Inductive Programming

Richard B. Kieburtz

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 93-001

January 1993

Inductive Programming*

Richard B. Kieburtz

Oregon Graduate Institute
of Science & Technology
19600 N.W. von Neumann Dr.
Beaverton, OR 97006 USA

Abstract

When a datatype is characterized categorically, attention is focused upon the homomorphisms of the algebras induced by the signature of the datatype. The inductive definition of the datatype leads to a natural recursion structure for functions that realize its homomorphisms. This structure is familiar for the homomorphisms of initial datatype algebras, but is also available for non-initial algebras. Such recursion schemes can be captured in combinators for any datatype, and form a basis for inductive programming. Programs structured in this way have natural and intuitive proof rules.

The dual concept, of coinductive types, also leads to useful recursive control combinators with proof rules that derive weakest preconditions for propositions about their results. Furthermore, lazy functional programs that use only coinductive combinators never engender space leaks. We suggest a means for incorporating both inductive and coinductive programming styles.

1 Introduction

Although the syntax of functional programs resembles that of an equational logic, all programmers rely upon a rather complex interpretation of the equational syntax to provide the control necessary for computation. Attempts to reconcile the superficial resemblance to equational logic with the operational imperative of a programming language generated heated debates during early phases of the Haskell language design and this question has presumably engaged the interest of others as well. The idea that the functions defined in a program are operators, axiomatically defined by systems of equations, suggests that computation is inherently algebraic. Let's explore further some consequences of that notion.

*this research was partially funded by NSF grant No. CCR-91011721

Under an algebraic interpretation, a functional program defines an abstract algebra intended to be realized with a carrier set selected for convenience of implementation. The carrier is an internal representation of sub-algebras in terms of the available data algebras of an underlying computer. In a denotational semantics, the *storable values* correspond to the carrier. This algebraic view of computation is also supported to some extent at the level of computer hardware, which implements certain subalgebras such as fixed-radix integer arithmetic and floating-point arithmetic.

To obtain a finite presentation of operator definitions, most functional languages allow recursion. Unfortunately, unconstrained recursion schemes are not always well founded, and the operations so defined may only be partial. In a denotational semantics, sets of storable values are extended with bottom elements to represent the denotations of expressions whose computations diverge, but the bottom elements are not themselves storable values. Thus the algebraic nature of computations is sacrificed. What remains is partial algebras.

With partial algebras, the equational syntax of operator definitions must not be naively interpreted. Substitutivity is only valid if every term actually denotes a storable value. The need to qualify the interpretation of the equational syntax weakens, if not destroys, the relation between the programming notation and an equational logic. It requires that order-of-evaluation must be specified to obtain an unambiguous interpretation of the programming language. It means that program equivalence judgements must generally be qualified by termination judgements, and these are tedious to prove at best; undecidable at worst.

At one time, some researchers (myself included) thought that lazy functional languages might solve this problem. By restricting the evaluation of applicative expressions to use normal-order reduction, the substitutivity of equations remains valid. In fact, the equational syntax of a simple language can be interpreted as term-rewriting rules to give the language an operational meaning. The domain of storable values is enlarged to a domain of suspended computations that can denote an expression. However, this tactic, while it solves the problem for a pure lambda calculus, cannot avoid all problems when a programming language incorporates specific algebras. An algebraic programming language involves some operators that are necessarily strict, such as the conditional, strict in its boolean typed argument. The presence of algebraic operators in an otherwise non-strict programming language leads to other problems, two of which are order-of-evaluation in pattern matching, and space leaks.

In the present paper, we explore alternatives to the use of

unconstrained recursion for expressing operator definitions. Exploiting inductive (and dually, coinductive) control structures leads to a disciplined style of functional programming, appropriate to a wide variety of problems. Following this discipline encourages the verification of proof obligations in conjunction with program development, and guides the programmer in avoiding the dual hazards of nontermination and space leaks.

The concept of inductive programming is based upon the realization that datatypes determine algebras in addition to determining data structures. Control structures appropriate for most algorithms that we wish to formulate can be derived as homomorphisms (or cohomomorphisms) of the appropriate algebras. The role of datatype algebras is most readily seen when datatypes are modeled as objects of a category.

1.1 A categorical view of datatypes

The possibilities for higher-order functional programming structured by datatypes have been discovered by several authors [Hag87, CS92, Wad90, MFP91, HKS92], many of whom have called it "categorical programming" because it arises from viewing a datatype constructor as a functor from a bicartesian category to itself. In this view, the function map^T for a datatype constructor T, a generalization of the familiar Lisp function **mapcar**, is simply the morphism mapping part of T as a functor. Two properties of map^T, namely that

$$\begin{aligned} map^T\, id_\alpha &= id_{T(\alpha)} \\ (map^T\, f) \circ (map^T\, g) &= map^T\, (f \circ g) \end{aligned}$$

are immediate from the characterization of T as a functor. There are other functions that arise from the definitions of freely constructed datatypes. A generalization of the list reduction function

$$red^{List} : (\beta \times (\alpha \times \beta \to \beta)) \to List(\alpha) \to \beta,$$

to arbitrary datatype constructors has been used by Hagino [Hag87] as the basis for a formal characterization of an inductively defined datatype.

An inductive datatype constructor is expressed categorically as the limit of a sequence of functors. The sequence is generated by a bifunctor, $E(\_,\_)$, that consists of a *finite sum* of covariant bifunctors. The first argument of each bifunctor is reserved for a parameter type, α, and the second argument is a parameter that ranges over terms of the sequence. Expressions with the structure dictated by $E(\alpha, \beta)$ are injected into the datatype β by an injection morphism, **in**. A datatype constructor, L_E is then defined by

$$L_E(\alpha) = \lim \mathbf{in}\,1,\ \mathbf{in}\,E(\alpha, \mathbf{in}\,1),\ \mathbf{in}\,E(\alpha, \mathbf{in}\,E(\alpha, \mathbf{in}\,1)), \cdots$$

where 1 designates a terminal object in the category (this object is interpreted as a datatype with a single value, such as the type **unit** of SML). The inductive property of the limiting datatype constructor is equivalent to the property that the natural transformation $\mathbf{in}_{\alpha,\beta} : E(\alpha, \beta) \to \beta$ is an isomorphism. When **in** is an isomorphism, the limit is a fixed point of the functor $\mathbf{in}\,E(\alpha, \_)$.

A consequence of the inductive definition of $L_E(\alpha)$ is that its homomorphisms have a canonical recursion scheme. The dashed arrows in Figure 1 are uniquely determined by the other data of the diagram.

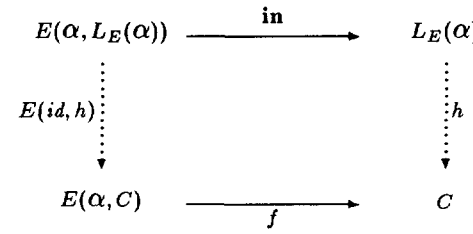


Figure 1—Commuting diagram for homomorphisms of an initial datatype

The dashed arrow h in Figure 1 is a homomorphism from the L_E-algebra $(L_E(\alpha), \mathbf{in}_\alpha)$ to another L_E-algebra (C, f). Uniqueness of the homomorphisms is the condition that $(L_E(\alpha), \mathbf{in}_\alpha)$ is an *initial* algebra, i.e. an initial object in a category whose objects are L_E-algebras and whose arrows are homomorphisms of those algebras. Initial datatypes are analogous to covariant types as they would be defined in a second-order logic such as System F [Gir71] or the Calculus of Constructions [CH86]. The induction principle for such types [Hue87] is expressed in the commuting diagram above.

More concretely, if the bifunctor $E(\_,\_)$ is an n-fold coproduct, then the injection morphism is a n-way coproduct injection whose components, $c_1, \ldots, c_n$ are *data constructors* of the datatype $L_E(\alpha)$. Thus $\mathbf{in} = \{c_1, \ldots, c_n\}$ is an isomorphism whose inverse is $\mathbf{case}^{L_E}$, which performs n-way discrimination on the data constructors. This is the characteristic of a *free* datatype. The unique derived homomorphism, h, is $red^T(f_1, \ldots, f_n)$, where the component functions are typed as $f_i : E_i(A, C) \to C$. That is, the component functions f_i are typed analogously to the types of the data constructors. The diagram of Figure 1 then specializes to:

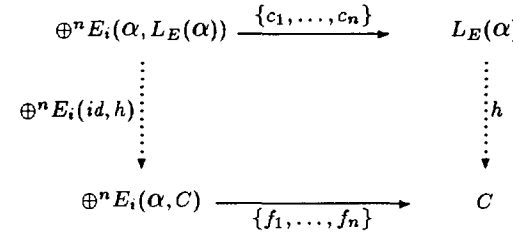


Figure 2—Homomorphisms of a freely constructed datatype

An example of an inductively defined datatype constructor is *List*. The component bifunctors defining *List* are

$$E_1(\alpha, \_) = 1 \qquad E_2(\alpha, \_) = \alpha \times \_$$

and its data constructors are

$$Nil : 1 \to List(\alpha) \qquad Cons : \alpha \times List(\alpha) \to List(\alpha)$$

For *List* types, the homomorphism diagram specializes further to Figure 3, in which it is easy to recognize that

$$h = red^{List}(f_1, f_2)$$

satisfies the recursion scheme for reduction over a list.

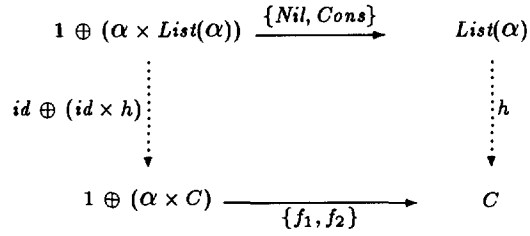


Figure 3—Homomorphisms of an initial *List*-algebra

Program derivation using homomorphisms of initial datatype algebras is a familiar technique [MFP91, Bir88, HKS92]. Some examples of functions definable as initial *List* homomorphisms are:

$$sum = red^{List}(0, (+))$$
$$length = red^{List}(0, \lambda(x, y).y + 1)$$
$$append(x, y) = red^{List}(y, Cons)\, x$$

Initial algebra homomorphisms have recursion schemes analogous to structural induction rules for logical inference of properties of typed data. But what of the homomorphisms of non-initial algebras? We shall see that these correspond to more general induction schemes.

2 Homomorphisms of non-initial algebras

Initial datatype homomorphisms are well-behaved and well known program structures, but not every algorithm can be expressed in terms of them. In particular, there are many examples of functions that, when realized as an initial datatype homomorphism, perform significantly worse than if realized by a different algorithm. For example, when the *predecessor* function is expressed as a primitive recursive algorithm, its evaluation takes time linear in the value of the argument, although a logarithmic time algorithm for this function is known. The desired algorithm can be expressed as a homomorphism of an appropriately chosen algebra. More precisely, given a datatype (constructor) T, we are interested in formulating homomorphisms of T-algebras that are not initial.

For simplicity of notation, we shall assume that T is a datatype, rather than a datatype constructor, in the development that follows. Correspondingly, we take the generating functor, E, to be a simple functor rather than a bifunctor. These conventions can easily be generalized to the case of type constructors by reintroducing the additional type parameter.

Let $g : E(A) \rightarrow A$ be the operator of a T-algebra, (A, g) and let $p : E(A) \rightarrow A$ be a left inverse for g, satisfying

$$p \circ g = id_{E(A)}$$

This may be the case, for instance, if (T, Σ^T) is the free term algebra generated by a signature Σ^T, but we are interested in a non-free algebra, (A, Σ^T). If there is an effectively computable projection, p, then the T-algebra homomorphisms can be calculated as (least) fixed points of the functional

$$H = \lambda h.\, f \circ E(h) \circ p$$

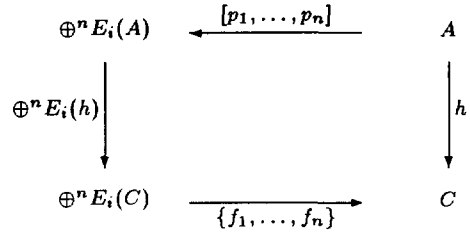
in analogy to an initial datatype. In practice, this occurs when the generating functor, E, is an n-fold sum-of-products and the T-algebra operator is factorized, $f = \{f_1, \ldots, f_n\}$.

We seek a representation of p as a function that effects an n-fold classification of its argument into mutually exclusive classes. We refer to such a function as a *classifier* for the T-algebra.

Lacking a complete set of constructor patterns for classification, suppose instead that p can be represented as the union of a set of n partial functions

$$p = \{p_i : A \rightarrow \iota_i(E_i(A))\}$$

whose domains partition the type A. Here, $\{\iota_j\}$, for $j \in 1..n$, is the set of injections into an n-fold coproduct. The following diagram then describes a T-algebra homomorphisms from (A, g) to (C, f):



The components of the classifier derive their typings from the types of the constructors of the initial datatype. For each index $i \in 1..n$,

$$\begin{array}{ll} \text{if} & c_i : (t_1 \times \ldots \times t_{m_i}) \rightarrow T(\alpha) \\ \text{then} & p_i : A \rightarrow (s_1 \times \ldots \times s_{m_i}) \\ \text{where } s_j = & \begin{cases} A & \text{if } t_j = T \\ t_{ij} & \text{otherwise} \end{cases} \end{array}$$

It is useful to introduce a homomorphism combinator, hom^T which given a T-algebra classifier and a T-algebra operator, constructs a homomorphism. The homomorphism h of the diagram above can then be represented as $hom^T([p_1, \ldots, p_n], \{f_1, \ldots, f_n\})$. The homomorphism combinator for a datatype T induced by a functor E has the typing

$$hom^T : (A \rightarrow E(A)) \times (E(C) \rightarrow C) \rightarrow A \rightarrow C$$

and satisfies the recursion equation

$$hom^T(p, f) = f \circ E(hom^T(p, f)) \circ p$$

Example 1: Consider the function

$$upto\_n : Nat \rightarrow List(Nat)$$

that, given an argument m, generates a list of the interval of natural numbers from m to n. This function is a *List*-algebra homomorphism. Its structure is similar to the function that copies a list, except that its classifier uses the predicate $(\leq n) : Nat \rightarrow Nat \oplus Nat$ to construct

$$[\text{if } (\leq n) \Rightarrow \iota_2 \circ \langle id, succ \rangle;\ \iota_1 \circ \Diamond] : Nat \rightarrow 1 \oplus Nat \times Nat$$

where $\Diamond_{Nat} : Nat \rightarrow 1$.

Following the prescription for building a *List* algebra homomorphism, *upto_n* is expressed in terms of a classifier and a *List*-algebra operator,

$$upto\_n = hom^{List}([\text{if } (\leq n) \Rightarrow \iota_2 \circ \langle id, succ \rangle;\ \iota_1 \circ \Diamond], (Nil, Cons))$$

When elaborated as a three-stage composition, this becomes

$$\begin{aligned} upto\_n = \quad & \{Nil,\ Cons\} \circ \\ & (id_1 \oplus (id_{Nat},\ upto\_n)) \circ \\ & [\mathbf{if}\,(\leq n) \Rightarrow \iota_2 \circ \langle id,\ succ \rangle;\ \iota_1 \circ \Diamond] \end{aligned}$$

Applying the laws for distribution over the conditional and for reduction of case selections, we obtain a form more familiar as a recursive program,

$$\begin{aligned} upto\_n = [\mathbf{if} \quad & (\leq n) \Rightarrow \\ & Cons \circ \langle id,\ upto\_n \circ succ \rangle; \\ & Nil] \end{aligned}$$

□

This is an example of what is called an *anamorphism* by [MFP91], that is, a homomorphism of a non-initial T-algebra in which the codomain, rather than the domain, is the type T. However, from our perspective, the fact that the codomain is T is no more than coincidence. The important notion is that the function has the structure of a homomorphism of a particular T-algebra whose carrier is not T, but in the example above, happens to be *Nat*. Knowledge of the structure of the homomorphism enables us to construct the function.

More generally, the classifier of a carrier type A need not effect a partition of A, but may only partition a subobject. Since there is no effective way, in general, to characterize the subobject, we instead extend the codomain of the classifier to indicate whether the classification succeeds or fails. A conditional classifier can be expressed with a codomain $M(E(A))$, where

$$M(\alpha) = \alpha \oplus \mathbf{1}$$

This corresponds to the *Maybe* type constructor introduced by Spivey [Spi90] to model exceptions in programs through the use of an inductive datatype[1]. Its data constructors are:

$$\begin{aligned} Just &: \ \alpha \to M(\alpha) \\ Nothing &: \ 1 \to M(\alpha) \end{aligned}$$

A classifier that induces an n-fold partition can be expressed as a sequential composition of classification trials,

$$p = [p_1; \ldots; p_n] : A \to M(\oplus^n E_i(\_))$$

in which each component, p_i, is typed as $p_i : A \to M(E_i(\_))$.

Example 2: Factors of a power of two in a positive integer. We seek to define a function

$$factors\text{-}of\text{-}2 : Nat \to M(List(Nat))$$

that when applied to a positive integer, will produce a list of its factors by powers of two, in descending order. If its argument is zero or is negative, the function will fail, yielding *Nothing*.

This function can be defined in terms of a list homomorphism using a classifier whose domain is the positive integers. The classifier must determine whether or not the number given as its argument is divisible by two, and if it is, perform the division, creating two copies of the quotient. One copy is an element of the result and the other is the value on which to continue the recursion.

$$\begin{aligned} & div2 \ : \ Nat \to \mathbf{1} \oplus (Nat \times Nat) \\ & div2 = [\mathbf{if}\ eq0 \circ (\mathbf{mod}\ 2) \Rightarrow \iota_2 \circ \langle (\mathbf{div}\ 2),\ (\mathbf{div}\ 2) \rangle;\ \iota_1 \circ \Diamond] \end{aligned}$$

Then *factors_of_2* can be expressed as a list homomorphism:

$$\begin{aligned} factors\_of\_2 = [\mathbf{if}(> 0) \Rightarrow \quad & Just \circ hom^{List}(div2,\ (Nil,\ Cons)); \\ & Nothing \circ \Diamond] \end{aligned}$$

□

It is interesting to compare the categorically-inspired definitions of sum-of-products types generated by a set of n component functors, $E_i(\_)$, with the analogous types defineable in second-order lambda calculus. The analogous type definition is

$$T \ = \ \Lambda C.\,(E_1(C) \to C) \to \cdots \to (E_n(C) \to C) \to C$$

in which the product constructors in E_i have also been replaced by $\to$. The induction principle for the type is implicit in its definition, reading $\to$ as an implication symbol according to the propositions-as-types analogy. The analogy has previously been noticed by Wraith, who derives expressions in the polymorphically typed lambda-calculus to correspond to Hagino's datatypes [Wra89]. Extending the analogy to non-initial datatype schemes, the classifiers, p_i, are introduced as effective tests of the n hypotheses of the induction principle for the type.

2.1 Non-initial datatypes lead to conditionally terminating recursion schemes

A program constructed with the T-algebra homomorphism combinator from a suitably typed classifier and operator may still fail to be a homomorphism because its computations diverge. To prove its uniform termination, an additional condition is required, namely, a demonstration that its recursion scheme is well-founded.

Example 2′: Returning to Example 2 of the previous section, a termination condition for a list homomorphism is

$$\forall x \in \{x' : Nat \mid x' > 0\}.\, p\, x = Just(y, x') \Rightarrow x' \prec x$$

in which $\prec$ designates a well-ordering relation over *Nat*.

For the list homomorphism used in Example 2, the guard in the definition of *factors_of_2* ensures that the argument is positive, so the condition becomes

$$\forall x > 0.\, x \bmod 2 = 0 \Rightarrow x \operatorname{\mathbf{div}} 2 \prec x$$

It is obvious that choosing $\prec$ to be a subordering of the total ordering on integers yields a well-ordering, as the domain is restricted to positive integers.

□

The verification condition for uniform termination can be stated formally, for the homomorphism scheme of an arbitrary datatype. Suppose the data constructors of an inductive type, T, have typings

$$c_i \ : \ t_{i1} \times \ldots \times t_{im_i} \to T$$

Then the termination verification condition for a homomorphism scheme $hom^T([p_1, \ldots, p_n], (f_1, \ldots, f_n)) \ : \ A \to C$ is

[1] The same type exists under a different name in Standard ML, where the type constructor is called option.

that there exists a well-ordering of A for which

$$x : A \Rightarrow \begin{array}{l} p_1\, x = \mathit{Just}(x_1, \ldots, x_{m_1}) \wedge \\ \quad \forall j \in \{1..m_1 \mid t_{1j} = T\}.\ x_j \prec x \\ \bigvee \cdots \\ \bigvee\ p_n\, x = \mathit{Just}(x_1, \ldots, x_{m_n}) \wedge \\ \quad \forall j \in \{1..m_n \mid t_{nj} = T\}.\ x_j \prec x \end{array}$$

Recalling that the classifier is $p = [p_1, \ldots, p_n]$, we see that the termination condition given above is equivalent to requiring that the retract $\oplus^n E_i(A) < A$ via (g, p) is compatible with a well-founded ordering on A.

Given evidence of termination, functional properties of T-algebra homomorphisms can be verified by inductive proof. The proof rule for a well-founded induction is

$$x : A \vdash \frac{\begin{array}{l} p_1\, x = \mathit{Just}(x_1, \ldots, x_{m_1}) \wedge \\ \quad (\forall j \in \{1..m_1 \mid t_{1j} = T\}.\ P(x_j)) \Rightarrow P(x) \\ \bigvee \cdots \\ \bigvee\ p_n\, x = \mathit{Just}(x_1, \ldots, x_{m_n}) \wedge \\ \quad (\forall j \in \{1..m_n \mid t_{nj} = T\}.\ P(x_j)) \Rightarrow P(x) \end{array}}{P(x)}$$

When such a rule is invoked without evidence of algorithmic termination, it is said to be a rule for a logic of partial correctness. In the present framework, partial correctness means that the proof rule is not known to be inductive. A property proved by non-inductive reasoning may be unsatisfiable.

Example 2″: We illustrate the use of inductive proof by returning again to Example 2. Let $Q(n, y, z)$ be the proposition

"y equals the n^{th} element of the list z"

As a three-place predicate, Q is formally defined by:

$$\begin{array}{l} Q(0, y, z) \equiv \mathit{ff} \\ Q(n, y, \mathit{Nil}) \equiv \mathit{ff} \\ Q(1, y, \mathit{Cons}(y, z)) \equiv \mathit{tt} \\ Q(n+1, y, \mathit{Cons}(x, z)) \equiv Q(n, y, z) \end{array}$$

Then define a property of *factors_of_2* in terms of a predicate over its integer domain,

$$P(x) \equiv \forall y\!:\!\mathit{Int}.\, \forall n\!:\!\mathit{Nat}.\, Q(n, y, \mathit{factors\_of\_2}\ x) \Rightarrow x = y * 2^n$$

The classifier used in the definition of *factors_of_2* provides two clauses that must be verified as hypotheses of the inductive proof rule for *List*:

(C1) $x \bmod 2 \neq 0 \wedge P(x)$

(C2) $x \bmod 2 = 0 \wedge (P(x \operatorname{div} 2) \Rightarrow P(x))$

Elaborating $P(x)$ and making use of the properties of $\mathit{hom}^{\mathit{List}}$ we find that (C1) is equivalent to:

$$\begin{array}{ll} & x \bmod 2 \neq 0 \wedge (Q(n, y, \mathit{Nil}) \Rightarrow x = y * 2^n) \\ \equiv & x \bmod 2 \neq 0 \wedge (\mathit{ff} \Rightarrow x = y * 2^n) \\ \equiv & x \bmod 2 \neq 0 \wedge \mathit{tt} \\ \equiv & x \bmod 2 \neq 0 \end{array}$$

For (C2), there is a proof by list induction:

$$\begin{array}{ll} & x \bmod 2 = 0 \wedge (P(x \operatorname{div} 2) \Rightarrow P(x)) \\ \equiv_{\{\mathit{defn\ of\ } P\}} & x \bmod 2 = 0 \wedge ((Q(n, y, \mathit{factors\_of\_2}\,(x \operatorname{div} 2)) \Rightarrow \\ & \qquad (x \operatorname{div} 2) = y * 2^n) \Rightarrow P(x)) \\ \equiv_{\{\mathit{defn\ of\ } Q\}} & x \bmod 2 = 0 \wedge ((Q(n+1, y, \mathit{Cons}(x \operatorname{div} 2, \\ & \qquad \mathit{factors\_of\_2}\,(x \operatorname{div} 2))) \Rightarrow \\ & \qquad (x \operatorname{div} 2) = y * 2^n) \Rightarrow P(x)) \\ \equiv_{\{\mathit{homomorphism}\}} & x \bmod 2 = 0 \wedge ((Q(n+1, y, \mathit{factors\_of\_2}\,((x \operatorname{div} 2) * 2)) \Rightarrow \\ & \qquad (x \operatorname{div} 2) * 2 = y * 2^n * 2) \Rightarrow P(x)) \\ \equiv_{\{x \bmod 2 = 0 \Rightarrow (x \operatorname{div} 2) * 2 = x\}} & x \bmod 2 = 0 \wedge ((Q(n+1, y, \mathit{factors\_of\_2}\ x) \Rightarrow \\ & \qquad x = y * 2^{n+1}) \Rightarrow P(x)) \end{array}$$

But since $Q(0, y, \mathit{factors\_of\_2}\ x) \equiv \mathit{ff}$, we have that

$$(Q(0, y, \mathit{factors\_of\_2}\ x) \Rightarrow x = y * 2^0) \equiv \mathit{tt}$$

Thus the clause can be generalized by natural induction to yield $\forall n : \mathit{Nat}.\, Q(n, y, \mathit{factors\_of\_2}\ x) \Rightarrow x = y * 2^n$, giving a derivation from (C2) of

$$x \bmod 2 = 0 \wedge (P(x) \Rightarrow P(x)) \ \equiv\ x \bmod 2 = 0$$

By the inductive proof rule we conclude

$$\forall x > 0.\ P(x)$$

3 Coinductive types and final algebras

Although many problems are resolved by inductive programming, some important ones are not. A program that responds interactively to an unbounded stream of input stimuli, or which incrementally generates an unbounded stream of output does not fit the inductive paradigm. There is, however, a dual to the notion of inductive datatype definition, called *coinductive* types. Formally, the idea is easy to express. Rather than types that are sums we examine types that are products (of tensor sums). In place of the homomorphism diagrams, we reverse all the arrows and draw cohomomorphism diagrams.

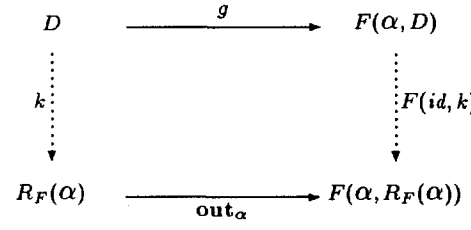


Figure 4—Final homomorphisms of a co-inductive type

When the natural transformation $\mathbf{out}_\alpha$ is an isomorphism, the co-algebra of the coinductive type is *final*, and the cohomomorphisms are uniquely determined by the projection operator, indicated by g in the diagram. The inverse transformation, $\mathbf{out}^{-1}$, can be considered as the datatype constructor of the coinductive type. It encloses all components of the product in its domain. We have less experience in

programming with coinductive types than with the familiar inductive datatypes. Some examples are:

$$\begin{array}{rll} R & = 1 \times R \simeq R & \textit{infinity} \\ \textbf{out} & = id \quad \textbf{out}^{-1} = id & \\ R(\alpha) & = \alpha \times R(\alpha) & \alpha\textit{-streams} \\ \textbf{out} & = \langle shd, stl \rangle \quad \textbf{out}^{-1} = \textit{str-cons} & \\ R(\alpha) & = \alpha \times R(\alpha) \times R(\alpha) & \textit{infinite binary trees} \\ \textbf{out} & = \langle elt, left, right \rangle \quad \textbf{out}^{-1} = \textit{node} & \end{array}$$

These types have computational interpretations that are most easily understood in terms of lazy functional languages. Recall that the categorical product requires a non-strict data constructor in a programming language in which computations may diverge.

The (final) co-homomorphisms of these types are *generators* for the types. Generators are the duals of the *reduce* homomorphisms of initial datatype algebras. Each is specified by the application of a combinator, gen^{R_F} to a projection function, g,

Example 3: For the coinductive datatype of streams, the *unit* of the stream monad satisfies the recursive equation

$$unit^{Stream}\, x = \textit{str-cons}(x,\ unit^{Stream}\, x)$$

The projection for this recursion scheme is the duplicator, Δ, thus we can write $unit^{Stream} = gen^{Stream}(id, id)$.

Example 4: To obtain a more interesting stream generator, choose a projection function that is not polymorphic, such as *succ*, which generates the stream co-homomorphism *from*, satisfying

$$\begin{array}{rl} \textit{from} & = gen^{Stream}(id, succ) \\ & = \textit{str-cons}\langle id, \textit{from} \circ succ \rangle \end{array}$$

Example 5: The generator that copies a stream has as its projection function

$$\langle sth, stl \rangle \ :\ Stream(\alpha) \rightarrow Stream(\alpha) \times Stream(\alpha)$$

Example 6: For the final datatype of infinite binary trees, the unit is

$$unit^{BinTree} = gen^{BinTree}(id, id, id)$$

3.1 Coinductive types that are not final

The elements of a coinductive datatype are infinitary structures that can only be partially analyzed by programs that consume finite resources. Of greater computational interest are the non-final cohomomorphisms of type co-algebras. Lifting this requirement means that the transformation **out** can be replaced by a *splitting* function, h, that has a left inverse, but is not an isomorphism. Let $f : F(\alpha, T(\alpha)) \rightarrow T(\alpha)$ be a function satisfying

$$f \circ h = id_{T(\alpha)}$$

This allows f to be a projection from an n-fold product. The projection defined by f may be conditional on the data.

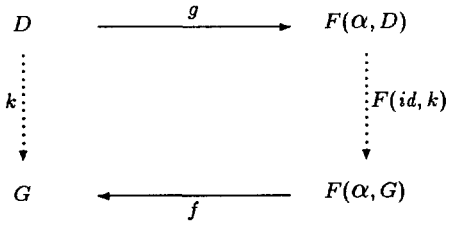


Figure 5—Non-final cohomomorphisms

Search algorithms can be expressed as co-homomorphisms of non-final algebras. In these examples, the recursion scheme of the search is induced by a *Stream* type, although the datatype $R(\alpha)$ is no longer a final type, but merely a carrier type for a co-algebra.

To express cohomomorphisms, we introduce another combinator,

$$cohom^{R_F} \ :\ (D \rightarrow F(\alpha, D)) \times (F(\alpha, G) \rightarrow G) \rightarrow D \rightarrow G$$

which takes as arguments a pair of a splitting function, g, and a projection function, f.

It satisfies the recursion equation

$$cohom^{R_F}(g, f) = f \circ F(id, cohom^{R_F}(g, f) \circ g$$

Example 7: Sequential search follows the recursion scheme of streams.

$$\begin{array}{l} F(\alpha, \beta) \ = \ \alpha \times \beta \\ G \ = \ \{x : \alpha \mid p\, x = tt\} \\ f \ = \ [\textbf{if}\ p \circ \pi_0 \Rightarrow \pi_0;\ \pi_1] \\ g \ = \ \langle g_0, g_1 \rangle \text{ where } g_0 : D \rightarrow \alpha,\ g_1 : D \rightarrow D \\ search \ = \ [\textbf{if}\ p \circ g_0 \Rightarrow g_0;\ search \circ g_1] \ :\ D \rightarrow G \\ search \ = \ cohom^{Stream}(g, f) \end{array}$$

in which π_0 and π_1 designate the first and second projections from a pair.

Example 8: Binary search follows the recursion scheme of binary trees.

$$\begin{array}{l} F(\alpha, \beta) \ = \ \alpha \times \beta \times \beta \\ G \ = \ \{x : \alpha \mid p_0\, x = tt\} \\ f \ = \ [\textbf{if}\ p_0 \circ \pi_0 \Rightarrow \pi_0;\ [\textbf{if}\ p_1 \circ \pi_0 \Rightarrow \pi_1;\ \pi_2]] \\ g \ = \ \langle g_0, g_1, g_2 \rangle \text{ where } g_0 : D \rightarrow \alpha,\ g_1, g_2 : D \rightarrow D \\ \textit{bin-search} \ = \ [\textbf{if}\ p_0 \circ g_0 \Rightarrow g_0; \\ \qquad\qquad [\textbf{if}\ p_1 \circ g_0 \Rightarrow \ \textit{bin-search} \circ g_1; \\ \qquad\qquad\qquad\qquad \textit{bin-search} \circ g_2]] \ :\ D \rightarrow G \\ \textit{bin-search} \ = \ cohom^{Stream}(g, f) \end{array}$$

Notice in these examples that the projection functions analyze data after they have been transformed by a component of g. Contrast this with the case of non-initial datatype homomorphisms in which the classification functions, $\{p_i\}$, are applied to "raw" data that may subsequently be mapped to a result.

Inductive recursion schemes accumulate (on a stack) the data they have classified until a base case is encountered. The test made by a classifier is applied prior to the completion of any recursive invocation. Coinductive recursion schemes can correspond to simple iteration, as a **while** loop, for instance. Coinductive schemes can produce results incrementally; inductive schemes cannot. These distinctions are important to keep in mind when one wishes to separate inductive from coinductive recursion. The clues provided by the types of domain and codomain are often misleading in the classification of recursion schemes that are neither initial nor final.

3.2 Coinductive proof rules

Suppose $F(\alpha, R_F(\alpha)) = t_1 \times \ldots \times t_n$, where each t_i is either α or R. This form of a product functor is not the most general one from which to form coinductive types, but it will serve to illustrate the formulation of proof rules. Let

$$I_\alpha = \{i \in 1..n \mid t_i = \alpha\} \qquad I_R = \{i \in 1..n \mid t_i = R(\alpha)\}$$

These are index sets that distinguish the components of the product that correspond to the parameter type or to the recursive type, respectively.

Further, let $g : D \to F(\alpha, D)$ be a splitting function and $f : F(\alpha, G) \to G$ be a projection function. Suppose f has the form of an n-fold conditional

$$f = [p_0 \Rightarrow f_0; \ldots [p_{n-1} \Rightarrow f_{n-1}] \ldots]$$

whose predicates are nonoverlapping. Each predicate has the typing

$$p_i : F_{I_\alpha}(\alpha, G) \to \mathit{Bool}$$

where by F_I we designate the projection of F on the index set I. This restriction is imposed so that the conditional projection, f, can be calculated from data produced by the splitting function, without further recursion. Define a boundedness predicate, B, for $\mathit{cohom}^{R_F}(g, f)$ as follows.

$$(\forall i.\, (p_i\, x = tt) \Rightarrow B(g_i\, x)) \Rightarrow B(x)$$

This is a transfinite induction rule. If B is compatible with a well-founded ordering $(D, \prec)$, such that $\forall x \in D.\, x \prec g_i\, x$, then for all $x \in D$, $\mathit{cohom}^{R_F}(g, f)\, x$ consumes only bounded resources.

The idea behind this rule is that the components of g are generalized predecessors. The subobject of D for which no recursion occurs under an application of $\mathit{cohom}^{R_F}(g, f)$ constitutes a basis. Termination is an important property to determine of a cohomomorphism because unlike a generator, its codomain is not necessarily of a coinductive type. It cannot be assumed that a value produced by applying a cohomomorphism will not be totally analyzed by other functions in a program.

We could also give a transfinite induction rule for more general properties of coinductive functions. Instead, we offer a rule that is easier to use. This rule is compatible with natural induction but is strictly less powerful than transfinite induction as a rule of logic.

$$\{x{:}D \mid B(x)\} \vdash \frac{\begin{array}{c}\bigwedge_{i \in I_\alpha} (p_i\, x = tt) \Rightarrow P(x) \Rightarrow Q(f_i\, x) \\ \bigwedge_{i \in I_R} (p_i\, x = tt) \Rightarrow P(x) \Rightarrow P(g_i\, x)\end{array}}{P(x) \Rightarrow Q(\mathit{cohom}^{R_F}(g, f)\, x)}$$

3.3 Space leaks

Space leaks occur when applications of projection functions (including conditional projections) are suspended. Since a lazy functional language specifies that suspended application is the default, with exception made only for applications of explicitly strict operators, space leaks are difficult to avoid. However, *coinductive programming does not engender space leaks.* A cohomomorphism combinator applies projections directly to tuples, rather than to arbitrary expressions, and does not need to be suspended. Only individual components of a tuple may require suspension, as when a component represents the recursive application of a cohomomorphism.

Contrast this situation with that of an inductive program written in a lazy programming language. There, the form of a homomorphism has an outermost operator that may as readily be a datatype constructor as a strict function. Suspending applications of datatype constructors often introduces space leaks. But suspending an application of a T-algebra operator is also unnecessary, as the operator never controls recursion. Recursion control is effected by the classifier of an inductive homomorphism. Our conclusion is that lazy languages are ill-suited for inductive programming. The use of an inductive programming style with lazy languages actually promotes leaky programs. On the other hand, inductive algorithms are very natural for a wide variety of problems.

3.4 Coinductive programming with explicit suspensions

If programmers wish to mix inductive and coinductive programming styles, how should they resolve the conflicting goals of preventing nontermination and space leaks? Since space leaks appear to be more subtle than nontermination, one obvious suggestion is to use a strict programming language and to create explicit suspensions for those applications that involve coinduction. There are combinators that will help to structure programs in this way.

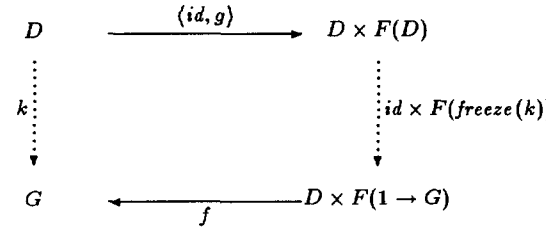


Figure 6—Coinduction with explicit suspension

Figure 6 relates to Figure 5 in the following way. The components of the splitting function that were typed as $g_i : D \to \alpha$ have been removed, leaving g. Instead, a copy of the argument (of type D) is propagated untransformed for analysis by the projection function, f. Recursive applications of the cohomomorphism under the product functor, F, are explicitly suspended with *freeze*. This allows the use of a strict pair constructor. The projection function f now employs predicates that act on the untransformed argument of type D, and most significantly, whenever a projection selects a component that is a suspended computation, it must be unfrozen explicitly.

A cohomomorphism combinator for a type

$$R(\alpha) \simeq \mathbf{out}_{-1}(t_1 \times \ldots \times t_n)$$

is constructed as follows. As before, let I_α designate the set $\{i \mid t_i = \alpha\}$ and I_R designate the set $\{i \mid t_i = R\alpha\}$. The combinator

$$\mathit{cohom}'(g, f) : D \to G$$

is constructed with a splitting function

$$g : D \to D^{I_R}$$

and projection function

$$f : D \times G^{I_R} \to G$$

that consists of a an $|I_R|+1$-fold conditional,

$$f = [p_0 \circ \pi_0 \Rightarrow f_0 \circ \pi_0; \ldots [p_{n-2} \circ \pi_0 \Rightarrow f_{n-2} \circ \pi_1;\ f_{n-1} \circ \pi_1] \ldots]$$

where $f_0 : D^{I_\alpha} \to G$ and for $i > 0$, $f_i : (1 \to G)^{I_R} \to G$.

For example, the cohomomorphism for *Stream* types is

$$\begin{aligned} &\text{cohom}'^{\,Stream} : (D \to D) \times (D \times (1 \to G) \to G) \to D \to G \\ &\text{cohom}'^{\,Stream}(g, f)\, x = f\,(x,\ \lambda().\, \text{cohom}'^{\,Stream}(g, f)\,(g\, x)) \end{aligned}$$

and that for *Bin-tree* types is

$$\begin{aligned} &\text{cohom}'^{\,Bin-tree} : \\ &\quad (D \to D) \times (D \times (1 \to G) \times (1 \to G) \to G) \to D \to G \\ &\text{cohom}'^{\,Bin-tree}(g_1, g_2, f)\, x = \\ &\qquad f\,(x,\ \ \lambda().\, \text{cohom}'^{\,Bin-tree}(g_1, g_2, f)\,(g_1\, x), \\ &\qquad\qquad \lambda().\, \text{cohom}'^{\,Bin-tree}(g_1, g_2, f)\,(g_2\, x)) \end{aligned}$$

The most familiar example of stream cohomorphism is an iteration function:

$$\begin{aligned} &\mathit{while} : ((D \to \mathit{Bool}) \times (D \to D)) \to D \to D \\ &\mathit{while}(p, r) = \mathit{cohom}'^{\,Stream}(r,\ \lambda(x, f).\text{if } p\, x \text{ then } f\,() \text{ else } x) \end{aligned}$$

Its proof rule is

$$\{x : D \mid B(x)\} \vdash \frac{\bigwedge \begin{array}{l} (p\, x = \mathit{ff}) \Rightarrow P(x) \Rightarrow Q(x) \\ (p\, x = \mathit{tt}) \Rightarrow P(x) \Rightarrow P(r\, x) \end{array}}{P(x) \Rightarrow Q(\mathit{while}\,(p, r)\, x)}$$

As a larger example, we illustrate an algorithm to construct a stream of prime numbers using the sieve of Eratosthenes. A sieve is represented as a list of pairs, (p, y), where p is a prime number and y is some multiple of p. A sieve s is said to be *well maintained* for a number n if it satisfies the predicate

$$\mathit{Sieve}(n, s) \equiv \forall (p, y) \in s.\, P_n(p, y)$$

where

$$P_n(p, y) \equiv \exists q.\ (y = q * p) \wedge ((y - p) < n \vee y < n)$$

This property assures that for any pair (p, y), that y is no greater than $n + p - 1$. It allows a simple test to determine whether or not n is a multiple of p. Namely, compare n with y. If $n > y$ then increase y in increments of p until the result is greater than or equal to n. If the comparison shows equality, n is a multiple of p, otherwise it is not.

To increment y, let

$$\mathit{incr}\, n\, p\, y = \mathit{while}\,((<n), (+p))\, y$$

Using the proof rule for *while*, together with the algebra of addition and inequality, we can conclude:

$$P_n(p, y) \wedge y' = \mathit{incr}\, j\, p\, y \Rightarrow P(p, y') \wedge y \geq n$$

Next, we construct a function that, given a number n and a sieve well maintained for n, produces a pair of a new sieve and a number. The number in the result either is n, in case n is not a multiple of any number in the original sieve, or else it is a smaller number of which n is a multiple. The new sieve is well maintained for n and in addition, is assured to contain a number of which n is a multiple. The new sieve also contains pairs that agree in the first component with each of the pairs in the old sieve.

$$\begin{aligned} &\mathit{update\_sieve} : \mathit{Nat} \to \mathit{List}(\mathit{Nat} \times \mathit{Nat}) \to \mathit{Nat} \times \mathit{List}(\mathit{Nat} \times \mathit{Nat}) \\ &\mathit{update\_sieve}\, n = \\ &\quad \mathit{cohom}'^{\,Stream} \\ &\qquad (\mathit{tl},\ \lambda(s, f).\ \text{case } s \text{ of} \\ &\qquad\qquad [] \Rightarrow (n, [(n, n)]) \\ &\qquad\qquad \mid\ ((p, y) :: t) \Rightarrow \\ &\qquad\qquad\qquad \text{let } m = \mathit{incr}\, n\, p\, y \text{ in} \\ &\qquad\qquad\qquad \text{if } m = n \\ &\qquad\qquad\qquad\quad \text{then } (p,\ (p, m) :: t) \\ &\qquad\qquad\qquad\quad \text{else } \mathit{box}\,(p, m)\,(f\,()) \end{aligned}$$

where $\mathit{box}\, x\, (n, xs) = (n,\ (x :: xs))$

A conclusion we wish to draw from this definition is that the result of the function satisfies

$$\begin{aligned} Q_{n,s}(n', s') \equiv \quad & \forall (p, y) \in s.\, \exists (p', y') \in s'.\, p' = p \\ \wedge\ (\quad & (n' = n \wedge \mathit{Sieve}(n, s') \wedge s' \neq [] \wedge \\ & \text{let } ((p, y) :: t) = \mathit{reverse}\, s' \text{ in} \\ & \quad p = n \wedge (\forall (p', y') \in t.\, p' < n \wedge y' \neq n)) \\ \vee \quad & (n' < n \wedge (\exists q.\, n = q * n') \wedge \mathit{Sieve}(n, s'))) \end{aligned}$$

We must formulate a precondition $R_n(s)$ that will enable us to discharge the following proof obligations:

(a) $R_n([]) \Rightarrow Q_{n,s}(n, [(n, n)])$

(b) $R_n((p, y) :: t) \wedge \mathit{incr}\, n\, p\, y = n \Rightarrow Q_{n,s}(p,\ (p,\ \mathit{incr}\, n\, p\, y) :: t)$

(c) $R_n((p, y) :: t) \Rightarrow R_n(t)$

(d) $P_n((p, y) :: t)\ \wedge \mathit{incr}\, n\, p\, y \neq n$
$\qquad \wedge (n', s') = \mathit{update\_sieve}\, n\, t$
$\qquad \Rightarrow Q_{n,s}(n', s') \Rightarrow Q_{n,s}(n',\ (p,\ \mathit{incr}\, n\, p\, y) :: s')$

Since $Q_{n,s}(n, [(n, n)])$ is valid (from the definition of Q), (a) imposes no condition on R. To establish (b), we require that

$$R_n(s) \Rightarrow \mathit{Sieve}(n, s)$$

Requirement (d) further suggests the condition

$$R_n(s) \Rightarrow \forall (p, y) \in s.\, p < n$$

Condition (c) is obviously satisfied if we let

$$R_n \equiv \mathit{Sieve}(n, s) \wedge \forall (p, y) \in s.\, p < n$$

Proving these clauses allows us to use the proof rule for stream coinduction to conclude:

(L) $\qquad R_n(s) \Rightarrow Q_{n,s}(\mathit{update\_sieve}\, n\, s)$

It is necessary to point out that none of these assertions has been proved formally. Their complexity and technicality illustrates the need for a good, automated proof assistant.

The final coinductive computational step constructs a sieve for the next prime number from a candidate number and a sieve well conditioned for the candidate.

$$\begin{aligned} &\mathit{nextsieve} : \mathit{Nat} \to \mathit{List}(\mathit{Nat} \times \mathit{Nat}) \to \mathit{Nat} \times \mathit{List}(\mathit{Nat} \times \mathit{Nat}) \\ &\mathit{nextsieve} = \quad \mathit{cohom}'^{\,Stream} \\ &\qquad ((+2),\ \lambda(n, f).\, \lambda s. \\ &\qquad\qquad \text{let } (n', s') = \mathit{update\_sieve}\, n\, s \text{ in} \\ &\qquad\qquad \text{if } n' = n \text{ then } (n', s') \text{ else } f\,() \end{aligned}$$

A property that describes the intended result of this function is

$$R_n(s) \wedge (n', s') = \mathit{nextsieve}\, n\, s \Rightarrow S(n, s, (n', s'))$$

$$\begin{aligned}\text{where } S(n, s, (n', s')) \equiv\ & \mathit{Sieve}(n', s') \wedge \\ & \forall (p, y) \in s.\, \exists (p', y') \in s'.\, p' = p \wedge \\ & \text{let } ((p, y) :: t) = \mathit{reverse}\, s' \text{ in} \\ & \quad (p, y) = (n', n') \wedge \\ & \quad \forall (p', y') \in t.\, p' < n' \wedge y' \neq n'\end{aligned}$$

Given the precondition

$$O(n) = \mathit{odd}\, n \wedge n > 1$$

we generate the proof obligations

$$\begin{aligned}(P1)\quad & O(n) \Rightarrow\ (R_n(s) \wedge (n', s') = \mathit{update\_sieve}\, n\, s \wedge \\ & \qquad n' = n \Rightarrow S(n, s, (n', s')))\end{aligned}$$

$$(P2)\quad O(n) \Rightarrow O(n+2)$$

Using proposition (L) established for *update_sieve* as a lemma and the equality $n' = n$, $(P1)$ can be reduced to

$$O(n) \Rightarrow Q_{n,s}(n, s') \Rightarrow S(n, s, (n, s'))$$

which is directly verifiable from the definitions of the predicates $Q_{n,s}$ and S. Condition $(P2)$ is true of numbers. We conclude:

$$O(n) \Rightarrow R_n(s) \Rightarrow S(n, s, \mathit{nextsieve}\, n\, s)$$

The reader will notice that the above proposition falls short of asserting that the function calculates a sieve of prime numbers. All the computational aspects are accounted for, but the primeness of the result is not, and would indeed require additional argument. It is obvious that any increasing sequence of candidate numbers would do as well in establishing computational properties of the algorithm as the choice made here, to use all the odd numbers.

The program to calculate a steam of primes is completed by defining

$$\begin{aligned}&\mathit{nextprime} = \langle \pi_0,\ ((+2) \times \mathit{id}) \circ \mathit{eval} \circ (\mathit{nextsieve} \times \mathit{id}) \rangle \\ &\text{where}\quad \mathit{eval} : (\alpha \to \beta) \times \alpha \to \beta \\ &\qquad\ \mathit{eval}(f, x) = f\, x\end{aligned}$$

$$\mathit{primes} = \mathit{str\text{-}cons'}(2,\ \lambda().\, \mathit{gen}^{\mathit{Stream}}\, \mathit{nextprime}\, (3, []))$$

Notice that the type of the stream constructor has been adapted for inclusion in a strict programming language,

$$\mathit{str\text{-}cons'} :\ \alpha \times (1 \to \mathit{Stream}(\alpha)) \to \mathit{Stream}(\alpha)$$

4 Conclusions

Unstructured recursion is to functional programmers as the much-maligned **goto** command is to imperative programmers. It affords the power to code algorithms that are often difficult to understand. In the realm of functional programming, "difficult to understand" can mean that logical properties are not manifest or easily derived, or that termination is uncertain, or that unsuspected space leaks may occur in execution. This paper has proposed several combinators for structuring recursive programs according to the control implicit in datatype algebras (and coalgebras).

We have insufficient experience in using these combinators to make strong claims for their superiority, but it is encouraging that they seem to have natural and intuitive proof rules. Furthermore, they suggest styles of programming that allow one to anticipate sources of possible non-termination and of space leaks, and to take appropriate measures to avoid them.

References

[Bir88] Richard S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, volume 52 of *NATO Series F*. Springer-Verlag, 1988.

[CH86] Thierry Coquand and Gérard Huet. The Calculus of Constructions. Technical Report 530, INRIA, May 1986.

[CS92] J. R. B. Cockett and D. Spencer. Strong categorical datatypes. In R. A. G. Seely, editor, *International Meeting on Category Theory, 1991*. AMS, 1992.

[Gir71] J.-Y. Girard. Une extension de l'interprètation fontionnelle de Gödel à l'analyse et son application à l'èlimination des coupures dans l'analyse et la théorie des types. In J. F. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North Holland, 1971.

[Hag87] T. Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.

[HKS92] James Hook, Richard B. Kieburtz, and Timothy Sheard. Generating programs by reflection. Technical Report OGI CSE-092-015, Oregon Graduate Institute, July 1992.

[Hue87] Gérard Huet. Induction principles formalized in the Calculus of Constructions. In *TAPSOFT'87*, volume 249 of *Lecture Notes in Computer Science*, pages 276–286. Springer-Verlag, 1987.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. of 5th ACM Conf. on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, August 1991.

[Spi90] Mike Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14:25–42, 1990.

[Wad90] Philip Wadler. Comprehending monads. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 61–78, 1990.

[Wra89] G. C. Wraith. A note on categorical datatypes. In D. H. Pitt, D. E. Rydeheard, P. Dybjer, A. M. Pitts, and A. Poigné, editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 118,127. Springer-Verlag, 1989.