Inductive, Projective, and Retractive Types

Brian T. Howard
Institute for Research in Cognitive Science
University of Pennsylvania
bhoward@saul.cis.upenn.edu

Abstract

We give an analysis of classes of recursive types by presenting two extensions of the simply-typed lambda calculus. The first language only allows recursive types with built-in principles of well-founded induction, while the second allows more general recursive types which permit non-terminating computations. We discuss the expressive power of the languages, examine the properties of reduction-based operational semantics for them, and give examples of their use in expressing iteration over large ordinals and in simulating both call-by-name and call-by-value versions of the untyped lambda calculus. The motivations for this work come from category theoretic models.

1 Introduction

An examination of the common uses of recursion in defining types reveals that there are two distinct classes of operations being performed. The first class of recursive type contains what are generally known as the "inductive" types, as well as their duals, the "coinductive" or "projective" types. The distinguishing characteristic of the types in this class is that they each have an associated rule of well-founded induction. Common inductive types are lists, trees, and the natural numbers, whereas standard examples of projective types are computable infinite streams and the natural numbers plus a point at infinity. The second class of recursive type contains what we refer to as the "retractive" types, whose distinguishing characteristic is that they contain elements which represent non-terminating computations. The standard example of a retractive type is the universal type which can be given to terms of an untyped lambda calculus.

We will present and discuss two functional languages containing these recursive types. The first, which we refer to as $\lambda^{\mu\nu}$, adds only the inductive and projective types to a simply-typed lambda calculus with finite products and sums. This language is essentially identical to the language $\lambda^{MM\mu\nu}$ developed independently by Greiner [Gre92]; both $\lambda^{\mu\nu}$ and $\lambda^{MM\mu\nu}$ were inspired by Hagino's work with categorical datatypes [Hag87a, Hag87b]. The obvious non-deterministic operational semantics for $\lambda^{\mu\nu}$ is shown to be both confluent and strongly normalizing, hence the non-determinism is inessential and any of the common reduction strategies will be normalizing.

The second language, $\lambda^{\perp\rho}$, extends $\lambda^{\mu\nu}$ in two ways. It includes the retractive types as well as the inductive and projective recursive types, and it also adds an explicit lifting constructor, for introducing types for which evaluation may not terminate. Lifting is essential in describing which type expressions are allowed in the body of a retractive type definition. The standard approach to introducing recursive types in a language involves modifying the semantics of some of the type constructors so that all recursive domain equations will have solutions; typically this means that the sum or product types will not be "extensional" (see for example [GS90, Plo85, SP82]). Our work developed from a desire to have recursive types in a language whose equational semantics includes extensionality for both products and sums (*i.e.*, both products and sums are "categorical"). It is well-known that such a semantics in which all recursive domain equations have solutions must be inconsistent, hence we must restrict the class of solvable domain equations. Explicitly introducing lifted types enables us to do this by defining the class of "pointed" type expressions, which intuitively are those that contain a "bottom" element, representing non-termination.

Since the operational semantics of $\lambda^{\mu\nu}$ is strongly normalizing, only total functions are representable. We show that all of the natural number functions which are provably total in Peano Arithmetic are expressible in $\lambda^{\mu\nu}$. There are also expressible functions which are not provably total in PA, hence the language is more expressive than Gödel's system **T** of primitive recursive functionals. We show that all of the functions expressible in $\lambda^{\mu\nu}$ are provably total in second-order arithmetic, by a translation into Girard's system **F**; however, there are *algorithms* expressible in $\lambda^{\mu\nu}$ which do not seem possible in system **F**, such as a constant-time predecessor.

A fixed point operator may be defined for pointed types (such as the flat natural numbers) in $\lambda^{\perp\rho}$, hence we may express all partial recursive functions over such types. Because of the possibility of non-termination, reduction order becomes important in giving an operational semantics for $\lambda^{\perp\rho}$. We show that a leftmost reduction strategy is normalizing. Using this fact, we give translations of both the call-by-value and call-by-name versions of the untyped lambda calculus into $\lambda^{\perp\rho}$ and show that the translations preserve the reduction relation.

By distinguishing these classes of recursive types, we have thus exhibited a Turing-equivalent functional programming language which contains a very expressive terminating sublanguage. By using the type system in this way to identify all the possible sources of non-termination in a program, we hope to make feasible more effective techniques for proving properties of programs, and perhaps also allow more powerful optimization procedures.

This is a revised and condensed version of material which appeared in the author's doctoral dissertation [How92].

2 Syntax of the language $\lambda^{\mu\nu}$

Type expressions may be any of the following, where we use $\sigma, \tau, \ldots$ as metavariables for type expressions:

$$t \mid 0 \mid 1 \mid \sigma + \tau \mid \sigma \times \tau \mid \sigma \to \tau \mid \mu t.\, \sigma \mid \nu t.\, \sigma.$$

The types of the language will be the closed type expressions, *i.e.*, all occurrences of type variables t must be in the scope of a binding operator μ or ν naming the same variable. Not every type expression σ may be the body of an inductive (μ) or projective (ν) type — the bound type variable t may only occur *strictly positively* in σ, that is, it may not appear in any subterm of the left argument of an arrow.

We now give an inference system for typing assertions $\Gamma \triangleright M{:}\, \sigma$ for well-formed terms of $\lambda^{\mu\nu}$. We start with the usual rules for variables, λ-abstraction and application:

(var) $$x{:}\, \sigma \triangleright x{:}\, \sigma$$

($add\ var$) $$\frac{\Gamma \triangleright M{:}\, \sigma}{\Gamma, x{:}\, \tau \triangleright M{:}\, \sigma.}$$

($\to$ *Intro*) $$\frac{\Gamma, x{:}\, \sigma \triangleright M{:}\, \tau}{\Gamma \triangleright (\lambda x{:}\, \sigma.\, M){:}\, \sigma \to \tau}$$

($\to$ *Elim*) $$\frac{\Gamma \triangleright M{:}\, \sigma \to \tau, \quad \Gamma \triangleright N{:}\, \sigma}{\Gamma \triangleright MN{:}\, \tau.}$$

The constants and term constructors corresponding to the given type constructors come from the categorical interpretation of the types. For example, the (+ *Elim*) rule provides the mediating arrow from the sum $\sigma + \tau$ to an arbitrary type υ, given arrows from each of σ and τ to υ.

(0 *Elim*) $$\emptyset \triangleright \Box^{\upsilon}{:}\, 0 \to \upsilon.$$

(+ *Intro*$_1$) $$\emptyset \triangleright \iota_1^{\sigma+\tau}{:}\, \sigma \to \sigma + \tau$$

(+ *Intro*$_2$) $$\emptyset \triangleright \iota_2^{\sigma+\tau}{:}\, \tau \to \sigma + \tau,$$

(+ *Elim*) $$\frac{\Gamma \triangleright M{:}\, \sigma \to \upsilon, \quad \Gamma \triangleright N{:}\, \tau \to \upsilon}{\Gamma \triangleright [M, N]{:}\, \sigma + \tau \to \upsilon.}$$

(1 *Intro*) $$\Gamma \triangleright \Diamond{:}\, 1$$

($\times$ *Elim*$_1$) $$\emptyset \rhd \pi_1^{\sigma\times\tau} \colon \sigma \times \tau \to \sigma$$

($\times$ *Elim*$_2$) $$\emptyset \rhd \pi_2^{\sigma\times\tau} \colon \sigma \times \tau \to \tau$$

($\times$ *Intro*) $$\frac{\Gamma \rhd M \colon \sigma, \quad \Gamma \rhd N \colon \tau}{\Gamma \rhd \langle M, N\rangle \colon \sigma \times \tau.}$$

We interpret the action of substituting a type τ for a free variable t in a type expression σ, which we write as $\{\tau/t\}\sigma$, as the application of a functor to an object. For example, the interpretation of $\sigma \equiv t \times \upsilon$ is the endofunctor F such that if the objects interpreting τ and υ are A and B, respectively, then the object interpreting $\{\tau/t\}\sigma \equiv \tau \times \upsilon$ is $F(A) = A \times B$. By abuse of notation we will often write $F(\tau)$ for the type $\{\tau/t\}\sigma$; correspondingly we will write μF instead of $\mu t.\,\sigma$ (and likewise for ν), where the underlying type expression σ is left implicit.

Finally, we have the terms associated with the inductive and projective types. The categorical interpretation of an inductive type μF is an initial F-algebra, *i.e.*, an arrow φ from $F(\mu F)$ to μF such that for any other F-algebra $f \colon F(X) \to X$ there is a unique F-algebra morphism $h_f \colon \varphi \to f$, that is, an arrow $h_f \colon \mu F \to X$ such that the following diagram commutes:

$$\begin{tikzcd}
F(\mu F) \arrow[r, "F(h_f)"] \arrow[d, "\varphi"'] & F(X) \arrow[d, "f"] \\
\mu F \arrow[r, "h_f"'] & X
\end{tikzcd}.$$

We therefore introduce the following terms corresponding to φ and h_f, for each "functor" F:

(μ *Intro*) $$\emptyset \rhd \mathit{fold}_{\mu F} \colon F(\mu F) \to \mu F$$

(μ *Elim*) $$\emptyset \rhd \mathit{it}^{\tau}_{\mu F} \colon (F(\tau) \to \tau) \to \mu F \to \tau.$$

It has long been known that the initial algebra φ is actually an isomorphism; indeed, the arrow $h_{F(\varphi)}$ names its inverse (we will prove this below within the terms of $\lambda^{\mu\nu}$). Thus we do not need to add a term to $\lambda^{\mu\nu}$ which provides an inverse to $\mathit{fold}_{\mu F}$; however, to obtain a better simulation of primitive recursion it will prove useful to consider an expanded language $\lambda^{\mu\nu\prime}$ which also contains a family of unfolding operators:

(μ *Elim*$'$) $$\emptyset \rhd \mathit{unfold}_{\mu F} \colon \mu F \to F(\mu F).$$

This is discussed further in Section 5.

We have the dual situation for projective types. The interpretation of νF is a terminal F-coalgebra, *i.e.*, an arrow $\psi \colon \nu F \to F(\nu F)$ such that for every F-coalgebra $g \colon Y \to F(Y)$ there is a unique arrow $k_g \colon Y \to \nu F$ such that

$$\begin{tikzcd}
Y \arrow[r, "k_g"] \arrow[d, "g"'] & \nu F \arrow[d, "\psi"] \\
F(Y) \arrow[r, "F(k_g)"'] & F(\nu F)
\end{tikzcd}$$

commutes. The constants corresponding to ψ and k are

(ν *Elim*) $$\emptyset \rhd \mathit{unfold}_{\nu F} \colon \nu F \to F(\nu F)$$

(ν *Intro*) $$\emptyset \rhd \mathit{new}^{\tau}_{\nu F} \colon (\tau \to F(\tau)) \to \tau \to \nu F.$$

In the language $\lambda^{\mu\nu\prime}$ we also add the inverse to $\mathit{unfold}_{\nu F}$, just as we added the inverse for $\mathit{fold}_{\mu F}$ above:

(ν *Intro*$'$) $$\emptyset \rhd \mathit{fold}_{\nu F} \colon F(\nu F) \to \nu F.$$

To extend the category-theoretic intuition behind the terms of $\lambda^{\mu\nu}$, we will write the composition of two "arrows" (terms of function type) $M\colon\tau\to\upsilon$ and $N\colon\sigma\to\tau$ as $M\circ N\colon\sigma\to\upsilon$, which is an abbreviation for $(\lambda x\colon\sigma.\, M(Nx))$, where x does not occur free in either M or N. We will also make use of the identity term for this composition (up to provable equality, as described in the next section), defining $id^\sigma \equiv (\lambda x\colon\sigma.\, x)$, for each type σ.

We have already mentioned the interpretation of substitution in a type expression as being the application of a functor to an object; we may extend this effect in a natural way to define the application of a functor to an arrow (*i.e.*, a term). For our purposes, it will be more useful to apply a functor to an internal arrow, that is, a term of function type; because our intended model is a cartesian closed category (hence we have extensional finite products and function spaces) this is entirely equivalent to defining functor application on arbitrary terms and their typing contexts. Thus, given a functor F and a term $M\colon\sigma\to\tau$, we will produce a term $F(M)\colon F(\sigma)\to F(\tau)$. In addition, after the equations are introduced in the next section, we will be able to prove that $F(id) = id$ and $F(M\circ N) = F(M)\circ F(N)$, completely justifying our referring to F as a functor. The basic idea for this comes from Hagino [Hag87a, Hag87b], although his presentation is somewhat more difficult to read and treats only strictly positive functors.

The definition of $F(M)$ proceeds by cases on the structure of the body type $F(t)\equiv\sigma$:

- if $F(t) = \upsilon$, where t does not occur free in υ, then $F(M) = id^\upsilon$;
- if $F(t) = t$, then $F(M) = M$;
- if $F(t) = G(t) + H(t)$, then $F(M) = [\iota_1\circ G(M), \iota_2\circ H(M)]$;
- if $F(t) = G(t)\times H(t)$, then $F(M) = \lambda\langle x\colon G(\sigma), y\colon H(\sigma)\rangle.\, \langle G(M)x, H(M)y\rangle$;
- if $F(t) = G(t)\to H(t)$, then $F(M) = \lambda f\colon G(\sigma)\to H(\sigma).\, H(M)\circ f\circ G(M^{op})$ (see below);
- if $F(t) = \mu s.\, G(s,t)$, then $F(M) = it^{F(\tau)}_{\mu s.G(s,\sigma)}(fold_{\mu s.G(s,\tau)}\circ G(F(\tau), M))$;
- if $F(t) = \nu s.\, G(s,t)$, then $F(M) = new^{F(\sigma)}_{\nu s.G(s,\tau)}(G(F(\sigma), M)\circ unfold_{\nu s.G(s,\sigma)})$.

For the system as defined in this section, F will always be strictly positive, so the type expression $G(t)$ will not depend on t in the case $F(t) = G(t)\to H(t)$ above. To handle the language $\lambda^{\perp\rho}$, however, we include the mechanism for dealing with arbitrary covariant functors F. The metanotation M^{op} is meant to indicate a term of type $\tau\to\sigma$, the *opposite* of $M\colon\sigma\to\tau$. Since in general we have no way of forming an opposite term, the only rules we have are that opposite is an *anti-involution*, *i.e.*, the opposite of the opposite is the original term: $(M^{op})^{op}\equiv M$, and opposite is contravariant with respect to composition: $(M\circ N)^{op}\equiv N^{op}\circ M^{op}$ (from which it is easy to prove that also $id^{op}\equiv id$). In forming the subterm $G(M^{op})$, if G is contravariant then by using this rule we will only ever need M; if M^{op} appears in the fully expanded term $F(M)$, then F must not have been covariant.

3 Equational proof system for $\lambda^{\mu\nu}$

Now that we have introduced the syntax of $\lambda^{\mu\nu}$, it is time to give a formal semantics for the language, in the form of a set of equational axioms and proof rules. The categorical interpretation will continue to be our guide in describing the equations that hold between terms. We start by listing the usual structural rules necessary to have an equivalence relation that respects term formation, including adding and renaming variables:

(*ref*) $$\Gamma\rhd M = M : \sigma$$

(*sym*) $$\frac{\Gamma\rhd M = N : \sigma}{\Gamma\rhd N = M : \sigma}$$

(*trans*) $$\frac{\Gamma\rhd M = N : \sigma,\quad \Gamma\rhd N = P : \sigma}{\Gamma\rhd M = P : \sigma}$$

(*abs*) $$\frac{\Gamma, x{:}\,\sigma \triangleright M = N : \tau}{\Gamma \triangleright (\lambda x{:}\,\sigma.\, M) = (\lambda x{:}\,\sigma.\, N) : \sigma{\to}\tau}$$

(*app*) $$\frac{\Gamma \triangleright M = N : \sigma{\to}\tau, \quad \Gamma \triangleright P = Q : \sigma}{\Gamma \triangleright MP = NQ : \tau}$$

(*add var*) $$\frac{\Gamma \triangleright M = N : \sigma}{\Gamma, x{:}\,\tau \triangleright M = N : \sigma}$$

(α) $$\Gamma \triangleright (\lambda x{:}\,\sigma.\, M) = (\lambda y{:}\,\sigma.\, \{y/x\}M) : \sigma{\to}\tau, \text{ if } y \notin \mathrm{FV}(M).$$

The rest of the axioms and inference rules come in two forms; in terms of the categorical interpretations of the type constructors, the (β) axioms state that the arrows provided for the type make a particular diagram commute, while the (η) axioms and rules establish that those arrows do so uniquely. For example, the ($+\beta$) axioms assert that the mediating morphism $[M, N]{:}\, \sigma + \tau{\to}\upsilon$ properly factors the arrows $M{:}\, \sigma{\to}\upsilon$ and $N{:}\, \tau{\to}\upsilon$ through $\sigma + \tau$, while the ($+\eta$) axiom ensures that all terms with this property must be equal.

(0η) $$\Gamma \triangleright \Box^{\upsilon} = M : 0{\to}\upsilon.$$

($+\beta_1$) $$\Gamma \triangleright [M, N] \circ \iota_1 = M : \sigma{\to}\upsilon$$

($+\beta_2$) $$\Gamma \triangleright [M, N] \circ \iota_2 = N : \tau{\to}\upsilon$$

($+\eta$) $$\Gamma \triangleright [M \circ \iota_1, M \circ \iota_2] = M : \sigma + \tau{\to}\upsilon.$$

(1η) $$\Gamma \triangleright \Diamond = M : 1.$$

($\times\beta_1$) $$\Gamma \triangleright \pi_1\langle M, N\rangle = M : \sigma$$

($\times\beta_2$) $$\Gamma \triangleright \pi_2\langle M, N\rangle = N : \tau$$

($\times\eta$) $$\Gamma \triangleright \langle \pi_1 M, \pi_2 M\rangle = M : \sigma \times \tau$$

(${\to}\beta$) $$\Gamma \triangleright (\lambda x{:}\,\sigma.\, M)N = \{N/x\}M : \tau,$$

(${\to}\eta$) $$\Gamma \triangleright (\lambda x{:}\,\sigma.\, Mx) = M : \sigma{\to}\tau, \text{ for } x \notin \mathrm{FV}(M).$$

($\mu\beta$) $$\Gamma \triangleright (\mathit{it}^{\tau}_{\mu F}\, M) \circ \mathit{fold}_{\mu F} = M \circ F(\mathit{it}^{\tau}_{\mu F}\, M) : F(\mu F){\to}\tau,$$

($\mu\eta$) $$\frac{\Gamma \triangleright N \circ \mathit{fold}_{\mu F} = M \circ F(N) : F(\mu F){\to}\tau}{\Gamma \triangleright N = \mathit{it}^{\tau}_{\mu F}\, M : \mu F{\to}\tau.}$$

($\nu\beta$) $$\Gamma \triangleright \mathit{unfold}_{\nu F} \circ (\mathit{new}^{\tau}_{\nu F}\, M) = F(\mathit{new}^{\tau}_{\nu F}\, M) \circ M : \tau{\to}F(\nu F)$$

($\nu\eta$) $$\frac{\Gamma \triangleright \mathit{unfold}_{\nu F} \circ N = F(N) \circ M : \tau{\to}F(\nu F)}{\Gamma \triangleright N = \mathit{new}^{\tau}_{\nu F}\, M : \tau{\to}\nu F.}$$

For the extended language $\lambda^{\mu\nu\prime}$ we also have axioms asserting that the extra terms $\mathit{unfold}_{\mu F}$ and $\mathit{fold}_{\nu F}$ are inverses for $\mathit{fold}_{\mu F}$ and $\mathit{unfold}_{\nu F}$, respectively:

($\mu\beta'$) $$\Gamma \triangleright \mathit{unfold}_{\mu F} \circ \mathit{fold}_{\mu F} = \mathit{id}^{F(\mu F)} : F(\mu F){\to}F(\mu F)$$

($\mu\eta'$) $$\Gamma \triangleright \mathit{fold}_{\mu F} \circ \mathit{unfold}_{\mu F} = \mathit{id}^{\mu F} : \mu F{\to}\mu F$$

($\nu\beta'$) $$\Gamma \triangleright \mathit{unfold}_{\nu F} \circ \mathit{fold}_{\nu F} = \mathit{id}^{F(\nu F)} : F(\nu F){\to}F(\nu F)$$

($\nu\eta'$) $$\Gamma \triangleright \mathit{fold}_{\nu F} \circ \mathit{unfold}_{\nu F} = \mathit{id}^{\nu F} : \nu F{\to}\nu F$$

As promised in the previous section, we may now prove some lemmas about the behavior of our abbreviated terms.

Lemma 3.1 *Application of a "functor" F to a term preserves composition and identities, i.e., $F(M \circ N) = F(M) \circ F(N)$ and $F(id) = id$ (thus justifying use of the term functor for substitution in a type).*

Proof. By induction on the structure of F; we will only show a few of the more interesting cases. For the induction to go through, we actually need to prove more — namely, that this result can be extended to functors with more than one argument, *e.g.*, $G(M \circ N, P \circ Q) = G(M, P) \circ G(N, Q)$; this extension is quite easy and giving the full details would not add enough to the presentation to be worth the extra notation. We assume that $\Gamma \triangleright M\colon \tau \to \upsilon$ and $\Gamma \triangleright N\colon \sigma \to \tau$ are well-formed for some Γ.

- if $F(t) = G(t) + H(t)$, then

$$\begin{aligned} F(M) \circ F(N) &= [F(M) \circ F(N) \circ \iota_1, F(M) \circ F(N) \circ \iota_2] \\ &= [F(M) \circ \iota_1 \circ G(N), F(M) \circ \iota_2 \circ H(N)] \\ &= [\iota_1 \circ G(M) \circ G(N), \iota_2 \circ H(M) \circ H(N)] \\ &= [\iota_1 \circ G(M \circ N), \iota_2 \circ H(M \circ N)] \\ &= F(M \circ N); \end{aligned}$$

  also, $F(id^{\sigma}) = [\iota_1, \iota_2] = id^{F(\sigma)}$.

- if $F(t) = G(t) \to H(t)$, then

$$\begin{aligned} F(M \circ N) &= \lambda f\colon F(\sigma).\, H(M \circ N) \circ f \circ G((M \circ N)^{op}) \\ &= \lambda f\colon F(\sigma).\, H(M) \circ H(N) \circ f \circ G(N^{op}) \circ G(M^{op}) \\ &= \lambda f\colon F(\sigma).\, H(M) \circ (F(N) f) \circ G(M^{op}) \\ &= \lambda f\colon F(\sigma).\, F(M)(F(N) f) \\ &= F(M) \circ F(N); \end{aligned}$$

  for the identity, we have

$$\begin{aligned} F(id^{\sigma}) &= \lambda f\colon F(\sigma).\, H(id^{\sigma}) \circ f \circ G((id^{\sigma})^{op}) \\ &= \lambda f\colon F(\sigma).\ id^{H(\sigma)} \circ f \circ id^{G(\sigma)} \\ &= id^{F(\sigma)}\,. \end{aligned}$$

- if $F(t) = \mu s.\, G(s, t)$, then

$$\begin{aligned} & F(M) \circ F(N) \circ \mathit{fold}_{F(\sigma)} \\ = \ & F(M) \circ \mathit{fold}_{F(\tau)} \circ G(F(\tau), N) \circ G(F(N), \sigma) \\ = \ & \mathit{fold}_{F(\upsilon)} \circ G(F(\upsilon), M) \circ G(F(M), \tau) \circ G(F(\tau), N) \circ G(F(N), \sigma) \\ = \ & \mathit{fold}_{F(\upsilon)} \circ G(F(\upsilon), M) \circ G(F(\upsilon), N) \circ G(F(M), \sigma) \circ G(F(N), \sigma) \\ = \ & \mathit{fold}_{F(\upsilon)} \circ G(F(\upsilon), M \circ N) \circ G(F(M) \circ F(N), \sigma), \end{aligned}$$

  where the interchange $G(F(M), \tau) \circ G(F(\tau), N) = G(F(\upsilon), N) \circ G(F(M), \sigma)$ in the middle is possible because both sides are equal to $G(F(M), N)$, noticing that, *e.g.*, $G(F(M), \tau) = G(F(M), id^{F(\tau)})$ and using the multiple argument form of the induction hypothesis. From the above we may then use $(\mu\eta)$ to deduce that $F(M) \circ F(N) = it^{F(\upsilon)}_{F(\sigma)}(\mathit{fold}_{F(\upsilon)} \circ G(F(\upsilon), M \circ N)) = F(M \circ N)$. For the identity, we must show that $F(id^{\sigma}) = it^{F(\sigma)}_{F(\sigma)}\, \mathit{fold}_{F(\sigma)}$ is the identity; but $id^{F(\sigma)} \circ \mathit{fold}_{F(\sigma)} = \mathit{fold}_{F(\sigma)} \circ G(id^{F(\sigma)}, \sigma)$ by the induction hypothesis, so $id^{F(\sigma)} = it^{F(\sigma)}_{F(\sigma)}\, \mathit{fold}_{F(\sigma)}$ by $(\mu\eta)$.

■

We may now use this lemma about functors to prove the statement above that the terms corresponding to the arrows $h_{F(\varphi)}$ and $k_{F(\psi)}$ are inverses for $\mathit{fold}_{\mu F}$ and $\mathit{unfold}_{\nu F}$, respectively.

Lemma 3.2 *The term $it^{F(\mu F)}_{\mu F}\ F(\mathit{fold}_{\mu F})$ is an inverse for $\mathit{fold}_{\mu F}$, and $new^{F(\nu F)}_{\nu F}\ F(\mathit{unfold}_{\nu F})$ is an inverse for $\mathit{unfold}_{\nu F}$.*

Proof. For variety, we will show the proof for the projective type νF; the inductive case is quite similar. Let us refer to the term $\mathit{new}_{\nu F}^{F(\nu F)}\, F(\mathit{unfold}_{\nu F})$ as *refold*; then what we wish to show is that $\mathit{refold} \circ \mathit{unfold}_{\nu F} = \mathit{id}^{\nu F}$ and $\mathit{unfold}_{\nu F} \circ \mathit{refold} = \mathit{id}^{F(\nu F)}$. First note that $\mathit{unfold}_{\nu F} \circ \mathit{refold} = F(\mathit{refold}) \circ F(\mathit{unfold}_{\nu F}) = F(\mathit{refold} \circ \mathit{unfold}_{\nu F})$ by $(\nu\beta)$ and the previous lemma; thus the second equation follows from the first. Since $\mathit{unfold}_{\nu F} \circ \mathit{refold} \circ \mathit{unfold}_{\nu F} = F(\mathit{refold} \circ \mathit{unfold}_{\nu F}) \circ \mathit{unfold}_{\nu F}$, we may use the $(\nu\eta)$ rule to conclude that $\mathit{refold} \circ \mathit{unfold}_{\nu F} = \mathit{new}_{\nu F}^{\nu F}\, \mathit{unfold}_{\nu F}$. The right-hand side of this equation is equal to the identity, by using $(\nu\eta)$ again on the equation $\mathit{unfold}_{\nu F} \circ \mathit{id}^{\nu F} = F(\mathit{id}^{\nu F}) \circ \mathit{unfold}_{\nu F}$, hence we are done. ∎

4 The reduction system $\lambda_r^{\mu\nu}$

The reduction rules we will take for $\lambda_r^{\mu\nu}$ are essentially the (β) axioms of the previous section, oriented in the direction of "computation." Since we do not include any (η) rules, the form of some of the axioms will be changed to better match our notion of normal form — instead of dealing with functional terms and composition, we will apply such terms to dummy arguments to get rid of the $\circ$'s. Here are the reduction rules:

$$(+\beta_1)_r \qquad [M,N](\iota_1 P) \longrightarrow MP$$

$$(+\beta_2)_r \qquad [M,N](\iota_2 P) \longrightarrow NP$$

$$(\times\beta_1)_r \qquad \pi_1\langle M,N\rangle \longrightarrow M$$

$$(\times\beta_2)_r \qquad \pi_2\langle M,N\rangle \longrightarrow N$$

$$(\rightarrow\beta)_r \qquad (\lambda x{:}\,\sigma.\, M)N \longrightarrow \{N/x\}M$$

$$(\mu\beta)_r \qquad \mathit{it}_{\mu F}^{\tau}\, M(\mathit{fold}_{\mu F}\, P) \longrightarrow M(F(\mathit{it}_{\mu F}^{\tau}\, M)P)$$

$$(\nu\beta)_r \qquad \mathit{unfold}_{\nu F}(\mathit{new}_{\nu F}^{\tau}\, MP) \longrightarrow F(\mathit{new}_{\nu F}^{\tau}\, M)(MP).$$

For the language extended with $\mathit{unfold}_{\mu F}$ and $\mathit{fold}_{\nu F}$ we have two additional rules, producing the system $\lambda_r^{\mu\nu\prime}$:

$$(\mu\beta')_r \qquad \mathit{unfold}_{\mu F}(\mathit{fold}_{\mu F}\, P) \longrightarrow P$$

$$(\nu\beta')_r \qquad \mathit{unfold}_{\nu F}(\mathit{fold}_{\nu F}\, P) \longrightarrow P.$$

The first important theorems of this section are that $\lambda_r^{\mu\nu}$ is confluent and strongly normalizing. As a result, we will be able to speak of the unique normal form $\lambda_r^{\mu\nu}(M)$ of an arbitrary term M, independent of any specific reduction strategy.

Theorem 4.1 (Strong Normalization) *There is no infinite sequence of reductions $M_1 \longrightarrow M_2 \longrightarrow M_3 \longrightarrow \cdots$ in $\lambda_r^{\mu\nu}$.*

Proof. In the next section we give a translation of $\lambda^{\mu\nu}$ into Girard's System **F**, with the property that if $M \longrightarrow N$ in $\lambda_r^{\mu\nu}$ then $\overline{M} \longrightarrow^+ \overline{N}$ in **F**. Since **F** is strongly normalizing [GLT89], this establishes that $\lambda_r^{\mu\nu}$ is also, since otherwise we would be able to construct the infinite reduction sequence $\overline{M_1} \longrightarrow^+ \overline{M_2} \longrightarrow^+ \overline{M_3} \longrightarrow^+ \cdots$ in **F**. ∎

Theorem 4.2 (Confluence) *If $M \twoheadrightarrow N$ and $M \twoheadrightarrow P$, then there is a term Q such that $N \twoheadrightarrow Q$ and $P \twoheadrightarrow Q$, for reduction in $\lambda_r^{\mu\nu}$.*

Proof. We only need to show that $\lambda_r^{\mu\nu}$ is weakly confluent, *i.e.*, if $M \longrightarrow N$ and $M \longrightarrow P$ then N and P have a common reduct, since by a version of Newman's theorem [New42] the fact that it is also strongly normalizing implies that it is confluent. It is easy to see that $\lambda_r^{\mu\nu}$ is weakly confluent, since there are no critical pairs, hence we are done. ∎

Although we have not worked out the proofs in detail, it should be easy to show that $\lambda_r^{\mu\nu\prime}$ is also confluent and strongly normalizing. The proof of strong normalization is made somewhat clumsy by the fact that there is no obvious way of translating the extra reductions of $\lambda_r^{\mu\nu\prime}$ into System **F**.

The author's doctoral dissertation [How92] discusses in more detail the relation between the operational semantics $\lambda_r^{\mu\nu}$ and the equational proof system given in the previous section. Briefly, we first take the observable types to be 1, $\sigma + \tau$, $\sigma \times \tau$, and μF, where σ, τ, and $F(\upsilon)$ are all observable types (provided υ is observable). Then we define a *result* to be a closed normal form of observable type, and find that $\lambda_r^{\mu\nu}$ is adequate for computing results of programs:

Theorem 4.3 *If $\emptyset \triangleright P = R : \sigma$ is provable for R a result, then $P \longrightarrow\!\!\!\!\!\rightarrow R$ in $\lambda_r^{\mu\nu}$.*

Proof. It is sufficient to show that if $\emptyset \triangleright P \overset{\eta}{=}_1 Q \colon \sigma$ is provable, and $Q \longrightarrow\!\!\!\!\!\rightarrow R$ for R a result, then $P \longrightarrow\!\!\!\!\!\rightarrow R$, where $\overset{\eta}{=}_1$ means that exactly one (η) rule is used (along with whatever structural rules are needed). The theorem then follows by the normal form property of $\longrightarrow\!\!\!\!\!\rightarrow$ and induction on the number of (η) rules.

The problem then is to construct a reduction sequence $P \longrightarrow\!\!\!\!\!\rightarrow R$ from the given sequence $Q \longrightarrow\!\!\!\!\!\rightarrow R$. If the (η) rule is $(+\eta)$, $(\times\eta)$, or $(\rightarrow\eta)$, then this chiefly consists of mimicking the reduction from Q on P, either adding or deleting steps corresponding to the destruction of an (η) redex by a (β) reduction. One difficulty arises in these cases from the non-linearity of $(+\eta)$ and $(\times\eta)$ — if Q contains the subterm $[M \circ \iota_1, M \circ \iota_2]$ where P only has M, for example, then we must choose to follow the reduction on only the first, say, of the two components of the choice in reducing P; since R is a result and reduction is confluent, we may make this choice arbitrarily.

We are thus left with the case of the (η) step for one of the recursive types. We will consider $(\mu\eta)$ — the situation for $(\nu\eta)$ is entirely similar. If P contains a subterm M and Q contains $it^{\tau}_{\mu F}\, N$ in the same position, then in constructing the reduction from P we will need to use the hypothesis from the (η) rule, *i.e.*, $M \circ \mathit{fold}_{\mu F} = N \circ F(M)$. Now, for every $(\mu\beta)$ step in the original reduction of the form $it^{\tau}_{\mu F}\, N'(\mathit{fold}_{\mu F}\, K) \longrightarrow N'(F(it^{\tau}_{\mu F}\, N')K)$, we must replace it with the *equational* step $M'(\mathit{fold}_{\mu F}\, K) = N'(F(M')K)$, where M' and N' are corresponding residuals of M and N. We must then go back and apply the current theorem to convert this new equational proof of $P = R$ into a reduction $P \longrightarrow\!\!\!\!\!\rightarrow R$; we avoid circularity by noting that the height of the new proof tree for $P = R$ is shorter than before, measured in the number of nested applications of the $(\mu\eta)$ and $(\nu\eta)$ rules. In the situation where P contains $it^{\tau}_{\mu F}\, N$ and Q contains M we go through the same process, with the additional requirement that the reduction from Q must be rearranged so that if M is a λ-abstraction it will only be β-reduced when applied to arguments of the form $\mathit{fold}_{\mu F}\, K$. ■

Corollary 4.4 *If $\emptyset \triangleright P = Q : \sigma$ is provable at observable type σ, then $\lambda_r^{\mu\nu}(P) \equiv \lambda_r^{\mu\nu}(Q)$.*

Proof. Trivial, since reduction is confluent and strongly normalizing. ■

If we define *observational congruence* as $\Gamma \triangleright M \simeq N \colon \sigma$ if $\lambda_r^{\mu\nu}(\mathcal{P}[M]) \equiv \lambda_r^{\mu\nu}(\mathcal{P}[N])$ for every well-formed program context $\mathcal{P}[\ ]$, then we find that the full equational proof system, including the extensional rules, is sound for reasoning about programs:

Corollary 4.5 *The equational proof system for $\lambda^{\mu\nu}$ is sound for observational congruence.*

Proof. Follows directly from the previous corollary. ■

Therefore, we conclude that $\lambda_r^{\mu\nu}$ provides a suitable operational semantics for the language $\lambda^{\mu\nu}$.

Some examples of observable types are:

- $\mathit{bool} \equiv 1 + 1$, the booleans; the only results of this type are $\mathit{true} \equiv \iota_1\diamond$ and $\mathit{false} \equiv \iota_2\diamond$.
- $\mathit{nat} \equiv \mu t.\, 1 + t$, the set of natural numbers; results of this type take the form $\mathit{zero} \equiv \mathit{fold}(\iota_1\diamond)$ or $\mathit{succ}(n) \equiv \mathit{fold}(\iota_2 n)$, where n is another such result.
- $\mathit{natlist} \equiv \mu t.\, 1 + \mathit{nat} \times t$, the type of lists of natural numbers; the results of this type are of the form $\mathit{nil} \equiv \mathit{fold}(\iota_1\diamond)$ and $\mathit{cons}(n, \ell) \equiv \mathit{fold}(\iota_2\langle n, \ell\rangle)$.

5 Comparison with Systems T and F

The question naturally arises of what functions are expressible in $\lambda^{\mu\nu}$. We speak here of functions on the natural numbers, as given by the (observable) type *nat* described in the previous section. Since $\lambda^{\mu\nu}_r$ is strongly normalizing, all the functions must be total. We can enumerate the terms of $\lambda^{\mu\nu}$, therefore we must not be able to represent all of the total functions (since that set is not recursively enumerable). In this section we will see that all of the functions that are provably total in Peano arithmetic are definable, as well as some that are not. We will also show that $\lambda^{\mu\nu}_r$ is strongly normalizing by simulating it in System **F**; since the functions expressible in **F** are exactly those which are provably total in second-order arithmetic, we thus have a range in which the answer must fall:

Theorem 5.1 *The class of functions definable in $\lambda^{\mu\nu}$ properly includes those which are provably total in Peano arithmetic, and is included in the class of functions provably total in second-order arithmetic.*

The rest of this section will be devoted to the proof of this theorem.

5.1 Simulation of System T

To prove the first part of this theorem, we first show how to simulate Gödel's system **T** of primitive recursive functionals of finite type [Göd58] in $\lambda^{\mu\nu}$, as it is well-known that the functions expressible in **T** are precisely those which are provably total in Peano arithmetic. We have already seen how to represent the natural numbers with the type $nat \equiv \mu t.\, 1 + t$; the only difficulty is finding an appropriate term to implement a general recursor, since $\lambda^{\mu\nu}$ only provides an iterator. That is, we need a term $\mathbf{R}^\tau \colon \tau \to (\tau \to nat \to \tau) \to nat \to \tau$, for each type τ, such that

$$\begin{aligned} &\mathbf{R}^\tau\, a\, f\, 0 \twoheadrightarrow a \\ &\mathbf{R}^\tau\, a\, f\, (succ\, n) \twoheadrightarrow f(\mathbf{R}^\tau\, a\, f\, n)\, n. \end{aligned}$$

In $\lambda^{\mu\nu}$ we are only given an iterator — for the type *nat* we get a term $\mathbf{Z}^\tau \colon \tau \to (\tau \to \tau) \to nat \to \tau$ by defining $\mathbf{Z}^\tau\, a\, f \equiv it^\tau_{nat}[\lambda\Diamond.\, a, f]$; its reduction behavior is given by

$$\begin{aligned} &\mathbf{Z}^\tau\, a\, f\, 0 \twoheadrightarrow a \\ &\mathbf{Z}^\tau\, a\, f\, (succ\, n) \twoheadrightarrow f(\mathbf{Z}^\tau\, a\, f\, n). \end{aligned}$$

The extra argument n to the function f in the inductive case of the recursor is what differentiates the two; it allows the predecessor function to be defined with the recursor simply by $\mathbf{R}^{nat}\, 0\, (\lambda x\colon nat.\, \lambda y\colon nat.\, y)$.

Since we have product types in $\lambda^{\mu\nu}$, we can use a standard trick to simulate the recursor with the iterator — we define an auxilliary function which will return a pair consisting of the desired function result as well as the corresponding argument value, thus making the second argument to f available. That is, we may define the same function on natural numbers as given by $\mathbf{R}^\tau\, a\, f$ with the term $\pi_1 \circ \mathbf{Z}^\tau\, a'\, f'$, where $a' \equiv \langle a, 0\rangle$ and $f'\, x\, n \equiv \langle f\, x\, n, succ\, n\rangle$. Extensionally this produces the same function, but intensionally it is not the same algorithm as given by the recursor. To see this, construct the definition of the predecessor function using the iterator and observe that it takes n steps to build up the predecessor of $(succ\, n)$, whereas the recursor can find the answer in constant time.

In the extended language $\lambda^{\mu\nu\prime}$ we can avoid this mismatch by using $unfold_{nat}$ to define the predecessor: $pred = [\lambda\Diamond.\, 0, id] \circ unfold_{nat}$. Using this we may now define a recursor which has the same reduction behavior as $\mathbf{R}^\tau$, by essentially passing the needed second argument down from the top, instead of building it up from the bottom. If we define the term $\mathbf{R}^\tau_{it}\, a\, f \colon nat \to \tau$ to be

$$(\lambda n\colon nat.\, \mathbf{Z}^{nat\to\tau}(\lambda x\colon nat.\, a)(\lambda g\colon nat\to\tau.\, \lambda m\colon nat.\, f(g(pred\, m))(pred\, m))\, n\, n),$$

then we may prove the following theorem:

Theorem 5.2 *A function defined with the recursor $\mathbf{R}^\tau_{it}$ will have the same running time in $\lambda^{\mu\nu\prime}_r$ (within a constant factor) as the equivalent function defined with $\mathbf{R}^\tau$ will have in $\mathbf{T}$.*

Proof. All we need to show is that the two reduction rules for $\mathbf{R}^\tau$ in $\mathbf{T}$ can be performed in constant time in $\lambda_r^{\mu\nu\prime}$. We omit a number of steps showing the details of reducing $\mathbf{Z}$ and *pred*; it is easy to verify that the total number of steps remains independent of the size of the input. Also, we abbreviate the term $\mathbf{Z}^{nat\rightarrow\tau}(\lambda x\colon nat.\, a)(\lambda g\colon nat\rightarrow\tau.\, \lambda m\colon nat.\, f(g(pred\, m))(pred\, m))$ as $\mathbf{Q}^\tau_{a,f}$.

$$\begin{array}{rcl}
\mathbf{R}^\tau_{it}\, a\, f\, 0 & \longrightarrow & \mathbf{Q}^\tau_{a,f}\, 0\, 0 \\
& \twoheadrightarrow & (\lambda x\colon nat.\, a)\, 0 \\
& \longrightarrow & a
\end{array}$$

$$\begin{array}{rcl}
\mathbf{R}^\tau_{it}\, a\, f\, (succ\, p) & \longrightarrow & \mathbf{Q}^\tau_{a,f}(succ\, p)(succ\, p) \\
& \twoheadrightarrow & (\lambda g\colon nat\rightarrow\tau.\, \lambda m\colon nat.\, f(g(pred\, m))(pred\, m))(\mathbf{Q}^\tau_{a,f}\, p)(succ\, p) \\
& \longrightarrow & (\lambda m\colon nat.\, f(\mathbf{Q}^\tau_{a,f}\, p\, (pred\, m))(pred\, m))(succ\, p) \\
& \longrightarrow & f(\mathbf{Q}^\tau_{a,f}\, p\, (pred(succ\, p)))(pred(succ\, p)) \\
& \twoheadrightarrow & f(\mathbf{Q}^\tau_{a,f}\, p\, p)\, p.
\end{array}$$

This last term is one $(\rightarrow\beta)$ step from $f(\mathbf{R}^\tau_{it}\, a\, f\, p)\, p$, which corresponds to the term we would get after one reduction step from $\mathbf{R}^\tau\, a\, f\, (succ\, p)$ in $\mathbf{T}$. ∎

5.2 Iteration over large ordinals

We have already mentioned the relation between System $\mathbf{T}$ and the functions that are provably total in Peano arithmetic. Another characterization of the functions expressible in $\mathbf{T}$ is that they are the functions definable by transfinite recursion up to some ordinal $\alpha < \epsilon_0$, where ϵ_0 is the least ordinal ϵ such that $\epsilon = \omega^\epsilon$ (see [Kre59] or [Sch75]; a good text covering this subject is [Ros84]). We will show that $\lambda^{\mu\nu}$ is more expressive than $\mathbf{T}$, thus completing the proof of the first part of Theorem 5.1, by constructing the Hardy function H_{ϵ_0}, which requires iteration up to ϵ_0 itself [BW87]. The method we use to represent ordinal numbers and construct the hierarchy of Hardy functions is based on an example given by Coquand and Paulin [CP90].

To define H_α for $\alpha \le \epsilon_0$ we will need to choose a *fundamental sequence* for each limit ordinal $\le \epsilon_0$; that is, for each limit ordinal λ we need an increasing, natural number indexed sequence $\langle \lambda[0], \lambda[1], \ldots\rangle$ of ordinals less than λ whose limit is λ. A convenient choice makes use of the Cantor normal form, which writes each ordinal $\alpha < \epsilon_0$ uniquely as $\omega^{\alpha_1} + \cdots + \omega^{\alpha_k} + m$, for some natural numbers k and m and ordinals (themselves in normal form) $0 < \alpha_k \le \ldots \le \alpha_1 < \alpha$. If α is a limit ordinal, then we take the ordinal $\omega^{\alpha_1} + \cdots + \omega^{\alpha_k}[n]$ as the nth element of the fundamental sequence for α, where $\omega^{\beta+1}[n] = \omega^\beta \cdot n$ and $\omega^\lambda[n] = \omega^{\lambda[n]}$ for λ a limit. We extend this definition to ϵ_0 by taking $\epsilon_0[0] = 1$ and $\epsilon_0[n+1] = \omega^{\epsilon_0[n]}$. Now we may define the functions H_α as follows:

$$\begin{array}{rcl}
H_0(n) & = & n \\
H_{\alpha+1}(n) & = & H_\alpha(n+1) \\
H_\lambda(n) & = & H_{\lambda[n]}(n).
\end{array}$$

The type of notations for countable ordinals may be specified as the inductive type $ord \equiv \mu t.\, 1 + t + (nat\rightarrow t)$; the constructors are thus

$$\begin{array}{l}
ordzero \equiv fold_{ord} \circ \iota^3_1\colon 1\rightarrow ord \\
ordsucc \equiv fold_{ord} \circ \iota^3_2\colon ord\rightarrow ord \\
lim \equiv fold_{ord} \circ \iota^3_3\colon (nat\rightarrow ord)\rightarrow ord.
\end{array}$$

(Although we have not formally introduced them, the constants ι^n_k are the natural extensions of the binary injection functions to the case of n-ary sums; they may be declared as syntactic sugar for the obvious definition in terms of the binary case, depending only on whether $+$ is taken to associate to the right or the left.) The interpretation of *lim* is that it creates an ordinal given a function which specifies the fundamental sequence for the ordinal; for example, we may define $\omega \equiv lim\ inord$, where $inord \equiv it^{ord}_{nat}[ordzero, ordsucc]$

is the natural injection from *nat* to *ord*, since $\langle 0, 1, 2, \ldots \rangle$ is the fundamental sequence for ω. Addition, multiplication, and exponentiation of ordinals may be defined as follows:

$$\begin{aligned}
\mathit{ordplus} &\equiv (\lambda\alpha{:}\, \mathit{ord}.\ \mathit{it}^{\mathit{ord}}_{\mathit{ord}}[(\lambda\Diamond.\, \alpha), \mathit{ordsucc}, \mathit{lim}])\\
\mathit{ordtimes} &\equiv (\lambda\alpha{:}\, \mathit{ord}.\ \mathit{it}^{\mathit{ord}}_{\mathit{ord}}[\mathit{ordzero}, (\lambda\beta{:}\, \mathit{ord}.\ \mathit{ordplus}\, \beta\alpha), \mathit{lim}])\\
\mathit{ordexp} &\equiv (\lambda\alpha{:}\, \mathit{ord}.\ \mathit{it}^{\mathit{ord}}_{\mathit{ord}}[\mathit{ordone}, (\lambda\beta{:}\, \mathit{ord}.\ \mathit{ordtimes}\, \beta\alpha), \mathit{lim}]),
\end{aligned}$$

where $\mathit{ordone} \equiv \mathit{ordsucc} \circ \mathit{ordzero}$. A function that creates an exponential stack of n ω's when applied to a natural number n is $\mathit{omegaexp} \equiv \mathit{it}^{\mathit{ord}}_{\mathit{nat}}[\mathit{ordone}, \mathit{ordexp}\, \omega]$; we may thus define a notation for ϵ_0 by stating $\epsilon_0 \equiv \mathit{lim}\ \mathit{omegaexp}$.

The following term represents the Hardy function $H{:}\, \mathit{ord}{\rightarrow}\mathit{nat}{\rightarrow}\mathit{nat}$ in $\lambda^{\mu\nu}$:

$$\mathit{it}^{\mathit{nat}\rightarrow\mathit{nat}}_{\mathit{ord}}[(\lambda\Diamond.\ \mathit{id}^{\mathit{nat}}), (\lambda f{:}\, \mathit{nat}{\rightarrow}\mathit{nat}.\, f \circ \mathit{succ}), (\lambda g{:}\, \mathit{nat}{\rightarrow}\mathit{nat}{\rightarrow}\mathit{nat}.\ \lambda n{:}\, \mathit{nat}.\ gnn)].$$

We will demonstrate the use of these definitions by evaluating $H_{\epsilon_0}(0)$:

$$\begin{aligned}
H\epsilon_0\, 0 &\equiv H(\mathit{lim}\ \mathit{omegaexp})\, 0\\
&\twoheadrightarrow (H \circ \mathit{omegaexp})\, 0\, 0\\
&\twoheadrightarrow H(\mathit{ordone}\Diamond)\, 0\\
&\equiv H(\mathit{ordsucc}(\mathit{ordzero}\Diamond))\, 0\\
&\twoheadrightarrow ((H(\mathit{ordzero}\Diamond)) \circ \mathit{succ})\, 0\\
&\twoheadrightarrow (\mathit{id}^{\mathit{nat}} \circ \mathit{succ})\, 0\\
&\twoheadrightarrow \mathit{succ}\, 0.
\end{aligned}$$

Being able to construct H_{ϵ_0} is sufficient for showing that $\lambda^{\mu\nu}$ can express more than the primitive recursive functionals of **T**, but there is no reason to stop at ϵ_0. Indeed, since we may define the Veblen hierarchy of functions $\varphi_\alpha{:}\, \mathit{ord}{\rightarrow}\mathit{ord}$ for all $\alpha{:}\, \mathit{ord}$, we have a system of notation for all the ordinals less than Γ_0, the first "strongly critical" ordinal (see [Gal91] for a very readable discussion of the significance of Γ_0). The particular Veblen hierarchy to which we refer is that starting from $\varphi_0(\beta) = \omega^\beta$; then the function φ_α for $\alpha > 0$ enumerates the common fixed points of all the functions φ_γ for $\gamma < \alpha$. For example, φ_1 enumerates the fixed points of $\varphi_0 = \lambda\beta.\, \omega^\beta$; these are known as the epsilon numbers, and indeed the first fixed point $\varphi_1(0)$ is the ordinal ϵ_0 discussed above. We will not give the term which computes φ here, but a detailed description of how to define it in terms of fundamental sequences is given in [CW83].

The ordinal Γ_0 is still not the largest ordinal we can express in $\lambda^{\mu\nu}$. As Miller shows in [Mil76], we may extend the definition of φ_α to uncountable ordinals α that satisfy certain conditions. For example, φ_Ω is the function which enumerates the strongly critical ordinals (so $\varphi_\Omega(0) = \Gamma_0$), where Ω is the least uncountable ordinal. We can express Ω, and many other uncountable ordinals, in $\lambda^{\mu\nu}$ by introducing the type $\mathit{ord}_1 \equiv \mu t.\, 1 + t + (\mathit{nat}{\rightarrow}t) + (\mathit{ord}{\rightarrow}t)$. As for *ord*, the first three components of the body of the recursive type represent zero, successor ordinals, and (*nat*-indexed) limit ordinals. But ord_1 also has a fourth component which allows *ord*-indexed limits. If we define $\mathit{lim}^1_1 \equiv \mathit{fold}_{\mathit{ord}_1} \circ \iota^4_4{:}\, (\mathit{ord}{\rightarrow}\mathit{ord}_1){\rightarrow}\mathit{ord}_1$, and make the obvious definition for $\mathit{inord}_1{:}\, \mathit{ord}{\rightarrow}\mathit{ord}_1$, then we may set $\Omega \equiv \mathit{lim}^1_1\, \mathit{inord}_1$. We may then define the usual arithmetic operations on ord_1, allowing the construction of such ordinals as Ω^2, Ω^Ω, and even $\epsilon_{\Omega+1} = \Omega^{\Omega^{\cdot^{\cdot^{\cdot}}}}$, with which we may go back and construct the countable (but *very* large) ordinal $\varphi_{\epsilon_{\Omega+1}+1}(0)$, known as "Howard's ordinal."[1]

We may make one more step in the production of ever-larger ordinals. By generalizing the construction of the types *ord* and ord_1, we may construct the class ord_n of *abstract tree ordinals* (see [Wai89]), all of which will have cardinality $\aleph_n$, by the inductive type $\mu t.\, 1 + t + (\mathit{nat}{\rightarrow}t) + (\mathit{ord}{\rightarrow}t) + (\mathit{ord}_1{\rightarrow}t) + \cdots + (\mathit{ord}_{n-1}{\rightarrow}t)$. Elements of this type may be defined as limits of order type up to Ω_n, the nth regular ordinal beyond $\Omega_0 \equiv \omega$. Further discussion of these ordinals is far beyond the scope of this paper; for more details see for example [Mil76, Wai89].

[1]No relation to the author.

5.3 Simulation in System F

The second part of Theorem 5.1 follows from a translation of $\lambda^{\mu\nu}$ into Girard's system **F** [Gir71, GLT89, Rey74]. The essential part of this translation is the well-known representation of finite sums and products and initial and terminal fixed points in **F** (see for example [GLT89] or [Has89]), although the details of the translation for inductive and projective types are original. We have already noted that the functions expressible in **F** are precisely those that are provably total in second order arithmetic, so the fact that all the functions computable by $\lambda^{\mu\nu}_{\tau}$ are also computable in **F** gives us an upper limit on the expressibility of $\lambda^{\mu\nu}$. We suspect that the inclusion is proper, although we do not know an example of a function computable in **F** and not in $\lambda^{\mu\nu}_{\tau}$. Interestingly, there are *algorithms* computable in $\lambda^{\mu\nu\prime}_{\tau}$ which do not seem to be computable in **F** — the simplest example is the constant time predecessor. This is a symptom of a more general lack in System **F**, namely, that types such as products, sums, and least fixed points are not extensional; as a result, many desirable equations between terms are not provable.

System **F** extends the simply typed lambda calculus $\lambda^{\rightarrow}$ with type variables and the polymorphic type $\forall t.\, \sigma$. Formally, we add the following term formation rules:

$$(\forall\ \textit{Intro}) \qquad \frac{\Gamma \triangleright M\!:\sigma}{\Gamma \triangleright (\Lambda t.\, M)\!: \forall t.\, \sigma} \text{ for } t \text{ not free in } \Gamma$$

$$(\forall\ \textit{Elim}) \qquad \frac{\Gamma \triangleright M\!: \forall t.\, \sigma}{\Gamma \triangleright M\tau\!: \{\tau/t\}\sigma.}$$

The metavariables σ and τ now refer to arbitrary type expressions formed with $\rightarrow$ and $\forall$; that is, they may contain free type variables. The type abstraction operator Λ binds type variables in its body in the same manner as λ binds regular variables, thus we need equational rules for Λ analogous to those for λ:

$$(\forall\ \textit{abs}) \qquad \frac{\Gamma \triangleright M = N : \sigma}{\Gamma \triangleright (\Lambda t.\, M) = (\Lambda t.\, N) : \forall t.\, \sigma}$$

$$(\forall\ \textit{app}) \qquad \frac{\Gamma \triangleright M = N : \forall t.\, \sigma}{\Gamma \triangleright M\tau = N\tau : \{\tau/t\}\sigma}$$

$$(\forall\alpha) \qquad \Gamma \triangleright (\Lambda t.\, M) = (\Lambda s.\, \{s/t\}M) : \forall t.\, \sigma, \text{ if } s \text{ not free in } M$$

$$(\forall\beta) \qquad \Gamma \triangleright (\Lambda t.\, M)\tau = \{\tau/t\}M : \{\tau/t\}\sigma$$

$$(\forall\eta) \qquad \Gamma \triangleright (\Lambda t.\, Mt) = M : \forall t.\, \sigma, \text{ for } t \text{ not free in } M.$$

As usual, the reduction relation obtained by directing the $(\rightarrow\beta)$ and $(\forall\beta)$ axioms from left to right will be denoted $\longrightarrow$; if a term M reduces to another term N in one or more steps, then we write $M \longrightarrow^{+} N$.

We will now start to give a translation from $\lambda^{\mu\nu}$ into **F**, such that if $\Gamma \triangleright M\!:\sigma$ is a well-formed term in $\lambda^{\mu\nu}$, then $\overline{\Gamma} \triangleright \overline{M}\!:\overline{\sigma}$ is a well-formed term of **F**. Type and term variables will stay the same under translation, as will lambda abstraction and application. The translation of the binary sum type $\sigma + \tau$ will be $\forall t.\, (\overline{\sigma}\rightarrow t)\rightarrow(\overline{\tau}\rightarrow t)\rightarrow t$, where t is not free in $\overline{\sigma}$ or $\overline{\tau}$; the empty type 0 is represented by $\forall t.\, t$. Dually, the binary product $\sigma \times \tau$ translates to $\forall t.\, (\overline{\sigma}\rightarrow\overline{\tau}\rightarrow t)\rightarrow t$, while the singleton type 1 becomes $\forall t.\, t\rightarrow t$. The translation of terms for these types is given in Table 1, where of course it is understood that all of the variables introduced on the right hand side are new.

Before we give the translation for the inductive and projective types, we will need a few abbreviations. It will be convenient to continue our practice of treating a type expression σ as a functor with respect to substitution for a type variable t, thus we will write $\overline{F}(\overline{\tau})$ to mean $\{\overline{\tau}/t\}\overline{\sigma}$; it is an easy exercise to show that $\overline{F(\tau)} \equiv \overline{F}(\overline{\tau})$. Similarly, if M is a term of type $\tau\rightarrow\upsilon$, then we write $\overline{F}(\overline{M})$ for the term $\overline{F(M)}$ of type $\overline{F(\tau)}\rightarrow\overline{F(\upsilon)}$. The "functor" $\overline{F}$ will no longer preserve composition or identities, since most types under the translation into **F** are no longer extensional (see below), but it will suffice for the purposes of this section because we are only interested in showing that the β reductions of $\lambda^{\mu\nu}$ are strongly normalizing.

We will also need the existentially quantified type $\exists t.\, \sigma$. This may be expressed in terms of $\forall$ and $\rightarrow$ as $\forall s.\, (\forall t.\, \sigma\rightarrow s)\rightarrow s$, where s is a fresh type variable. A term of the existential type $\exists t.\, \sigma$ is like a pair $\langle\tau, M\rangle$

$M{:}\,\sigma$	$\overline{M}{:}\,\overline{\sigma}$
$x{:}\,\sigma$	$x{:}\,\overline{\sigma}$
$(\lambda x{:}\,\sigma.\,M){:}\,\sigma\to\tau$	$(\lambda x{:}\,\overline{\sigma}.\,\overline{M}){:}\,\overline{\sigma}\to\overline{\tau}$
$MN{:}\,\tau$	$\overline{M}\,\overline{N}{:}\,\overline{\tau}$
$\iota_1{:}\,\sigma\to\sigma+\tau$	$(\lambda x{:}\,\overline{\sigma}.\,\Lambda t.\,\lambda f{:}\,\overline{\sigma}\to t.\,\lambda g{:}\,\overline{\tau}\to t.\,fx){:}\,\overline{\sigma}\to\overline{\sigma+\tau}$
$\iota_2{:}\,\tau\to\sigma+\tau$	$(\lambda x{:}\,\overline{\tau}.\,\Lambda t.\,\lambda f{:}\,\overline{\sigma}\to t.\,\lambda g{:}\,\overline{\tau}\to t.\,gx){:}\,\overline{\tau}\to\overline{\sigma+\tau}$
$[M,N]{:}\,\sigma+\tau\to\upsilon$	$(\lambda x{:}\,\overline{\sigma+\tau}.\,x\overline{\upsilon}\,\overline{M}\,\overline{N}){:}\,\overline{\sigma+\tau}\to\overline{\upsilon}$
$\square^{\upsilon}{:}\,0\to\upsilon$	$(\lambda x{:}\,\overline{0}.\,x\overline{\upsilon}){:}\,\overline{0}\to\overline{\upsilon}$
$\pi_1{:}\,\sigma\times\tau\to\sigma$	$(\lambda x{:}\,\overline{\sigma\times\tau}.\,x\overline{\sigma}(\lambda y{:}\,\overline{\sigma}.\,\lambda z{:}\,\overline{\tau}.\,y)){:}\,\overline{\sigma\times\tau}\to\overline{\sigma}$
$\pi_2{:}\,\sigma\times\tau\to\tau$	$(\lambda x{:}\,\overline{\sigma\times\tau}.\,x\overline{\tau}(\lambda y{:}\,\overline{\sigma}.\,\lambda z{:}\,\overline{\tau}.\,z)){:}\,\overline{\sigma\times\tau}\to\overline{\tau}$
$\langle M,N\rangle{:}\,\sigma\times\tau$	$(\Lambda t.\,\lambda f{:}\,\overline{\sigma}\to\overline{\tau}\to t.\,f\overline{M}\,\overline{N}){:}\,\overline{\sigma\times\tau}$
$\Diamond{:}\,1$	$(\Lambda t.\,\lambda x{:}\,t.\,x){:}\,\overline{1}$

Table 1: Translation of function, sum, and product terms

of a type τ and a term M of type $\{\tau/t\}\sigma$; we will exploit this analogy by introducing (as in [GLT89]) the syntactic sugar

$$\begin{aligned} \langle\tau,N\rangle &\equiv (\Lambda s.\,\lambda x{:}\,\forall t.\,\sigma\to s.\,x\tau N){:}\,\exists t.\,\sigma \\ (\lambda\langle t,x{:}\,\sigma\rangle.\,M) &\equiv (\lambda y{:}\,\exists t.\,\sigma.\,y\upsilon(\Lambda t.\,\lambda x{:}\,\sigma.\,M)){:}\,(\exists t.\,\sigma)\to\upsilon, \end{aligned}$$

where in the latter definition the term M has type υ. It will be convenient to have this pattern matching syntax for terms of (translated) product type as well; thus, the **F** term $(\lambda\langle x{:}\,\sigma,y{:}\,\tau\rangle.\,M)$ will be an abbreviation for

$$(\lambda p{:}\,\forall t.\,(\sigma\to\tau\to t)\to t.\,pv(\lambda x{:}\,\sigma.\,\lambda y{:}\,\tau.\,M).$$

If we also write $\langle N,P\rangle$ for $(\Lambda t.\,\lambda f{:}\,\sigma\to\tau\to t.\,f\,N\,P)$, then it is easy to see that both of these pattern matching terms behave as expected under reduction, *i.e.*,

$$\begin{aligned} (\lambda\langle t,x{:}\,\sigma\rangle.\,M)\langle\tau,N\rangle &\longrightarrow^{+} \{N/x\}\{\tau/t\}M \\ (\lambda\langle x{:}\,\sigma,y{:}\,\tau\rangle.\,M)\langle N,P\rangle &\longrightarrow^{+} \{P/y\}\{N/x\}M. \end{aligned}$$

The general translation for an inductive type $\mu t.\,\sigma$ was essentially given by Böhm and Berarducci [BB85], although they were mainly concerned with representing iteratively defined functions over the term algebra of an algebraic signature, touching only briefly on iteration at higher types. The corresponding translation for a projective type $\nu t.\,\sigma$ was given independently by Hasegawa [Has89] and Wraith [Wra89]. Here are the translations for the types:

$$\begin{aligned} \overline{\mu t.\,\sigma} &\equiv \forall t.\,(\overline{\sigma}\to t)\to t \\ \overline{\nu t.\,\sigma} &\equiv \exists t.\,\overline{(t\to\sigma)\times t} \\ &\equiv \exists t.\,\forall u.\,((t\to\overline{\sigma})\to t\to u)\to u \end{aligned}$$

The translations of the terms for these types are given in Table 2.

It is now a simple matter to verify that this translation preserves reduction in $\lambda_r^{\mu\nu}$.

Lemma 5.3 *If $M\longrightarrow N$ in $\lambda_r^{\mu\nu}$, then $\overline{M}\longrightarrow^{+}\overline{N}$.*

Proof. We will show two of the cases; the rest of the proof is entirely similar. We omit most types for brevity.

$M\colon\sigma$	$\overline{M}\colon\overline{\sigma}$
$\mathit{fold}_{\mu F}$	$(\lambda x\colon \overline{F}(\overline{\mu F}).\, \Lambda t.\, \lambda f\colon \overline{F}(t)\to t.\, f(\overline{F}(\overline{it}\,t\,f)\,x))\colon \overline{F}(\overline{\mu F})\to\overline{\mu F}$
$\mathit{it}^{\tau}_{\mu F}$	$\overline{it}\,\overline{\tau} \equiv (\Lambda s.\, \lambda f\colon \overline{F}(s)\to s.\, \lambda x\colon \overline{\mu F}.\, x\,s\,f)\overline{\tau}\colon (\overline{F}(\overline{\tau})\to\overline{\tau})\to\overline{\mu F}\to\overline{\tau}$
$\mathit{unfold}_{\nu F}$	$(\lambda\langle t, \langle f\colon t\to\overline{\sigma}, x\colon t\rangle\rangle.\, \overline{F}(\overline{new}\,t\,f)(f\,x))\colon \overline{\nu F}\to\overline{F}(\overline{\nu F})$
$\mathit{new}^{\tau}_{\nu F}$	$\overline{new}\,\overline{\tau} \equiv (\Lambda s.\, \lambda f\colon s\to\overline{F}(s).\, \lambda x\colon s.\, \langle s, \langle f, x\rangle\rangle)\overline{\tau}\colon (\overline{\tau}\to\overline{F}(\overline{\tau}))\to\overline{\tau}\to\overline{\nu F}$

Table 2: Translation of inductive and projective terms

$$\begin{array}{rcl}
\overline{[M,N](\iota_1 P)} & \equiv & (\lambda x.\, x\overline{v}\,\overline{M}\,\overline{N})((\lambda y.\, \Lambda t.\, \lambda f.\, \lambda g.\, f\,y)\,\overline{P}) \\
& \longrightarrow^{+} & (\Lambda t.\, \lambda f.\, \lambda g.\, f\,\overline{P})\,\overline{v}\,\overline{M}\,\overline{N} \\
& \longrightarrow^{+} & \overline{M}\,\overline{P} \\
& \equiv & \overline{MP} \\
\overline{\mathit{unfold}(\mathit{new}^{\tau}\,M\,N)} & \equiv & (\lambda\langle t, \langle f, x\rangle\rangle.\, \overline{F}(\overline{new}\,t\,f)(f\,x)) \\
& & \quad((\Lambda s.\, \lambda g.\, \lambda y.\, \langle s, \langle g, y\rangle\rangle)\,\overline{\tau}\,\overline{M}\,\overline{N}) \\
& \longrightarrow^{+} & (\lambda\langle t, \langle f, x\rangle\rangle.\, \overline{F}(\overline{new}\,t\,f)(f\,x))\langle\overline{\tau}, \langle\overline{M}, \overline{N}\rangle\rangle \\
& \longrightarrow^{+} & \overline{F}(\overline{new}\,\overline{\tau}\,\overline{M})(\overline{M}\,\overline{N}) \\
& \equiv & \overline{F(\mathit{new}^{\tau}\,M)(MN)}
\end{array}$$

■

This lemma is precisely what we needed to complete the proof of strong normalization for $\lambda^{\mu\nu}_{r}$, since if there were an infinite reduction sequence from a term M in $\lambda^{\mu\nu}_{r}$ then we would be able to construct an infinite reduction from $\overline{M}$ in **F**. System **F** is strongly normalizing (see [GLT89], for example), so we are done.

6 Syntax of the language $\lambda^{\perp\rho}$

Now we describe the language $\lambda^{\perp\rho}$, which can handle mixed-variant recursive type expressions at the cost of introducing the potential for non-termination. We start by adding two type constructors to $\lambda^{\mu\nu}$: the *lifted* type $\sigma_{\perp}$, which corresponds to the operation of adding a bottom element to a cpo, and the *retractive* type $\rho t.\, \sigma$, which corresponds to the recursive type found by the usual Smyth-Plotkin construction [SP82]. Note that, whereas the types of $\lambda^{\mu\nu}$ could be thought of as sets, the motivating example for the types of $\lambda^{\perp\rho}$ is the category **CPO** of complete partial orders (not necessarily with least elements). A more general categorical setting in which to find examples is the class of algebraically complete **CPO**-categories, as described by Freyd in [Fre90, Fre91, Fre92]; we differ from Freyd in considering cpo's that may not have least elements, hence our **CPO**-categories include his as the special case where everything is pointed. In this setting, we may view the retractive types as being constructed in the algebraically compact subcategory of pointed objects and strict maps; we will write more about this connection in a future paper.

Not all type expressions σ may appear as the body of a retractive type, just as the bodies of inductive and projective types were restricted in the previous chapter to type expressions strictly positive in the type variable being bound. To state the restriction for retractive types we need to formally introduce the concept of a *pointed* type. Intuitively, a pointed type is one which contains a bottom element, *i.e.*, a least element with respect to the information-content ordering on the type. Following our cpo interpretation of the type constructors, we may see that the types 1 and $\sigma_{\perp}$ are always pointed; it will also turn out that the retractive type $\rho t.\, \sigma$ is always pointed. If the types σ and τ are both pointed, then the product $\sigma \times \tau$ will be pointed, since the pair consisting of the bottom elements of σ and τ will be the least element under the pointwise ordering of $\sigma \times \tau$. Similarly, if τ is pointed, then the function type $\sigma\to\tau$ will be pointed for any σ, since the constant bottom function is less defined than any other element of $\sigma\to\tau$. The empty type 0 and the disjoint sum $\sigma + \tau$ will never be pointed; we will not consider a type such as $\sigma_{\perp} + 0$ to be pointed, despite the fact that its cpo representation has a least element, since an element of a sum type necessarily conveys at least the information that it comes from one or the other summand.

Since a recursive type $\mu t.\,\sigma$ (or, mutatis mutandis, $\nu t.\,\sigma$ or $\rho t.\,\sigma$) is isomorphic to the unfolded type $\{\mu t.\,\sigma/t\}\sigma$, it is reasonable to consider a recursive type to be pointed whenever the body σ is. We have two choices when defining the pointedness of type expressions with free variables. If σ is pointed no matter what types are substituted for the free variables, then it is said to be *unconditionally pointed*. If the pointedness of σ depends on that of a free variable t, as for example in $\tau\to t$, then it is *conditionally pointed with respect to* t. Thus, our rule for inductive types will be that $\mu t.\,\sigma$ is pointed if σ is unconditionally pointed. We may go further with projective and retractive types, saying that $\nu t.\,\sigma$ and $\rho t.\,\sigma$ will be pointed if σ is conditionally pointed with respect to t. The reason for this difference comes from the respective constructions of the recursive types in **CPO**: an inductive type μF essentially results from an infinite number of applications of the functor F to the initial object 0, while the projective and retractive types start from 1. If F is only conditionally pointed, *i.e.*, $F(\tau)$ is only pointed if τ is, then none of the finite approximations 0, $F(0)$, $F^2(0)$, $\ldots$, to μF will be pointed; by continuity we thus expect that μF itself will not be pointed. By contrast, all of the approximations 1, $F(1)$, $F^2(1)$, $\ldots$, to νF and ρF are pointed, so the limit types will be as well. Finally, for a retractive type $\rho t.\,\sigma$ to be well-formed, t may be of mixed variance in σ, but σ must be conditionally pointed with respect to t.

For a retractive type ρF, we have terms which give the two directions of the isomorphism $\rho F \simeq F(\rho F)$:

(ρ *Intro*) $$\emptyset \triangleright \mathit{fold}_{\rho F}\colon F(\rho F)\to\rho F$$

(ρ *Elim*) $$\emptyset \triangleright \mathit{unfold}_{\rho F}\colon \rho F\to F(\rho F).$$

If σ is a pointed type, then we may use these terms to define a least fixed point operator $\mathit{fix}^{\sigma}\colon(\sigma\to\sigma)\to\sigma$:

$$(\lambda x\colon\upsilon.\,\lambda f\colon\sigma\to\sigma.\,f(\mathit{unfold}_{\upsilon}\,xxf))(\mathit{fold}_{\upsilon}(\lambda x\colon\upsilon.\,\lambda f\colon\sigma\to\sigma.\,f(\mathit{unfold}_{\upsilon}\,xxf))),$$

where $\upsilon \equiv \rho t.\,(t\to(\sigma\to\sigma)\to\sigma)$. This is essentially a typed version of Turing's combinator Θ (see [Bar84], for example), where the retractive type allows the self-application of x to be typed.

We motivate the terms for the lifted type $\sigma_\bot$ by considering the left adjoint L to the forgetful functor U into **CPO** from the subcategory $\mathbf{CPPO}_\bot$ of pointed cpo's and strict continuous functions. Lifting arises from this adjunction by taking the interpretation of $\sigma_\bot$ to be the application of the endofunctor UL to the object corresponding to σ; the effect of this is to add a bottom element to σ. One way of describing the adjunction $L \dashv U$ is by giving a natural transformation $\eta\colon\mathbf{CPO}\overset{.}{\to}UL$, the *unit* of the adjunction, such that each arrow $\eta_X\colon X\to UL(X)$ is universal from X to U, *i.e.*, for any other arrow $f\colon X\to U(Y)$ there is a unique arrow $g\colon L(X)\to Y$ such that $f = U(g)\circ\eta_X$ (see [Mac71], for example). If we have a judgement $\triangleright\,\mathtt{ptd}\,\sigma$ which asserts that σ is a pointed type, then we obtain the following term judgement rules:

(*bottom*) $$\frac{\triangleright\,\mathtt{ptd}\,\sigma}{\emptyset\triangleright\bot^{\sigma}\colon\sigma.}$$

($\bot$ *Intro*) $$\frac{\Gamma\triangleright M\colon\sigma}{\Gamma\triangleright\lfloor M\rfloor\colon\sigma_\bot.}$$

($\bot$ *Elim*) $$\frac{\Gamma, x\colon\sigma\triangleright M\colon\tau,\quad \triangleright\,\mathtt{ptd}\,\tau}{\Gamma\triangleright(\lambda\lfloor x\colon\sigma\rfloor.\,M)\colon\sigma_\bot\to\tau.}$$

In comparing our language to Moggi's Computational Lambda Calculus, we note that our treatment of lifting differs from Moggi's in two important respects. First, because we are dealing with the specific operation of lifting instead of an arbitrary monad of computations, we will get more terms and more provable equations. Specifically, by introducing strict functions and defining lifting as an adjunction, we will find some equations that hold for all pointed types, whereas if we only defined lifting as a monad they would only hold for lifted types. The second difference is that in Moggi's system, all functions return values of the computation type; we require lifting to be indicated more explicitly, so that functions are expressible which do not represent computations of the lifted type (indeed, the entire sublanguage $\lambda^{\mu\nu}$ is concerned with defining functions which do not return lifted types, because they always terminate).

7 Equational proof system for $\lambda^{\perp\rho}$

We will extend the proof system given earlier for $\lambda^{\mu\nu}$ to provide a formal semantics for the types and terms just introduced. Because $\lambda^{\perp\rho}$ can express all partial recursive functions on the natural numbers, we cannot hope to obtain a complete proof system. We will only give a fairly weak system here; a stronger system involving approximation orderings and fixed point induction is given in the author's thesis [How92].

First we must add a congruence rule for the strict abstraction, since it is a new variable-binding operation:

$$(\mathit{str\ \ abs}) \qquad \frac{\Gamma, x{:}\,\sigma \rhd M = N : \tau}{\Gamma \rhd (\lambda\lfloor x{:}\,\sigma\rfloor.\, M) = (\lambda\lfloor x{:}\,\sigma\rfloor.\, N) : \sigma_\perp \to \tau} \ \text{ if } \tau \text{ is pointed.}$$

For a retractive type ρF, we simply take the two equations establishing that $\mathit{fold}_{\rho F}$ and $\mathit{unfold}_{\rho F}$ are inverses:

$$(\rho\beta) \qquad \emptyset \rhd \mathit{unfold}_{\rho F} \circ \mathit{fold}_{\rho F} = \mathit{id}^{F(\rho F)} : F(\rho F) \to F(\rho F)$$

$$(\rho\eta) \qquad \emptyset \rhd \mathit{fold}_{\rho F} \circ \mathit{unfold}_{\rho F} = \mathit{id}^{\rho F} : \rho F \to \rho F.$$

For the lifted type, the adjunction gives us the following:

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

$$(\perp\beta') \qquad \Gamma \rhd (\lambda\lfloor x{:}\,\sigma\rfloor.\, M)\perp^{\sigma_\perp} = \perp^\tau : \tau$$

$$(\perp\eta) \qquad \frac{\Gamma \rhd M\perp^{\sigma_\perp} = \perp^\tau : \tau}{\Gamma \rhd (\lambda\lfloor x{:}\,\sigma\rfloor.\, M\lfloor x\rfloor) = M : \sigma_\perp \to \tau} \ \text{ for } x \notin \mathrm{FV}(M).$$

However, after some experience using these rules we discover that they are not quite strong enough to represent our intuition about lifting. In particular, they do not seem to reflect the desired property that the *only* elements of the lifted type $\sigma_\perp$ are the elements of σ plus bottom. Therefore, we will use the following more powerful inference rule in place of $(\perp\eta)$:

$$(\perp \mathit{ext}) \qquad \frac{\Gamma \rhd M\perp^{\sigma_\perp} = N\perp^{\sigma_\perp} : \tau, \quad \Gamma, x{:}\,\sigma \rhd M\lfloor x\rfloor = N\lfloor x\rfloor : \tau}{\Gamma \rhd M = N : \sigma_\perp \to \tau.}$$

It is easy to see that taking $N \equiv (\lambda\lfloor x{:}\,\sigma\rfloor.\, M\lfloor x\rfloor)$ in this rule lets us derive $(\perp\eta)$. We do not have any good categorical motivation for this rule, yet without it we have been unable to prove equations such as

$$(\lambda\lfloor x{:}\,\sigma\rfloor.\, (\lambda\lfloor y{:}\,\tau\rfloor.\, M)P)Q = (\lambda\lfloor y{:}\,\sigma\rfloor.\, (\lambda\lfloor x{:}\,\tau\rfloor.\, M)Q)P,$$

where x and y are not free in P or Q. Our intuition about strict abstraction says that this should hold, *i.e.*, that the order of strict evaluation should not matter as long as both arguments must be evaluated, and indeed this equation is easy to prove using the "reasoning by cases" made possible by the $(\perp \mathit{ext})$ rule. We take this defect of the categorically motivated rules as an indication that more work needs to be done to fully understand the lifted types.

Now we may prove a partial extension of the fact that substituting a term in a type behaves like functor application; note that a stronger proof system is required to handle the case where the functor contains a retractive type:

Lemma 7.1 *If $F(t)$ does not contain any subterms of the form $\rho s.\, v$, where t occurs free in v, then application of F to a term preserves composition and identities, i.e., $F(M \circ N) = F(M) \circ F(N)$ and $F(id) = id$.*

Proof. Most of this lemma was proved in Section 3 as Lemma 3.1; we only need to consider the case where $F(t)$ is a lifted type expression. If $F(t) = G(t)_\perp$, then

$$\begin{aligned}
F(M) \circ F(N) &= F(M) \circ (\lambda\lfloor x{:}\,G(\sigma)\rfloor.\, \lfloor G(N)x\rfloor) \\
&= (\lambda\lfloor x{:}\,G(\sigma)\rfloor.\, F(M)((\lambda\lfloor x{:}\,G(\sigma)\rfloor.\, \lfloor G(N)x\rfloor)\lfloor x\rfloor)) \\
&= (\lambda\lfloor x{:}\,G(\sigma)\rfloor.\, (\lambda\lfloor y{:}\,G(\tau)\rfloor.\, \lfloor G(M)y\rfloor)\lfloor G(N)x\rfloor) \\
&= (\lambda x{:}\,G(\sigma).\, \lfloor G(M)(G(N)x)\rfloor) \\
&= F(M \circ N);
\end{aligned}$$

similarly, $F(id^\sigma) = (\lambda\lfloor x{:}\,G(\sigma)\rfloor.\, \lfloor x\rfloor) = id^{F(\sigma)}$. ∎

The impact on previous results of the restriction on the form of F in this lemma is that the proof of Lemma 3.2, which states that the defined terms for $\mathit{unfold}_{\mu F}$ and $\mathit{fold}_{\nu F}$ are inverses for the primitives $\mathit{fold}_{\mu F}$ and $\mathit{unfold}_{\nu F}$, is only valid for functors of the restricted form. This provides another argument for including the inverses as primitives, as in $\lambda^{\mu\nu\prime}$.

In the interpretation of $\lambda^{\perp\rho}$ in the category **CPO**, we observe that application of the lifting functor to the initial object, *i.e.*, the empty cpo, yields a terminal object, *i.e.*, a cpo with exactly one element. It is an interesting consequence of our rules for lifting that this is true in all models of $\lambda^{\perp\rho}$:

Lemma 7.2 *The types* $0_\perp$ *and* 1 *are isomorphic.*

Proof. We will show that the terms $\perp^{0_\perp \to 1}$ and $\perp^{1 \to 0_\perp}$ are inverses. Since both terms are strict, either way of composing them will result in a bottom function, hence we only need to show that $\perp^{1\to 1} = id^1$ and $\perp^{0_\perp \to 0_\perp} = id^{0_\perp}$. The first equation is trivial, since by the (1η) rule, all terms of type $1\to 1$ are equal to $(\lambda\Diamond.\,\Diamond)$. For the second equation, since both sides are strict functions, if we can prove $x{:}\,0 \rhd \perp^{0_\perp\to 0_\perp}\lfloor x\rfloor = id^{0_\perp}\lfloor x\rfloor : 0_\perp$, then by using the (derived) strict abstraction congruence rule and $(\perp\eta)$ we are done. But this last equation is true, since for any term $x{:}\,0 \rhd M{:}\,\sigma$ we find by $(\to\beta)$ and (0η) that $M = (\lambda x{:}\,0.\,M)x = \Box^\sigma x$ is true, *i.e.*, all such terms are equal. ■

This isomorphism is an instance of a general condition for showing that lifting is a *monad with zero*, a concept introduced by Wadler [Wad90]. The monad natural transformations $\eta{:}\,\mathcal{C} \dot{\to} T$ and $\mu{:}\,TT \dot{\to} T$ receive their names "unit" and "multiplication" in part because of an analogy with the corresponding concepts in a monoid; the diagrams which must commute for a monad may then be seen as statements that multiplication is associative and has the unit as an identity:

$$\begin{tikzcd}
TTTX \arrow[r, "T\mu_X"] \arrow[d, "\mu_{TX}"'] & TTX \arrow[d, "\mu_X"] \\
TTX \arrow[r, "\mu_X"'] & TX
\end{tikzcd}
\qquad
\begin{tikzcd}
TX \arrow[r, "\eta_{TX}"] \arrow[dr, "id"'] & TTX \arrow[d, "\mu_X"] & TX \arrow[l, "T\eta_X"'] \arrow[dl, "id"] \\
& TX &
\end{tikzcd} .$$

Wadler observed that many common monads also have zeroes with respect to multiplication. In a category with a terminal object, we may present a zero for a monad by giving a natural transformation $\zeta{:}\,1 \dot{\to} T$ such that the following diagram commutes:

$$\begin{tikzcd}
1 \arrow[r, "\zeta_{TX}"] \arrow[dr, "\zeta_X"'] & TTX \arrow[d, "\mu_X"] & T1 \arrow[l, "T\zeta_X"'] \arrow[dl, "\zeta_X \circ \Diamond_{T1}"] \\
& TX &
\end{tikzcd} ,$$

where $\Diamond_{T1}$ is the unique arrow from $T1$ to 1. For the lifting monad, it is easy to see that the bottom arrow from 1 to $X_\perp$ will act as a zero, since the multiplication μ_X collapses both the bottom element $\perp^{(X_\perp)_\perp}$ and the lifted bottom element $\lfloor \perp^{X_\perp}\rfloor$ down to the bottom of $X_\perp$.

This author and Michael Johnson [Wad91] independently noticed the following special case of this definition: if a category with a monad (T, η, μ) and a terminal object 1 also has an initial object 0 and if there is an isomorphism $\perp{:}\,1 \to T0$, then the natural transformation whose X component is the arrow $T\Box_X \circ \perp{:}\,1 \to TX$ will act as a zero for the monad. This follows by a simple diagram chase, using the uniqueness of the arrows $\Diamond_X{:}\,X \to 1$ and $\Box_X{:}\,0 \to X$, plus the fact that $\Diamond_{T0}$ is an inverse for the given arrow $\perp$. The preceding lemma thus allows us to confirm the above observation about the zero of the lifting monad, since the term corresponding to $T\Box_X \circ \perp$ is $(\lambda\lfloor x{:}\,X\rfloor.\,\lfloor \Box^X x\rfloor) \circ \perp^{1\to 0_\perp}$, which is equal to $\perp^{1 \to X_\perp}$ by the $(\perp\beta')$ rule.

8 The reduction system $\lambda_r^{\perp\rho}$

We may now consider an operational semantics for $\lambda^{\perp\rho}$, given as the obvious extension to $\lambda_r^{\mu\nu}$. The full reduction system $\lambda_r^{\perp\rho}$ will still be confluent, although it will of course no longer be normalizing. A notion of evaluating a term must therefore be relative to some strategy for choosing a reduction path. We will show that a lazy strategy is computationally adequate with respect to finding normal forms of programs.

As usual, we obtain the reduction rules from the equational proof system by directing the (β) axioms in the direction of decreasing complexity of terms. Here is the full list of rules:

$$
\begin{array}{ll}
(+\beta_1)_r & [M,N](\iota_1 P) \longrightarrow MP \\
(+\beta_2)_r & [M,N](\iota_2 P) \longrightarrow NP \\
(\times\beta_1)_r & \pi_1\langle M,N\rangle \longrightarrow M \\
(\times\beta_2)_r & \pi_2\langle M,N\rangle \longrightarrow N \\
(\rightarrow\beta)_r & (\lambda x{:}\,\sigma.\, M)N \longrightarrow \{N/x\}M \\
(\mu\beta)_r & \mathit{it}_{\mu F}\, M(\mathit{fold}_{\mu F}\, P) \longrightarrow M(F(\mathit{it}_{\mu F}\, M)P) \\
(\nu\beta)_r & \mathit{unfold}_{\nu F}(\mathit{new}_{\nu F}\, MP) \longrightarrow F(\mathit{new}_{\nu F}\, M)(MP) \\
(\mu\beta')_r & \mathit{unfold}_{\mu F}(\mathit{fold}_{\mu F}\, M) \longrightarrow M \\
(\nu\beta')_r & \mathit{unfold}_{\nu F}(\mathit{fold}_{\nu F}\, M) \longrightarrow M \\
(\rho\beta)_r & \mathit{unfold}_{\rho F}(\mathit{fold}_{\rho F}\, M) \longrightarrow M \\
(\bot\beta)_r & (\lambda\lfloor x{:}\,\sigma\rfloor.\, M)\lfloor N\rfloor \longrightarrow \{N/x\}M \\
(\bot\beta')_r & (\lambda\lfloor x{:}\,\sigma\rfloor.\, M)\bot^{\sigma_\bot} \longrightarrow \bot^{\tau}.
\end{array}
$$

For the proof of confluence we may simply observe that $\lambda_r^{\bot\rho}$ is a regular combinatory reduction system (CRS), as introduced by Klop [Klo80]. A combinatory reduction system (CRS), as introduced by Klop [Klo80], is a generalization of term rewriting systems to include variable binding operators and substitution. Our system $\lambda_r^{\bot\rho}$ is a CRS with two binding forms, each of which binds one variable: $(\lambda x.\,\cdot)$ and $(\lambda\lfloor x\rfloor.\,\cdot)$. In fact it is a regular CRS, because it is left-linear (no meta-variable appears more than once on the left-hand side of a rule) and non-ambiguous (there are no critical pairs).

Theorem 8.1 (Confluence) *If $M \twoheadrightarrow N$ and $M \twoheadrightarrow P$, then there is a term Q such that $N \twoheadrightarrow Q$ and $P \twoheadrightarrow Q$.*

Proof. Immediate from [Klo80], Theorem II.3.11. ■

Klop also shows that if a regular CRS has the additional property that it is *left-normal*, then it satisfies a standardization theorem. The definition of left-normal is that all of the constants be to the left of any meta-variables in the reduction rules. Intuitively, if a system is left-normal, then evaluating terms from left to right will not allow any redexes to be missed.

Our system $\lambda_r^{\bot\rho}$ as it stands is not left-normal; for example, in the $(+\beta_1)_r$ rule, the constant ι_1 appears to the right of the meta-variables M and N. To see concretely how this would affect a left-to-right reduction strategy, consider the term $[\mathit{id}, \mathit{fix}\ \mathit{id}](\mathit{id}(\iota_1\diamond))$. If we employed a strategy of strict left-to-right evaluation, then the reduction would get hung in a loop trying to evaluate the right arm of the choice, never reaching the $(\rightarrow\beta)_r$ redex at the right, which must be evaluated to complete the $(+\beta_1)_r$ redex and reach the normal form $\diamond$.

The non-left-normal rules are $(+\beta_1)_r$, $(+\beta_2)_r$, $(\mu\beta)_r$, $(\bot\beta)_r$, and $(\bot\beta')_r$. There are two easy ways to fix these and obtain a left-normal reduction system. Either we may rearrange the syntax of terms, so that these rules become left-normal by Klop's definition, or we may modify the meaning of "left" with respect to the ordering of subterms. We will describe the second method here; the first is explored in the author's doctoral dissertation [How92].

To change the left-to-right order of evaluation, we will alter the definition of when one redex, R, is to the left of another, S, written $R \prec S$. The effect of this will be to direct reduction into the appropriate part of a potential redex (for example, the test P of a choice expression $[M,N]P$) so that the reduction does not get needlessly sidetracked. The definition of $R \prec S$ that we start with is that a subterm R of a term M (which we write $R \subseteq M$; if $R \not\equiv M$ then as usual we write $R \subset M$) is to the left of another subterm $S \subseteq M$ if one of the following conditions holds:

- $S \subset R$;
- $R \subseteq M_1$ and $S \subseteq M_2$ for some subterm $[M_1, M_2]$ of M;
- $R \subseteq M_1$ and $S \subseteq M_2$ for some subterm $\langle M_1, M_2\rangle$ of M; or
- $R \subseteq M_1$ and $S \subseteq M_2$ for some subterm $M_1 M_2$ of M.

Our modification will be to the last clause, based on whether or not the term M_1 is a strict abstraction, choice, or iterated function:

- if $R \subseteq M_1$ and $S \subseteq M_2$ for some $M_1 M_2 \subseteq M$, then $R \prec S$ if $M_1 \not\equiv (\lambda\lfloor x\colon\sigma\rfloor.\, P)$, $[P, Q]$, or $it_{\mu F}\, P$, otherwise $S \prec R$.

Now, following Klop, we may define standard reductions and the process of standardization. If we have a reduction sequence $\mathcal{R}\colon M_0 \longrightarrow M_1 \longrightarrow \ldots$, then let us form a labelled sequence by the following process: if the redex contracted in the term M_i is R, then we label every redex $S \prec R$; descendents of labelled redexes keep their labels until they are themselves reduced. For example, here is the labelled form of a reduction from a term discussed above:

$$\begin{array}{rcl}
[id, \mathit{fix}\ id](id(\iota_1\diamond))^* & \longrightarrow\!\!\!\!\rightarrow & [id, id(\mathit{fix}\ id)](id(\iota_1\diamond))^* \\
 & \longrightarrow & ([id, id(\mathit{fix}\ id)](\iota_1\diamond))^* \\
 & \longrightarrow & ([id, \mathit{fix}\ id](\iota_1\diamond))^* \\
 & \longrightarrow & id\,\diamond \\
 & \longrightarrow & \diamond.
\end{array}$$

A *standard reduction* is one in which no labelled redexes are contracted. The above reduction is not standard because two labelled redexes contract, producing the second and fourth lines respectively. An equivalent standard reduction is

$$\begin{array}{rcl}
[id, \mathit{fix}\ id](id(\iota_1\diamond)) & \longrightarrow & [id, \mathit{fix}\ id](\iota_1\diamond) \\
 & \longrightarrow & id\,\diamond \\
 & \longrightarrow & \diamond.
\end{array}$$

An *anti-standard pair* of reduction steps is a reduction $\mathcal{R}_1\colon M_0 \longrightarrow M_1 \longrightarrow M_2$ which is not standard. The process of *meta-reduction* is the replacement in a reduction $\mathcal{R}$ of an anti-standard pair $\mathcal{R}_1$ by an equivalent standard reduction $\mathcal{R}_2$ to obtain a new reduction $\mathcal{R}'$; we write $\mathcal{R} \Longrightarrow \mathcal{R}'$. For example, in the reduction sequence

$$\mathcal{R}\colon (\lambda x\colon\sigma.\ id\langle x, x\rangle)(id\,\diamond) \longrightarrow (\lambda x\colon\sigma.\ id\langle x, x\rangle)\diamond \longrightarrow id\langle\diamond, \diamond\rangle \longrightarrow \langle\diamond, \diamond\rangle,$$

the first two reduction steps form an anti-standard pair; the pair may be replaced by a standard reduction to obtain the reduction sequence

$$\begin{array}{rcl}
\mathcal{R}'\colon (\lambda x\colon\sigma.\ id\langle x, x\rangle)(id\,\diamond) & \longrightarrow & id\langle id\,\diamond, id\,\diamond\rangle \longrightarrow id\langle\diamond, id\,\diamond\rangle \\
 & \longrightarrow & id\langle\diamond, \diamond\rangle \longrightarrow \langle\diamond, \diamond\rangle,
\end{array}$$

which is intuitively closer to a standard reduction than $\mathcal{R}$. One can imagine how repeating this process of meta-reduction will eventually produce a standard reduction sequence.

This intuition is formalized by the following theorem:

Theorem 8.2 (Standardization) *For every $\lambda_r^{\perp\rho}$ reduction sequence $\mathcal{R}\colon M \longrightarrow\!\!\!\!\rightarrow N$ there is a standard reduction sequence $\mathcal{R}_{st}\colon M \longrightarrow\!\!\!\!\rightarrow N$, obtained as a normal form of meta-reduction.*

Proof. See [Klo80], Section II.6.2.8. In fact, Klop does not give the proof for the fully general case of a left-normal regular combinatory reduction system, he only indicates that it is possible. However, a minor modification to $\lambda_r^{\perp\rho}$ will put it in the form of a left-normal term rewriting system plus lambda abstraction, for which Klop does provide a full proof. The required change is that the binding operation of strict abstraction $(\lambda\lfloor x{:}\,\sigma\rfloor.\, M)$ be replaced by the ordinary lambda abstraction $(\lambda z{:}\,\sigma_\perp.\, \mathit{test}_\sigma^\tau z(\lambda x{:}\,\sigma.\, M))$, where $\mathit{test}_\sigma^\tau{:}\,\sigma_\perp{\to}(\sigma{\to}\tau){\to}\tau$ is a new function constant. The action of $\mathit{test}_\sigma^\tau$ is given by the following reduction rules:

$$\begin{array}{rcl} \mathit{test}_\sigma^\tau\lfloor M\rfloor N & \longrightarrow & NM \\ \mathit{test}_\sigma^\tau\perp N & \longrightarrow & \perp; \end{array}$$

that is, it strictly evaluates the first argument and then applies the second argument to the result. It is easy to see that this modified system will have essentially the same reduction behavior as $\lambda_r^{\perp\rho}$; in particular, the standard reductions in the two systems will be isomorphic. ∎

As a corollary we find that a leftmost strategy for $\lambda_r^{\perp\rho}$ is normalizing, using the adjusted definition of "left":

Corollary 8.3 (Normalization) *If there is a $\lambda_r^{\perp\rho}$ reduction $M \twoheadrightarrow N$ where N is a normal form, then the reduction sequence starting at M in which the leftmost redex is contracted at each step will eventually reach N.*

Proof. By the Standardization theorem, there is a standard reduction $\mathcal{R}_{st}$ from M to N. Suppose that there is some step in $\mathcal{R}_{st}$ in which the contracted redex is not leftmost; then since the reduction is standard, no descendents of the redexes to the left can ever be contracted, nor can they be erased, hence N must contain uncontracted redexes, contradicting the fact that it is a normal form. Therefore $\mathcal{R}_{st}$ is the desired normal reduction sequence. ∎

Note that this result is weaker than the computational adequacy result for $\lambda^{\mu\nu}$. If there is some reduction from a term to a normal form then we are guaranteed to find it, but there is no guarantee that the reduction system is strong enough to produce a normal form whenever the initial term is provably equal to one.

9 Example: call-by-name and call-by-value

In this section we present two retractive types n and v which serve respectively as universal types for call-by-name and call-by-value versions of the untyped lambda calculus [Plo75]. This provides a syntactic solution to the problem of finding non-trivial universal types which corresponds to the domain models presented by Boudol in [Bou91].

The type n is given by $\rho t.\, (t{\to}t)_\perp$, which reflects the fact that the only way the call-by-name calculus can fail to terminate is when trying to evaluate the head of an application to get an abstraction. By contrast, the type v is given by $\rho t.\, t{\to}t_\perp$, which reflects the fact that it is the process of application of an abstraction to a term which may not terminate, if the argument never reduces to a value. Since in the call-by-value calculus we also have the possibility that evaluating the head may not terminate either, we actually use $v_\perp$ as the universal type.

To put this in more concrete terms, consider the following translations from Λ to $\lambda^{\perp\rho}$:

$$\begin{array}{rcl} \mathcal{N}(x) & \equiv & x{:}\,n \\ \mathcal{N}(\lambda x.\, M) & \equiv & \mathit{fold}_n\lfloor\lambda x{:}\,n.\,\mathcal{N}(M)\rfloor{:}\,n \\ \mathcal{N}(MN) & \equiv & \mathit{App}^N\,\mathcal{N}(M)\mathcal{N}(N){:}\,n \\ \\ \mathcal{V}(x) & \equiv & \lfloor x\rfloor{:}\,v_\perp \\ \mathcal{V}(\lambda x.\, M) & \equiv & \lfloor\mathit{fold}_v(\lambda x{:}\,v.\,\mathcal{V}(M))\rfloor{:}\,v_\perp \\ \mathcal{V}(MN) & \equiv & \mathit{App}^V\,\mathcal{V}(M)\mathcal{V}(N){:}\,v_\perp, \end{array}$$

where $App^N \equiv (\lambda \mathit{fold}_n \lfloor f{:}\, n{\to}n \rfloor .\, f)$ and $App^V \equiv (\lambda \lfloor \mathit{fold}_v (f{:}\, v{\to}v_\bot) \rfloor .\, \lambda \lfloor z{:}\, v \rfloor .\, fz)$ In evaluating an application MN in either call-by-name or call-by-value, M must first reduce to an abstraction. The strict abstractions on f in the *App* combinators will ensure that this evaluation takes place first. In the case of the call-by-value translation, the additional strict abstraction on z forces the evaluation of the argument next; in call-by-name, the function f is simply applied to the argument right away.

To formalize the connection with the call-by-name and call-by-value calculi, recall from [Plo75] that the reductions $\xrightarrow{cbn}\!\!\!\!\rightarrow$ and $\xrightarrow{cbv}\!\!\!\!\rightarrow$ are built up from the axioms

$$(\beta) \qquad (\lambda x.\, M)N \longrightarrow \{N/x\}M$$

$$(\beta_V) \qquad (\lambda x.\, M)V \longrightarrow \{V/x\}M, \quad V \text{ a value,}$$

respectively, where a *value* is either a variable or an abstraction (we will omit the optional set of constants and δ-rules for simplicity). Then we may use standardization to prove the following theorem:

Theorem 9.1 *$M \xrightarrow{cbn}\!\!\!\!\rightarrow N$ iff $\mathcal{N}(M) \longrightarrow\!\!\!\!\rightarrow \mathcal{N}(N)$, and $M \xrightarrow{cbv}\!\!\!\!\rightarrow N$ iff $\mathcal{V}(M) \longrightarrow\!\!\!\!\rightarrow \mathcal{V}(N)$.*

Proof. The forward direction of each part proceeds by induction on the length of the reduction; we only need to show that $\mathcal{N}((\lambda x.\, M)N) \longrightarrow\!\!\!\!\rightarrow \mathcal{N}(\{N/x\}M)$ and $\mathcal{V}((\lambda x.\, M)V) \longrightarrow\!\!\!\!\rightarrow \mathcal{V}(\{V/x\}M)$, for V a value. We will only show the details of the second, since the first is very similar (and easier):

$$\begin{array}{rcl} \mathcal{V}((\lambda x.\, M)V) & \equiv & App^V \lfloor \mathit{fold}_v(\lambda x{:}\, v.\, \mathcal{V}(M)) \rfloor \mathcal{V}(V) \\ & \longrightarrow\!\!\!\!\rightarrow & (\lambda \lfloor z{:}\, v \rfloor .\, (\lambda x{:}\, v.\, \mathcal{V}(M))z)\mathcal{V}(V); \end{array}$$

since V is a value, $\mathcal{V}(V)$ must be of the form $\lfloor N \rfloor$ for some term N, hence the strict application may reduce, leading to $\{N/x\}\mathcal{V}(M)$. Now, examination of the definition of $\mathcal{V}(M)$ reveals that this substitution is identical to the term $\mathcal{V}(\{V/x\}M)$, as desired.

In the other direction we will make use of the standardization theorem of the previous section. Again, we will only give the details for the more complex call-by-value case. We will proceed by induction on the structure of M and the length of the reduction sequence. If $M \equiv x$, then $\mathcal{V}(M) \equiv \lfloor x \rfloor$, which is a normal form, so $M \equiv N$ and we are done. If $M \equiv (\lambda x.\, P)$, then $\mathcal{V}(M) \equiv \lfloor \mathit{fold}_v(\lambda x{:}\, v.\, \mathcal{V}(P)) \rfloor$, so the only possible reduction is to $\lfloor \mathit{fold}_v(\lambda x{:}\, v.\, Q) \rfloor$. For this to be $\mathcal{V}(N)$ we must have that $Q \equiv \mathcal{V}(P')$ for $N \equiv (\lambda x.\, P')$; by the induction hypothesis then we know that $P \xrightarrow{cbv}\!\!\!\!\rightarrow P'$ and hence $M \xrightarrow{cbv}\!\!\!\!\rightarrow N$.

The remaining case is that $M \equiv PQ$. If $\mathcal{V}(M) \equiv App^V\, \mathcal{V}(P)\mathcal{V}(Q)$ reduces to $\mathcal{V}(N)$, then either $\mathcal{V}(N) \equiv App^V\, \mathcal{V}(P')\mathcal{V}(Q')$, whence $M \xrightarrow{cbv}\!\!\!\!\rightarrow P'Q' \equiv N$ by the induction hypothesis, or the App^V must participate in the reduction. For this to be true, the standardization of the reduction must look like the following:

$$\begin{array}{rcl} App^V\, \mathcal{V}(P)\mathcal{V}(Q) & \longrightarrow\!\!\!\!\rightarrow & App^V \lfloor \mathit{fold}_v(\lambda x{:}\, v.\, \mathcal{V}(P')) \rfloor \mathcal{V}(Q) \\ & \longrightarrow\!\!\!\!\rightarrow & (\lambda \lfloor z{:}\, v \rfloor .\, (\lambda x{:}\, v.\, \mathcal{V}(P'))z)\mathcal{V}(Q) \\ & \longrightarrow\!\!\!\!\rightarrow & (\lambda \lfloor z{:}\, v \rfloor .\, (\lambda x{:}\, v.\, \mathcal{V}(P'))z)\lfloor Q_1 \rfloor \\ & \longrightarrow\!\!\!\!\rightarrow & \{Q_1/x\}\mathcal{V}(P') \equiv \mathcal{V}(\{Q'/x\}P') \\ & \longrightarrow\!\!\!\!\rightarrow & \mathcal{V}(N), \end{array}$$

where $\mathcal{V}(Q') \equiv \lfloor Q_1 \rfloor$. This uses the fact that in any standard reduction from $\mathcal{V}(Q)$, the first term in the reduction sequence which is of the form $\lfloor Q_1 \rfloor$ must in fact be $\mathcal{V}(Q')$ for some value Q'. By the induction hypothesis we thus have the corresponding reduction sequence

$$\begin{array}{rcl} M \equiv PQ & \xrightarrow{cbv}\!\!\!\!\rightarrow & (\lambda x.\, P')Q \\ & \xrightarrow{cbv}\!\!\!\!\rightarrow & (\lambda x.\, P')Q' \\ & \xrightarrow{cbv} & \{Q'/x\}P' \\ & \xrightarrow{cbv}\!\!\!\!\rightarrow & N. \end{array}$$

■

References

[Bar84] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1984.

[BB85] C. Böhm and A. Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[Bou91] G. Boudol. Lambda-calculi for (strict) parallel functions. Technical Report 1387, INRIA, January 1991.

[BW87] W. Buchholz and S.S. Wainer. Provably computable functions and the fast growing hierarchy. In S.G. Simpson, editor, *Logic and Combinatorics*, volume 65 of *Contemporary Mathematics*, pages 179–198. American Mathematical Society, 1987.

[CP90] T. Coquand and C. Paulin. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *COLOG-88*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1990.

[CW83] E.A. Cichon and S.S. Wainer. The slow-growing and the Grzegorczyk hierarchies. *Journal of Symbolic Logic*, 48(2):399–408, 1983.

[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.

[Göd58] K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958. An English translation by W. Hodges and B. Watson appeared in *Journal of Philosophical Logic*, 9:133–142, 1980.

[Gal91] J.H. Gallier. What's so special about Kruskal's theorem and the ordinal Γ_0? A survey of some results in proof theory. *Annals of Pure and Applied Logic*, 53(3):199–260, 1991.

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

[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.

[GS90] C.A. Gunter and D.S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.* North-Holland, 1990.

[Hag87a] T. Hagino. *A Categorical Programming Language.* PhD thesis, University of Edinburgh, 1987.

[Hag87b] T. Hagino. A typed lambda calculus with categorical type constructors. In *Category Theory in Computer Science*, pages 140–157, 1987.

[Has89] R. Hasegawa. Parametric polymorphism and internal representations of recursive type definitions. Master's thesis, Research Institute for Mathematical Science, Kyoto University, 1989.

[How92] B.T. Howard. *Fixed Points and Extensionality in Typed Functional Programming Languages.* PhD thesis, Stanford University, 1992.

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

[Kre59] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In A. Heyting, editor, *Constructivity in Mathematics*, pages 101–128. North-Holland, 1959.

[Mac71] S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics.* Springer-Verlag, 1971.

[Mil76] L.W. Miller. Normal functions and constructive ordinal numbers. *Journal of Symbolic Logic*, 41(2):439–459, 1976.

[New42] M.H.A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.

[Plo75] G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo85] G.D. Plotkin. Denotational semantics with partial functions. Lecture notes, C.S.L.I. Summer School, Stanford, 1985.

[Rey74] J.C. Reynolds. Towards a theory of type structure. In *Paris Colloquium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[Ros84] H.E. Rose. *Subrecursion: Functions and Hierarchies*, volume 9 of *Oxford Logic Guides.* Oxford University Press, 1984.

[Sch75] H. Schwichtenberg. Elimination of higher type levels in definitions of primitive recursive functions by means of transfinite recursion. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium, '73*, pages 279–303. North-Holland, 1975.

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

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

[Wad91] P. Wadler. Private communication, March 1991.

[Wai89] S.S. Wainer. Slow growing versus fast growing. *Journal of Symbolic Logic*, 54(2):608–614, 1989.

[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.