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

1. Introduction.

Since the publication of two influential papers on lazy evaluation in 1976 [Henderson and Morris, Friedman and Wise], the idea has gained widespread acceptance among language theoreticians -- particularly among the advocates of "functional programming" [Henderson80, Backus78]. 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, which 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. As a result, many simple theorems about ordinary data objects do not hold in the context of lazy evaluation. For example, although the function reverse∘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 $\perp$. Replacing conventional data constructors by their lazy counterparts radically alters the structure of the data domain. As a result, reasoning about programs defined over lazy spaces is a subtle endeavor. 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 [Scott81, Scott76] 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 [Friedman and Wise76, Henderson and Morris76]. 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 [Scott81, Scott76]) such as the $P\omega$ model for the untyped lambda calculus. Specifically, we show that $P\omega$ (with the standard primitive operations **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)

This research has been partially supported by NSF grants MCS-7805850 and MCS-8104209 and by Xerox Corporation.

in terms of the constants **true** and ⊥, 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 [Guttag78, Kamin80] 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 **Pω**, 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

(1) the principle of structural induction for finite objects;

(2) the existence of least upper bounds for directed sets; and

(3) the continuity of functions.

To demonstrate the deductive power of the system, we embed the higher-order logic LCF [Gordon77] in our system and derive a generalized induction rule (analogous to fixed point induction in LCF) for admissible predicates called lazy induction which 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 six sections. Section 2 provides a brief overview of Scott's theory of data domains [Scott81,Scott76]. 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 the process, we prove that algebraic specification is too weak to accomplish the task and that lazy spaces have the same rich "higher-order" structure as **Pω**. Finally, in Section 6, we present a simple first order theory for lazy data domains and demonstrate that it is as least as powerful as the corresponding theory formulated in the higher-order logic LCF. Section 7 assesses the intuitive significance of our results and speculates about promising directions for future research.

2. Background

In Scott's theory of data domains, a domain D is a set of abstract data objects partially ordered under an approximation relation ≤ that satisfies the following three properties:

1. Every directed set[1] has a least upper bound within the domain.

2. The domain has a minimum element ⊥ ("bottom"). In intuitive terms, ⊥ corresponds to the result of a non-terminating or erroneous computation.

3. The domain D has a countable subset $B = \{b_i \mid i \in N\}$ (where N denotes the set of natural numbers {0, 1, ...}), called the basis or the finite elements of D, such that:

   a. $\perp \in B$ and B is closed under the least upper bound operation on finite consistent[2] subsets.

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

   $$\forall x \in D \quad x = \mathbf{lub}\{y \in B \mid y \leq x\}.$$

   c. For every element $x \in B$, the set $\{y \in B \mid y \leq x\}$ is finite.

Note that the structure of D is completely determined by the structure of B; D is isomorphic to the set of filters[3] over B under the subset ordering. Any element of D that is not a basis element is called a limit point or total element of D.

2.1. Computability

The domain D corresponding to the basis set B is computable if and only if it satisfies the following additional constraints:

1. For every element $x \in D$, the subset of B that approximates x is recursively enumerable. In an implementation of D, x is represented by a concrete description (such as a Goedel number or lazy list) of this recursively enumerable set.

2. On the basis set B, the ternary relation

   $z = \mathbf{lub}\{x,y\}$

---

[1] A set S is directed iff for every finite subset $E \subseteq S$, S contains an upper bound for E.

[2] A subset $S \subseteq D$ is consistent under the partial ordering ≤ iff there exists an upper bound for S in D.

[3] A filter over B is a set F such that (i) $\forall x,y \in F$ $\mathbf{lub}\{x,y\} \in F$, and (ii) $\forall x \in F, y \in B$ $y \leq x \Rightarrow y \in F$.

and the binary relation

$\exists k\ k \in B\ k \leq x \wedge k \leq y$

are both decidable assuming that we represent basis elements b_i by their integer indices i.

The computable functions on the data domain D can be implemented solely in terms of the two decidable relations and ordinary computable (partial recursive) functions over the natural numbers. In this paper, we will confine our attention computable domains.

Assume that we are given the computable domains D_1, D_2 with corresponding bases B_1, B_2. A computable mapping $f \subseteq D_1 \rightarrow D_2$ is a recursively enumerable binary relation $f \subseteq B_1 \times B_2$ such that:

(1) $\perp_1\ f\ \perp_2$;

(2) $x\ f\ y \wedge x\ f\ y' \Rightarrow x\ f\ \mathbf{lub}\{y, y'\}$;

(3) $x\ f\ y \wedge x \leq x'$ and $y' \leq y \Rightarrow x'\ f\ y'$.

Every computable mapping $f \subseteq D_1 \rightarrow D_2$ determines a corresponding computable function $f: D_1 \rightarrow D_2$ defined by

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

The function f is computable in the sense that given an arbitrary element $d \in D_1$ (represented by a description of the recursively enumerable subset of B_1 approximating d), we can effectively generate (a description of) the recursively enumerable subset of B_2 approximating f(d).

There is a one-one correspondence between computable mappings (relations) over $B_1 \times B_2$ and computable functions over $D_1 \rightarrow D_2$. A particularly appealing property of Scott's theory of data domains is that the set of computable mappings (functions) between computable domains is a computable domain in its own right. We will discuss this issue in more detail below.

2.2. Sample Data Domains

Many common data domains such as the natural numbers and ordinary (industrious) lists are degenerate in the sense that they contain no limit points; in these domains the basis set is the entire domain. For example, let Nat be defined as the set

$\{\perp, 0, 1, 2, \ldots\}$

under the partial ordering $\leq_{Nat}$ defined by

$$x \leq_{Nat} y \Leftrightarrow x = y \vee x = \perp .$$

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

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

under the partial ordering $\leq_{Bool}$ defined by

$$x \leq_{Bool} y \Leftrightarrow x = y \vee x = \perp .$$

An example of a more interesting data domain is $\mathbf{P}\omega$, the power set of the natural numbers under the partial ordering ordering $\subseteq$ (set inclusion). The finite (basis) elements of $\mathbf{P}\omega$ are precisely the finite sets of natural numbers. $\mathbf{P}\omega$ obviously is not computable because the set of finite sets approximating an arbitrary element of $\mathbf{P}\omega$ is not necessarily recursively enumerable. On the other hand, the recursively enumerable elements of $\mathbf{P}\omega$ form a computable subdomain of $\mathbf{P}\omega$.

2.3. Simple Domain Constructions

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

Given computable domains D_1, D_2 with bases B_1, B_2, the Cartesian product $D_1 \times D_2$ is the computable domain generated by the basis set

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

under the partial ordering $\leq_{D_1 \times D_2}$ defined by

$$(x_1, y_1) \leq_{D_1 \times D_2} (x_2, y_2) \Leftrightarrow x_1 \leq_{D_1} x_2 \wedge y_1 \leq_{D_2} y_2 .$$

The bottom element of $D_1 \times D_2$ is $(\perp_{D_1}, \perp_{D_2})$.

The second fundamental domain constructor is the formation of the space of computable mappings. Given the computable domains D_1, D_2 with corresponding bases B_1, B_2, the domain of computable mappings $D_1 >> D_2$ is the domain determined by the basis set consisting of the finite computable mappings under the partial ordering generated by set inclusion, i.e. $f_1 \leq f_2 \Leftrightarrow f_1 \subseteq f_2$. The corresponding set of computable functions $f: D_1 \rightarrow D_2$ also forms a domain, but we will focus on computable mappings instead since they are more intuitive from a computational viewpoint.

2.4. Retractions on the Universal Domain

A fairly rich collection of domains can be constructed by starting with a few very simple primitive domains (such as N and B) and constructing more complex domains using the Cartesian product and computable mapping constructions. However, it is easy to devise domains such as infinite cartesian products of primitive domains that are beyond the scope of this simple scheme.

Scott has developed a much more comprehensive approach to the problem of domain construction based on the concept of a universal domain. A universal domain **U** is a computable data domain such that that every data domain D (as defined above) is isomorphic to a subdomain[4] of **U**. Moreover, if D is computable, both the projection function P_D mapping D into **U** and the inverse projection function are computable.

Since every domain D has an isomorphic image within the universal domain, Scott proposes defining data domains directly as subspaces of the universal domain. To simplify this endeavor, Scott suggests using computable retractions on the universal domain to identify subdomains. A retraction on **U** is a function **a**: **U** → **U** such that a∘**a** = a. Given that a is computable, the range of a (image of **U** under a_D) specifies a subdomain of **U**. Moreover, if D is a computable domain then there is a computable retraction a_D: **U** → **U** such that the range of a_D is the isomorphic image of D in **U**. Hence, there is a one-one correspondence between computable subdomains of **U** and computable retractions over **U**.

There are many different possible choices for the universal domain. For our purposes, the particular choice is unimportant. All we need is an arbitrary universal domain **U** determined by the basis $\mathbf{U}_B$ with the primitive rectractions R_{Bool}, R_X, and $R_{>>}$ identifying the subdomains Bool (Boolean truth values), **U**×**U**, and **U**>>**U**, and the following operations (which are definable in terms of the primitive operations provided by the universal domain):

true, **false** ∈ Bool
def: **U** → Bool
cond: Bool → (**U** → (**U** → **U**))
and: Bool → (Bool → Bool)
or: Bool → (Bool → Bool)
por: Bool → (Bool → Bool)
not: Bool → Bool
pair: **U** → (**U** → **U** × **U**))
$proj_1$: **U** × **U** → **U**
$proj_2$: **U** × **U** → **U**
apply: **U**>>**U** → (**U** → **U**)
S: (**U**>>**U**) → (**U**>>**U** → **U**>>**U**)
K: **U** → (**U**>>**U**)

that obey the following axioms:

def(⊥) = ⊥
x≠⊥ ⇒ def(x)=**true**
cond(**true**)(x)(y) = x
cond(**false**)(x)(y) = y
cond(⊥)(x)(y) = ⊥
and(x)(y) = cond(x)(y)(**false**)
or(x)(y) = cond(x)(**true**)(y)
x≠**true** ∧ y≠**true** ⇒ por(x,y)=or(x,y)
x=**true** ∨ y=**true** ⇒ por(x,y)=**true**
not(x) = cond(x)(**false**)(**true**)
$R_X(x)=x \Rightarrow pair(proj_1(x))(proj_2(x)) = x$
$proj_1(pair(x)(y)) = x$
$proj_2(pair(x)(y)) = y$
$R_{>>}(f)=f \Rightarrow$
  apply(f,x) = **lub**{y∈$\mathbf{U}_B$ | ∃ u∈$\mathbf{U}_B$ u≤x ∧ u f y}
apply(S(x))(y)(z) =
  apply(apply(x)(z))(apply(y)(z))
apply(K(x))(y) = x .

With the exception of **por**, S, and B, these operations are generalizations of familiar operations from lazy LISP (where car, cdr, and cons correspond to $proj_1$, $proj_2$, and pair). The declared domain for each operation is its intended domain of usage. Each operation is actually defined over the entire universal domain U; domain declarations are enforced by projecting argument values outside the declared domain onto the declared domain (using the retraction R_D) Note that since each operation f listed above is computable, there is a corresponding computable mapping map_f such that

$apply(map_f) = f$.

The S and K operations can be used to construct an arbitrary computable mapping, including the computable mapping map_Y corresponding to the least fixed point operator Y defined by

Y: **U**>>**U** → **U** = λu. (λx. apply(u)(apply(x)(x)))
  (λx. apply(u)(apply(x,x))) .

It is well known [Barendregt 77] that any closed term (no free variables) in the (untyped) λ calculus can be expressed as a composition of the operations S and K. As a notational convenience, we will use explicit λ-abstraction instead compositions of S and K, but on a formal level these abstractions are simply abbreviations for the corresponding compositions of S and K.

In the remainder of the paper, we will also use the standard infix abbreviations for Boolean operations:

if x **then** y **else** z = cond(x)(y)(z)
x **and** y = **and**(x)(y)
x **or** y = **or**(x)(y)
x **por** y = **por**(x)(y).

3. The Construction of Lazy Spaces

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

[4] A subset S of the domain D with basis B is a subdomain of D iff i) ⊥∈D, ii) S∩B is closed under the **lub** operation on finite consistent (in D) subsets, and iii) S is a domain under the ordering $\leq_D$ with the basis set S∩B.

implicitly determines whether the composite space corresponds to lazy or non-lazy computations.

Let D_1 and D_2 be arbitrary computable subdomains of our universal space $\mathbf{U}$ with corresponding retractions R_1 and R_2 mapping $\mathbf{U} \to \mathbf{U}$. Using the Cartesian pairing function pair: $\mathbf{U} \to \mathbf{U} \to \mathbf{U}\times\mathbf{U}$, we can form a surprisingly wide variety of simple composite domains using the following domain constructions.

1. Ordinary product. $D_1 \times D_2 = \{\langle x,y\rangle \mid x\in D_1, y\in D_2\}$. The corresponding primitive operations are:

$P_\times$: $D_1 \to (D_2 \to D_1\times D_2) = \lambda x.\ \lambda y.\ \text{pair}(x)(y)$
$\text{fst}_\times$: $D_1\times D_2 \to D_1 = \lambda z.\ \text{proj}_1(z)$
$\text{snd}_\times$: $D_1\times D_2 \to D_2 = \lambda z.\ \text{proj}_2(z)$
$R_\times$: $\mathbf{U} \to D_1\times D_2 =$
$\quad \lambda x.\ P_\times(R_1(\text{fst}_\times(x)))(R_2(\text{snd}_\times(x)))$

2. Coalesced product. $D_1 \circledast D_2 = \{\langle x,y\rangle \mid x\in D_1, y\in D_2, x\neq\perp, y\neq\perp\} \cup \{\perp\}$. The corresponding primitive operations are:

$P_\circledast$: $D_1 \to (D_2 \to D_1\circledast D_2 =$
$\quad \lambda x.\ \lambda y.$ **if** def(x) **and** def(y) **then** pair(x)(y)
$\qquad$ **else** $\perp$
$\text{fst}_\circledast$: $D_1\circledast D_2 \to D_1 = \lambda z.\ \text{proj}_1(z)$
$\text{snd}_\circledast$: $D_1\circledast D_2 \to D_2 = \lambda z.\ \text{proj}_2(z)$
$R_\circledast$: $\mathbf{U} \to D_1\circledast D_2 = \lambda x.\ P_\circledast(R_1(\text{fst}_\circledast(x)))(R_2(\text{snd}_\circledast(x)))$

3. Separated product. $D_1 \times_\perp D_2 = \{\langle$**true**$,\langle x,y\rangle\rangle \mid x\in D_1, y\in D_2\}$. The corresponding primitive operations are:

$P_{\times_\perp}$: $D_1 \to (D_2 \to D_1\times_\perp D_2 =$
$\quad \lambda x.\ \lambda y.\ \text{pair}($**true**$)(\text{pair}(x)(y))$
$\text{fst}_{\times_\perp}$: $D_1\times_\perp D_2 \to D_1 = \lambda z.\ \text{proj}_1(\text{proj}_2(z))$
$\text{snd}_{\times_\perp}$: $D_1\times_\perp D_2 \to D_2 = \lambda z.\ \text{proj}_2(\text{proj}_2(z))$
$R_{\times_\perp}$: $\mathbf{U} \to D_1\times_\perp D_2 =$
$\quad \lambda x.\ P_{\times_\perp}(R_1(\text{fst}_{\times_\perp}(x))))(R_2(\text{snd}_{\times_\perp}(x))))$

4. Coalesced sum. $D_1 \circledcirc D_2 = \{\langle$**true**$,x\rangle \mid x\in D_1, x\neq\perp\} \cup \{\langle$**false**$,y\rangle \mid y\in D_2, y\neq\perp\} \cup \{\perp\}$. The corresponding primitive operations are:

$\text{InL}_\circledcirc$: $D_1 \to D_1\circledcirc D_2 =$
$\quad \lambda x.$ **if** def(x) **then** pair(**true**)(x) **else** $\perp$
$\text{InR}_\circledcirc$: $D_2 \to D_1\circledcirc D_2 =$
$\quad \lambda x.$ **if** def(x) **then** pair(**false**)(x) **else** $\perp$
$\text{OutL}_\circledcirc$: $D_1\circledcirc D_2 \to D_1 = \lambda z.\ \text{proj}_2(z)$
$\text{OutR}_\circledcirc$: $D_1\circledcirc D_2 \to D_2 = \lambda z.\ \text{proj}_2(z)$
$\text{IsL}_\circledcirc$: $D_1\circledcirc D_2 \to \text{Bool} = \lambda z.\ \text{proj}_1(z)$
$\text{IsR}_\circledcirc$: $D_1\circledcirc D_2 \to \text{Bool} = \lambda z.$ **not** $\text{proj}_1(z)$
$R_\circledcirc$: $\mathbf{U} \to D_1\circledcirc D_2 =$
$\quad \lambda x.$ **if** $\text{IsL}_\circledcirc(x))$ **then** $\text{InL}_{(p_\perp}(R_1(\text{OutL}_\circledcirc(x))$
$\qquad$ **else** $\text{InR}_\circledcirc(R_2(\text{OutR}_\circledcirc(x))$

5. Separated sum. $D_1+D_2 = \{\langle$**true**$,x\rangle \mid x\in D_1\} \cup \{\langle$**false**$,y\rangle \mid y\in D_2\} \cup \{\perp\}$ The corresponding primitive operations are:

InL_+: $D_1 \to D_1+D_2 = \lambda x.$ pair(**true**)(x)
InR_+: $D_2 \to D_1+D_2 = \lambda x.$ pair(**false**)(y)
OutL_+: $D_1+D_2 \to D_1 = \lambda z.\ \text{proj}_2(z)$
OutR_+: $D_1+D_2 \to D_2 = \lambda z.\ \text{proj}_2(z)$
IsL_+: $D_1+D_2 \to \text{Bool} = \lambda z.\ \text{proj}_1(z)$
IsR_+: $D_1+D_2 \to \text{Bool} = \lambda z.$ **not** $\text{proj}_1(z)$
R_+: $\mathbf{U} \to D_1+D_2 =$
$\quad \lambda x.$ **if** IsL_+ **then** $\text{InL}_+(R_1(\text{OutL}_+(x)))$
$\qquad$ **else** $\text{InR}_+(R_2(\text{OutR}_+(x)))$

6. Lifted domain. $D_\perp = \{\langle$**true**$,x\rangle \mid x\in D\} \cup \{\perp\}$. Let R_D be the retraction corresponding to D. The primitive operations corresponding to $D_\perp$ are:

lift: $D \to D_\perp = \lambda x.$ pair(**true**)(x)
drop: $D_\perp \to D = \lambda z.\ \text{proj}_2(z)$
$R_\perp$: $\mathbf{U} \to \mathbf{U} = \lambda x.\ \text{lift}(R_1(\text{drop}(x)))$

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

1. A composite object (an ordered pair) containing an undefined component may be identified with the undefined object in the constructed domain. Coalesced products ($\circledast$) and sums ($\circledcirc$) obey this convention.

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

3. A composite object may always be distinguished from the bottom element of the constructed domain. In this case, the bottom element is outside the range of the constructor function corresponding to the composite domain. Separated products ($\times_\perp$), separated sums ($+$), and lifted domains ($_\perp$) 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" evaluation: 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). 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 $_\perp$ provides an explicit mechanism for constructing a domain of "suspended" or "unevaluated" elements corresponding to a given domain D. Note that the composition of the lifted domain construction with the ordinary product construction is identical to the separated product construction, i.e.

$$D_1 \times_\perp D_2 = D_{1_\perp} \times D_{2_\perp} .$$

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

$$D_1 + D_2 = D_{1_\perp} \oplus D_{2_\perp} .$$

Consequently, without loss of generality, we can confine our attention when it is convenient to the four domain construction mechanisms: $\times$ (ordinary product), $\otimes$ (coalesced product), $\oplus$ (coalesced sum), and $_\perp$ (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 domain (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 domains in detail. In particular, we are interested in determining and classifying the possible variations on the data industrious domain definition

(0) List = Atom $\oplus$ (List $\otimes$ List)

corresponding to the retraction

$$R_{List} = Y(\lambda f.\ \lambda u.\ \textbf{if } Is L_\otimes(u) \textbf{ then } InL_\otimes(R_{Atom}(OutL_\otimes(u)) \textbf{ else } InR_\otimes(P_\otimes(fst_\otimes(OutR_\otimes(u)))(snd_\otimes(OutR_\otimes(u))))))$$

where Atom is a given flat subdomain of $\mathbb{U}$. In the process, we would like to identify which value space corresponds to the implementation-oriented semantics presented in the literature [Henderson and Morris76, Friedman and Wise76]. Our investigation will demonstrate that that apparently innocuous variations in the definition of recursive data domains have profound semantic consequences. For the sake of simplicity, we will the domain Nat (the domain of natural numbers) as the Atom domain in all of our examples.

The obvious syntactic variations on industrious List domain defined above replace $\oplus$ by + or $\otimes$ by $\times$ or $\times_\perp$. They are:

(1) List = Nat + (List $\times$ List)
(2) List = Nat + (List $\otimes$ List),
(3) List = Nat $\oplus$ (List $\times$ List)
(4) List = Nat $\oplus$ (List $\times_\perp$ List)
(5) List = Nat + (List $\times_\perp$ List)

We will subsequently consider other possible variations that involve the explicit use of the $_\perp$ 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 expression

(0) $Y(\lambda u.\ InR_\otimes(P_\otimes(InL_\otimes(2))(u)))$

denotes the undefined element of the industrious domain (0) while the corresponding expressions

(1) $Y(\lambda u.\ InR_+(P_\times(InL_+(2))(u)))$
(2) $Y(\lambda u.\ InR_+(P_\otimes(InL_+(2))(u)))$
(3) $Y(\lambda u.\ InR_\otimes(P_\times(InL_\otimes(2))(u)))$
(4) $Y(\lambda u.\ InR_\otimes(P_{\times_\perp}(InL_\otimes(2))(u)))$
(5) $Y(\lambda u.\ InR_+(P_{\times_\perp}(InL_+(2))(u)))$

in the domains (1)-(5) respectively, all denote the infinite linear list of 2's -- a fact which can be easily checked from our definitions.

Within the class of domains that support infinite objects, we can analyze what kind of infinite and undefined objected that lists may contain. By using this form of analysis, we can determine that the first four spaces (1)-(4) have fundamentally different semantics and that space (5) is indistinguishable from space (1), unless we consider computations that return paired lists (objects in the space List$\times_\perp$List) instead of lists.

In space (1), lists can contain both undefined lists and undefined atoms. In space (2), lists can contain undefined atoms but not undefined lists. In space (3), lists can contain undefined lists but not undefined atoms. Space (4) is similar to space (3) except that it distinguishes the list consisting of two undefined lists from the undefined list. Space (5) is isomorphic to space (1), but the corresponding component spaces List$\times_\perp$List and List$\times$List are not.

By inspecting a few simple examples, we can easily prove that all five spaces are semantically

distinct (corresponding computations will yield different answers). In space (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 the undefined atom

as follows:

(a) $Y(\lambda u.\ InR_+(P_\times(u)(u)))$, and

(b) $Y(\lambda u.\ InR_+(P_\times(\perp)(InR_+(P_\times(InL_+(0))(u)))))$

(c) $InL_+(Y\lambda u.\ u)$.

However, in all the other spaces except (5) at least one of the corresponding lists does not exist. In space (2), expression (b) (with $P_\otimes$ substituted for $P_\times$) denotes the degenerate expression $InR_+(\perp)$; lists may not contain undefined lists. In space (3), both expression (a) (substituting $InR_\oplus$ for InR_+) and expression (c) (substituting $InL_\oplus$ for InL_+) denote the undefined list $\perp$; every defined list must contain a defined atom. In space (4), expression (c) (substituting $InL_\otimes$ for InL_+) denotes the undefined list $\perp$; lists not contain undefined atoms although undefined lists are permitted. Note that no two of the spaces (1), (2), (3), and (4) are isomorphic; the notion of finite element (list) is fundamentally different in each case.

The final lazy space presented above (5) is identical to (1) except that it contains a redundant level of delayed evaluation in paired lists. Hence, the meaning of the expression

$$OutR_+(InR_+(P_\times(\perp,\perp)))$$

in space (1) is $P_\times(\perp,\perp) = \perp$ while the corresponding computation in (5) (substituting $P_{\times_\perp}$ for $P_\times$) yields $P_{\times_\perp}(\perp,\perp) \neq \perp$. Obviously, the semantic difference between these two spaces is slight since it requires a computation over the corresponding paired list space to demonstrate any difference in behavior between the two spaces.

With the aid of the $\perp$ operator, we can define an even wider class of lazy list spaces. First, we can define pairing operators that are lazy in only one argument (unlike $P_\times$, $P_{\times_\perp}$). Second, we can delay the evaluation of atoms without delaying the evaluation of paired lists. 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 space in this last class of lazy domains is isomorphic to a space outside the class (assuming we ignore affiliated component spaces), we will not discuss it any further.

To facilitate classifying the extra spaces, we redefine the five lazy list spaces that we have already examined in terms in terms of the $\perp$ operator, ordinary cartesian product ($\times$), and the the coalesced sum and product ($\otimes$ and $\oplus$):

(1) $List = Nat_\perp \oplus (List \times List)_\perp$
(2) $List = Nat_\perp \oplus (List \otimes List)_\perp$
(3) $List = Nat \oplus (List \times List)$
(4) $List = Nat \oplus (List \times List)_\perp$
(5) $List = Nat_\perp \oplus (List \times List)_{\perp_\perp}$.

When the domain definitions are expressed in this simplified form, the close relationship between space (5) and space (1) is evident.

The remaining interesting variations on lazy lists are:

(6) $List = Nat \oplus (List_\perp \otimes List)$
(7) $List = Nat \oplus (List \otimes List_\perp)$
(8) $List = Nat_\perp \oplus (List_\perp \otimes List)$
(9) $List = Nat_\perp \oplus (List \otimes List_\perp)$
(10) $List = Nat_\perp \oplus (List \otimes List)$.

Variations (6), (7), (8), (9) all delay the evaluation of only one argument of a paired list. As a result, spaces (6) and (8) allow infinitely deep lists but not infinitely long ones while spaces (7) and (9) do the opposite. Spaces (6) and (7) prohibit undefined atoms while spaces (8) and (9) accommodate them. Finally, lazy space (10) does not accommodate infinite lists or undefined sublists within lists, but atoms may be undefined.

At this point, the question arises: which denotational definition of lazy lists corresponds to the standard implementation-oriented definition given in the literature [Friedman and Wise76]? The answer is (4), where we interpret

(1) $cons(x, y)$ as $InR_\otimes(Lift(P_\times(x)(y)))$,
(2) $car(x)$ as $fst_\times(Drop(OutR_\otimes(x)))$,
(3) $cdr(x)$ as $snd_\times(Drop(OutR_\otimes(x)))$, and
(4) atomic constants α as $InL_\otimes(\alpha)$.

The situation is somewhat more complicated in the case of the semantics presented in [Henderson and Morris76]. Their semantic definition describes a space isomorphic to (1), but the syntax of their language prohibits the construction of undefined atoms. Hence, they could used space (4) instead without affecting the semantics of their lazy LISP dialect.

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 formally specify a lazy space like

$$List = Atom + List \times List$$

(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 (or alternatively the corresponding final algebra). 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 $d_0 \leq d_1 \leq d_2 \leq \ldots$ (where $\leq$ denotes the approximation relation introduced in Section 3) 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 S-expressions), 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) radically alters 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. In short, the fundamental difference between lazy and industrious domains is that lazy spaces contain limit points while industrious spaces do not.

Since lazy spaces include limit points, we can construct domain with a more interesting topological structure using lazy domain constructors (such as $\times$) than their industrious counterparts (such as $\otimes$). An important illustration of this phenomenon is the following observation: the lazy data domain

$$\text{trivseq} = (\text{triv} \times \text{trivseq})$$

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

$$\alpha(x) = \{i \mid x_i = \textbf{true}\}$$

where x_i denotes the ith element of $x = \langle x_0, x_1, \ldots, x_j, \ldots \rangle$. Furthermore, the primitive operations of P_ω

$$0 \in P_\omega$$
$$\text{succ}: P_\omega \to P_\omega$$
$$\text{pred}: P_\omega \to P_\omega$$
$$\text{cond}: P_\omega \to (P_\omega \to (P_\omega \to P_\omega))$$
$$K: P_\omega \to (P_\omega >> P_\omega)$$
$$S: P_\omega \to (P_\omega >> (P_\omega >> P_\omega))$$
$$\text{apply}: (P_\omega >> P_\omega) \times P_\omega \to P_\omega$$

are all definable by first order recursion equations (recursive definitions) expressed in terms of the constants **true** and $\perp$, and the binary functions **por** and **and** on **triv**, and the constructor and selector functions for **trivseq**: cons (P_x) $\text{hd}(\text{fst}_x)$, and $\text{tl}(\text{snd}_x)$. The definitions appear in the **Appendix**.

In addition, the computable mappings (elements of $P_\omega >> P_\omega$) map_{succ}, map_{pred}, map_{cond}, map_K, and map_S corresponding to the primitive functions succ, pred, cond, K, and S are all definable by ground terms in the same first order theory; we can simply use the same scheme for translating λ abstraction used in the definition of S and K in the **Appendix**. The details are left to the reader.

Since P_ω together with the operations $S, K \in P_\omega >> P_\omega$ and apply: $P_\omega >> P_\omega \times P_\omega \to P_\omega$ forms a model for the (untyped) lambda calculus (excluding η-reduction), the lazy space **trivseq** with the corresponding operations also constitutes a model for the untyped lambda calculus. Lazy spaces provide a simple mechanism for treating higher order objects (functions) exactly like ordinary ones. Moreover, any first-order characterization of lazy spaces will indirectly provide a first-order theory of the lambda calculus and higher order objects.

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 lazy spaces.

Proof: We will prove the theorem for the specific lazy space **trivseq**, but it is clear that **trivseq** can be implemented (using an inverse homomorphism) within any non-trivial lazy space.

The initial algebra corresponding to a recursively enumerable set of equations A is the set of equivalence classes of ground terms under the relation MustEqual, where MustEqual(x,y) is true iff x=y is derivable from A by first order deduction. Hence the equality relation on ground 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 relation CannotEqual where CannotEqual(x,y) is true iff $x \neq y$ is derivable from A ∪ {**true≠false**} by first order deduction. Note that if A has no final algebra, then 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.

Q.E.D.

Since lazy spaces are so similar in structure to $\mathbf{P}_\omega$, an obvious approach is to use a least fixed point logic similar to Edinburgh LCF that conveniently expresses the properties of $\mathbf{P}_\omega$. (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 a 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 Spaces

The chief obstacle to extending ordinary first-order structural induction theories to lazy spaces is that conventional structural induction is applicable only to sets without limit points, yet lazy spaces under the (proper) substructure ordering $\lhd$ include limit points. If we develop a candidate axiomatization containing

(1) implications between equations relating the primitive domain operations (constructors, selectors, characteristic functions, cond, computable equality);

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

(3) axioms describing the substructure ordering $\sqsubset$ (a predicate), the approximation ordering $\leq$, and the characteristic predicate Isfin for finite objects (members of the domain's basis); and

(4) the structural induction scheme

$$\forall x\ [\forall x'\ (x' \sqsubset x \Rightarrow \theta(x')) \Rightarrow \theta(x)] \Rightarrow \forall z\ \theta(z);$$

then the specified domain contains only the finite objects of the lazy space.[5] The structural induction scheme (4) has the effect of banning the infinite objects from the domain. In fact, we can prove that

$$\forall x\ \mathrm{IsFin}(x)$$

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

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

is contradictory, because we can prove by structural induction that

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

including $x = \perp$!

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

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

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.

6.1. 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 and

(6) every term t(x) is continuous in the variable x.

---

[5] Non-standard models may contain "infinite objects", but their behavior does not resemble that of lazy data objects.

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

$$\forall x,y[\theta(x)\wedge\theta(y) \Rightarrow \exists z[\theta(z) \wedge x\leq t(z) \wedge y\leq t(z)]]$$

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

$$\forall x\ [(\theta(x) \Rightarrow t(x)\leq v) \wedge \forall z(\forall x\theta(x) \Rightarrow t(x)\leq z) \Rightarrow t(x)\leq v]$$

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

(5) (the existence of least upper bounds)

$$\textbf{Dir}\{t(u)|\theta(u)\} \Rightarrow \exists v\ [\textbf{lub}\{t(u)|\theta(u)\}(v)]$$

(6) (the continuity of functions)

$$\textbf{lub}\{t(u)|\theta(u)\}(v) \Leftrightarrow \exists v\ [\textbf{lub}\{x|\theta(x)\}(v) \wedge v=t(u)].$$

where $t(u)$ and $\theta(u)$ are an arbitrary term and formula containing no free variable other than u. This scheme asserts that the function $\lambda u.\ t(u)$ is continuous at the definable point **lub**$\{u|\theta(u)\}$ assuming it exists.

Although there are no blatant sources of incompleteness in this axiomatization [6] ((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 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 **Pω**. 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

$$\theta(\perp) \wedge \forall f[\theta(f)\Rightarrow\theta(\tau(f))]) \Rightarrow \theta(Y(\lambda f.\tau(f)))$$

where $\theta(f)$ is a formula that admits induction on f. It is applicable only to admissible formulas, where admissibility is a complex syntactic test that analyzes the types of terms within the formula (see [Gordon77] for a precise definition). The closest analog of structural induction in LCF is fixed point induction on the retraction characterizing the domain of interest.

After studying the two systems, we were surprised to discover that our system is at least as strong as LCF both in expressiveness and deductive power. We can systematically translate LCF statements into equivalent first order statements, by converting all lambda expressions into equivalent expressions formed using the standard S and K combinators, which are definable in our first-order system using the construction shown in the **Appendix.** Moreover, 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 and the continuity scheme (6). We call this first order analog of fixed point induction, lazy induction. 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.

Lazy induction is a particularly useful rule for reasoning about a lazy data domains when it is applied to the retraction characterizing the lazy space. In this case, the premises of the rule are identical to those of conventional structural induction. Hence, if a formula is admissible, conventional structural induction establishes the formula holds for all objects, not just finite ones!

6.2. A Sample Program Proof.

Consider the recursive definition

```
app(x,y) = if atom x then y
           else cons(car x,app(cdr x,y))
```

over the data domain List × List where

List = Atom + List×List

[6] For a non-trivial lazy space (e.g. **trivseq**) the axiomatization is obviously not complete by Godel's first incompleteness theorem.

$car = fst_x \circ OutR_+$
$cdr = snd_x \circ OutR_+$
$cons = InR_+ \circ P_x$
$atom = IsL_+$.

The following formula

$\forall x,y,z{:}List\ app(x,app(y,z)) = app(app(x,y),z).$

is obviously true on the domain of finite objects. 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.

7. 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 $P\omega$ 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 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. 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.

8. References

[ADJ76]
Goguen, J., J. Thatcher and E. Wagner. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. IBM Research Report RC-6478, Yorktown Heights, 1976.

[ADJ77]
Goguen, J., J. Thatcher, E. Wagner and J. Wright. Initial Algebra Semantics and Continuous Algebras. JACM 24(1977), pp. 68-95.

[Backus78]
Backus, J. Can Programming be Liberated from the vonNeumann Style? A Functional Style and its Algebra of Programs. CACM 21(1978), pp. 613-641.

[Barendregt77]
Barendregt, H. The Type Free Lambda Calculus. Handbook of Mathematical Logic, J. Barwise, ed., North-Holland, Amsterdam, pp. 1091-1132.

[Boyer75]
Boyer, R.S., and Moore, J S.; "Proving Theorems About LISP Functions," JACM 22(1975), pp. 129-144.

[Boyer79]
Boyer, R.S., and Moore, JS. A Computational Logic, Academic Press, New York, 1979.

[Cartwright76]
Cartwright, R. User-Defined Data Types as an Aid to Verifying LISP Programs. Automata, Languages and Programming. Edinburgh Press, 1976.

[Cartwright80]
Cartwright, R. A Constructive Alternative to Axiomatic Data Type Definitions. Proceedings 1980 LISP Conference, Stanford, 1980.

[Enderton72]
Enderton, H.B. A Mathematical Introduction to Logic. Academic Press, New York, 1972.

[Friedman and Wise76]
Friedman, D. and D. Wise. CONS Should Not Evaluate Its Arguments. Automata, Languages and Programming, Edinburgh University Press, 1976, pp.257-284.

[Giles78]
Giles. An LCF Axiomatization of Lazy Lists. CSR-31-78, Computer Science Department, Edinburgh University.

[Gordon77]
Gordon, M., R. Milner and C. Wadsworth. Edinburgh LCF. CSR-11-77. Computer Science Department, Edinburgh University.

[Guttag78]
Guttag, J. and J. Horning. The Algebraic Specification of Abstract Data Types. Acta Informatica 10(1978), pp. 27-52.

[Henderson80]
Henderson, P. Functional Programming: Application and Implementation. Prentice-Hall, London, 1980.

[Henderson and Morris76]
Henderson, P and J. Morris, Jr. A Lazy Evaluator. Record Third Symposium on Principles of Programming Languages (1976), pp. 95-103.

[Kamin80]
Kamin, S. Final Data Type Specifications: A New Data Type Specification Method. Record Seventh Symposium on Principles of Programming Languages (1980), pp. 131-138.

[Scott76]
Scott, D. Data Types as Lattices. SIAM J. Computing 5(1976), pp. 522-587.

[Scott81]
Scott, D. Lectures on a Mathematical Theory of Computation. Technical Monograph PRG-19, Oxford University Computing Laboratory, Oxford.

[Stoy77]
Stoy, J. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, 1977.

9. Appendix: Embedding P_ω in trivseq

For the sake of clarity, all of the recursive definitions in this appendix obey the following syntactic conventions.

1. The names of **triv** operations (functions that return values of type **triv**) are capitalized; the names of **trivseq** operations (functions that return values of type **trivseq**) other than S and K are not.

2. Variables ranging over **trivseq** that are intended to denote arbitrary sets in P_ω are capitalized. Variables ranging over **trivseq** that are intended to denote individual natural numbers (singleton sets) are not. No variables range over **triv**.

3. Each **triv** operation is annotated with comment (a string of the form # ...) specifying the object in **triv** that the operation computes.

4. Each **trivseq** operation is annotated with a comment specifying the the abstract object in P_ω corresponding to the element of **trivseq** that the operation computes.

Recursive definitions for all the primitive operations of P_ω (**0**, pred, succ, cond, S, K, apply) appear below.

```
0 = cons(true,⊥)        # {0}
succ(X) = cons(⊥,X)     # {u+1 | u ∈ α(X)}
pred(X) = tl X          # {u | u+1 ∈ α(X)}

cond(X,Y,Z) =
  # {u ∈ α(Y)|0 ∈ α(X)} ∪ {v ∈ α(Y)|∃ w w+1 ∈ α(X)}
    cons([hd X and hd Y] por [Def tl X and hd Z],
         cond(X, tl Y, tl Z))

# let <u,v> = [(u+v)(u+v+1)]/2 + u
pair(U,V) =   # {<u,v> | ∃u ∈ α(U),v ∈ α(V)}
  plus(halve(times(plus(U,V),
                   plus(plus(U,V),succ(0))))),
       U)

fst(Z) = fst1(0,Z)   # {u | ∃v <u,v> ∈ α(Z)}
fst1(i,Z) =          # {u-i | ∃v <u,v> ∈ α(Z)}
  cons(Check2(i,0,Z),fst1(succ(i),Z))
Check2(i,j,Z) =      # ∃ k≥j <i,k> ∈ α(Z)
  Consist(pair(i,j),Z) por Check2(i,succ(j),Z)
Consist(X,Y) =       # ∃i i ∈ α(X) ∧ i ∈ α(Y)
  hd X and hd Y por Consist(tl X,tl Y)

snd(X) = snd1(0,Z)   # {v | ∃u <u,v> ∈ α(Z)}
snd1(j,Z) =          # {v-j | ∃u <u,v> ∈ α(Z)}
  cons(Check1(0,j,Z),snd1(succ(j),Z))
Check1(i,j,Z) =      # ∃ k≥i <k,j> ∈ α(Z)
  Consist(pair(i,j),Z) por Check1(succ(i),j,Z)

plus(X,Y) =          # {u+v | u ∈ α(X) ∧ v ∈ α(Y)}
  cons(hd X and hd Y,
       cons([hd tl X and hd Y] por
            [hd X and hd tl Y],
          plus(tl X, tl Y)))

times(X,Y) =         # {u*v | u ∈ α(X) ∧ v ∈ α(Y)}
  cons([Def X and hd Y] por [hd X and Def Y],
       plus(tl X, times(X, tl Y)))

Def(X) = hd x por Def tl X  # ∃u ∈ α(X)

top = cons(true,top)     # {i}
K(X) = pair(top,filter(X))  # {<u,v> | v ∈ α(X)}

# let e_i\*U denote the finite set in Pω
#   corresponding to code i
filter(X) = filter1(X,0)    # {i | e_i⊆α(X)}
filter1(X,i) =              # {j-i | e_j⊆α(X)}
  cons(Approx(i,X),filter1(X,succ(i)))

Approx(i,X) =               # e_i⊆α(X)
  hd i por
    [([odd i and hd X] por odd tl i) and
        Approx(halve(i),tl X)]

odd(X) = hd tl x por odd tl tl X  # ∃i 2i+1 ∈ α(X)
halve(X) =     # {i|2i ∈ α(X)} ∪ {j|2j+1∈α(X)}
  cons(hd X por hd tl X, halve tl tl X)

s2(X,Y,Z) = apply(apply(X,Z),apply(Y,Z))
s1(X,Y) = abstr2(X,Y,0)     # λZ. s2(X,Y,Z)
abstr2(X,Y,i) =
  # {<u,v>-i | v ∈ α(s2(X,Y,α^-1(e_u)))}
    cons(Approx(snd i,s2(X,Y,fst i)),
         abstr2(X,Y,succ(i)))
S(X) = abstr1(X,0)          # λY. s1(X,Y)
abstr1(X,i) = # {<u,v>-i | v ∈ α(s1(X,α^-1(e_u)))}
  cons(Approx(snd i,s1(X,fst i)),
       abstr1(X,succ(i)))

apply(F,X) =  # {v | ∃u <u,v>∈F ∧ e_u⊆X}
  snd(apply1(0,F,X))
apply1(i,F,X) =
  # {p | ∃p≥i ∧ p∈F ∧ e_fst(p)⊆X}
    (cons(Test(i,X,F),apply1(succ(i),F,X)))
Test(p,X,F) = # p∈F ∧ e_fst(p)⊆X}
  Consist(p,F) and Approx(fst(p),X)
```