Inductive, Coinductive, and Pointed Types

Brian T. Howard
Department of Computer and Information Sciences
Kansas State University
bhoward@cis.ksu.edu

Abstract

An extension of the simply-typed lambda calculus is presented which contains both well-structured inductive and coinductive types, and which also identifies a class of types for which general recursion is possible. The motivations for this work are certain natural constructions in category theory, in particular the notion of an algebraically bounded functor, due to Freyd. We propose that this is a particularly elegant core language in which to work with recursive objects, since the potential for general recursion is contained in a single operator which interacts well with the facilities for bounded iteration and coiteration.

1 Introduction

In designing typed languages that include recursion, there has long been a tension between the structure provided by types based on well-founded induction and the freedom permitted by types based on general recursion. Very few languages outside of purely theoretical studies have chosen a strictly inductive system (one exception is **charity** [CF92]), partly because the logical price to be paid for ensuring that all recursion is well-founded is the necessity that all computations terminate, hence the language cannot be Turing-equivalent. On the other hand, the prevalence of inductively defined structures in computer science makes it natural to structure many algorithms in terms of iteration over elements of an inductive datatype. This natural structure is lost in common programming languages, where iteration is at best a syntactic sugaring for an application of a fixpoint operator.

In this paper we present an extension of the simply-typed lambda calculus with inductive types which also allows general recursion in a controlled manner, thus preserving many of the benefits of well-founded structural induction in a Turing-equivalent language. There are two key ideas which permit this:

- In addition to inductive types and iteration over their elements, we also consider the dual notion, the *coinductive* types, along with their associated natural operation of coiteration.
- All the potential for unbounded recursion in the language is confined to a subset of the types which are syntactically identified as "pointed"—a generalization of the standard notion of a lifted type, along the lines of Moggi's computation types. General recursion over pointed types follows from adding a function which forces evaluation of an element of a pointed coinductive type, either producing a value of the corresponding inductive type or failing to terminate.

All of the components of the language are motivated by constructions in category theory. This reflects our belief that the language of categories can be a useful source of inspiration and guidance in the design of programming language features. Our goal is to apply several recent results and trends in category theory to issues in practical language design, producing an elegant, powerful base language with the full natural structure afforded by inductive and coinductive types. This structure is well-behaved with respect to a powerful set of equational reasoning principles, ensuring that many useful program transformations will be sound.

As an example of a program written in this language, consider first the following term *fibs*, of the coinductive type $\nu t.(1 + nat \times t)_\perp$ (which may be thought of as a type of streams of natural numbers. The details of the language will be presented in Section 2; we will use a minor amount of syntactic sugaring which will not be discussed further):

$$fibs = \mathrm{gen}(\lambda\langle m, n\rangle.\, \lfloor \iota_2\langle m, \langle n, m+n\rangle\rangle \rfloor)\langle 1, 1\rangle$$

This generates the stream $1, 1, 2, 3, 5, \ldots$ of Fibonacci numbers by coiteration; no non-termination is yet involved, because it only generates successive numbers in the stream on demand. Now consider the related inductive type $\mu t.(1 + nat \times t)_\perp$, which is essentially a type of finite lists of natural numbers (the purpose of the lifting will be explained in a moment). A reasonable function defined by iteration over this type is *headUpto*, which when given a natural number k will take a list and return the longest prefix consisting entirely of numbers less than k:

$$\begin{aligned} headUpto = \lambda k.\ \mathrm{it}(\lambda\lfloor x \rfloor.\ &\mathbf{case}\ x\ \mathbf{of} \\ &\iota_1\Diamond\colon nil \\ &\iota_2\langle n, \ell\rangle\colon\ \mathbf{if}\ n < k \\ &\qquad\qquad \mathbf{then}\ cons(n, \ell) \\ &\qquad\qquad \mathbf{else}\ nil), \end{aligned}$$

where nil and $cons(n, \ell)$ are abbreviations for $fold\lfloor \iota_1\Diamond \rfloor$ and $fold\lfloor \iota_2\langle n, \ell\rangle \rfloor$, respectively. Again, this is a perfectly well-founded operation, because all lists of the inductive type

ICFP '96 5/96 PA, USA

are finite. If we wish to use this function to find the list of all Fibonacci numbers less than 100, we must first force *fibs* from a stream into a list; it is this action of "observing" the value of a coinductive object which introduces the only possible source of non-termination in the language, and the only reason it is allowed in this program is that we made an explicit provision for it with the lifting operator $\perp$, creating a pointed type. The correct function application is thus $\mathit{headUpto}\,100\,(\mathit{force}\,\mathit{fibs})$; with an appropriate reduction strategy (which only needs to be lazy in applying the *force* function), this evaluates to the desired list $(1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89)$. The above solution is natural and elegant, and would not have been nearly so easy to produce in a strictly iterative style. On the other hand, a solution using the standard machinery of fixpoint operators would have obscured or overpowered the intuitive construction of the iterative and coiterative pieces of the computation.

1.1 Related work

A preliminary attempt was made in the author's PhD thesis [How92] (see also [How93]) to reconcile inductive types with types for general recursion. The solution there was to introduce two different kinds of recursive type, corresponding to the choice between induction and recursion. This system did not provide a close integration of the two kinds of recursive type, and suffered from a reliance on the heavy machinery of fixpoint induction for reasoning about terms involving general recursion.

Following recent work by Freyd and others on algebraic compactness [Fre90, Fre91, Fre92, Bar92, Sim92, Fio94], the more elegant solution presented in this paper was developed. In brief, Freyd showed how to reduce the problem of finding solutions for general recursive domain equations to that of building inductive types, provided the functors involved are algebraically bounded, *i.e.*, the inductive and coinductive types are canonically isomorphic. By syntactically identifying a class of type expressions which will correspond in a model to algebraically bounded functors, we may apply this construction to develop a programming language which accounts for general recursive functions while only dealing with inductive (and coinductive) types.

In spirit, this work also follows in the footsteps of Crole and Pitts [CP92], who present a metalanguage which accounts for general recursion by obtaining a *fixpoint object* which allows unbounded iteration under control of Moggi's computation type (see also [Mog95] for recent work of Moggi along similar lines).

Independently, Launchbury and Paterson [LP95] have developed a type system similar to that described here to keep track of values which may be unboxed in the implementation of a lazy functional language. Their notion of a pointed type (which must be boxed) exactly corresponds with ours except they only allow recursive types which are pointed. It is difficult to see how a value of one of our unpointed inductive types could usefully be unboxed in general; however, it is interesting to note that a natural unboxed representation of the type $\mu t.\, 1 + \mathrm{char} \times t$ is exactly the traditional null-terminated string used in C.

1.2 Structure of the paper

In Section 2 we present the details of our proposed language and its categorical motivations: first a basic language with functions, sums, and products is described, then inductive, coinductive, and pointed types are added in turn. The major novelty of the language is the system of pointed types and the explicit forcing operation they permit—this is described in section 2.1. Following this are two sections of examples and comparisons to related work: a fixpoint combinator over pointed types is constructed in Section 3, based on the fixpoint object of Crole and Pitts [CP92]; then Freyd's construction of recursive types from inductive types [Fre90] is applied in Section 4 to provide universal types for call-by-value and call-by-name versions of the untyped lambda calculus; and finally, in Section 5, we compare our system to several recent proposals advocating a categorical style of program construction and manipulation based on iteration and coiteration [MFP91, FM91, Mei92, Kie93].

2 The language $\lambda^{\mu\nu\perp}$

Figure 1 presents a convenient formulation of the syntax of our base language with finite sums and products. As usual, a typing judgment $\Gamma \triangleright M : \sigma$ means that the term M has type σ, given the context Γ (a list of free variables and their types). Figure 2 lists the axioms governing these terms; they are derived from the equations which hold among the corresponding arrows in a closed category with finite products and coproducts. Observe that the term metavariables M, N, ..., are restricted to range over only terms of the appropriate type (so that, for example, the axiom (1η) does not imply that *all* terms are equal to $\diamond$, but only all terms of type 1). In association with standard rules about equality and substitution, these axioms provide an equational semantics. A non-deterministic operational semantics for the language may be obtained by directing the β axioms from left to right. For more details about this system and the relation between its equational and operational semantics, consult [How92]; related systems are considered many places in the literature, for example [GLT89, LS86, Mit90].

In a standard way, we may interpret a type expression σ containing a free type variable t as a *functor*; that is, it provides a map from types to types by substitution for t, and it may be extended to a map on terms of function type (because the intended model of this language is a cartesian closed category, we will feel free to abuse the distinction between arrows and elements of an exponential object, and to switch between external and internal views of functors). For example, if $\sigma \equiv 1 + t$, then as a functor it maps the type τ to $1 + \tau$ and it maps a term M of type $\tau \to \upsilon$ to the term $[\lambda x{:}\,1.\, \iota_1^{1+\upsilon}\diamond, \lambda y{:}\,\tau.\, \iota_2^{1+\upsilon}(My)]$ of type $1 + \tau \to 1 + \upsilon$. When talking about σ as a functor, we will find it convenient to name these maps F_σ, or simply F; thus, we would write $F(\tau) = 1 + \tau$ and $F(M) = [\lambda x.\, \iota_1\diamond, \lambda y.\, \iota_2(My)]$ (dropping type annotations for brevity).

A solution to the recursive type equation $t = \sigma$ is a type τ such that there is an isomorphism between τ and $F(\tau)$, *i.e.*, τ is a fixpoint of F. A well-known technique for finding a fixpoint, attributed to Lambek, is to consider the category of F-algebras, whose objects are (in our case) functions of type $F(\upsilon) \to \upsilon$, for any type υ; given F-algebras $f{:}\,F(\tau) \to \tau$ and $g{:}\,F(\upsilon) \to \upsilon$, an arrow from f to g is a function h of type $\tau \to \upsilon$ such that the following diagram commutes:

$$(hyp)\quad \frac{}{\Gamma, x{:}\sigma \triangleright x{:}\sigma}$$

$$(\to I)\quad \frac{\Gamma, x{:}\sigma \triangleright M{:}\tau}{\Gamma \triangleright (\lambda x{:}\sigma.\, M){:}\sigma \to \tau} \qquad \frac{\Gamma \triangleright M{:}\sigma \to \tau \quad \Gamma \triangleright N{:}\sigma}{\Gamma \triangleright MN{:}\tau}\quad (\to E)$$

$$(1I)\quad \frac{}{\Gamma \triangleright \Diamond{:}1} \qquad \frac{\Gamma \triangleright M{:}\sigma \times \tau}{\Gamma \triangleright \pi_1 M{:}\sigma}\quad (\times E_1)$$

$$(\times I)\quad \frac{\Gamma \triangleright M{:}\sigma \quad \Gamma \triangleright N{:}\tau}{\Gamma \triangleright \langle M, N\rangle{:}\sigma \times \tau} \qquad \frac{\Gamma \triangleright M{:}\sigma \times \tau}{\Gamma \triangleright \pi_2 M{:}\tau}\quad (\times E_2)$$

$$(+I_1)\quad \frac{\Gamma \triangleright M{:}\sigma}{\Gamma \triangleright \iota_1^{\sigma+\tau} M{:}\sigma + \tau} \qquad \frac{}{\Gamma \triangleright \square^{\upsilon}{:}0 \to \upsilon}\quad (0E)$$

$$(+I_2)\quad \frac{\Gamma \triangleright M{:}\tau}{\Gamma \triangleright \iota_2^{\sigma+\tau} M{:}\sigma + \tau} \qquad \frac{\Gamma \triangleright M{:}\sigma \to \upsilon \quad \Gamma \triangleright N{:}\tau \to \upsilon}{\Gamma \triangleright [M, N]{:}\sigma + \tau \to \upsilon}\quad (+E)$$

Figure 1: Syntax of the basic language

$$(\to \beta)\quad (\lambda x{:}\sigma.\, M)N = \{N/x\}M \qquad (\lambda x{:}\sigma.\, Mx) = M,\ x \text{ not free in } M\quad (\to \eta)$$

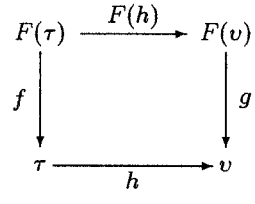
$$(\times\beta_1)\quad \pi_1\langle M, N\rangle = M \qquad M = \Diamond\quad (1\eta)$$

$$(\times\beta_2)\quad \pi_2\langle M, N\rangle = N \qquad M = \langle \pi_1 M, \pi_2 M\rangle\quad (\times\eta)$$

$$(+\beta_1)\quad [M, N](\iota_1^{\sigma+\tau} P) = MP \qquad M = \square^{\upsilon}\quad (0\eta)$$

$$(+\beta_2)\quad [M, N](\iota_2^{\sigma+\tau} P) = NP \qquad M = [\lambda x{:}\sigma.\, M(\iota_1^{\sigma+\tau} x), \lambda y{:}\tau.\, M(\iota_2^{\sigma+\tau} y)],\ x \text{ and } y \text{ not free in } M\quad (+\eta)$$

Figure 2: Axioms of the basic language



If $f{:}F(\tau) \to \tau$ is an initial object in this category, then in fact τ is a fixpoint of F, and f is the desired isomorphism. If $g{:}F(\upsilon) \to \upsilon$ is the isomorphism for any other fixpoint υ of F, then the initiality of f implies that the arrow h in the above diagram gives a unique way to map τ into υ; in this respect, τ is the *least* fixpoint of F.

A dual solution to $t = \sigma$ may be found by taking a terminal object in the category of F-coalgebras, which are simply functions of type $\upsilon \to F(\upsilon)$. Reversing all the arrows in the above diagram, if $f{:}\tau \to F(\tau)$ is such a terminal object, then τ is the *greatest* fixpoint of F.

If we extend our assumptions about the category underlying $\lambda^{\mu\nu\perp}$ to suppose that at least every F which corresponds to a type expression σ has both least and greatest fixpoints (that is, it is algebraically complete and cocomplete in a relevant sense), then we may augment the language to include these fixpoints as the types $\mu t.\, \sigma$ and $\nu t.\, \sigma$, respectively, which we will frequently write as μF and νF. An important point to note is that the type expression $t \to \sigma$ does not produce a (covariant) functor—given a function $f{:}\tau \to \upsilon$, there is no general way to produce a function of type $(\tau \to \sigma) \to (\upsilon \to \sigma)$ (consider $\sigma = \tau = 0$ and $\upsilon = 1$, and note that the existence of a function of type $1 \to 0$ leads to inconsistency). This contravariance in the first argument of $\to$ leads us to restrict the types σ for which we can find fixpoints to those in which t occurs only *positively*, that is, to the left of an even number of function arrows.

The terms and proof rules corresponding to these least and greatest fixpoint types (which are commonly called *inductive* and *coinductive* types, respectively) are given in Figure 3. For μF, the term $fold_{\mu F}$ is (the isomorphism of) the initial F-algebra, and the application $\mathbf{it}_{\mu F}\, M$ produces the unique F-algebra morphism from $fold_{\mu F}$ to the given function M. Dually, $unfold_{\nu F}$ is the terminal F-coalgebra, and $\mathbf{gen}_{\nu F}\, M$ produces the unique morphism from M to $unfold_{\nu F}$.[1]

The language described to this point is called $\lambda^{\mu\nu}$ in [How92]; it is shown there that the reduction rules for this language are confluent and strongly normalizing, hence only total functions may be computed. In fact, the class of total functions expressible in $\lambda^{\mu\nu}$ is quite large, containing precisely those functions provably total in the logic $ID_{<\omega}$,

[1] For efficiency's sake, we should also add terms and axioms providing explicit inverses to $fold_{\mu F}$ and $unfold_{\nu F}$; although $\mathbf{it}_{\mu F}\, F(fold_{\mu F})$ (and dually $\mathbf{gen}_{\nu F}\, F(unfold_{\nu F})$) has the right behavior, it works in time proportional to the size of its argument instead of constant time. This is related to the well-known linear-time predecessor problem.

$$(\mu I)\quad \frac{}{\Gamma \triangleright \mathit{fold}_{\mu F}\colon F(\mu F) \to \mu F} \qquad \frac{\Gamma \triangleright M\colon F(\tau) \to \tau}{\Gamma \triangleright \mathbf{it}_{\mu F}\, M\colon \mu F \to \tau}\quad (\mu E)$$

$$(\nu I)\quad \frac{\Gamma \triangleright M\colon \tau \to F(\tau)}{\Gamma \triangleright \mathbf{gen}_{\nu F}\, M\colon \tau \to \nu F} \qquad \frac{}{\Gamma \triangleright \mathit{unfold}_{\nu F}\colon \nu F \to F(\nu F)}\quad (\nu E)$$

$$(\mu\beta)\quad \mathbf{it}_{\mu F}\, M(\mathit{fold}_{\mu F}\, N) = M(F(\mathbf{it}_{\mu F}\, M)N) \qquad \frac{P(\mathit{fold}_{\mu F}\, N) = M(F(P)N)}{P = \mathbf{it}_{\mu F}\, M}\quad (\mu\eta)$$

$$(\nu\beta)\quad \mathit{unfold}_{\nu F}(\mathbf{gen}_{\nu F}\, M\, N) = F(\mathbf{gen}_{\nu F}\, M)(MN) \qquad \frac{\mathit{unfold}_{\nu F}(PN) = F(P)(MN)}{P = \mathbf{gen}_{\nu F}\, M}\quad (\nu\eta)$$

Figure 3: Syntax and axioms/inference rules for inductive and coinductive types

which is first order arithmetic augmented by finitely-iterated inductive definitions (see [BFPS81] for details about this logic; the relation to $\lambda^{\mu\nu}$ was presented in [How94][2]). This almost certainly contains every total function that would ever be needed for practical purposes, as it contains at an early stage every function bounded by Ackermann's notoriously fast-growing function. However, from a theoretical viewpoint this is nowhere near the class of all computable functions (and as long as all computations are terminating it can never hope to cover the entire class, because to do so would solve the Halting Problem), and from a practical viewpoint the proof of expressibility of any total function bounded by some fast-growing function does not lead to an efficient program, since the result will have the running time of the bounding function!

2.1 Contravariance and pointed types

The usual Smyth-Plotkin construction of fixpoints of mixed-variant functors in categories enriched with an order structure [SP82] reveals a coincidence between least and greatest fixpoints. Recent work of Freyd [Fre90, Fre91, Fre92] shows that this coincidence is the essential property needed to handle contravariance. Specifically, all that is needed to construct a fixed point for a contravariant endofunctor F is to show that the covariant functor F^2 is *algebraically bounded*, meaning that it has both an initial algebra and a terminal coalgebra, and they are canonically isomorphic. Consequently, the only addition needed to $\lambda^{\mu\nu}$ to allow the representation of fixpoints of contravariant functors is a function expressing this isomorphism.

We do not want to assert that *all* corresponding least and greatest fixpoints are isomorphic; for instance, if there were *any* function from $\nu t.\, t$ to $\mu t.\, t$ then all terms of any given type would be provably equal (given that we desire categorical finite sums as well as cartesian closure). Following Simpson [Sim92], we will identify the algebraically bounded functors by considering a faithful commutative strong monad whose functor T is algebraically bounded. Call a type τ *pointed* if it is the underlying object of an Eilenberg-Moore T-algebra, *i.e.*, if there is a retraction $\rho_\tau\colon T(\tau) \to \tau$ for the unit $\eta_\tau\colon \tau \to T(\tau)$ such that $\rho_\tau \circ T(\rho_\tau) = \rho_\tau \circ \mu_\tau$, where $\mu_\tau\colon T^2(\tau) \to T(\tau)$ is the monad multiplication. Say that a functor F is pointed if $F(\tau)$ is pointed for any τ, and say that it is conditionally pointed if $F(\tau)$ is pointed whenever τ is; then we have the following proposition:

Proposition 1 *For any interpretation of $\lambda^{\mu\nu\perp}$ in a cartesian closed category with finite sums, least and greatest fixpoints for all relevant endofunctors, and a monad as described above:*

1. *If σ and τ are pointed types, and υ is an arbitrary type, then 1, $T\upsilon$, $\sigma \times \tau$, and $\upsilon \to \tau$ are all pointed types.*
2. *If F is conditionally pointed, then νF is pointed.*
3. *If F is pointed, then μF and νF are canonically isomorphic (and pointed).*

Since the motivating example is the category of predomains and total functions, with the standard lifting monad identifying the pointed objects as the domains,[3] we will write $\tau_\perp$ for the type $T(\tau)$. The corresponding additions to the terms and proof rules are given in Figure 4. The *lift* $\lfloor M \rfloor$ corresponds to an application of the unit, while the *pointed abstraction* $\lambda\lfloor x\colon \sigma \rfloor.\, M$ corresponds to applying the functor T to the function $\lambda x.\, M\colon \sigma \to \tau$ and post-composing with the retract from $\tau_\perp$ to τ. These generalize the similar constructions in Moggi's computational lambda calculus [Mog89] in two ways:

- We include the total function space constructor as well as the Kleisli exponential; this is consistent with our other type constructors because we only allow general recursion on pointed types (see below).
- As a result, the body of the pointed abstraction may be of any pointed type, not just a computation type $T(\tau)$.

The term $\mathit{force}_{\mu F}$ gives the promised isomorphism from νF to μF (there is always a morphism in the other direction, described by $\mathbf{it}_{\mu F}\, \mathit{unfold}_{\nu F}^{-1}$ or, equivalently, $\mathbf{gen}_{\nu F}\, \mathit{fold}_{\mu F}^{-1}$); the axiom $(\nu\mu\beta)$ simply expresses the fact that it is an F-(co)algebra morphism. The intuition for calling this function *force* is that it provides the interface between the naturally lazy coinductive type νF and the more concrete, observable (at least to the extent that other components of

[2] An interesting feature of this correspondence is that coinductive types and non-strictly-positive inductive types (where t occurs to the left of a function arrow) do not add to the expressiveness—all computations over these types may be coded up using a family of ordinal notations described by strictly-positive inductive types.

[3] This is not quite correct; to avoid cardinality problems with types such as $\mu t.\, (t \to \mathit{bool}) \to \mathit{bool}$, where $\mathit{bool} \equiv 1 + 1$, we need to work in an effective version of domains. An appropriate category of PERs should work fine.

$$(\perp I)\quad \frac{\Gamma \triangleright M:\sigma}{\Gamma \triangleright \lfloor M\rfloor:\sigma_\perp} \qquad \frac{\Gamma, x:\sigma \triangleright M:\tau, \quad \tau \text{ pointed}}{\Gamma \triangleright (\lambda\lfloor x:\sigma\rfloor.\, M):\sigma_\perp \to \tau}\quad (\perp E)$$

$$(\nu\mu)\quad \frac{F \text{ pointed}}{\Gamma \triangleright \mathit{force}_{\mu F}:\nu F \to \mu F}$$

$$(\perp\beta)\quad (\lambda\lfloor x:\sigma\rfloor.\, M)\lfloor N\rfloor = \{N/x\}M$$

$$(\nu\mu\beta)\quad \mathit{force}_{\mu F}\, M = \mathit{fold}_{\mu F}(F(\mathit{force}_{\mu F})(\mathit{unfold}_{\nu F}\, M))$$

Figure 4: Syntax and axioms for pointed types

F are observable types such as products or sums) inductive type μF, coercing one to the other perhaps at the expense of non-termination if an attempt is made to force evaluation of an "infinite" element.

2.2 Properties of reduction

For reference, we collect the reduction rules for $\lambda^{\mu\nu\perp}$ in Figure 5. Using standard results about such reduction systems (see [Klo80, How92]) it is easy to prove that reduction is confluent and that a lazy reduction strategy (*i.e.*, outermost reduction of the principal subterm of each elimination operator, reducing *force* only when it is the argument of an iteration) is normalizing. In fact, since the reduction system without $(\nu\mu\beta)$ is strongly normalizing, the only operator that needs to be treated lazily is *force*.

If we identify the class of *observable* types as those which do not contain $\to$, ν, or $\perp$ (intuitively, these are the types whose values may be "printed out"), then a reasonable definition of observational equivalence says that two terms are equivalent if they have the same result (normal form or non-termination) under lazy reduction in all closed contexts of observable or lifted observable type, with the proviso that we do not distinguish values of types isomorphic to 1 (since these are the only types which may be both observable and pointed). We conjecture that all of the equational rules listed above (in particular, all of the (η)-rules) are sound with respect to this observational equivalence, therefore any program transformation justified by these rules will also be sound. The restriction to observable and lifted observable types is not harsh: taking the example of the *head-Upto* function of the Introduction, which produced values of type $\mu t.(1 + \mathit{nat} \times t)_\perp$, we may easily convert such values into lifted lists, $(\mu t.\, 1 + \mathit{nat} \times t)_\perp$, by applying the function $\mathbf{it}(\lambda\lfloor x\rfloor.\, \lfloor \mathit{fold}_{\mu t.\, 1+\mathit{nat}\times t}\, x\rfloor)$. Similar observation functions may be defined for all types not containing $\to$ by applying Mulry's notion of the *lifting* of a functor [Mul93].

3 A fixpoint object

In [CP92], a *fixpoint object* for a monad (T, η, μ) is defined to be a structure $(\Omega, \varsigma, \omega)$, where $\varsigma: T\Omega \to \Omega$ is an initial T-algebra and $\omega: 1 \to T\Omega$ picks out the unique fixpoint of the arrow $\eta_\Omega \circ \varsigma: T\Omega \to T\Omega$. This is used as the basis for a logical system for reasoning about fixpoint computations. For example, given a fixpoint object for T, they construct a fixpoint combinator for any type of the form $\alpha \to T\beta$.

In $\lambda^{\mu\nu\perp}$, we may find a fixpoint object for any monad (T, η, μ) such that μT is pointed. The type Ω is just μT, and ς is fold_Ω. We may construct ω by first coiterating the T-coalgebra $\eta_1: 1 \to T1$ to obtain the function $\mathbf{gen}_{\nu T}\, \eta_1: 1 \to \nu T$, and then applying it to the seed $\diamond$ and forcing the result over to Ω: $\mathit{force}_\Omega(\mathbf{gen}_{\nu T}\, \eta_1\, \diamond)$. This gives an "infinite" element in Ω; call it ∞. The desired global object ω is then just $\omega \equiv (\lambda x:1.\, \lfloor\infty\rfloor)$.

For the special case $T(\tau) = \tau_\perp$, where η_τ is $(\lambda x:\tau.\, \lfloor x\rfloor)$ (and μ_τ is $(\lambda\lfloor y:\tau_\perp\rfloor.\, y)$), if we define the term ∞ as above, then we may use it to construct a fixpoint combinator for an arbitrary pointed type σ by defining

$$\mathit{fix}_\sigma: (\sigma \to \sigma) \to \sigma \equiv (\lambda f:\sigma \to \sigma.\ \mathbf{it}_\Omega(\lambda\lfloor x:\sigma\rfloor.\, fx)\,\infty).$$

That is, we simply iterate $f^*: \sigma_\perp \to \sigma$ over the object ∞. This is as direct an explanation of finding fixpoints in a typed language as this author has seen (an anonymous referee of an earlier version of this paper pointed out that this is essentially the construction given by Mulry in Theorem 3.12 of [Mul92]).

4 Example: Recursive types reduced to inductive types

It is a relatively simple matter to reproduce in $\lambda^{\mu\nu\perp}$ the proofs from [Fre90] that the process of finding a fixpoint of an arbitrary type expression, in which the type variable may appear both covariantly and contravariantly, may be reduced solely to the problem of finding fixpoints for the covariant case, provided the resulting covariant functors are algebraically bounded. The process we follow is that, given a bifunctor T which is contravariant in its left argument and covariant in its right, first we show that $\mu t.\, T(-, t)$ is a contravariant functor whose fixpoints are also fixpoints of T itself—that is, we may find fixpoints one variable at a time. Next, we need to show that if F is a contravariant functor, then the fixpoints of F are the same as the fixpoints of the covariant functor F^2, provided F^2 is algebraically bounded.

We omit the details of these constructions here, and just note an example of this process. In [How92, How93] this author presented types which correspond to universal types for call-by-value and call-by-name versions of the untyped lambda calculus. Specifically, if $V = V \to V_\perp$ and $N = (N \to N)_\perp$, then we may use V and N respectively to give types to the cbv and cbn calculi. In $\lambda^{\mu\nu\perp}$, we may find solutions to these type equations by taking

$$V \equiv \mu r.\, \mu s.\, (\mu t.\, r \to t_\perp) \to s_\perp \equiv \mu r.\, F^2(r),$$
$$\text{for } F(r) \equiv \mu s.\, r \to s_\perp$$

$$N \equiv \mu r.\, \mu s.\, ((\mu t.\, (r \to t)_\perp) \to s)_\perp \equiv \mu r.\, G^2(r),$$
$$\text{for } G(r) \equiv \mu s.\, (r \to s)_\perp.$$

Figure 6 exhibits the remaining definitions needed for the simulation. For each untyped lambda term M, the transla-

$$(\to\beta)\quad (\lambda x{:}\sigma.\, M)N \longrightarrow \{N/x\}M \qquad (\lambda\lfloor x{:}\sigma\rfloor.\, M)\lfloor N\rfloor \longrightarrow \{N/x\}M\quad (\bot\beta)$$

$$(\times\beta_1)\quad \pi_1\langle M, N\rangle \longrightarrow M \qquad [M, N](\iota_1^{\sigma+\tau} P) \longrightarrow MP\quad (+\beta_1)$$

$$(\times\beta_2)\quad \pi_2\langle M, N\rangle \longrightarrow N \qquad [M, N](\iota_2^{\sigma+\tau} P) \longrightarrow NP\quad (+\beta_2)$$

$$(\mu\beta)\quad \mathbf{it}_{\mu F}\, M(\mathit{fold}_{\mu F}\, N) \longrightarrow M(F(\mathbf{it}_{\mu F}\, M)N)$$

$$(\nu\beta)\quad \mathit{unfold}_{\nu F}(\mathbf{gen}_{\nu F}\, M\, N) \longrightarrow F(\mathbf{gen}_{\nu F}\, M)(MN)$$

$$(\nu\mu\beta)\quad \mathit{force}_{\mu F}\, M \longrightarrow \mathit{fold}_{\mu F}(F(\mathit{force}_{\mu F})(\mathit{unfold}_{\nu F}\, M))$$

Figure 5: Reduction rules of $\lambda^{\mu\nu\bot}$

tions $\mathcal{V}[\![M]\!]{:}V_\bot$ and $\mathcal{N}[\![M]\!]{:}N$ will be terms of $\lambda^{\mu\nu\bot}$; it is shown in [How92] that a simple argument based on standardization of reductions will verify that $\mathcal{V}[\![M]\!] \twoheadrightarrow \mathcal{V}[\![M']\!]$ iff $M \twoheadrightarrow_{cbv} M'$, and similarly that $\mathcal{N}[\![M]\!] \twoheadrightarrow \mathcal{N}[\![M']\!]$ iff $M \twoheadrightarrow_{cbn} M'$.

5 Hylomorphisms and inductive programming

There have been several recent efforts in the programming language community to use programming techniques based on combinations of iteration and coiteration. Two which we will relate to our system are Meijer's hylomorphisms and Kieburtz's use of weakly initial algebras and their duals. Both of these proposals stay within the realm of pointed types, where inductive and coinductive types coincide, so neither is able to take full advantage of the separation between the well-founded operations of iteration and coiteration and the potential unboundedness of an explicit *force* operation. Nevertheless, we have been significantly influenced by their suggested style of programming, which reflects our intuitions about how the structure of data should guide the structure of programs.

In the language of the Squiggol group, the functions defined using **it** are *catamorphisms* and those using **gen** are *anamorphisms* ([FM91, MFP91, Mei92]). A *hylomorphism* is a combination of these concepts which first uses an anamorphism to build up what Meijer refers to as a "call-tree", and then uses a catamorphism to reduce this tree to a final result (compare the Fibonacci example of the introduction). A requirement for this is that the inductive types under consideration are all isomorphic to the corresponding coinductive types. They observe that hylomorphisms necessarily introduce the possibility of partial functions; when put in the framework of $\lambda^{\mu\nu\bot}$, where an explicit use of $\mathit{force}_{\mu F}$ is needed to tie together $\mathbf{gen}_{\nu F}\, M{:}\sigma \to \nu F$ and $\mathbf{it}_{\mu F}\, N{:}\mu F \to \tau$ to obtain the hylomorphism from σ to τ, this follows from the necessity of μF being a pointed type. By making this coercion explicit, our system also allows consideration of purely inductive or coinductive types, for which all functions are total.

Kieburtz [Kie93] also uses hylomorphisms (although not by name) when he demonstrates that a useful notion for inductive programming is that of finding homomorphisms from *weak* initial F-algebras. That is, we may have a function $g{:}F(\tau) \to \tau$ which has a left inverse $p{:}\tau \to F(\tau)$, *i.e.*, $p(g(x)) = x$ for all $x{:}F(\tau)$; if g is weakly initial then we may construct an F-algebra morphism from g to any given $f{:}F(\sigma) \to \sigma$ (which will not necessarily be unique). This will be possible if we can just find a fixpoint of the map which takes $h{:}\tau \to \sigma$ into $f \circ F(h) \circ p$. But this is just the hylomorphism which first coiterates p and then iteratively applies f to reduce the call-tree back down. Therefore, in $\lambda^{\mu\nu\bot}$, we may find weak initial F-algebras just by finding a left inverse, provided F is algebraically bounded. Kieburtz also describes the dual case, but this still involves a hylomorphism so it is not fundamentally different (just a shift of view between which of the given functions is the algebra/coalgebra and which is the left/right inverse).

6 Conclusions

We have demonstrated how recent developments in category theory may be used as guidance in designing a programming language with well-behaved recursive types. The language facilities for recursion concentrate on the natural inductive/coinductive structure which is common to many of the objects of interest to computer science. When general, unbounded recursion is needed, it is introduced in a controlled manner through a function which forces the evaluation of a coiteration process. To avoid inconsistency in a model which includes both extensional (categorical) products and sums, this forcing operation is only allowed on a class of types which have been identified as "pointed". We believe that the result is an elegant language in which to describe and examine recursive objects and algorithms.

References

[Bar92] Michael Barr. Algebraically compact functors. *Journal of Pure and Applied Algebra*, 82:211–231, 1992.

[BFPS81] Wilfried Buchholz, Solomon Feferman, Wolfram Pohlers, and Wilfried Sieg. *Iterated Inductive Definitions and Subsystems of Analysis: Recent Proof-Theoretical Studies.* Number 897 in Lecture Notes in Mathematics. Springer-Verlag, 1981.

[CF92] R. Cockett and T. Fukushima. About CHARITY. Technical Report 92/480/18, University of Calgary, June 1992.

[CP92] Roy L. Crole and Andrew M. Pitts. New foundations for fixpoint computations: FIX-hyperdoctrines and the FIX-logic. *Information and Computation*, 98(2):171–210, June 1992.

$$wrap^V : F(V) \to V \equiv force_V \circ \mathbf{gen}^{F(V)}_{\nu F^2}\, F(fold_V)$$
$$unwrap^V : V \to F(V) \equiv \mathbf{it}^{F(V)}_V\, F(fold_V^{-1})$$
$$wrap^N : G(N) \to N \equiv force_N \circ \mathbf{gen}^{G(N)}_{\nu G^2}\, G(fold_N)$$
$$unwrap^N : N \to G(N) \equiv \mathbf{it}^{G(N)}_N\, G(fold_N^{-1})$$

$$in^V : (V \to V_\perp) \to V \equiv wrap^V \circ fold_{F(V)} \circ (V \to unwrap^V_\perp)$$
$$out^V : V \to (V \to V_\perp) \equiv (V \to wrap^V_\perp) \circ fold^{-1}_{F(V)} \circ unwrap^V$$
$$in^N : (N \to N)_\perp \to N \equiv wrap^N \circ fold_{G(N)} \circ (N \to unwrap^N)_\perp$$
$$out^N : N \to (N \to N)_\perp \equiv (N \to wrap^N)_\perp \circ fold^{-1}_{G(N)} \circ unwrap^N$$

$$\mathcal{V}[\![x]\!] \equiv \lfloor x \rfloor$$
$$\mathcal{V}[\![\lambda x.M]\!] \equiv \lfloor in^V (\lambda x{:}V.\, \mathcal{V}[\![M]\!]) \rfloor$$
$$\mathcal{V}[\![M_1 M_2]\!] \equiv (\lambda \lfloor f{:}V \rfloor.\, \lambda \lfloor x{:}V \rfloor.\; out^V\, f\, x)\mathcal{V}[\![M_1]\!]\mathcal{V}[\![M_2]\!]$$

$$\mathcal{N}[\![x]\!] \equiv x$$
$$\mathcal{N}[\![\lambda x.M]\!] \equiv in^N \lfloor \lambda x{:}N.\, \mathcal{N}[\![M]\!] \rfloor$$
$$\mathcal{N}[\![M_1 M_2]\!] \equiv (\lambda \lfloor f{:}N \to N \rfloor.\, f)(out^N\, \mathcal{N}[\![M_1]\!])\mathcal{N}[\![M_2]\!]$$

Figure 6: Simulation of call-by-value and call-by-name untyped lambda calculi

[Fio94] Marcello P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps.* PhD thesis, University of Edinburgh, 1994.

[FM91] Maarten M. Fokkinga and Erik Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, January 1991.

[Fre90] Peter Freyd. Recursive types reduced to inductive types. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 498–507, 1990.

[Fre91] Peter Freyd. Algebraically complete categories. In A. Carboni, M.C. Pedicchio, and G. Rosolini, editors, *Category Theory: Proceedings, Como 1990*, pages 95–104. Springer-Verlag, 1991.

[Fre92] Peter Freyd. Remarks on algebraically compact categories. In M.P. Fourman, P.T. Johnstone, and A.M. Pitts, editors, *Applications of Categories in Computer Science*, number 177 in London Mathematical Society Lecture Note Series, pages 95–106. Cambridge University Press, 1992. Proceedings of the LMS Symposium, Durham 1991.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[How92] Brian T. Howard. *Fixed Points and Extensionality in Typed Functional Programming Languages.* PhD thesis, Stanford University, 1992. Published as Stanford Computer Science Department Technical Report STAN-CS-92-1455.

[How93] Brian T. Howard. Inductive, projective, and retractive types. Technical Report MS-CIS-93-14, Department of Computer and Information Science, University of Pennsylvania, 1993.

[How94] Brian T. Howard. The expressive power of inductive and coinductive types. Presented at the Tenth Workshop on the Mathematical Foundations of Programming Semantics, March 1994.

[Kie93] Richard B. Kieburtz. Inductive programming. Technical Report CS/E 93-001, Oregon Graduate Institute of Science and Technology, 1993.

[Klo80] Jan Willem Klop. *Combinatory Reduction Systems.* PhD thesis, University of Utrecht, 1980. Published as Mathematical Center Tract 129.

[LP95] John Launchbury and Ross Paterson. Parametricity and unboxing with unpointed types. Available at `http://www.cse.ogi.edu/~jl/Papers/point.ps`, 1995.

[LS86] J. Lambek and P.J. Scott. *Introduction to Higher-Order Categorical Logic.* Number 7 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1986.

[Mei92] Erik Meijer. *Calculating Compilers.* PhD thesis, University of Nijmegen, 1992.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the Fifth ACM Conference on Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 124–144. Springer-Verlag, 1991.

[Mit90] John C. Mitchell. Type systems for programming languages. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 8, pages 365–458. Elsevier, 1990.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23, 1989.

[Mog95] Eugenio Moggi. Metalanguages and applications. Available as `ML-notes.dvi.gz` by anonymous ftp from `theory.doc.ic.ac.uk` in directory `tfm/papers/MoggiE`, October 1995.

[Mul92] Philip S. Mulry. Strong monads, algebras and fixed points. In M.P. Fourman, P.T. Johnstone, and A.M. Pitts, editors, *Applications of Categories in Computer Science*, number 177 in London Mathematical Society Lecture Note Series, pages 202–216. Cambridge University Press, 1992. Proceedings of the LMS Symposium, Durham 1991.

[Mul93] Philip S. Mulry. Lifting theorems for kleisli categories. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Ninth International Conference on Mathematical Foundations of Programming Semantics*, number 802 in Lecture Notes in Computer Science, pages 304–319. Springer-Verlag, 1993.

[Sim92] Alex K. Simpson. Recursive types in kleisli categories. Available as `kleisli.dvi.Z` by anonymous ftp from `ftp.dcs.ed.ac.uk` in directory `pub/als`, August 1992.

[SP82] Michael B. Smyth and Gordon D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11:761–783, 1982.