Fixpoint Computations and Coiteration (Extended Abstract)

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 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 language with the full natural structure afforded by inductive and coinductive types.

As an example of a program written in this language, consider first the following term *fibs*, of the (pointed) coinductive type $\nu t.\,(1 + \mathit{nat} \times t)_\perp$ (which may be thought of as a type of streams of natural numbers. The details of the language will be presented in Section 2; to aid in reading the example, **gen** is the coiteration operator, **it** is the iterator, ι_1 and ι_2 are the injections into the sum type, $\lfloor x \rfloor$ is the insertion of x into the lifted type—as a pattern with λ it produces a strict abstraction, and $\diamond$ is the unique value of type 1. We also use a minor amount of syntactic sugaring which will not be discussed further):

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

This generates the stream of Fibonacci numbers $(1, 1, 2, 3, 5, \ldots)$ by coiteration; no non-termination is yet involved, because it only generates successive numbers in the stream on demand. Now consider the associated (pointed) inductive type $\mu t.\,(1 + \mathit{nat} \times t)_\perp$, which is essentially a type of finite lists of natural numbers. A reasonable function defined by iteration over this type is *headUpto*, which takes a natural number k and a list ℓ and returns the largest prefix of ℓ consisting entirely of numbers less than k:

$$\mathit{headUpto} = \lambda k.\,\mathbf{it}(\lambda\lfloor x\rfloor.\,[\lambda\diamond.\,\mathit{fold}\lfloor\iota_1\diamond\rfloor, \lambda\langle n, \ell\rangle.\,\begin{pmatrix}\text{if } n < k \\ \text{then } \mathit{fold}\lfloor\iota_2\langle n, \ell\rangle\rfloor \\ \text{else } \mathit{fold}\lfloor\iota_1\diamond\rfloor\end{pmatrix}]x)$$

Again, this is a perfectly well-founded operation, because all lists of the inductive type 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 at this point 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 recursively defined 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 Barr on algebraic compactness [Fre90, Fre91, Fre92, Bar92], the more elegant solution presented in this paper was developed. In brief, Freyd showed how to reduce the problem of finding solutions to 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 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 [CP90], who present a language which accounts for general recursion by obtaining a *fixpoint object* which allows unbounded iteration under the control of Moggi's computation type. In our opinion, their system suffers from a somewhat unnatural choice of proof rules, similar to our earlier problems with an unfortunate mixing of general fixpoint induction with the well-founded rules for inductive types. More will be said about the connection with the current system below.

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 2.2. Following this are three 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 [CP90]; 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 $\lambda^{\mu\nu\bot}$

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 functional 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). For example, if $\sigma \equiv 1 + t$, then 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,

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

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

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

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

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

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

Figure 1: Syntax of the basic language

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

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

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

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

$$(+\beta_2)\quad [M, N](\iota_2^{\sigma+\tau} P) = NP \qquad M = [\lambda x{:}\,\sigma.\, M(\iota_1^{\sigma+\tau} x), \lambda y{:}\,\tau.\, M(\iota_2^{\sigma+\tau} y)] \quad (+\eta)$$

Figure 2: Axioms of the basic language

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

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

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

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

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

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 $f\colon F(\tau) \to \tau$ and $g\colon F(\upsilon) \to \upsilon$, an arrow from f to g is a function h of type $\tau \to \upsilon$ such that the following diagram commutes:

$$\begin{tikzcd} F(\tau) \arrow[r, "F(h)"] \arrow[d, "f"'] & F(\upsilon) \arrow[d, "g"] \\ \tau \arrow[r, "h"'] & \upsilon \end{tikzcd} .$$

If $f\colon F(\tau) \to \tau$ is an initial object in this category, then in fact τ is a fixpoint of F, and f is the desired isomorphism. If $g\colon F(\upsilon) \to \upsilon$ 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 $f\colon \tau \to F(\tau)$ 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, 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 $f\colon \tau \to \upsilon$, 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 $\mathit{fold}_{\mu F}$ is the initial F-algebra, and the operator $\mathbf{it}_{\mu F}$ produces the unique F-algebra morphism from $\mathit{fold}_{\mu F}$ to the given function M. Dually, $\mathit{unfold}_{\nu F}$ is the terminal F-coalgebra, and $\mathbf{gen}_{\nu F}$ produces the unique morphism from M to $\mathit{unfold}_{\nu F}$.

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, including all functions which are provably total in the logic $ID_{<\omega}$, 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]). This almost certainly contains every total function that would ever be needed for practical purposes, as it contains at an early stage every function which grows no faster than 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 in recursive types

A standard way to construct fixpoints of both covariant and contravariant functors is to consider categories whose homsets have been enriched by an ordering structure [Wan79, SP82]. Under certain additional conditions, these categories allow contravariant functors to be converted into ordinary functors on a related category of embedding-projection pairs; the fixpoints of these functors will then transfer back to the original category. This permits solutions to equations such as $t = t \to t$, giving a "universal" type which allows typing of terms from the untyped lambda calculus. For another example, if we could solve the equation $t = t \to \sigma$, obtaining a fixpoint τ and isomorphism $\varphi\colon \tau \to (\tau \to \sigma)$, we would be able to type the self-application involved in the Turing fixpoint combinator Θ_υ of type $\sigma \equiv (\upsilon \to \upsilon) \to \upsilon$:

$$(\lambda w\colon \tau.\, \lambda f\colon \upsilon \to \upsilon.\, f(\varphi\, w\, w\, f))(\varphi^{-1}(\lambda w\colon \tau.\, \lambda f\colon \upsilon \to \upsilon.\, f(\varphi\, w\, w\, f))).$$

Unfortunately, this construction is inconsistent with our desire to have categorical products and sums. In particular, we could use the above fixpoint combinator to find a term $\Theta_\upsilon(\lambda x\colon \upsilon.\, x)$ of type υ for *any* υ, including the (supposedly) empty type 0.

However, following recent work of Freyd [Fre90, Fre91, Fre92], this heavy machinery is not necessary to handle contravariance. In fact, 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*, which means that it has both an initial algebra and a terminal coalgebra, and they are isomorphic (Barr uses the term *algebraically compact* for the same concept in [Bar92]). Consequently, the only addition needed to $\lambda^{\mu\nu}$ to allow fixpoints of contravariant functors to be represented is a function expressing this isomorphism.

2.2 Pointed types

We do not want to assert that *all* corresponding least and greatest fixpoints are isomorphic because that would lead to the same inconsistency mentioned above. Instead, we will restrict the class of functors for which the fixpoints will be isomorphic. The mechanism by which we choose to do this is to identify a reflective subcategory of *pointed* objects and arrows, and say that only a least fixpoint which is pointed will be isomorphic to its greatest fixpoint. We will not be more specific about precisely which objects and arrows will be considered pointed, but the motivating example is when we are dealing with a category of predomains, *i.e.*, complete partial orders which do not necessarily have bottom elements, and we take as the subcategory all those predomains which do have bottoms, with only strict (bottom-preserving) functions as the arrows. In this case, the reflector is the common lifting functor which adds a bottom element to a cpo.

A reflective subcategory $\mathcal{A} \subseteq \mathcal{C}$ is determined by a functor $R\colon \mathcal{C} \to \mathcal{A}$, the *reflector*, such that there is an isomorphism of the homsets $\mathcal{A}(RX, Y) \cong \mathcal{C}(X, UY)$, natural in X and Y, where $U\colon \mathcal{A} \to \mathcal{C}$ is the inclusion. (Because we are talking about arbitrary categories for the moment, we will use upper-case letters to refer to objects, rather than the Greek letters which correspond specifically to types as objects.) That is, R is left adjoint to U, which can also be stated by saying that we have a natural transformation $\eta\colon \mathcal{C} \stackrel{.}{\to} UR$ such that each arrow $\eta_X\colon X \to URX$ is universal from X to U. Note that since U is the inclusion functor, we may as well speak of R as an endofunctor on $\mathcal{C}$ (with values in $\mathcal{A}$); in this case the data we must provide are, for each object X of $\mathcal{C}$,

- an object RX,
- an arrow $\eta_X\colon X \to RX$, and

- the map $-^*$ which takes any $f\colon X \to Y$, for Y in $\mathcal{A}$, and produces the unique $f^*\colon RX \to Y$ such that $f^* \circ \eta_X = f$.

As constructs in the programming language, this entails adding the type $\sigma_\perp$ for each type σ, a term constructor $\lfloor M \rfloor$ corresponding to η, and another term constructor $(\lambda\lfloor x\colon\sigma\rfloor.\, M)$, which corresponds to applying * to the abstraction $(\lambda x\colon\sigma.\, M)$ and which we call the *pointed abstraction.* These additions, together with their associated proof rules, are given in Figure 4. We do not specify rules here for when a type is pointed, because in general it will depend on the particular choice of $\mathcal{A}$ and $\mathcal{C}$ in the intended model, but we expect that suitable syntactic conditions may be applied in a given situation which will at least establish that $\sigma_\perp$ and $\mu t.\, \sigma_\perp$ are pointed. A set of conditions corresponding to the predomain model is given in [How92].

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

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

$$(\perp\beta)\quad (\lambda\lfloor x\colon\sigma\rfloor.\, M)\lfloor N\rfloor = \{N/x\}M \qquad\qquad \frac{M\colon \sigma_\perp \to \tau \text{ pointed, } x \text{ not free in } M}{(\lambda\lfloor x\colon\sigma\rfloor.\, M\lfloor x\rfloor) = M}\quad (\perp\eta)$$

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

Figure 4: Syntax and axioms/inference rules for pointed types

Observe that the data for R make it a monad on $\mathcal{C}$, with unit η and multiplication given by $\mathit{id}^*_{RX}\colon RRX \to RX$ for each X; this is accompanied by an intentional similarity between our additions to the term language and the computational lambda calculus described by Moggi [Mog89]. Of course, this result is not a surprise, since every adjunction produces a monad; however, the fact that R is a reflector of a subcategory of $\mathcal{C}$ allows us to take advantage of some extra properties. In particular, because the collection of pointed objects may extend beyond those which are obtainable as an image RX of some object X, there may be more opportunities to remove the computation type constructor. That is, for an arbitrary monad functor T, once it has been applied to an object there is no general way to remove it, since the multiplication $\mu\colon TT \xrightarrow{\cdot} T$ can only collapse two T's into one. However, the functor R may be absorbed into any pointed type Y, by the arrow $\mathit{id}^*_Y\colon RY \to Y$. An example is the aforementioned category of predomains, where not only the lift of an arbitrary predomain is pointed, but also the cartesian product of any two pointed predomains (whose bottom object is just the ordered pair of the bottoms of the two factors).

Now we are ready to finish our description of the language $\lambda^{\mu\nu\perp}$ by adding the function $\mathit{force}_{\mu F}\colon \nu F \to \mu F$ whenever μF is pointed; see Figure 4. Together with an axiom that says that $\mathit{force}_{\mu F}$ is indeed an F-(co)algebra morphism from $\mathit{unfold}_{\nu F}$ to $\mathit{fold}_{\mu F}$, this is sufficient to establish an isomorphism between μF and νF (since there is always a morphism in the other direction, given by either $\mathbf{it}_{\mu F}\, \mathit{unfold}^{-1}_{\nu F}$ or $\mathbf{gen}_{\nu F}\, \mathit{fold}^{-1}_{\mu F}$; see below for the inverses to *fold* and *unfold*). 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 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.

To summarize the assumptions we have made about a category which will model $\lambda^{\mu\nu\perp}$, it must be cartesian closed, with finite sums; it must be algebraically complete and cocomplete for some class of (covariant) endofunctors at least including all those which correspond to type constructors; and it must contain a reflective subcategory of "pointed" objects and arrows, such that the functors with pointed initial algebras are algebraically bounded. Unfortunately, the category of predomains discussed before does not quite satisfy these requirements, because the functor $FX = (X \to B) \to B$, where $B \equiv 1 + 1$ is the type of booleans, does not have a fixpoint (at least using ordinary set theory) since the carrier sets of X and FX must have different cardinalities. If we relax the completeness requirement slightly to disallow such functors then predomains are a valid example; establishing general conditions which cover this is a subject for future work. Another interesting example (which requires the same restriction to functors which will not lead to a cardinality blow-up) is simply the category of sets, where the only pointed objects are the one-element sets, hence we may take the constant 1 functor as the reflector. In this case there is no interest in the terms for pointed types, so $\lambda^{\mu\nu\perp}$ reduces to the strongly normalizing language $\lambda^{\mu\nu}$.

For reference, we collect the reduction rules for $\lambda^{\mu\nu\perp}$ in Figure 5.

$$
\begin{array}{lllr}
(\to\beta) & (\lambda x{:}\,\sigma.\,M)N \longrightarrow \{N/x\}M & (\lambda\lfloor x{:}\,\sigma\rfloor.\,M)\lfloor N\rfloor \longrightarrow \{N/x\}M & (\perp\beta) \\
(\times\beta_1) & \pi_1\langle M,N\rangle \longrightarrow M & [M,N](\iota_1^{\sigma+\tau}P) \longrightarrow MP & (+\beta_1) \\
(\times\beta_2) & \pi_2\langle M,N\rangle \longrightarrow N & [M,N](\iota_2^{\sigma+\tau}P) \longrightarrow NP & (+\beta_2)
\end{array}
$$

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

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

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

Figure 5: Operational semantics of $\lambda^{\mu\nu\perp}$

3 A fixpoint object

In [CP90], 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 $T = R$, where η_τ is $(\lambda x{:}\, \tau.\, \lfloor x\rfloor)$ (and μ_τ is $(\lambda\lfloor y{:}\, \tau_\perp\rfloor.\, y)$, although it is not used here), 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.

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 for this summary, 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 translations $\mathcal{V}[\![M]\!]: V_\perp$ and $\mathcal{N}[\![M]\!]: N$ will be terms of $\lambda^{\mu\nu\perp}$; 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 $\mathcal{N}[\![M]\!] \twoheadrightarrow \mathcal{N}[\![M']\!]$ 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 what we have identified as the pointed subcategory, 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 $\lambda^{\mu\nu\perp}$, where an explicit use of $force_{\mu F}$ 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

$$\begin{aligned}
&\mathit{wrap}^V\colon F(V) \to V \equiv \mathit{force}_V \circ \mathbf{gen}^{F(V)}_{\nu F^2}\, F(\mathit{fold}_V)\\
&\mathit{unwrap}^V\colon V \to F(V) \equiv \mathbf{it}^{F(V)}_V\, F(\mathit{fold}^{-1}_V)\\
&\mathit{wrap}^N\colon G(N) \to N \equiv \mathit{force}_N \circ \mathbf{gen}^{G(N)}_{\nu G^2}\, G(\mathit{fold}_N)\\
&\mathit{unwrap}^N\colon N \to G(N) \equiv \mathbf{it}^{G(N)}_N\, G(\mathit{fold}^{-1}_N)
\end{aligned}$$

$$\begin{aligned}
&\mathit{in}^V\colon (V \to V_\perp) \to V \equiv \mathit{wrap}^V \circ \mathit{fold}_{F(V)} \circ (V \to \mathit{unwrap}^V_\perp)\\
&\mathit{out}^V\colon V \to (V \to V_\perp) \equiv (V \to \mathit{wrap}^V_\perp) \circ \mathit{fold}^{-1}_{F(V)} \circ \mathit{unwrap}^V\\
&\mathit{in}^N\colon (N \to N)_\perp \to N \equiv \mathit{wrap}^N \circ \mathit{fold}_{G(N)} \circ (N \to \mathit{unwrap}^N)_\perp\\
&\mathit{out}^N\colon N \to (N \to N)_\perp \equiv (N \to \mathit{wrap}^N)_\perp \circ \mathit{fold}^{-1}_{G(N)} \circ \mathit{unwrap}^N
\end{aligned}$$

$$\begin{aligned}
&\mathcal{V}[\![x]\!] \equiv \lfloor x \rfloor\\
&\mathcal{V}[\![\lambda x.\, M]\!] \equiv \lfloor \mathit{in}^V(\lambda x\colon V.\, \mathcal{V}[\![M]\!]) \rfloor\\
&\mathcal{V}[\![M_1 M_2]\!] \equiv (\lambda \lfloor f\colon V \rfloor.\, \lambda \lfloor x\colon V \rfloor.\ \mathit{out}^V\, f\, x)\mathcal{V}[\![M_1]\!]\mathcal{V}[\![M_2]\!]
\end{aligned}$$

$$\begin{aligned}
&\mathcal{N}[\![x]\!] \equiv x\\
&\mathcal{N}[\![\lambda x.\, M]\!] \equiv \mathit{in}^N \lfloor \lambda x\colon N.\, \mathcal{N}[\![M]\!] \rfloor\\
&\mathcal{N}[\![M_1 M_2]\!] \equiv (\lambda \lfloor f\colon N \to N \rfloor.\, f)(\mathit{out}^N\, \mathcal{N}[\![M_1]\!])\mathcal{N}[\![M_2]\!]
\end{aligned}$$

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

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 $g\colon F(\tau) \to \tau$ which has a left inverse $p\colon \tau \to F(\tau)$, *i.e.*, $p(g(x)) = x$ for all $x\colon F(\tau)$; if g is weakly initial then we may construct an F-algebra morphism from g to any given $f\colon F(\sigma) \to \sigma$ (which will not necessarily be unique). This will be possible if we can just find a fixpoint of the map which takes $h\colon \tau \to \sigma$ 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 $\lambda^{\mu\nu\perp}$, 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] M. Barr. Algebraically compact functors. Available as `algcomp.dvi.Z` by anonymous ftp from `triples.math.mcgill.ca` in directory `pub/barr`, March 1992.

[BFPS81] W. Buchholz, S. Feferman, W. Pohlers, and W. Sieg. *Iterated Inductive Definitions and Subsystems of Analysis: Recent Proof-Theoretical Studies*, volume 897 of *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.

[CP90] R.L. Crole and A.M. Pitts. New foundations for fixpoint computations. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 489–497, 1990. A revised version appeared in *Information and Computation*, 98(2):171–210, 1992.

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

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

[Fre91] P. 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] P. Freyd. Remarks on algebraically compact categories. In M.P. Fourman, P.T. Johnstone, and A.M. Pitts, editors, *Applications of Categories in Computer Science: Proceedings of the London Mathematical Society Symposium, Durham, 1991*, pages 95–106. Cambridge University Press, 1992.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[Gre92] J. Greiner. Programming with inductive and co-inductive types. Technical Report CMU-CS-92-109, Carnegie Mellon University, January 1992.

[How92] B.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] B.T. Howard. Inductive, projective, and retractive types. Technical Report MS-CIS-93-14, Department of Computer and Information Science, University of Pennsylvania, 1993.

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

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

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

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

[MFP91] E. Meijer, M. Fokkinga, and R. 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*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.

[Mit90] J.C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North-Holland, 1990.

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

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

[Wan79] M. Wand. Fixed-point constructions in order-enriched categories. *Theoretical Computer Science*, 8:13–30, 1979.