logic for lazy spaces is to use a higher order logic based on the lambda calculus (similar to Edinburgh LCF) that conveniently expresses the properties of **P**ω. (See [Giles78] for an LCF axiomatization of lazy lists.)

However, we would prefer not to abandon first-order logic for two reasons. First, first-order systems (such as first-order Peano arithmetic) based on structural induction provide a simple, elegant characterization of ordinary data spaces. The highly successful Boyer-Moore LISP Verifier [Boyer75, 79] is based on such a first-order system. We would like to extend this approach to handle lazy lists as well. Second, the completeness theorem for first order logic provides an invaluable tool for analyzing the deductive power of any theory. If a first order theory is too weak to establish a particular theorem, there must be a non-standard model in which that theorem is false. In higher order logics, on the other hand, a theory may be too weak to prove an important theorem, yet there may be no model that refutes it.

6. A First-Order Theory of Lazy Domains

The chief obstacle to extending ordinary first-order structural induction theories to lazy domains is that conventional structural induction is applicable only to well-founded sets, yet lazy spaces under the (proper) containment (substructure) ordering determined the constructors are not well-founded because a limit element (e.g., **BigTree**) can properly contain itself. Let **D** = <D, **G**> be a data domain with signature **G** such that:

(i) **G** contains two constants **true** and **false** denoting inconsistent finite elements of D and the standard ternary conditional function **cond** defined as in Section 3.

(ii) **G** contains a finite set of constructor functions **C** = $\{c_1, ..., c_n\}$ that *generate* the basis of D. In other words, **C** satisfies the following properties:

(a) For every basis element $b \in B$, there exists a term ρ_b composed solely from operations in **C** such that ρ_b denotes b.

(b) For all $\mathbf{c} \in \mathbf{C}$, $\forall\ x_1, ..., x_{\#c} \in B\ \ \mathbf{c}(x_1, ..., x_{\#c}) \in B$.

(c) For all $c_i, c_j \in \mathbf{C}$,

$$\forall x_1, ..., x_{\#ci}, y_1, ..., y_{\#cj} \in B\ [c_i(x_1, ..., x_{\#ci}) \sqsubseteq c_j(y_1, ..., y_{\#cj}) \Rightarrow c_i(x_1, ..., x_{\#ci}) = \perp \vee (\ i=j \wedge x_1 \sqsubseteq y_1 \wedge ... \wedge x_{\#ci} \sqsubseteq y_{\#ci})\ ]$$

(iii) For each constructor $\mathbf{c} \in \mathbf{C}$, **G** contains selector functions s_j, $j=1, ..., \#c$ such that:

$$s_j(\mathbf{c}(x_1, ..., x_{\#c})\ ) = x_j \ \text{ if } \mathbf{c}(x_1, ..., x_{\#c}) \neq \perp$$

and a characteristic function **isc**: D → Bool such that:

$$\begin{aligned} \mathbf{isc}(x) &= \perp \quad \text{if } x=\perp \\ \mathbf{isc}(x) &= \mathbf{true} \quad \text{if } x \neq \perp \wedge \mathbf{c}(s_1(x), ..., s_{\#c}(x))=x \\ \mathbf{isc}(x) &= \mathbf{false} \quad \text{otherwise.} \end{aligned}$$

The basis B of D forms a well-founded set under the *substructure* ordering (which is *not* an approximation ordering) which is the transitive closure of the binary relation:

$$\bigcup_{c \in C} (\bigcup_{j=1, ..., \#c} \{ (x_j, \mathbf{c}(x_1, ..., x_{\#c}) \mid x_1, ..., x_{\#c} \in B \wedge \mathbf{c}(x_1, ..., x_{\#c}) \neq \perp \}$$

If **D** is industrious, then D = B, and the substructure ordering $\subset$ on D is the conventional well-founded ordering used in the structural induction scheme for D. It is a straightforward (but tedious, and error-prone) task to devise a first order axiomatization (comparable in deductive power to the first order formulation of Peano's axioms) for an industrious domain **D** consisting of:

(1) implications between equations relating the operations in **G** (e.g., constructors, selectors, characteristic functions, if-then-else);

(2) inequations asserting that the Boolean truth values **true**, **false**, and the undefined object $\perp$ are all distinct;

(3) axioms describing the substructure ordering $\sqsubset$ and the approximation ordering $\sqsubseteq$ (which are both predicates);

(4) the structural induction scheme:

$$\bigwedge_{c \in C} [\, \forall\, x_1, \ldots, x_{\#c} \;\; (\bigwedge_{i=1, \ldots, \#c} \varphi(x_i) \Rightarrow \varphi(c(\, x_1, \ldots, x_{\#c}))) \,] \Rightarrow \forall x\, \varphi(x)$$

or, equivalently,

$$\forall x\, [\forall x'\, (x' \sqsubset x \Rightarrow \varphi(x')) \Rightarrow \varphi(x)] \Rightarrow \forall z\, \varphi(z).$$

A detailed account of this process appears in [Cart80].

The corresponding problem for lazy domains **D** is much more subtle. If we construct the axiomatization described above for a lazy domain **D**, then the specified space contains only the finite objects (basis elements) of the lazy space. (Non-standard models may contain "infinite objects", but their behavior does not resemble that of lazy data objects.) The structural induction scheme (4) has the effect of banning infinite objects (limit points) from the domain. In fact, if we extend the axiomatized structure to include the characteristic predicate IsFin for finite objects and augment the axiomatization by a sentence asserting that constructors map finite objects to finite objects, then we can prove:

$$\forall x \; \mathrm{IsFin}(x) = \mathbf{true}$$

by structural induction.

As a result, recursive definitions over the domain may not have least fixed points because directed sets do not necessarily have least upper bounds. For example, if we consider a domain consisting the finite objects in Trivseq, the function definition:

$$\mathbf{f}(x) = \mathbf{cons}(\mathbf{true}, \mathbf{f}(x))$$

is contradictory, because we can prove by structural induction that:

$$\forall\; x, y \;\; x \neq \mathbf{cons}(y, x)$$

including $x = \perp$!

If we replace induction scheme (4) by an induction axiom scheme restricted to finite objects:

(4') $\forall x\,[\mathrm{IsFin}(x) \Rightarrow [\forall x'[x' \sqsubset x \Rightarrow \varphi(x')] \Rightarrow \varphi(x)]] \Rightarrow\ [\forall z\ \mathrm{IsFin}(z) \Rightarrow \varphi(z)],$

then the lazy space is a model for our axiomatization, but so is the subspace containing only finite objects. In such a theory, we could not prove any interesting statements about infinite objects.

7. A Satisfactory Axiomatization

The solution to the problem is to augment the axiomatization consisting of (1), (2), (3), and (4') above by two additional schemes asserting that:

(5) Every definable directed set has a least upper bound.

(6) Every term t(x) over the domain operations **G** is continuous in the variable x.

They are formalized as follows. Let $\varphi(u)$ and t(u) be an arbitrary formula and term respectively in the language of the data domain and let x, y, z be variables not free in either $\varphi(u)$ or t(u). Let **Dir**{t(u)|$\varphi(u)$} abbreviate the formula:

$$\forall x, y\, [\varphi(x) \wedge \varphi(y)\ \Rightarrow \exists z(\varphi(z) \wedge x \sqsubseteq \mathrm{t}(z) \wedge y \sqsubseteq \mathrm{t}(z))]$$

which asserts that {t(u)|$\varphi(u)$} is a directed set. Let **lub**{t(u)|$\varphi(u)$}(v) abbreviate the formula:

$$\forall x\,([\varphi(x) \Rightarrow \mathrm{t}(x) \sqsubseteq v] \wedge \forall z[\forall x\ \varphi(x) \Rightarrow \mathrm{t}(x) \sqsubseteq z] \Rightarrow \mathrm{t}(x) \sqsubseteq v$$

which asserts that v is the least upper bound of the set {t(u)|$\varphi(u)$}. (Note that u is not free in either **Dir**{t(u)|$\varphi(u)$} or **lub**{t(u)|$\varphi(u)$}(v)). Then the two additional schemes are:

(5) (the existence of least upper bounds)

$$\mathbf{Dir}\{\mathrm{t}(u)|\varphi(u)\} \Rightarrow \exists v\ [\mathbf{lub}\{\mathrm{t}(u)|\varphi(u)\}(v)]$$

(6) (the continuity of functions)

$$\mathbf{lub}\{u|\varphi(u)\}(v) \Rightarrow \mathbf{lub}\{\mathrm{t}(u)|\varphi(u)\}(\mathrm{t}(v)).$$

where t(u) and $\varphi(u)$ are an arbitrary term and formula containing no free variable other than u. Scheme (5) asserts that if the set {t(u)|$\varphi(u)$} is directed, then it has a least upper bound. Scheme (6) asserts that if the set {u|$\varphi(u)$} has a least upper bound v, then the function λu. t(u) is continuous at v.

Although there are no blatant sources of incompleteness in this axiomatization (consisting of (1), (2), (3), (4a), (4b), (5), (6)), it is not obvious that the system is strong enough to prove all of the important properties of particular lazy spaces. (For a non-trivial lazy space (e.g., Trivseq) the axiomatization is obviously not complete by Godel's first incompleteness theorem.) For this reason, it is interesting to compare the power of our first-order system with the corresponding theory in LCF, a logic specifically designed to accommodate "higher order" spaces like **P**ω. The LCF theory looks similar except:

1. It includes the typed lambda calculus in the term syntax for the logic.

2. The induction axiom scheme is fixed point induction on recursively defined functions. This scheme has the form:

$$\varphi(\bot) \wedge \forall f[\varphi(f) \Rightarrow \varphi(\tau[f])] \Rightarrow \varphi(Y(\lambda f.\ \tau[f]))$$

where $\varphi(f)$ is a formula that *admits* induction on f. Fixed-point induction is applicable only to *admissible* formulas, where admissibility is a complex syntactic test (described in [Gord77]) that analyzes the types of terms within the formula.

The closest analog of structural induction in LCF is fixed point induction on a retraction characterizing the domain of interest. The fixed point induction scheme has the form:

$$(7)\quad [\forall f\ \varphi(f) \Rightarrow \varphi(\tau[f])] \Rightarrow \varphi(Y(\lambda f.\ \tau[f]))$$

where f is a function of type T, τ is a functional mapping functions of type T to functions of type T, $\varphi(f)$ is an admissible formula containing no free variables other than f, and Y is the least fixed point operator.

After studying the two systems, we were surprised to discover that our system subsumes LCF both in expressiveness and deductive power. In particular, we can systematically translate arbitrary LCF statements into equivalent statements in our first order system by:

(i) Converting all lambda expressions into equivalent expressions formed using the standard S and K combinators.

(ii) Converting all function applications to explicit applications (using the primitive operation **Apply**) of corresponding mapping.

Unlike many translations between formal systems, this translation does not mutilate the syntactic structure of the original formula. In fact, if we use the abbreviated notation for terms described in Section 2, the first order translation of an LCF formula is identical to the original formula!

Under this translation, all of the LCF proof rules and axioms (expressed in terms of translated formulas) are derivable in our first-order system. In particular, we can derive the LCF fixed point induction scheme for admissible formulas. The derivation critically relies on the structural induction scheme for finite objects (4'), the least upper bound scheme (5), and the continuity scheme (6).

We call the first order analog of fixed-point induction, *lazy induction.* If we use the abbreviated notation described in Section 2, then the lazy induction scheme is identical in appearance to the fixed point scheme (7). The formal derivation of lazy induction within our system is a tedious induction on the structure of formulas that is beyond the scope of this paper, but the basic idea underlying the proof is instructive.

The admissibility test in LCF ensures that passing to the limit of a directed set (of lazy data objects) does not change the meanings of subformulas that determine the truth of the entire formula. The idea behind the derivation is that the metamathematical justification for fixpoint induction on a function within a particular admissible formula can be translated into a proof in our first order system consisting of two parts. The first part utilizes conventional structural induction to establish that the formula holds for all finite approximations to the function. The second part extends the result to the entire function (an infinite lazy object) by appealing to the definition of admissibility and the fact that all functions in the domain are continuous.

Although the admissibility test required for lazy induction is awkward, the rule can be a useful shortcut in certain situations. A particular important example is lazy induction on the retraction R_D characterizing the recursive data type D defined by the domain equation:

$$D = D^{n1} + \ldots + D^{nk}$$

where $n_1, \ldots, n_k$ are positive integers. For each component D^{ni} of D, let isc_i, c_i, and $s_{1,j}$, $j=1, \ldots, n_i$ denote the recognizer, constructor, and selector functions, respectively, used to identify, build, and tear apart objects of form D^{ni} within D. Then R_D is defined by the equation:

$$\begin{aligned} R_D = \lambda x.\ & \text{if } isc_1\ x \text{ then } c_1\ (R \circ s_{1,1}\ x) \ldots (R \circ s_{1,n1}\ x) \ldots \\ & \text{else if } isc_k\ x \text{ then } c_k\ (R \circ s_{k,1}\ x) \ldots\ (R \circ s_{k,nk}\ x) \text{ else } \bot. \end{aligned}$$

When we apply lazy induction to this retraction, the premises of the rule reduce to the premises of conventional structural induction for the finite objects of the space. Similarly, the conclusion of the rule reduces to an assertion that the hypothesis holds for all objects in D. Hence, if a formula is admissible, conventional structural induction establishes the formula holds for all objects in D, not just finite ones!

8. Sample Program Proofs

Consider the recursive definition:

$$\mathbf{append}(x, y) = \mathbf{if\ isAtom}\ x\ \mathbf{then}\ y\ \mathbf{else\ cons(car}(x), \mathbf{append}(\mathrm{cdr}(x), y))$$

over the data domain $\mathrm{List}_{(1)}$. The following formula:

$$\forall x, y, z\ \mathbf{append}(x, \mathbf{append}(y, z)) = \mathbf{append(append}(x, y), z).$$

is obviously true on the domain of finite objects (including $\bot$). The proof is a trivial induction on the structure of x. Does the same theorem hold for all lazy lists? The answer must be yes, because the formula stating the theorem is admissible! Lazy induction enables us to prove theorems about lazy spaces using conventional structural induction.

On the other hand, lazy induction is not sound if the induction formula is not admissible. For instance, consider the formula:

(8) $\forall\ x \in \text{List}\ (\perp \sqsubseteq \mathbf{zap}(x))$

where the function **zap** and the relation $\sqsubseteq$ are defined by the formulas:

$\mathbf{zap}(x) = \mathbf{if}\ \text{isAtom}(x)\ \text{then}\ \perp\ \mathbf{else}\ \mathbf{cons}(\mathbf{car}(x), \mathbf{zap}(\mathbf{cdr}(x)))$
$x \sqsubseteq y \Leftrightarrow (\mathrm{x}=\mathrm{y}) \vee (\mathrm{x} \sqsubset \mathrm{y})$.

By induction on x, we can trivially "prove" the formula (8), yet it is clearly false for lazy lists since:

$\mathbf{zap}(\mathbf{BigTree}) = \mathbf{BigTree}$

where **BigTree** is defined as in Section 4. In this case, lazy induction fails because the formula (8) is not admissible.

9. Conclusions and Future Research

Although implementation-oriented definitions of lazy evaluation provide some insight into the behavior of particular computations, they are inadequate as the basis of a logical theory of lazy spaces. They also blur subtle but important semantic distinctions between different forms of lazy evaluation. Our abstract characterization in terms of domain constructors provides a much clearer picture of the mathematical properties of lazy spaces and directly corresponds to a natural formal system for reasoning about them.

Since lazy spaces have essentially the same complex structure as Scott's $\mathbf{P}\omega$ model of the untyped lambda calculus, they cannot be specified by restrictive specification methods such as algebraic specification. One approach is to axiomatize lazy spaces within a least fixed point logic such as LCF. In this paper we have presented a first-order theory of lazy spaces that we prefer to higher order formalizations because it relies on conventional structural induction rather than fixed point induction as the fundamental axiom scheme. In our system, the admissibility test for fixed point induction is simply a sufficient set of conditions for its derivation. Moreover, our system extends conventional structural induction (as implemented in the Boyer-Moore LISP Verifier [Boyer75, 79]) to the context of lazy data domains, providing the programmer with a simple intuitive framework for reasoning about functions that manipulate lazy data objects.

Since computable functions have a natural extensional representation as lazily evaluated graphs (mappings), our first-order formalization of lazy spaces accommodates function spaces as well. (There are still multiple "partial" mappings corresponding to the same function, but the only difference between an arbitrary mapping and the canonical one for the equivalence class is that the canonical one contains every possible piece of redundant information.) However, we must overcome one major obstacle to make our treatment of functions intuitively accessible to programmers: our reliance on combinators rather than lambda expressions to denote computable mappings. In response to this issue, we are currently developing a collection of combinators that closely correspond to conventional lambda notation.

References

[ADJ76]

Goguen, J., Thatcher, J., and Wagner, E. An initial algebra approach to the specification, correctness and implementation of abstract data types. Yorktown Heights, NY: IBM Thomas J. Watson Research Center; 1976; Technical Report RC-6478.

[ADJ77]

Goguen, J., Thatcher, J., Wagner, E., and Wright, J. Initial algebra semantics and continuous algebras. Journal of the ACM. 1977 January; 24(1): 68-95.

[Back78]

Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Communications of the ACM. 1978 August; 21(8): 613-641.

[Bare77]

Barendregt, H. The type free lambda calculus. In: Barwise, J., ed. Handbook of mathematical logic. Amsterdam: North-Holland; 1091-1132.

[Boye75]

Boyer, R. S., and Moore, J S. Proving theorems about LISP functions. *Journal of the ACM.* 1975 January; 22(1): 129-144.

[Boye79]

Boyer, R. S. and Moore, J S. A computational logic. New York: Academic Press; 1979.

[Cart76]

Cartwright, R. User-defined data types as an aid to verifying LISP programs. Proceedings of the International Conference on Automata, Languages and Programming. Edinburgh Press, 1976.

[Cart80]

Cartwright, R. A. Constructive alternative to axiomatic data type definitions. Proceedings of the 1980 LISP Conference; Stanford, CA; 1980.

[Ende72]

Enderton, H. B. A mathematical introduction to logic. New York: Academic Press; 1972.

[Frie76]

Friedman, D. and Wise, D. CONS should not evaluate its arguments. Proceedings of the International Conference on Automata, Languages and Programming. Edinburgh University Press; 1976; 257-284.

[Gile78]

Giles. An LCF axiomatization of lazy lists. Edinburgh University; Computer Science Department; CSR-31-78.

[Gord77]

Gordon, M., Milner, R., and Wadsworth, C. Edinburgh LCF. Edinburgh University; Computer Science Department; CSR-11-77.

[Gutt78]

Guttag, J. and Horning, J. The algebraic specification of abstract data types. Acta Informatica. 1978; 10(1); pp. 27-52.

[Hend80]

Henderson, P. Functional programming: Application and implementation. London: Prentice-Hall; 1980.

[Hend76]

Henderson, P and Morris, J., Jr. A lazy evaluator. Proceedings of the Third Symposium on Principles of Programming Languages; 1976; ACM: 95-103.

[Kami80]

Kamin, S. Final data type specifications: A new data type specification method. Proceedings of the Seventh Symposium on Principles of Programming Languages; 1980 January 28-30; Las Vegas, NV. ACM: 131-138.

[Scot76]

Scott, D. Data types as lattices. SIAM Journal of Computing. 1976 September; 5(3): 522-587.

[Scot81]

Scott, D. Lectures on a mathematical theory of computation. Oxford, UK: Oxford University Computing Laboratory; Technical Monograph PRG-19.

[Scot83]

Scott, D. Domains for denotational semantics. Pittsburgh, PA: Carnegie-Mellon University; Computer Science Department; Technical Report; 1983.

[Stoy77]

Stoy, J. Denotational semantics: The Scott-Strachey approach to programming language theory. Cambridge, MA: MIT Press; 1977.

Appendix: Mapping Pω Onto the Lazy Space Trivseq

Each data object X in the lazy space Trivseq is an infinite sequence x_0, x_1, ..., x_i, ... in which each element x_i is either **true** or $\perp$. In effect, a member of Trivseq is a potentially infinite enumeration of natural numbers (the indices of the convergent elements). Consequently, the abstraction function α: Trivseq $\rightarrow$ Pω defined by:

$\alpha(X) = \{ \mathrm{i} \mid x_{\mathrm{i}} = \mathbf{true} \}$

establishes a natural isomorphism between the two spaces.

This appendix contains a recursive program defining the operations $\mathbf{Op}_{\mathbf{P}\omega}{}'$ over Trivseq corresponding to the basic operations $\mathbf{Op}_{\mathbf{P}\omega}$ of $\mathbf{P}\omega$. The style of this program is rather unusual because all computations over Trivseq are infinite enumerations in which the subcomputations determining individual elements are dovetailed (performed in parallel)—an unfamiliar phenomenon in conventional applicative languages such as Pure LISP.

For the sake of clarity, each individual recursive function definition in the program obeys the following syntactic conventions.

1. Each definition has the form:

   $\mathbf{f}(x) \equiv$ *informal-definition* $=$ *formal-definition*

   where an *informal-definition* is a mathematical description of the value of the function and *formal-definition* is the actual body of the function definition. If the *formal-definition* is transparent, then the *informal-definition* may be omitted.

2. The names of Trivseq operations (functions that return values of type Trivseq) are capitalized; the names of Triv operations (functions that return values of type Triv) are not. Triv operations are used as subfunctions within the definitions of the functions in $\mathbf{Op}_{\mathbf{P}\omega}{}'$.

3. Variables ranging over Trivseq that are intended to denote arbitrary sets in $\mathbf{P}\omega$ are capitalized. Variables ranging over Trivseq that are intended to denote individual natural numbers (singleton sets) are not. No variables range over Triv.

4. In every *unary* function application, the parentheses enclosing the argument are omitted. Note that this is *not* the same abbreviation we employed in connection with mappings in the main body of the paper. In the following program, every application within an expression is explicitly written down; consequently, a chain of unary applications $\mathbf{f}\ \mathbf{g}\ \mathbf{h}\ x$ associates to the right $[\mathbf{f}(\mathbf{g}(\mathbf{h}(x)))]$, rather than the left $[(((\mathbf{f}\ \mathbf{g})\ \mathbf{h})\ x)]$.

5. In informal definitions (comments), the following special notation appears.

   (a) The symbol ε_{i} denotes the finite set in $\mathbf{P}\omega$ corresponding to the binary coded integer i, i.e.,

   {j | bit j in the binary representation of i is **1**}

   where bits are numbered from right to left starting with 0.

   (b) The function symbol ρ denotes the inverse of the function α, i.e., $\rho\ s$ is the infinite sequence denoting the set of natural numbers s.

   (c) The bracketed pair $\langle \mathrm{i}, \mathrm{j} \rangle$ abbreviates the arithmetic expression $[(\mathrm{i}+\mathrm{j})*(\mathrm{i}+\mathrm{j}+1)]/2 + \mathrm{i}$. The binary function $\lambda \mathrm{i}, \mathrm{j}\ .\ \langle \mathrm{i}, \mathrm{j} \rangle$ is a commonly used bijective pairing function.

Auxiliary Operations

The following collection of auxiliary operations $\mathbf{O}_{\mathrm{Aux}}$ are used in the definition of the primitive operations $\mathbf{O}_{\mathbf{P}_\omega}$ of $\mathbf{P}\omega$.

def X $\equiv \exists i \in \alpha X$ =
hd x **por def Tl** X

Plus(I, J) $\equiv \{ i+j \mid i \in \alpha I \wedge j \in \alpha J \}$ =
Cons(hd I **and hd** J, **Cons([hd Tl** I **and hd** J**] por [hd** I **and hd Tl** J**], Plus(Tl** I, **Tl** J**)))**

Times(I, J) $\equiv \{ i*j \mid i \in \alpha I \wedge j \in \alpha J \}$ =
Cons([def I **and hd** J**] or [hd** I **and def** J**], Plus(Tl** I, **Times(**I, **Tl** J**)))**

Pair(I, J) $\equiv \{ \langle i, j\rangle \mid \exists i \in \alpha(I) \wedge \exists j \in \alpha(J) \}$ =
Plus(Halve Times(Plus(I, J**), Plus (Plus(**I, J**), Succ 0))),** I**)**

Fst X $\equiv \{ i \mid \exists j \langle i, j\rangle \in \alpha X \}$ =
Fst$_1$(0, X**)**

Fst$_1$(k, X) $\equiv \{ i\text{-}k \mid \exists j \langle i, j\rangle \in \alpha X \}$ =
Cons(anySnd(k, **0,** X**), Fst$_1$(Succ** k, X**))**

anySnd(i, k, X) $\equiv \exists j \geq k\, [\langle i, j\rangle \in \alpha X]$ =
Overlap(Pair(i, k**),** X**) por anySnd(**i, **Succ** k, X**)**

Snd X $\equiv \{ j \mid \exists i\, [\langle i, j\rangle \in \alpha X] \}$ =
Snd$_1$(0, X**)**

Snd$_1$(k, X) $\equiv \{ j\text{-}k \mid \exists i\, [\langle i, j\rangle \in \alpha X] \}$ =
Cons(anyFst(0, k, X**), Snd$_1$(Succ** k, X**))**

anyFst(k, j, X) $\equiv \exists i \geq k\, [\langle i, j\rangle \in \alpha X]$ =
Overlap(Pair(k, j**),** X**) por anyFst(Succ** k, j, X**)**

Overlap(I, J) $\equiv \exists i\; i \in [\alpha I \wedge i \in \alpha J]$ =
hd I **and hd** J **por Overlap(Tl** I, **Tl** J**)**

Top $\equiv \{i\}$ =
Cons(true, Top)

odd X $\equiv \exists i\, [2*i+1 \in \alpha X]$ =
hd Tl x **por odd Tl Tl** X

Halve X $\equiv \{ i \mid 2*i \in \alpha X \} \cup \{ j \mid 2*j+1 \in \alpha X \}$ =
Cons(hd X **por hd Tl** X, **Halve Tl Tl** X**)**

approx(i, X) $\equiv \varepsilon_i \subseteq \alpha X$ =
hd i por [([odd i and hd X] por odd Tl i) and approx(Halve i, Tl X)]

Primitive Operations of Pω

Recursive definitions for all the operations in $\mathbf{O}_{\mathbf{P}_\omega}$' = { **0**, *Succ*, *Pred*, *Cond*, *K*, *S*, **Apply** } in terms of the auxiliary operations $\mathbf{O}_{\mathrm{Aux}}$ appear below.

0 $\equiv$ {0} = **Cons(true, $\bot$)**

Succ = **GraphSucc 0**

GraphSucc k $\equiv$ { <i, j>-k | <i, j> $\geq$ k $\wedge$ j $\in$ [α **Succ** ρ ε_i] } =
Cons(approx(Snd k, **Succ Fst** k), **GraphSucc Succ** k)

Succ I $\equiv$ { i+1 | i $\in$ αI } = **Cons($\bot$,** I)

Pred $\equiv$ **GraphPred 0**

GraphPred k $\equiv$ { <i, j>-k | <i, j> $\geq$ k $\wedge$ j $\in$ [α **Pred** ρ ε_i] } =
Cons(approx(Snd k, **Pred Fst** k), **GraphPred Succ** k)

Pred I $\equiv$ { i | i+1 $\in$ αI } = **Tl** I

Cond = **GraphCond 0**

GraphCond k $\equiv$ { <i, j>-k | <i, j> $\geq$ k $\wedge$ j $\in$ [α $\mathbf{Cond}_1$ ρ ε_i] } =
Cons(approx(Snd k, $\mathbf{Cond}_1$ **Fst** k), **GraphCond Succ** k)

$\mathbf{Cond}_1$ X = $\mathbf{GraphCond}_1$(X, **0**)

$\mathbf{GraphCond}_1$(X, k) $\equiv$ { <i, j>-k | <i, j> $\geq$ k $\wedge$ j $\in$ α $\mathbf{Cond}_2$(X, ρ ε_i) } =
Cons(approx(Snd k, $\mathbf{Cond}_2$(X, **Fst** k)), $\mathbf{GraphCond}_1$(X, **Succ** k))

$\mathbf{Cond}_2$(X, Y) $\equiv$ λZ. **Cond**(X, Y, Z) = $\mathbf{GraphCond}_2$(X, Y, **0**)

$\mathbf{GraphCond}_2$(X, Y, k) $\equiv$ { <i, j>-k | <i, j> $\geq$ k $\wedge$ j $\in$ α **Cond**(X, Y, ρ ε_i) } =
Cons(approx(Snd k, **Cond**(X, Y, **Fst** k)), $\mathbf{GraphCond}_2$(X, Y, **Succ** k))

Cond(I, Y, Z) $\equiv$ { i $\in$ αY | 0 $\in$ αI } $\cup$ {j $\in$ αY | $\exists$ w w+1 $\in$ αI } =
Cons([hd I **and hd** Y] **por [def Tl** I **and hd** Z], **Cond**(I, **Tl** Y, **Tl** Z))

K X $\equiv$ { <i, j> | j $\in$ αX } = **Pair**(Top, **Filter** X)

Filter I $\equiv$ { i | $\varepsilon_i \subseteq \alpha$I } = $\mathbf{Filter}_1$(I, **0**)

$\mathbf{Filter}_1$(I, k) $\equiv$ { i-k | i $\geq$ k $\wedge$ $\varepsilon_i \subseteq \alpha$X } =
Cons(approx(k, I), **Filter1**(I, **Succ** k))

S = **GraphS(0)**

GraphS k $\equiv$ $\{ \langle i, j\rangle\text{-}k \mid \langle i, j\rangle \geq k \wedge j \in [\alpha\ S1\ \rho\ \varepsilon_i] \}$ =
Cons(approx(Snd k, S1 **Fst** k), **GraphS Succ** k)

S_1 X $\equiv$ $\lambda Y.\ S_2(X, Y)$ = **GraphS$_1$**(X, 0)

GraphS$_1$(X, k) $\equiv$ $\{ \langle i, j\rangle\text{-}k \mid \langle i, j\rangle \geq k \wedge j \in \alpha\ S1(X, \rho\ \varepsilon_i) \}$ =
Cons(approx(Snd k, S_2(X, **Fst** k)), **GraphS$_1$**(X, **Succ** k))

$S_2(X, Y)$ = $\lambda Z.\ S_3(X, Y, Z)$ = **GraphS$_2$**(X, Y, 0)

GraphS$_2$(X, Y, k) $\equiv$ $\{ \langle i, j\rangle\text{-}k \mid \langle i, j\rangle \geq k \wedge j \in \alpha\ S3(X, Y, \rho\ \varepsilon_i) \}$ =
Cons(approx(Snd k, S_3(X, Y, **Fst** k)), **GraphS$_2$**(X, Y, **Succ** k))

$S_3(X, Y, Z)$ = **Apply(Apply**(X, Z), **Apply**(Y, Z))

Apply(F, X) $\equiv$ $\{ j \mid \exists i\ \langle i, j\rangle \in F \wedge \varepsilon_i \subseteq X \}$ =
Snd Apply$_1$(0, F, X)

Apply$_1$(F, X, k) $\equiv$ $\{ p\text{-}k \mid p \geq k \wedge p \in F \wedge \varepsilon_{\mathrm{Fst}\ p} \subseteq X \}$ =
Cons(test(k, X, F), **Apply$_1$**(F, X, **Succ** k))

test(p, X, F) $\equiv$ $p \in F \wedge \varepsilon_{\mathrm{Fst}\ p} \subseteq X$ =
Overlap(p, F) **and approx(Fst** p, X)

XEROX

The Semantics of Lazy (And Industrious) Evaluation Robert Cartwright and James Donahue

CSL-83-9