Programming with Inductive and Co-Inductive Types

John Greiner
January 27, 1992
CMU-CS-92-109

Carnegie Mellon
Programming with Inductive and Co-Inductive Types

John Greiner
January 27, 1992
CMU-CS-92-109

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We look at programming with inductive and co-inductive datatypes, which are inspired theoretically by initial algebras and final co-algebras, respectively. A predicative calculus which incorporates these datatypes as primitive constructs is presented. This calculus allows reduction sequences which are significantly more efficient for two dual classes of common programs than do previous calculi using similar primitives. Several techniques for programming in this calculus are illustrated with numerous examples. A short survey of related work is also included.

This research was supported in part by the Office of Naval Research and in part by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under Contract N00014-84-K-0415, ARPA Order No. 5404, and in part by the ONR Graduate Fellowship Program.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of ONR, DARPA or the U.S. government.
Keywords: Data types and structures, program and recursion schemes, lambda calculus and related systems, induction, co-induction, initial algebras, final co-algebras
1 Introduction

This paper explores programming with inductive and co-inductive datatypes. Type expressions using the $\mu$ type constructor represent finitely representable inductive types (e.g., natural numbers, lists), while those using $\nu$ represent countably infinite or potentially infinite co-inductive types (e.g., potentially infinite streams, infinite depth binary trees). Each (co-)inductive type is associated with operators to build and manipulate terms of these types.

Encodings of such types in previous calculi have suffered from efficiency problems which prevent them from being as useful in practice as desired. The typical example is that of the predecessor function on the common Church numeral encoding in $F_2$, which requires linear-time, as shown by Parigot [24]. Most encodings in other calculi are closely related and suffer the same problem. Our calculus allows similar encodings of data types, but the calculus offers extensions admitting definitions of constant-time $\text{pred}$ and $\text{cdr}$, and related efficiency improvements.

Previous work has concentrated on inductive types. Very few examples of co-inductive functions have been given, so that their usefulness in practice is in question. This paper explores the duality of inductive and co-inductive types and presents a number of examples to attempt to show the usefulness of co-inductive types.

Induction and co-induction as presented here are obviously less powerful than recursion, as they guarantee termination. So, an essential question is whether these concepts are powerful enough for practical programming. This is the motivation for our extensive look at examples within the calculus.

These (co-)inductive type constructors are inspired by initial algebras and final co-algebras [8, 9, 19] of category theory. Some examples of this paper use simple commuting diagrams to help explain the definition of some functions in the calculus. A basic knowledge of category theory will be helpful, but not necessary, to the reader.

A summary of this theoretical motivation is given in Section 2. The calculus is introduced in section 3, with examples of inductive and co-inductive terms in Sections 4 and 5. In Section 6, related work is discussed. The appendices give additional details for interested readers.

2 Theoretical Inspiration

This section provides a theoretical motivation or inspiration for the calculus as presented in Section 3. As such, it also serves as a beginning point for developing a model for the calculus. More pragmatically, simple commuting diagrams lead to methods for programming in the calculus.

Assume that there exists an appropriate category $C$ of types, where arrows

\footnote{Some features of such a category are discussed in [2, 6].}
represent functions from one type to another and composition is function composition. We then look at $F$-algebras, for functors $F : C \to C$. An $F$-algebra is a pair $(\tau, g)$ consisting of an object $\tau : C$ and a map $g : F(\tau) \to \tau$. An $F$-homomorphism, an arrow in the category of $F$-algebras, is a $C$-arrow such that the following diagram commutes.

\[
\begin{array}{ccc}
F[\tau] & \xrightarrow{F(h)} & F[\tau'] \\
\downarrow & & \downarrow \\
g & & f \\
\tau & \xrightarrow{h} & \tau'
\end{array}
\]

If $(\tau, g)$ is an initial $F$-algebra, then there exists a unique $h$ for any given choice of $(\tau', f)$. Labelling the initial $F$-algebra as $(\mu(F), \text{in}(F))$ and the unique $h$ as $R(F)[\tau]f$ to emphasize their dependencies, we obtain the commuting diagram

\[
\begin{array}{ccc}
F[\mu(F)] & \xrightarrow{F(R(F)[\tau']f)} & F[\tau'] \\
\downarrow & & \downarrow \\
\text{in}(F) & & f \\
\mu(F) & \xrightarrow{R(F)[\tau']f} & \tau'
\end{array}
\]

When the above arguments are dualized, we obtain the final $F$-co-algebra $(\nu(F), \text{out}(F))$ such that the following diagram commutes for any choice of $(\tau', f)$.

\[
\begin{array}{ccc}
F[\tau'] & \xrightarrow{F(G(F)[\tau']f)} & F[\nu(F)] \\
\downarrow & & \downarrow \\
f & & \text{out}(F) \\
\tau' & \xrightarrow{G(F)[\tau']f} & \nu(F)
\end{array}
\]
The following theorems and corollaries are to establish motivation for some of the equality judgments of the calculus. Proofs are given only for inductive cases, as the co-inductive cases are analogous.

The first theorem is \( \beta \)-like and shows the main interaction of the induction morphisms. The second gives \( \eta \)-like equalities.

**Theorem 1** Principle of induction: \((R\{F\}[r]f) \circ \text{in}\{F\} = f \circ F(R\{F\}[r]f)\)

and of co-induction: \(\text{out}\{F\} \circ (G\{F\}[r]f) = F(G\{F\}[r]f) \circ f\).

**Proof:** Follow from the commutativity of the preceding diagrams. \(\square\)

**Theorem 2** \(R\{F\}[\mu(F)]\text{in}\{F\} = \text{Id}_{F}\) and \(G\{F\}[\nu(F)]\text{out}\{F\} = \text{Id}_{F}\).

**Proof:** Follow from the initiality (finality) of the \(F\)-(co-)algebra and commutativity. \(\square\)

**Theorem 3** \(\text{in}\{F\}\) and \(\text{out}\{F\}\) are isomorphisms.

**Proof:** We show that \(\text{in}\{F\}\) has left- and right-inverses. Consider the following diagram, where both squares commute:

\[
\begin{array}{ccc}
F[\mu(F)] & \xrightarrow{F(!)} & F[F[\mu(F)]] \\
\downarrow \text{in}(F) & & \downarrow F(\text{in}(F)) \\
\mu(F) & \xrightarrow{!} & F[\mu(F)] \\
\end{array}
\]

By the definition of an algebra, \(! = R\{F\}[F[\mu(F)]][F(\text{in}\{F\})]\) is the unique map such that

\[! \circ \text{in}\{F\} = F(\text{in}\{F\}) \circ F(!) = F(\text{in}\{F\} \circ !).\]

Since the inner squares commute, so does the outer rectangle. Then, by initiality,

\[\text{in}\{F\} \circ ! = \text{Id}_{F}.\]

Furthermore,

\[! \circ \text{in}\{F\} = F(\text{in}\{F\} \circ !) = F(\text{Id}_{F}) = F(\mu(F)) = \text{Id}_{F}.\]

But these equations simply mean that \(! is a left- and right-inverse of \(\text{in}\{F\}\), and thus, that \(\text{in}\{F\}\) is an isomorphism. \(\square\)
Corollary 1 There exist unique morphisms $\text{in}^{-1}\{F\}$ and $\text{out}^{-1}\{F\}$. And these inverses are expressible in terms of the other (co-)inductive morphisms as $R\{F\}[F[\mu(F)](F(\text{in}\{F\}))]$ and $G\{F\}[F[\nu(F)](F(\text{out}\{F\}))]$, respectively.

Example 1 For $F(X) = 1 + X$, $\text{in}\{F\}$ is the mapping $\text{[zero, succ]}$. Its inverse, $\text{in}^{-1}\{F\}$, is related to the mappings $\text{zero}$ and $\text{pred}$. Details are found in Example 11.

3 The Calculus $\lambda^{\text{MM}^\mu\nu}$

The calculus is based upon a restricted version of $\lambda^{\text{ML}}$ [11]. The higher kinds of that calculus are omitted to avoid complications that arise between type constructor application and the positivity requirement of $\mu$ and $\nu$. The explicit set injection is also omitted, for readability. Thus, this calculus is called $\lambda^{\text{MM}^\mu\nu}$, for "mini-$\lambda^{\text{ML}}$ with $\mu$ and $\nu$". The choice of calculus for a base is not critical; e.g., the Calculus of Constructions, $F_2$, and variants of $F_2$ have been used in other work.

3.1 Syntax

Given denumerable sets of variables $\text{typevar}$ and $\text{termvar}$, the calculus is defined by

\[
\begin{align*}
X & \in \text{typevar} \\
\tau & \in \text{types} \quad ::= \quad X \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau \rightarrow \tau' \mid \mu(u) \mid \nu(u) \\
u & \in \text{typescons}^3 \quad ::= \quad \lambda X.\tau \quad \text{s.t.} \quad \neg \text{Neg}(X, \tau) \\
\sigma & \in \text{typeschemes} \quad ::= \quad \tau \mid \sigma_1 \times \sigma_2 \mid \sigma_1 + \sigma_2 \mid \sigma \rightarrow \sigma' \mid \forall X.\sigma \\
x & \in \text{termvar} \\
t & \in \text{terms} \quad ::= \quad x \mid * \mid (t_1, t_2) \mid \pi_1 t \mid \pi_2 t \mid \\
\quad \text{in}\{t\} \mid \text{in}\{t\} \mid \text{case}(t, t_1, t_2) \mid \\
\quad \lambda x : \sigma.t \mid t_1 t_2 \mid \Lambda X.t \mid t[\tau] \mid \\
\quad \text{in}\{u\} \mid \text{in}\{u\} \mid R\{u\}[\tau]t \mid \\
\quad \text{out}\{u\} \mid \text{out}\{u\} \mid G\{u\}[\tau]t \\
\Gamma & \in \text{contexts} \quad ::= \quad \emptyset \mid \Gamma, v : \sigma
\end{align*}
\]

where $\text{Neg}(X, \tau) (\text{Pos}(X, \tau))$ holds if $X$ occurs "negatively" ("positively") in $\tau$.

\footnote{The implications are discussed briefly in Section 8.}

\footnote{The weak notion of "type constructor" used here corresponds to a constructor of kind $\text{Type} \rightarrow \text{Type}$. Also, $\mu$ and $\nu$ can be informally considered type constructors of kind $(\text{Type} \rightarrow \text{Type}) \rightarrow \text{Type}$.}
\( \text{Neg}(X, X) = \text{Neg}(X, 1) = \text{false} \)

\( \text{Neg}(X, \tau_1 \times \tau_2) = \text{Neg}(X, \tau_1 + \tau_2) = \text{Neg}(X, \tau_1) \lor \text{Neg}(X, \tau_2) \)

\( \text{Neg}(X, \tau \rightarrow \tau') = \text{Pos}(X, \tau) \lor \text{Neg}(X, \tau') \)

\( \text{Neg}(X, \mu(AX.r)) = \text{Neg}(X, \nu(AX.r)) = \text{false} \)

\( \text{Neg}(X, \mu(AY.r)) = \text{Neg}(X, \nu(AY.r)) = (X \neq Y) \land \text{Neg}(X, \tau), \text{if } X \neq Y \)

\( \text{Pos}(X, X) = \text{true} \)

\( \text{Pos}(X, 1) = \text{false} \)

\( \text{Pos}(X, \tau_1 \times \tau_2) = \text{Pos}(X, \tau_1 + \tau_2) = \text{Pos}(X, \tau_1) \lor \text{Pos}(X, \tau_2) \)

\( \text{Pos}(X, \tau \rightarrow \tau') = \text{Neg}(X, \tau) \lor \text{Pos}(X, \tau') \)

\( \text{Pos}(X, \mu(AX.r)) = \text{Pos}(X, \nu(AX.r)) = \text{false} \)

\( \text{Pos}(X, \mu(AY.r)) = \text{Pos}(X, \nu(AY.r)) = (X \neq Y) \land \text{Pos}(X, \tau), \text{if } X \neq Y \)

and where contexts are taken to be sets, with the comma as the extension operator.

The addition of the families of constants in\(^{-1}\{u\} \) and out\(^{-1}\{u\}, \) along with the association reductions in Section 3.4, is a primary contribution of this paper.

Here, we find the notation \( \mu(AX.r) \) (and similarly, \( \nu(AX.r) \)) more convenient, but it is equivalent to the more common notation \( \mu X.r. \)

3.2 Meta-notation

For readability, we make extensive use of notational definitions using the symbol \( \equiv. \) For example, \( \text{Id}^\equiv \equiv \lambda x : \sigma x, t^\equiv \equiv \lambda u : 1.f. \) Also, after the first examples, we omit type information when it is clear from context.

In the examples, we express desired properties of functions via equivalences. Observational equivalence between terms is denoted by \( \equiv. \) As usual, \( t \equiv t' \) iff \( P[t] \) and \( P[t'] \) evaluate to the same value, for all contexts \( P[] \) of type \( \text{Bool}. \)

Capture-avoiding substitution is denoted \( A[B/C]. \) Type constructor application \( u \tau \) is shorthand for its \( \beta \)-reduction, \( u[\tau/X]. \)

3.3 Type judgments

Type judgments of the form \( \Gamma \vdash t : \sigma \) state that \( t \) is a term of type \( \sigma. \)

\[
\begin{align*}
\Gamma \vdash t : \sigma & \quad \text{\( \Gamma \vdash \ast : 1 \)} \\
\Gamma \vdash t_i : \sigma_i & \quad (i = 1, 2) \\
\Gamma \vdash (t_1, t_2) : \sigma_1 \times \sigma_2 & \quad (i=1,2) \\
\Gamma \vdash \pi_i t : \sigma_i \\
\Gamma \vdash t : \sigma_1 \times \sigma_2 & \quad \Gamma \vdash \text{inl}^\sigma t : \sigma_1 + \sigma_2 \\
\Gamma \vdash \text{inr}^\sigma t : \sigma_1 + \sigma_2 & \quad \text{if } X \neq Y \land \text{Pos}(X, \tau), \text{if } X \neq Y
\end{align*}
\]

\(^4\)See Section 3.4 for the definitions of evaluation and values.
\[ \Gamma \vdash \text{case}(t_1, t_2) : \sigma \]

\[ \frac{\Gamma, x : \sigma \vdash t : \sigma'}{\Gamma \vdash (\lambda x. t) : \sigma \rightarrow \sigma'} \]

\[ \frac{\Gamma \vdash t_1 : \sigma \rightarrow \sigma' \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1 \cdot t_2 : \sigma'} \]

\[ \frac{\Gamma \vdash t : \sigma}{\Gamma \vdash \Lambda X.t : \forall X.\sigma} \]

\[ \frac{\Gamma \vdash \text{in}(u) : (u \mu(u)) \rightarrow \mu(u)}{\Gamma \vdash \text{out}(u) : \nu(u) \rightarrow (u \nu(u))} \]

\[ \frac{\Gamma \vdash \text{in}^{-1}(u) : \mu(u) \rightarrow (u \mu(u))}{\Gamma \vdash \text{out}^{-1}(u) : (u \nu(u)) \rightarrow \nu(u)} \]

\[ \frac{\Gamma \vdash t : (u \tau) \rightarrow \tau}{\Gamma \vdash R(u)[r]t : \mu(u) \rightarrow \tau} \]

\[ \frac{\Gamma \vdash t : \tau \rightarrow u \tau}{\Gamma \vdash G(u)[r]t : \tau \rightarrow \nu(u)} \]

Type and type constructor equality is simply α-equality.\(^5\)

### 3.4 Operational semantics

The values of the calculus are the closed terms of the form

\[ v \in \text{values} \quad ::= \quad * \mid (v_1, v_2) \mid \text{in}^\sigma v \mid \text{inr}^\sigma v \mid \lambda x.t \mid \Lambda X.t \mid \text{in}(u) \mid \text{in}(u)v \mid \text{in}^{-1}(u) \mid R(u)[r]v \mid \text{out}(u) \mid \text{out}^{-1}(u) \mid \text{out}^{-1}(u)v \mid G(u)[r]v \mid G(u)[r]v_1 v_2 \]

These are the values of the \(\lambda^ML\) core, plus the "partially applied" (co-)inductive forms.

The one-step evaluation function \(\longrightarrow\) is defined upon well-typed terms by the following judgments. Its reflexive and transitive closure is denoted by \(\longrightarrow^*\). Although lazy products and co-products would also improve efficiency, we use eager (co-)products and call-by-value which are sufficient for the desired efficiency results of induction and co-induction.

\[ t_1 \longrightarrow t_1' \quad t_2 \longrightarrow t_2' \quad t \longrightarrow t' \quad i = 1, 2 \]

\[ (t_1, t_2) \longrightarrow (t_1', t_2) \quad (v, t_2) \longrightarrow (t, t_2') \quad \pi_i t \longrightarrow \pi_i t' \]

\[ \text{inl}^\sigma t \longrightarrow \text{inl}^\sigma t' \quad \text{inr}^\sigma t \longrightarrow \text{inr}^\sigma t' \]

\(^5\)In particular, the calculus does not have the equality \(\Gamma \vdash \mu(u) = u(\mu(u))\) as found in many object-oriented type systems using "recursive" types. Here, these two expressions denote distinct, but isomorphic types, where \(\text{in}(u)\) and \(\text{in}^{-1}(u)\) are coercions.
\[
\begin{align*}
t &\to t' \\
\text{case}(t, t_1, t_2) &\to \text{case}(t', t_1, t_2) \\
t_1 &\to t'_1 \\
\text{case}(v, t_1, t_2) &\to \text{case}(v, t'_1, t_2) \\
t_2 &\to t'_2 \\
\text{case}(v, v_1, t_2) &\to \text{case}(v, v_1, t'_2)
\end{align*}
\]

\[
\begin{align*}
t &\to t' \\
t \to t'' &\to t' \to t'' \\
v &\to v' \\
t[u] &\to t'[u]
\end{align*}
\]

\[
\begin{align*}
t &\to t' \\
\text{in}^\times(u) &\to \text{in}^\times(u) t' \\
\text{in}^{-1}(u) &\to \text{in}^{-1}(u) t' \\
t &\to t' \\
\text{out}^\times(u) &\to \text{out}^\times(u) t' \\
\text{out}^{-1}(u) &\to \text{out}^{-1}(u) t'
\end{align*}
\]

\[
\begin{align*}
t &\to t' \\
\text{R} \{u\}[r] &\to \text{R} \{u\}[r] t' \\
\text{G} \{u\}[r] &\to \text{G} \{u\}[r] t' \\
\text{case}(\text{in}^\times v, v_1, v_2) &\to v_1 v \\
\text{case}(\text{in}^\times v, v_1, v_2) &\to v_2 v \\
\pi_t(v_1, v_2) &\to v_1 \\
(\lambda z : \sigma . t) v &\to t[v/x] \\
\text{in}^{-1}(u) &\to \text{in}^{-1}(u) v \\
\text{out}^\times(u) &\to \text{out}^\times(u) v \\
\text{R} \{u\}[r] v_1 &\to v_1 (\text{R} \{u\}[r] v_1) \\
\text{G} \{u\}[r] v_1 &\to \text{G} \{u\}[r] (v_1) \\
\text{out}^\times(u) &\to v_1 (\text{G} \{u\}[r] v_1) \\
\text{R} \{u\}[r] v_1 &\to \text{R} \{u\}[r] (v_1) \\
\text{G} \{u\}[r] v_1 &\to \text{G} \{u\}[r] (v_1)
\end{align*}
\]

These last \( \to \cdot \) rules represent infinite families of reductions indexed by \( u \), since \( \Phi \{u\}[r_1][r_2] \) if \( t \) is meta-notation defined by induction on the structure of \( u \) as below.\(^6\)

Together, the type constructor \( u \) and \( \Phi \{u\}[r_1][r_2] \) if \( t \) correspond to the object (type) morphism and map (term) morphism of the functor \( \Phi \) of Section 2. Since the latter is defined in terms of the former it is sufficient to index the terms such as \( \text{in}^\times(u) \) as such, rather than indexing by the functor as in category theory.

The definition of \( \Phi \{u\}[r_1][r_2] \) is

\(^6\)This definition is adapted in conjunction with Daniel Leivant and is a correction of the analogous definitions in Hagino [8, 9] and Leivant [12].
if $u \equiv \lambda X.X$, \quad f \quad t
if $u \equiv \lambda X.Y, X \neq Y$, \quad t
if $u \equiv \lambda X.1$, \quad * 
if $u \equiv \lambda X.r_1 \times r_2$, \quad \langle \Phi\{\lambda X.r_1\}[r_1]/r_2 \rangle f (\pi_1 t), \Phi\{\lambda X.r_2\}[r_1]/r_2 \rangle f (\pi_2 t))
if $u \equiv \lambda X.r_1 + r_2$, \quad \text{case}(t, \lambda x_1 : r_1[r_1/X].\text{inl}(\Phi\{\lambda X.r_1\}[r_1]/r_2 f x_1),$
\quad \lambda x_2 : r_2[r_1/X].\text{inr}(\Phi\{\lambda X.r_2\}[r_1]/r_2 f x_2))
if $u \equiv \lambda X.r_1 \rightarrow r_2$, \quad \lambda x : r_1[r_2/X].$
\quad \Phi\{\lambda X.r_2\}[r_1]/r_2 \rangle f (t(\Phi\{\lambda X.r_1\}[r_1]/r_2 f x))$
if $u \equiv \lambda X.\mu(u')$ \quad \text{R}\{u'[r_1/X]\}[\mu(u'[r_2/X])]$
and $Y$ fresh, \quad \langle \lambda x : u'[r_1/X].\mu(u'[r_2/X]).$
\quad \text{in}\{u'[r_2/X]\}$
\quad (\Phi\{\lambda X.u' Y\}[r_1]/r_2 f x)[\mu(u'[r_2/X])/Y])$
\quad t$
if $u \equiv \lambda X.\nu(u')$ \quad \text{G}\{u'[r_1/X]\}[^\nu(u'[r_2/X])]$
and $Y$ fresh, \quad \langle \lambda x : \nu(u'[r_1/X]).$
\quad \text{in}\{u'[r_1/X]\}$
\quad (\Phi\{\lambda X.u' Y\}[r_1]/r_2 f x)[\nu(u'[r_1/X])/Y])$
\quad t$
where the definition of $\Phi\{u\}[r_1]/r_2 f t$ is almost identical, except that $\Phi$ and $\overline{\Phi}$ are interchanged:

if $u \equiv \lambda X.Y, X \neq Y$, \quad t
if $u \equiv \lambda X.1$, \quad * 
if $u \equiv \lambda X.r_1 \times r_2$, \quad \langle \overline{\Phi}\{\lambda X.r_1\}[r_1]/r_2 \rangle f (\pi_1 t), \overline{\Phi}\{\lambda X.r_2\}[r_1]/r_2 \rangle f (\pi_2 t))$
if $u \equiv \lambda X.r_1 + r_2$, \quad \text{case}(t, \lambda x_1 : r_1[r_1/X].\text{inl}(\overline{\Phi}\{\lambda X.r_1\}[r_1]/r_2 f x_1),$
\quad \lambda x_2 : r_2[r_1/X].\text{inr}(\overline{\Phi}\{\lambda X.r_2\}[r_1]/r_2 f x_2))
if $u \equiv \lambda X.r_1 \rightarrow r_2$, \quad \lambda x : r_1[r_2/X].$
\quad \overline{\Phi}\{\lambda X.r_2\}[r_1]/r_2 \rangle f (t(\overline{\Phi}\{\lambda X.r_1\}[r_1]/r_2 f x))$
if $u \equiv \lambda X.\mu(u')$ \quad \text{R}\{u'[r_1/X]\}[\mu(u'[r_1/X])]$
and $Y$ fresh, \quad \langle \lambda x : u'[r_1/X].\mu(u'[r_1/X]).$
\quad \text{in}\{u'[r_1/X]\}$
\quad (\overline{\Phi}\{\lambda X.u' Y\}[r_1]/r_2 f x)[\mu(u'[r_1/X])/Y])$
\quad t$
if $u \equiv \lambda X.\nu(u')$ \quad \text{G}\{u'[r_1/X]\}[^\nu(u'[r_1/X])]
and $Y$ fresh, \quad \langle \lambda x : \nu(u'[r_1/X]).$
\quad \text{in}\{u'[r_1/X]\}$
\quad (\overline{\Phi}\{\lambda X.u' Y\}[r_1]/r_2 f x)[\nu(u'[r_1/X])/Y])$
The expression $\Phi(u)[\tau_1][\tau_2] f t : (u \tau_2)$ is well-defined when

- $f : \tau_1 \rightarrow \tau_2$,
- $t : u \tau_1$, and
- $X$ occurs only positively in $u X$, i.e., $\neg\text{Neg}(X,u X)$.

Intuitively, $f$ is applied to the appropriate subterms of $t$, as syntactically directed by the type constructor $u$. Similarly, $\Phi(u)[\tau_1][\tau_2] f t : (u \tau_1)$ is well-defined when

- $f : \tau_1 \rightarrow \tau_2$,
- $t : u \tau_2$, and
- $X$ occurs only negatively in $u X$.

The reductions which allow the one-step cancellation of inverse constants are the significant additions to previous work. In other papers, functions have had to simulate these constants via Corollary 1. But, without these inverse cancellation reductions, reduction sequences are significantly longer.

The operational semantics is type sound:

**Theorem 4** If $\Gamma \vdash t : \sigma$ and $t \leadsto v$, then $\Gamma \vdash v : \sigma$.

**Proof Sketch:** Each evaluation step is type consistent.  

It also guarantees that evaluation terminates:

**Conjecture 1** If $\vdash t : \sigma$, then there exists a unique value $v$ such that $t \leadsto v$.

**Proof Sketch:** By the translation given in Appendix B, all terms can be mapped to terms in $F_2$. The translation preserves reduction, i.e., $t \leadsto u$ implies $t \bigleadsto_{F_2} u$. Since $F_2$ is strongly normalizing, any evaluation sequence for $\lambda^{MM\mu\nu}$ which respects the $F_2$ translation and standard semantics must terminate. The evaluation function $\leadsto$ meets this requirement.

## 4 Inductive Types

Inductive types are those definable with the $\mu$ type constructor. They represent tree structures of finite depth. Some examples are

\[
\begin{align*}
\text{Void} & \equiv \mu(\lambda X.X) \\
\text{Nat} & \equiv \mu(\lambda X.1 + X) \\
\text{List}_A & \equiv \mu(\lambda X.1 + A \times X) \\
\text{BinaryTree}_A & \equiv \mu(\lambda X.A + X \times X)
\end{align*}
\]
binary tree with labels only on leaves

\[ A \text{FancyTree}_A \equiv \mu(\lambda X. A + A \times X \times X + A \times X \times X \times X) \]

binary/ternary tree with all nodes labelled

\[ \text{FancierTree}_{A, B} \equiv \mu(\lambda X. A + A \times (B \to X)) \]

tree with B-branching and A-labelled nodes

By convention, the definition of type \( (\cdot)_A \) has a free type variable \( A \), and \( (\cdot)_r \equiv (\cdot)_A[\tau/A] \).

While inductive types are usually defined in the form \( \mu(\lambda X. r_1 + \cdots + r_n) \), this is not necessary. For example, \( \text{FancierTree}_{A, B} \) is isomorphic to \( \mu(\lambda X. A \times (1 + (B \to X))) \). However, any inductive type not isomorphic to \text{Void} is isomorphic to some type given in the conventional form.

Observe that if \( X \) does not occur in \( r \), then \( \mu(\lambda X. r) \) is isomorphic to \( r \).

A number of the examples in this section are adapted from [26].

## 4.1 Maps to inductive types (and constants)

It is helpful to first examine the structure of inductive constants and constructors. Recall that from a categorial perspective, constants are isomorphic to constructors mapping from type 1. The patterns are most easily explained by example.

For \( \mu(u) \equiv \text{Nat} \), i.e., \( u \equiv \lambda X. 1 + X \):

\[
\begin{align*}
0 & \equiv \text{in}(u)(\text{inl } *)^7 \\
1 & \equiv \text{in}(u)(\text{inr } 0) \\
2 & \equiv \text{in}(u)(\text{inr } 1) \\
\text{succ} & \equiv \lambda n : \text{Nat}. \text{in}(u)(\text{inr } n)
\end{align*}
\]

For \( \mu(u) \equiv \text{List}_A \), i.e., \( u \equiv \lambda X. 1 + A \times X \):

\[
\begin{align*}
\text{null} & \equiv \lambda A. \text{in}(u)(\text{inl } *)^6 \\
[b] & \equiv \text{in}(u)(\text{inr}(b, \text{null}(A))) \\
[a, b] & \equiv \text{in}(u)(\text{inr}([a, b])) \\
\text{cons} & \equiv \lambda A. \lambda a l : A \times \text{List}_A. \text{in}(u)(\text{inr } a l)
\end{align*}
\]

For \( \mu(u) \equiv \text{BinaryTree}_A \) (abbreviated \( \text{BT}_A \)), i.e., \( u \equiv \lambda X. A + X \times X \):

\[
\begin{align*}
\cdot c & \equiv \text{in}(u)(\text{inl } c) \\
\cdot a \quad \cdot b & \equiv \text{in}(u)(\text{inr}(\text{in}(u)(\text{inl } a), \text{in}(u)(\text{inl } b)))
\end{align*}
\]

\[^7\]These terms are very similar to Church numerals if coproducts are encoded into the remaining calculus in the standard way. E.g., \( 0 \equiv \text{in}(u)(\lambda Z. \lambda s : Z. s : \text{Nat} \to Z. s 0) \) and \( 1 \equiv \text{in}(u)(\lambda Z. \lambda s : Z. s : \text{Nat} \to Z. s 0) \).

\[^6\]Remember that \( A \) is free in \( u \)!
leaf \equiv \Lambda A.\lambda a : A.\text{in}\{u\}(\text{inl} \ a)
m\makeBT \equiv \Lambda A.\lambda tt : BT_A \times BT_A.\text{in}\{u\}(\text{inr} \ tt)

As seen from these examples and from its type, \text{in}\{u\} is required to "package" a term into the type \(\mu(u)\).

The uncurried forms of constructors such as \text{cons} and \text{makeBT} are more natural as a result of using products in the definitions of \(\text{List}_A\) and \(\text{BinaryTree}_A\).

4.2 Inductive functions: Maps from inductive types

When defining an inductive\(^9\) function \(g \equiv R(u)[\tau]f\), it is often convenient to use one or both of the tools used here. One method is to give a set of recurrence equations and extract the function \(f\). This extraction can be aided by using the commuting diagrams of \(F\)-algebras, the second method.

The form of the recurrence equations for three common inductive types (each of these having two constructors) is given here.

\[
\begin{align*}
\text{Nat}: & \quad g(0) \equiv f_1[\ast] \\
& \quad g(\text{succ} \ n) \equiv f_2(g \ n) \\
\text{List}_A: & \quad g(\text{null}[A]) \equiv f_1[\ast] \\
& \quad g(\text{cons}[A](a, l)) \equiv f_2(a, g \ l) \\
\text{BinaryTree}_A: & \quad g(\text{leaf}[A] \ a) \equiv f_1[\ast] \\
& \quad g(\text{makeBT}[A](t_1, t_2)) \equiv f_3(g \ t_1, g \ t_2)
\end{align*}
\]

where \(f_i : \tau_i[\tau/X] \rightarrow \tau\), and \(f \equiv \lambda x : u.\text{case}(x, f_1, f_2)\).

Example 2 To illustrate, we define \(\text{even?} : \text{Nat} \rightarrow \text{Bool}\) (assume that standard Boolean functions are defined\(^10\)). In particular, we wish to satisfy the inductive recurrences

\[
\begin{align*}
\text{even?} \ 0 & \equiv \text{true} \\
\text{even?} \ (\text{succ} \ n) & \equiv \text{not}(\text{even?} \ n)
\end{align*}
\]

The first equivalence is the same as

\[
\text{even?} \ (0^1 [\ast]) \equiv \text{true}^1 [\ast]
\]

which fits the form of recurrences given at the beginning of the section. (On following examples, we leave this sort of expansion to the reader.) The above are equivalent to the commutativity of this diagram:

\(^9\)In this context, \textit{inductive} is equivalent to \textit{iterative}, rather than \textit{primitive recursive}. As will be shown in Section 4.3, iteration is as powerful as primitive recursion.

\(^10\)These definitions are straightforward from either \(\text{Bool} \equiv 1 + 1\) or \(\text{Bool} \equiv \mu(\lambda X.1 + 1)\). See Appendix A.
Translating these views of the desired definition into the syntax of the calculus, we first observe

\[
\Phi(\lambda N.1 + N)[Nat][Bool] f t \equiv \text{case}(t, \lambda u : 1.\text{inl} *, \lambda n : \text{Nat.inr}(f n))
\]

and then define \( f \) using \( f_1 \equiv \text{true} \) and \( f_2 \equiv \text{not} \):

\[
f \equiv \lambda x : 1 + \text{Bool}.\text{case}(x, \text{true}, \text{not})
\]

\[
even? \equiv R(\lambda N.1 + N)[\text{Bool}]f
\]

Omitting the unwieldy type information from the relevant \( \Phi \) above, the following evaluation sequence demonstrates iteration using these definitions.

\[
even? 1
\]

\[
\overset{*}{\longrightarrow} f(\Phi \ even? \ (\text{inr} \ 0))
\]

\[
\equiv f(\text{case}(\text{inr} \ 0, \lambda u : 1.\text{inl} *, \lambda n : \text{Nat.inr}(\text{even?} \ n)))
\]

\[
\overset{*}{\longrightarrow} f(\text{inr}(\text{even?} \ 0))
\]

\[
\overset{*}{\longrightarrow} f(\text{inr}(f(\Phi \ even? \ (\text{inl} \ *))))
\]

\[
\overset{*}{\longrightarrow} f(\text{inr}(f(\text{inl} \ *)))
\]

\[
\overset{*}{\longrightarrow} \text{not true}
\]

\[
\overset{*}{\longrightarrow} \text{false}
\]

**Example 3** The \( \text{car} \), or first element, of a list.\(^{11}\)

\[
\text{car}[A](\text{null}[A]) \equiv \text{inl} *
\]

\[
\text{car}[A](\text{cons}[A](a, l)) \equiv \text{inr}(\pi_1(a, \text{car}[A] l))
\]

\(^{11}\)This definition provides “error checking”, i.e., detection of traditionally erroneous \( \text{car}[A](\text{null}[A]) \), via coproducts. Alternatively, we could base a definition on the simpler relations

\[
\text{car}[A](\text{null}[A]) \equiv \text{error} \quad \text{car}[A](\text{cons}[A](a, l)) \equiv \pi_1(a, \text{car}[A] l)
\]

for some constant \( \text{error} : A \).
1 + A × List_A \xrightarrow{Id + Id × car[A]} 1 + A × (1 + A)

\[null[A]^t, cons[A]] \xrightarrow{inl + inr × \pi_1}

\[List_A \xrightarrow{car[A]} 1 + A\]

\[\Phi\{\lambda L.1 + A \times L\} f t \equiv \text{case}(t, \lambda u : 1.\text{inl} *, \lambda al : A \times List_A.\text{inr}(\pi_1 al, f(\pi_2 al)))\]

\[f \equiv \lambda x : 1 + A \times (1 + A).\text{case}(x, \lambda u : 1.\text{inl} u, \lambda y : A \times (1 + A) .\text{inr}(\pi_1 y))\]

\[car \equiv \Lambda A.R\{L, 1 + A \times L\}[1 + A]f\]

A typical evaluation sequence of an application of \textit{car} is

\[\text{car}[\text{Nat}][1, 2]\]

\[\xrightarrow{*} f(\Phi (R[1 + \text{Nat}]f) (\text{inr}(1,[2])))\]

\[\equiv f(\text{case}(\text{inr}(1,[2])), \lambda u : 1.\text{inl} *, \lambda al : A \times List_A.\text{inr}(\pi_1 al, R[1 + \text{Nat}]f(\pi_2 al)))\]

\[\xrightarrow{*} f(\text{inr}(1, R[1 + \text{Nat}]f [2]))\]

\[\xrightarrow{*} \text{inr}(\pi_1(1, 2))\]

\[\xrightarrow{\pi_1} \text{inr}}\]

This evaluation sequence requires a number of steps linear in the length of the list as, in the example, \(R[1 + \text{Nat}]f [2] \equiv \text{car}[\text{Nat}][2]\) must be evaluated. If pairing were lazy, this would not be the case, and \textit{car} would only take constant time. However, a better definition of \textit{car} can be given, as in Example 12, which requires only constant-time even with eager pairing.

Many inductive destructors can be defined in a similar way. The other destructors can be defined inductively as in Examples 6 and 7. In Section 4.4, we will show a much simpler way to define all of these destructors.

Example 4 Test whether all leaves in a binary tree are even numbers.

\[\text{leavesEven?}(\text{leaf}[\text{Nat}][n]) \equiv \text{even? } n\]
leavesEven?(makeBT[Nat](s, t)) ≡ and (leavesEven? s, leavesEven? t)

\[ \text{Nat} + BT_{Nat} \times BT_{Nat} \xrightarrow{Id + \text{leavesEven?}} \text{Nat} + \text{Bool} \times \text{Bool} \]

\[ [\text{leaf}[\text{Nat}], \text{makeBT}[\text{Nat}]] \]

\[ BT_{Nat} \xrightarrow{\text{leavesEven?}} \text{Bool} \]

Multiple argument functions can be defined via currying as in the following example.

Example 5 Addition of two natural numbers. The recurrences

\[
\begin{align*}
\text{plus } 0 \ n & \equiv n \\
\text{plus } (\text{succ } \ m) \ n & \equiv \text{plus } (\text{succ } n) \\
\end{align*}
\]

are equivalent to

\[
\begin{align*}
\text{plus } 0 & \equiv \text{Id} \\
\text{plus } (\text{succ } \ m) & \equiv \lambda n : \text{Nat}. \text{plus } (\text{succ } n) \\
\end{align*}
\]

The categorical equivalent of currying is exponentiation:

\[
\frac{1 + \text{Nat}}{\text{Id} + \text{Id} \times \text{plus}} \xrightarrow{1 + \text{Nat} \rightarrow \text{Nat}}
\]

\[
\begin{align*}
\text{[0, succ]} & \\
\text{[Id, } \lambda y : \text{succ]} & \\
\end{align*}
\]

\[
\frac{\text{Nat}}{\text{plus}} \xrightarrow{\text{Nat} \rightarrow \text{Nat}}
\]

\[
f \equiv \lambda x : 1 + \text{Nat} \rightarrow \text{Nat}. \\
\text{case}(x, \lambda u : 1. \text{Id}^{\text{Nat}}, \lambda y : \text{Nat} \rightarrow \text{Nat}. \lambda n : \text{Nat}. y(\text{succ } n))
\]

\[
\text{plus} \equiv \text{R}[\text{Nat} \rightarrow \text{Nat}]f
\]

4.3 Primitive recursion

Simple induction is a valuable tool, as the previous sections shows. However, it is also overly constrained in practice. In particular, we would like to use the more convenient notion of primitive recursion such that, for example, the following recurrence equations hold.
\( Nat: \ g 0 \equiv f_1 \ast \\
(\text{succ } n) \equiv f_2(n, g n) \)

\( \text{List}_A: \ g(\text{null}[A]) \equiv f_1 \ast \\
(\text{cons}[A](a, l)) \equiv f_3(a, (l, g l)) \)

\( \text{BT}_A: \ g(\text{leaf}[A] a) \equiv f_1 a \\
(g(\text{makeBT}[A](t_1, t_2)) \equiv f_2((t_1, g t_1), (t_2, g t_2)) \)

It is well-known that induction can implement primitive recursion, e.g., [20, 26]. However, such simulation is, in an intuitive sense, frequently too inefficient. This intuition has been formalized for (the Church numeral encoding of) natural numbers by Parigot in [24]. This section shows examples of this simulation, and how to even go beyond primitive recursion. Section 4.4 shows how this inefficiency sometimes can be avoided in \( \lambda MM\nu \).

**Example 6** Natural number predecessor.\(^{12}\)

\[
\begin{align*}
\text{pred} &\ 0 \equiv 0 \\
\text{pred} &\ (\text{succ } n) \equiv n
\end{align*}
\]

This is not in the form of an inductive definition, but is in the above primitive recursive form. So, since inductive types can also be encoded in pure \( F_2 \), it should not be surprising that functions such as \( \text{pred} \) and \( \text{cdr} \) may be defined using the same pairing technique common in \( F_2 \):

\[
\begin{align*}
\text{predPair} &\ 0 \equiv (0, 0) \\
\text{predPair} &\ (\text{succ } n) \equiv (\text{succ}(\pi_1(\text{predPair } n)), \pi_1(\text{predPair } n))
\end{align*}
\]

In particular, this definition implies

\[
\begin{align*}
\text{predPair} \ n &\ \equiv (n, \text{pred } n) \\
\text{pred } n &\ \equiv \pi_2(\text{predPair } n)
\end{align*}
\]

\[
\begin{array}{c}
\begin{array}{c}
1 + Nat \\
[0^1, \text{succ}]
\end{array} \xrightarrow{\text{Id} + \text{predPair}} \\
\begin{array}{c}
1 + Nat \times Nat \\
\text{predPair}
\end{array}
\end{array}
\]

\(^{12}\)We omit error checking, e.g.

\[
\begin{align*}
\text{pred} \ 0 &\ \equiv \text{inl } \ast \\
\text{pred} \ (\text{succ } n) &\ \equiv \text{inr } n
\end{align*}
\]

from this and following examples for simplicity.
\[ f \equiv \lambda x : 1 + \text{Nat} \times \text{Nat}. \]
\[
\text{case}(z, \lambda u : 1. (0, 0), \lambda n : \text{Nat} \times \text{Nat}. (\text{succ}(n), n))
\]
\[ \text{predPair} \equiv R[\text{Nat} \times \text{Nat}]f \]
\[ \text{pred} \equiv \lambda n : \text{Nat}. \pi_2(\text{predPair} n) \]

Like its F\textsubscript{2} counterpart, this definition requires linear time, even with alternative definitions of \(-\), since \text{predPair} n must be calculated to determine \text{predPair}(\text{succ} n).

Example 7 The \text{cdr} of a list (such that \text{cdr}[A](\text{null}[A]) \rightharpoonup \text{null}[A]), using the same technique. We define

\[
\text{cdrPair}[A](\text{null}[A]) \equiv (\text{null}[A], \text{null}[A])
\]
\[
\text{cdrPair}[A](\text{cons}[A](a, l)) \equiv (\text{cons}[A](a, \pi_1(\text{cdrPair}[A] l)), \pi_1(\text{cdrPair}[A] l))
\]

so that

\[
\text{cdrPair}[A] l \equiv (l, \text{cdr}[A] l)
\]
\[
\text{cdr}[A] l \equiv \pi_2(\text{cdrPair}[A] l)
\]

\[
1 + A \times \text{List}_A \xrightarrow{\text{Id} + \text{Id} \times \text{cdrPair}[A]} 1 + A \times (\text{List}_A \times \text{List}_A)
\]

\[
\xrightarrow{[\text{null}[A]^t, \text{cons}[A]]} \quad \xrightarrow{f}
\]

\[
\text{List}_A \xrightarrow{\text{cdrPair}[A]} \text{List}_A \times \text{List}_A
\]

f \equiv \lambda x : 1 + A \times (\text{List}_A \times \text{List}_A).
\[
\text{case}(z, \lambda u : 1. (\text{null}[A], \text{null}[A]), \lambda y : A \times (\text{List}_A \times \text{List}_A).
\]
\[
\text{cdrPair} \equiv \Lambda A. R[\text{List}_A \times \text{List}_A]f
\]
\[ \text{cdr} \equiv \lambda l : \text{List}_A. \pi_2(\text{cdrPair}[A] l) \]

Example 8 The factorial of a natural number.

\[
\text{fact} 0 \equiv 1 \quad \text{fact} (\text{succ} n) \equiv \text{times} n (\text{fact} n)
\]

Again, the above definition is not expressed in the inductive form, and we must use a pairing technique similar to before, with the relations

\[
\text{factPair} 0 \equiv (0, 1)
\]
The general form of primitive recursion, recurring over an inductive type \( \mu(u) \), resulting in a type \( \tau \), is given by the following commuting diagram [20].

Given a function \( f \) (this corresponds to \( (f_1, f_2) \) from the beginning of the section), we get

\[
\begin{align*}
\Phi\{u\} \pi_1 & \quad \Phi\{u\}(Id, pr\{u\}[\tau]f) \\
\Phi\{u\} \pi_1 & \quad u(\mu(u) \times \tau) \\
in\{u\} & \quad f' \\
\mu(u) & \quad f
\end{align*}
\]

So,

\[
pr\{u\}[\tau]f = \pi_2 \circ (R\{u\}[\mu(u) \times \tau]f')
\]

Example 9 Primitive recursion for natural numbers and lists. These definitions follow the tradition of separate base- and inductive-case functions \( f_i \), instead of a single \( f \).

\[
pr\{\lambda N.1 + N\} \equiv \\
\lambda A.\lambda f_1 : A.\lambda f_2 : (Nat \times A) \rightarrow A.\lambda n : Nat. \\
\pi_2(R\{\lambda N.1 + N\}[Nat \times A] \\
(\lambda y : 1 + Nat \times A. \\
\text{case}(y, \lambda u : 1.(0,f_1), \\
\lambda na : Nat \times A.(\mu\succ(\pi_1 na), f_2 na))) \\
\pi_1
\]

\[
pr\{\lambda L.1 + A \times L\} \equiv
\]
\[ \Lambda B. \lambda f_1 : B. \lambda f_2 : A \times (\text{List}_A \times B) \rightarrow B. \lambda l : \text{List}_A. \]
\[ \pi_2(\text{R}\{L, 1 + A \times L\}[\text{List}_A \times B]) \]
\[ (\lambda y : 1 + A \times (\text{List}_A \times B).) \]
\[ \text{case}(y, \lambda u : 1. (0, f_1),) \]
\[ \lambda \text{alb} : A \times (\text{List}_A \times B).) \]
\[ (\text{cons}[A](\pi_1 \text{alb}, \pi_1(\pi_2 \text{alb})),) \]
\[ f_2 \text{alb}))) \]

Using these we can define, for example,

\[ \text{pred} \equiv \text{pr} \{\lambda N. 1 + N\}[\text{Nat}] 0 (\lambda n n : \text{Nat} \times \text{Nat}. n n), \text{and} \]
\[ \text{cdr} \equiv \Lambda A. \text{pr} \{\lambda L. 1 + A \times L\}[\text{List}_A] \]
\[ \text{null}[A] (\lambda \text{all} : A \times (\text{List}_A \times \text{List}_A). \pi_1(\pi_2 \text{all})) \]

Extending this technique, we can generalize beyond the form of primitive recursion, allowing a function to depend on \( i \) previous recursive calls, for a given fixed \( i \). I.e.,

\[ g 0 \equiv f_1 \quad \ldots \quad g(i - 1) \equiv f_i \]
\[ g(i + n) \equiv f_{i + 1}(n, g n, \ldots, g(i + n - 1)) \]

Primitive recursion in the traditional number-theoretic sense corresponds to \( \text{prNat}[\text{Nat}] \). However, using induction (or primitive recursion) with a higher-order type, it is possible to define functions which are not primitive recursive in the traditional sense. The common example of such a function is Ackermann's function.

**Example 10** Ackermann's function.

\[ \text{ack} \equiv \lambda m : \text{Nat}. \]
\[ \text{R}(\lambda N. 1 + N)[\text{Nat} \rightarrow \text{Nat}] \]
\[ (\lambda y : 1 + \text{Nat} \rightarrow \text{Nat}. \]
\[ \text{case}(y, \text{succ}^! , \lambda f : \text{Nat} \rightarrow \lambda n : \text{Nat}. \]
\[ \text{R}(\lambda N. 1 + N)[\text{Nat}] \]
\[ (\lambda z : 1 + \text{Nat}. \text{case}(z, 1^!, f)) \]
\[ (\text{succ } n)) \]

4.4 Inductive destructors

Since \( \text{in}\{u\} \) is used to obtain the constructors for an inductive type, its inverse, \( \text{in}^{-1}\{u\} \), gives the corresponding destructors. For example, the destructor for \( \text{Nat} \) is \( \text{pred} \) and those for \( \text{List}_A \) are \( \text{car} \) and \( \text{cdr} \). Now observe that some destructors require linear time when using definitions such as those presented so far. Comparing these definitions to Corollary 1, we see that other definitions using \( \text{in}^{-1} \) are available. Due to the reduction rule using \( \text{in}^{-1} \), these new definitions will be more efficient.
Example 11 Using the inverse of \( \text{in}\{\lambda N.1 + N\} \).

\[
\begin{align*}
\text{in}^{-1}0 & \equiv \text{in}^{-1}(\text{in}(\text{inl} *)) & \rightarrow & \text{inl} * \\
\text{in}^{-1}(\text{succ } n) & \equiv \text{in}^{-1}(\text{in}(\text{inr } n)) & \rightarrow & \text{inr } n
\end{align*}
\]

Again defining \( \text{pred} 0 \rightarrow 0 \), we can define

\[
\begin{align*}
\text{zero}\? & \equiv \lambda n : \text{Nat}. \text{case}(\text{in}^{-1}n, \text{true}, \lambda n : \text{Nat}. \text{false}) \\
\text{pred} \quad & \equiv \lambda n : \text{Nat}. \text{case}(\text{in}^{-1}n, 0\!, \text{Id}_{\text{Nat}})
\end{align*}
\]

And, we can compare this to Corollary 1 with the following diagram (both triangles commute) and definitions:

\[
\begin{align*}
f & \equiv \lambda x : 1 + (1 + \text{Nat}). \text{case}(z, \lambda u : 1. \text{inl } u, \lambda y : 1 + \text{Nat}. \text{inr}(y)) \\
\text{in}^{-1} & \equiv \text{R}[1 + \text{Nat}]f
\end{align*}
\]

This operational equivalence empirically confirms the corollary.

Unlike the definitions for \( \text{pred} \) given in the previous section, this is constant-time. For example, where \( n \) is a value,

\[
\begin{align*}
\text{pred}(\text{succ } n) & \rightarrow \text{case}(\text{in}^{-1}(\text{in}(\text{inr } n)), 0\!, \text{Id}_{\text{Nat}}) \\
& \rightarrow \text{case}(\text{inr } n, 0\!, \text{Id}_{\text{Nat}}) \\
& \rightarrow n
\end{align*}
\]

Example 12 Using the inverse of \( \text{in}\{L, 1 + A \times L\} \).

\[
\begin{align*}
\text{in}^{-1}(\text{null}[A]) & \equiv \text{in}^{-1}(\text{in}(\text{inl} *)) & \rightarrow & \text{inl} * \\
\text{in}^{-1}(\text{cons}[A](a, l)) & \equiv \text{in}^{-1}(\text{in}(\text{inr}(a, l))) & \rightarrow & \text{inr}(a, l)
\end{align*}
\]

\[
\begin{align*}
\text{null}\? & \equiv \Lambda A. \lambda l : \text{List}_A. \text{case}(\text{in}^{-1}l, \text{true}, \lambda a : A \times \text{List}_A. \text{false}) \\
\text{car} & \equiv \Lambda A. \lambda l : \text{List}_A. \text{case}(\text{in}^{-1}l, \lambda u : 1. \text{inl } *, \lambda a : A \times \text{List}_A. \text{inr}(\text{in} u a)) \\
\text{cdr} & \equiv \Lambda A. \lambda l : \text{List}_A. \text{case}(\text{in}^{-1}l, \lambda u : 1. \text{inl } *, \lambda a : A \times \text{List}_A. \text{inr}(\text{in} u a))
\end{align*}
\]
5 Co-inductive Types

The dual of inductive types, co-inductive types are defined with the \( \nu \) type constructor. They represent tree structures of potentially countably infinite depth. Some examples are:

\[
\begin{align*}
Nat_\omega &\equiv \nu(\lambda N.1 + N) \\
&\text{natural numbers and their limit } \omega \\
Stream_A &\equiv \nu(\lambda S.1 + A \times S) \\
&\text{finite and infinite length streams} \\
InfStream_A &\equiv \nu(\lambda S.A \times S) \\
&\text{infinite length streams} \\
InfTree_A &\equiv \nu(\lambda X.A \times \text{List}_X) \\
&\text{finite branching, infinite depth trees} \\
FancierTree_{A,B} &\equiv \nu(\lambda X.A + A \times (B \rightarrow X)) \\
&\text{B-branching, potentially infinite depth trees}
\end{align*}
\]

For each co-inductive type, there are terms of that type which represent objects of infinite size. For types such as \( \text{Stream}_A \), there are also terms which represent finite-sized objects.

The dual types \( \mu(u) \) and \( \nu(u) \) represent similar collections of objects. For example, \( \text{List}_A \) and \( \text{Stream}_A \) both represent sequences of elements of type \( A \). The co-inductive type is isomorphic to its dual inductive type plus some infinite-sized objects. Another example of this correspondence is between \( \text{Nat} \) and \( \text{Nat}_\omega \).

5.1 Co-inductive functions and simple terms: Maps to co-inductive types

For co-inductive types, there are two convenient definition methods for simple terms and constructors. The first is based on dualizing inductive simple terms and constructors. Later examples will point out a method using the categorical idea that constants of type \( \sigma \) are obtained from maps of type \( \Gamma \rightarrow \sigma \).

For \( \nu(u) \equiv \text{Nat}_\omega \), i.e., \( u \equiv \lambda N.1 + N \):
\[
\begin{align*}
\text{zeroNat}_\omega &\equiv \text{out}^{-1}\{u\}(\text{inl } *) \\
\text{succNat}_\omega &\equiv \lambda n : \text{Nat}_\omega.\text{out}^{-1}\{u\}(\text{inr } n)
\end{align*}
\]
Note: \( \omega \) must be defined using the second method. See Example 20.

For \( \nu(u) \equiv \text{Stream}_A \) (abbreviated \( \text{Str}_A \)), i.e., \( u \equiv \lambda S.1 + A \times S \):
\[
\begin{align*}
\text{emptyStream} &\equiv \lambda A.\text{out}^{-1}\{u\}(\text{inl } *) \\
[a, b] &\equiv \text{out}^{-1}\{u\}(\text{inr}(\text{out}^{-1}\{u\}(\text{inr}(b, \text{emptyStream}[A])))) \\
\text{consStream} &\equiv \lambda A.\lambda as : A \times \text{Stream}_A.\text{out}^{-1}\{u\}(\text{inr } as)
\end{align*}
\]

For \( \nu(u) \equiv \text{InfStream}_A \) (abbreviated \( \text{IStr}_A \)), i.e., \( u \equiv \lambda X.A \times \text{List}_X \):
\[
\text{consIStr}\equiv \lambda A.\text{out}^{-1}\{u\}
\]
Due to duality, maps to co-inductive types are similar to maps from inductive types, in that such functions involve the induction operator and commuting diagrams.

However, the general recurrence patterns of co-induction are not as convenient as those of induction as shown in Section 4.2. To define a function 

\[ g \equiv G[X]f : X \rightarrow \nu(\lambda X.\tau) \]

the recurrence equations of co-induction for the above three types are

\[ \text{Nat}w: \quad g \ z \equiv \text{case}(f \ x, \text{zero} \ \text{Nat}w(y), \lambda y : X.\text{succ} \ \text{Nat}w(g \ y)) \]

\[ \text{Str}_A: \quad g \ z \equiv \text{case}(f \ x, \text{empty} \ \text{Stream}[A], \lambda ax : A \times X.\text{cons} \ \text{Stream}[A](\pi_1 \ ax, g(\pi_2 \ ax))) \]

\[ \text{IStr}_A: \quad g \ z \equiv \text{cons} \ \text{Stream}[A](\pi_1(f \ x), g(\pi_2(f \ x))) \]

Example 13 The empty stream (alternate method). Using the idea that the constant is essentially a map of type \( 1 \rightarrow \text{Stream}_A \), we use the general pattern of co-induction on streams.

\[ 1 + A \times 1 \xrightarrow{\text{Id} + \text{Id} \times \text{empty} \ \text{Stream}[A] \text{T}} 1 + A \times \text{Str}_A \]

\[ f = \text{inl} \quad \text{out} \]

\[ \Phi(\lambda S.A \times S)[1][\text{Str}_A] \ f \ t \equiv \text{case}(t, \lambda u : 1.\text{inl} *, \lambda au : A \times 1.\text{inr}((\pi_1 au, g(\pi_2 au)))) \]

\[ \text{empty} \ \text{Stream} \equiv \Lambda A.\Phi(\lambda S.1 + A \times S)[1](\lambda u : 1.\text{inl} u) * \]

Evaluation of out(\text{empty} \ \text{Stream}[A]):

\[
\text{out}(\text{empty} \ \text{Stream}[A]) \\
\quad \rightarrow \Phi(G[1](\lambda u : 1.\text{inl} u))((\lambda u : 1.\text{inl} u) *) \\
\quad \equiv \text{case}((\lambda u : 1.\text{inl} u) *, \\
\quad \quad \lambda u : 1.\text{inl} *, \\
\quad \quad \lambda au : A \times 1.\text{inr}((\pi_1 au, G[1](\lambda u : 1.\text{inl} u)(\pi_2 au)))) \\
\quad \rightarrow \text{inl} * 
\]
Example 14 Similarly, we can define \( \text{consIStream} \) using this alternate method, and compare the result to Corollary 1. It fits into the above recurrence pattern as

\[
\text{consIStream}[A](a, s) \cong \text{consIstream}[A](a, \text{consIStream}[A](\text{out } s))
\]

since \( \text{consIStream}[A] \cong \text{out}^{-1} \).

\[
\begin{array}{c}
A \times (A \times \text{Istr}_A) \\
\xrightarrow{\text{Id} \times \text{consIStream}[A]} \\
A \times \text{Istr}_A
\end{array}
\]

\[
\begin{array}{c}
\text{Id} \times \text{out} \\
\xrightarrow{\text{consIStream}[A]} \\
\text{out}
\end{array}
\]

\[
\begin{array}{c}
A \times \text{Istr}_A \\
\xrightarrow{\Phi[A \times \text{Istr}_A]} \\
\text{Istr}_A
\end{array}
\]

\[
\Phi[A \times \text{Istr}_A] (A \times \text{Istr}_A) = (f_1, f(f_2))
\]

\[
f \equiv \lambda as : A \times \text{Istr}_A. (f_1 as, \text{out}(f_2 as))
\]

\[
\text{consIStream} \equiv \Lambda A. G[A \times \text{Istr}_A] f
\]

Unfortunately, destructing a term built using this definition is computationally expensive. In essence, destructing a stream built with this \( \text{consIStream} \) propagates the \text{out} found in the above definition of \( f \). E.g., given any \( \text{Istr}_{\text{Nat}} \),

\[
\text{tailIstr}[A](\text{consIStream}[A](1, \text{is}))
\]

\[
\xrightarrow{\ast} \pi_2(\text{out}(\text{consIStream}[A](1, \text{is})))
\]

\[
\xrightarrow{\ast} \pi_2(\Phi[G[A \times \text{Istr}_A] f](f(1, \text{is})))
\]

\[
\equiv \pi_2((\pi_1(f(1, \text{is}))), G[A \times \text{Istr}_A] f(\pi_2(f(1, \text{is}))))
\]

\[
\xrightarrow{\ast} G[A \times \text{Istr}_A] f(\text{out } \text{is})
\]

The last term in this reduction sequence is observationally equivalent to \( \text{is} \), but computing the \text{headIstr} or \text{tailIstr} of this stream requires a longer reduction sequence than does destructing \( \text{is} \). Using the previous definition of \( \text{consIStream} \equiv \text{out}^{-1} \) avoids this problem since

\[
\pi_2(\text{tailIstr}[A](\text{out}^{-1}(1, \text{is}))) \xrightarrow{\ast} \text{is}.
\]

So, just as the inverse of \( \text{in}(u) \) efficiently defines an inductive type's destructors, the inverse of \( \text{out}(u) \) produces efficient co-inductive constructors.

---

22
Example 15 The infinite stream of natural numbers from a given one, e.g., \( \text{natsFrom } 2 \cong \{2,3,4,\ldots\} \). From the relation

\[
\text{natsFrom } n \cong \text{consStream}[\text{Nat}](n, \text{natsFrom}(\text{succ } n))
\]

we can obtain the equivalent

\[
\text{out } (\text{natsFrom } n) \cong (n, \text{natsFrom } (\text{succ } n))
\]

which is the natural counterpart for the commuting diagram

\[
\begin{array}{c}
\text{Nat} \times \text{Nat} \xrightarrow{\text{Id} \times \text{natsFrom}} \text{Nat} \times \text{IStr}_{\text{Nat}} \\
\text{(Id, succ)} \downarrow \quad \downarrow \text{out} \\
\text{Nat} \xrightarrow{\text{natsFrom}} \text{IStr}_{\text{Nat}}
\end{array}
\]

\[
\text{natsFrom } \equiv \text{G}[\text{Nat}](\lambda n : \text{Nat}.(n, \text{natsFrom } (\text{succ } n)) )
\]

Example 16 The infinite stream of a constant \( c : A \). The recurrence

\[
\text{constIS}[A]c \cong \text{consStream}[A](c, \text{constIS}[A]c)
\]

leads to the definition

\[
\text{constIS } \equiv \text{A}A.\lambda c : A.\text{G}\{\text{A}A.\lambda x (A \times S)x[1](\lambda u : 1.(c, u))\} \ast
\]

Example 17 The infinite stream of factorial numbers. We first define

\[
\text{factsHelp } (m, n) \cong \text{consStream}[\text{Nat}](n, \text{factsHelp } (\text{times } m n, \text{succ } n))
\]

which encapsulates the incremental computation of the factorials. To define the stream of all factorials, we must seed the computation with some numbers, in this case the first factorial number and the first number to multiply by:

\[
\text{facts } \equiv \text{factsHelp } (1, 1)
\]

\[
\begin{array}{c}
\text{Nat} \times (\text{Nat} \times \text{Nat}) \xrightarrow{\text{Id} \times \text{factsHelp}} \text{Nat} \times \text{IStr}_{\text{Nat}} \\
\text{f} \downarrow \quad \downarrow \text{out} \\
\text{Nat} \times \text{Nat} \xrightarrow{\text{factsHelp}} \text{IStr}_{\text{Nat}}
\end{array}
\]
\( f \equiv \lambda n n : \text{Nat} \times \text{Nat}. \)

\( (\pi_1 n n, (\text{times} (\pi_1 n n) (\pi_2 n n), \text{succ}(\pi_2 n n))) \)

\( \text{factsHelp} \equiv G[\text{Nat} \times \text{Nat}]f \)

\( \text{facts} \equiv \text{factsHelp} (1, 1) \)

**Example 18** The infinite stream of Fibonacci numbers

\( \text{fibsHelp} (m, n) \equiv \text{consIStream}[\text{Nat}](m, \text{fibsHelp} (n, \text{plus} m n)) \)

\( f \equiv \lambda n n : \text{Nat} \times \text{Nat}. (\pi_1 n n, (\pi_2 n n, \text{plus} (\pi_1 n n) (\pi_2 n n))) \)

\( \text{fibsHelp} \equiv G[\text{Nat} \times \text{Nat}]f \)

\( \text{fibs} \equiv \text{fibsHelp} (0, 1) \)

**Example 19**Appending two streams. The following definition is based on the relations

\[ \text{appStr}[A](e, e) \equiv e \]

\[ \text{appStr}[A](e, \text{consStr}[A](a, s)) \equiv \text{consStr}[A](a, \text{appStr}[A](e, s)) \]

\[ \text{appStr}[A](\text{consStr}[A](a, s), t) \equiv \text{consStr}[A](a, \text{appStr}[A](s, t)) \]

using the abbreviation \( e \equiv \text{emptyStream}[A] \).

Using destructors defined in Section 5.3 and boolean functions defined in Appendix A,

\( f \equiv \lambda s s : \text{Str}_A \times \text{Str}_A. \)

\( \text{ite} (\text{emptyStream}[A](\pi_1 s s)) \)

\( (\text{ite} (\text{emptyStream}[A](\pi_2 s s)) \)

\( (\text{inl} *, \text{inr}(\text{headStr}[A](\pi_2 s s), (\text{inl} *, \text{tailStr}[A](\pi_2 s s)))) \)

\( \text{inr}(\text{headStr}[A](\pi_1 s s), (\text{tailStr}[A](\pi_1 s s), \pi_2 s s))) \)

\( \text{appStr} \equiv \Lambda A. G[\text{Str}_A \times \text{Str}_A]f \)

**Example 20** The term \( \omega : \text{Nat} \omega \). The desired equality \( \omega \equiv \text{succNat} \omega \) \( \omega \) is equivalent to \( \text{out} (\omega^t *) \equiv \text{inc}(\omega^t *) \). So, we have \( \omega \equiv G[1]\lambda u : 1. \text{inr} u \) \( \ast \). Note than an alternate definition for \( \text{zeroNat} \omega \) is the very similar \( G[1]\lambda u : 1. \text{inl} u \) \( \ast \).

Since \( \text{Nat} \omega \) is isomorphic to \( \text{Stream}_1 \), we can adapt Example 19 to define (non-curried) addition on \( \text{Nat} \omega \):

\( f \equiv \lambda n n : \text{Nat} \times \text{Nat} \omega. \)

\( \text{ite} (\text{zeroNat} \omega ? (\pi_1 n n)) \)

\( (\text{ite} (\text{zeroNat} \omega ? (\pi_2 n n)) \)

\( (\text{inl} *, \text{inr}(\text{inl} *, \text{predNat} \omega (\pi_2 n n))) \)

\( \text{inr}(\text{predNat} \omega (\pi_1 n n), \pi_2 n n)) \)

\( \text{plusNat} \omega \equiv G[\text{Nat} \omega \times \text{Nat} \omega]f \)

24
5.2 Primitive co-recursion

Dualizing the diagram for primitive recursion, we obtain the following commuting diagram [20].

\[ \begin{array}{c}
\Phi \{ u \} \inl \\
\Phi \{ u \} \ [I d, p c o r \{ u \}[r] f] \\
u \ (\nu(u) + \tau) \\
\Phi \{ u \} \ [I d, p c o r \{ u \}[r] f] \\
\Phi \{ u \} \inl \\
u(u) \\
\end{array} \]

\[ \begin{array}{c}
\tau \rightarrow f \rightarrow f' \\
\nu(u) + \tau \rightarrow [I d, p c o r \{ u \}[r] f] \\
\inl \rightarrow \nu(u) \\
\end{array} \]

So, given a function \( f \), we find \( f' \) by

\[ f' = ((\Phi \{ u \} \ inl) \ o \ out\{ u \}, f) \]

Then, since the rectangle commutes,

\[ [I d, p c o r \{ u \}[r] f] = G\{ u \}[\nu(u) + \tau]f' \]

So,

\[ p c o r \{ u \}[r] f = \text{inr} \circ G\{ u \}[\nu(u) + \tau]f' \]

Recurrence equations for a primitive co-recursive function \( g \) would be based on the equation

\[ g \ x \cong \Phi \{ u \} \ (\lambda y : \nu(u) + X. \text{case}(y, Id'^{(u)}), g)) \ (f \ x) \]

for each constructor \( u \).

**Example 21** Translating the definition of \( p c o r \) into \( \lambda^M M^\nu \) syntax leads to, for example,

\[ p c o r \{ Natw \} \equiv \Lambda A. \lambda f : A \rightarrow 1 + (Natw + A). \lambda x : A. \\
inr(\text{G}\{ \text{Natw} \}[\text{Natw} + A] \\
(\lambda y : Natw + A. \\
\text{case}(y, \lambda n : Natw. \\
\text{case}(\text{out}(\lambda N.1 + N)n, \\
\lambda u : 1.\text{inl} *, \\
\lambda m : Natw.\text{inr}(\text{inl} m)), \\
f)) \\
x) \]
An example of pcor is the zip function, which maps a pair of streams into a stream of pairs.

\[
zipIS \equiv \lambda A.pcor_\{\lambda S.A \times S\}[IS_A \times IS_A] \\
(\lambda ss : IS_A \times IS_A.
  ((headIStr[A](\pi_1 ss), headIStr[A](\pi_2 ss)) \\
  \text{inr}(tailIStr[A](\pi_1 ss), tailIStr[A](\pi_2 ss))))
\]

Notice the similarity in structure to the definitions of appStr and plusNatw which could also be defined via primitive co-recursion.

5.3 Maps from co-inductive types

The use of out\{u\} is characteristic of such functions from co-inductive types, as it "unrolls" or "forces" an object one step, the only way to access the information packaged by G\{u\}. The simplest examples of its use are destructors and base-case tests.

**Example 22** Test for the empty stream.

\[
\text{emptyStream?} \equiv \lambda A.ss : Str_A.\text{case(out s, true, }\lambda as : A \times Str_A.\text{false)}
\]

**Example 23** The head and tail destructors for streams.

\[
\text{headStr }\equiv \lambda A.ss : Str_A.\text{case(out s, }\lambda u : 1.\text{inl s, }\lambda as : A \times Str_A.\text{inr(\pi_1 as))}
\]

\[
\text{tailStr }\equiv \lambda A.ss : Str_A.\text{case(out s, }\lambda u : 1.\text{inl s, }\lambda as : A \times Str_A.\text{inr(\pi_2 as))}
\]

\[
\text{headIStr }\equiv \lambda A.ss : IStr_A.\pi_1(\text{out s}) \\
\text{tailIStr }\equiv \lambda A.ss : IStr_A.\pi_2(\text{out s})
\]

**Example 24** Maps from Natw.

\[
\text{zeroNatw?} \equiv \lambda n : Natw.\text{case(out n, true, }\lambda m : Natw.\text{false)}
\]

\[
\text{predNatw }\equiv \lambda n : Natw.\text{case(out n, zero, }\lambda m : Natw.\text{m)}
\]

As desired, predNatw \(\rightarrow^*\) \(\omega\).

The function \(\omega\) cannot be defined. Since out\(^{-1}\) provides the only way to "unroll" and examine an object of type Natw, the only way to define such as test is by the above destructors. Thus, such a function would have to decrement its argument until zeroNatw was obtained, so it cannot be written in the calculus. Similarly, any such test of infiniteness on any co-inductive type is impossible.

**Example 25** The function nextnode returns the next node of the tree paired with a new tree with which to continue the search.
nezinode ≡ ΛA.λt :InfTree_A. 
  (π₁(out t),
   makeT_A(π₁(out(car[A]([π₂(out t)]))),
   append[InfTree_A]
   (cdr[A]([π₂(out t)]),
    π₂(out(car[A]([π₂(out t)])))))

where makeT_A ≡ out⁻¹ is the constructor for InfTree_A.

Repeated uses of neztnode results in breadth-first search of a finitely branch-
ing infinite tree:

search[A]t ≡ consiS[A](π₁(neztnode[A]t), search[A](neztnode[A]t))

search ≡ ΛA.G{IAS.A X S}[^TA](neztnode[A])

Thus, search[A]t is an infinite stream of the nodes of t in breadth-first order.

6 Summary of Related Work

Hagino [8, 9] uses a generalization of algebras called di-algebras. This allows
him not to assume any base types or the constructors + and ×. Instead, all
types are defined with a μ or ν constructor and a possibly empty list of functors
corresponding to the τᵢ's in μ(λX.τ₁ + ... + τₙ).

Coquand and Paulin [5], and Pfenning and Paulin-Mohring [25] are similar
in that they also do not assume any base types or the constructors + and ×.
However, they do not work in a category theoretic framework and they use
inductive, but not co-inductive types.

Mendler [20] explains primitive recursion and its dual in terms of category
theory, using a generalization of algebras. Using these as primitives instead of
(co-)induction, a calculus using the ideas outlined would allow constant-time
encodings of our inverses.

Pierce, Dietzen, and Michaylov [26] present an example-based tutorial on
programming in the F₁ hierarchy of calculi, using iterators for inductive types.

Both Leivant [12] and Parigot [22, 23, 24] view programs with inductive
data types as being derived from proofs. Leivant formalizes the extraction of
programs from several families of calculi, giving numerous examples. Instead
of extending a calculus to improve efficiency, Parigot examines alternate en-
codings of Nat in F₂ (optionally extended with fix) which allow constant-time
destuctors.

Michaylov and Pfenning [21] describe a process to compile F₂ terms to F₂
extended with constants for inductive constructors and recursors. It translates,
for example, the common pair-based pred function similar to Example 6 to a
constant-time function using recursors. A more systematic approach to defining
the extensions in the target calculus could be obtained from our \( \mu \) types and related terms.

Leivant [13] looks at Church numerals in a predicative version of \( F_2 \), describing precisely what computations can be defined on that type. When adding inductive types to this stratified calculus, he shows that the type \( \mu X. \tau \) is at the same level as \( \tau \). In \( \lambda MM \mu \), this means that \( \mu (\lambda X. \tau) \) is a type and not a type scheme. He also proves that the addition of inductive types "does not result in new functions being representable, but it does allow new algorithms".

Burstall [4] extends ML with an inductive case statement, and relates programming with inductive types to specification with abstract data types, another area which uses initial algebras and sometimes final co-algebras. Hagino [10] extends ML with a "codatatype" declaration and a "merge" statement which corresponds to G. Wraith [27] uses a rougher equivalent system. Both notations, however, assume definitions of co-inductive types are of the form \( \nu (\lambda X. \tau_1 \times \cdots \times \tau_n) \), which makes use of types such as \( \text{Stream}_A \) inconvenient.


With the assumption that \( C \) is the category of CPO's, Meijer, Fokkinga, and Paterson [15, 7] eliminate the distinction between least and greatest fixed points. Thus, \( \text{in}^{-1} \{ u \} = \text{out} \{ u \} \). This allows additional elegant recursion schemes, but introduces non-strictness.

7 Conclusions

Algebraic datatypes are a valuable abstraction for programming, as terms are easily defined directly from their specifications, i.e., recurrence equations or simple category theory diagrams. Using the morphisms \( \text{in}^{-1} \) and \( \text{out}^{-1} \) provides straightforward means of obtaining constant-time inductive destructors and co-inductive constructors, which significantly improves efficiency as compared to similar calculi. It has also been shown that conceptually infinite objects can be used with ease. However, when termination is guaranteed, the usefulness of co-inductive datatypes is significantly restricted, as many common functions cannot be defined.

8 Comments and Future Research

The calculus as presented is rather verbose from explicit types. Type inference should be explored to eliminate or reduce the amount of explicit type information necessary. A ML-like type declaration facility together with pattern matching, as in [4, 27, 10] would also be useful, but work still needs to be done for co-inductive types.

A formal model of the calculus would involve formalizing the points raised in Section 2, in particular, detailing the structure of the "category of all types".
It would be interesting to base the calculus on the full $\lambda^M L$ calculus, reintroducing higher kinds. In that case, it is more difficult to enforce the positivity constraint on (co-)inductive types that ensures that $\Phi$ is well-defined. Alternatively, dropping the positivity constraint altogether introduces non-termination and requires redefinition of the evaluation of $R$ and $G$, since $\Phi$ is not always definable.[14]

The syntax of the calculus creates one unfortunate semantic problem. It is not Church-Rosser when using the standard $\eta$ rule and an inductive $\eta$-like rule corresponding to Theorem 2, $R\{u\}[\mu(u)]\text{lin}\{u\}\; t \longrightarrow t$. In this case, the diamond property does not hold for $\lambda x : \mu(u).R\{u\}[\mu(u)]\text{lin}\{u\}\; t$. A simple fix is to only allow "fully applied" forms such as $R\{u\}[r]t$ to be terms. The problem does not seem to arise with the alternate inductive $\eta$-like rule, $R\{u\}[\mu(u)]\text{lin}\{u\} \longrightarrow I d^R(u)$.

More could be learned about (co-)inductive terms in $F_2$ by translating our examples, for example, as shown in Appendix B. Also, we would like to examine more closely the duality of types $\mu(u)$ and $\nu(u)$. Furthermore, our familiarity of co-inductive constructs is still not as developed as our understanding of programming with inductive types. More examples could come from extracting co-inductive programs from proofs.

Acknowledgements

Many thanks to Chris Colby, Daniel Leivant, the ever-questioning Mark Lillibridge, and Benjamin Pierce, and especially Bob Harper and Frank Pfenning for their ideas, suggestions, and proof-reading.

A Other functions

The most direct definition of $\text{Bool}$, with some typical functions:

$$\text{Bool} \equiv 1 + 1$$

$$\text{true} \equiv \text{inl } *$$

and $\equiv \lambda bb : \text{Bool} \times \text{Bool}.\text{case}(\pi_1 \; bb, \pi_2 \; bb, \text{false})$

$$\text{ite} \equiv \lambda b : \text{Bool}.\lambda aa : \text{A} \times \text{A}.\text{case}(b, \lambda u : 1.\pi_1 \; aa, \lambda u : 1.\pi_2 \; aa)$$

An alternative definition, allowing use of the iterator $R$ to define $\text{ite}$:

$$\text{Bool} \equiv \mu(\lambda X.1 + 1)$$

$$\text{true} \equiv \text{inl } *$$

and $\equiv \lambda bb : \text{Bool} \times \text{Bool}.\text{case}(\pi_1 (\text{in}^{-1} bb), \pi_2 (\text{in}^{-1} bb), \text{false})$

$$\text{ite} \equiv \lambda b : \text{Bool}.R[A \times A \rightarrow A](\lambda x : 1 + 1.\text{case}(x, \lambda u : 1.\lambda aa : A \times A.\pi_1 \; aa, \lambda u : 1.\lambda aa : A \times A.\pi_2 \; aa))$$

Some other basic functions on natural numbers; note the similarity of $\text{monus}$ and $\text{plus}$ (Exercise 5):
eq? ≡ R[Nat → Bool]
(λx : 1 + Nat → Bool.
  case(x, zero? x, λy : Nat → Bool.λn : Nat.y(pred m))

leq? ≡ R[Nat → Bool]
(λx : 1 + Nat → Bool.
  case(x, λu : 1.λn : Nat.true,
    λy : Nat → Bool.λn : Nat.
      ite (zero? n) (false, y(pred n))))

monusHelp ≡ R[Nat → Nat]
(λx : 1 + Nat → Nat.
  case(x, λu : 1.IdNat,
    λy : Nat → Nat.λn : Nat.y(pred m))

monus ≡ λn : Nat.λm : Nat.monusHelp m n

divrem ≡ R[Nat → (Nat × Nat)]
(λx : 1 + Nat → (Nat × Nat).
  case(x, λu : 1.λm : Nat.(0, 0),
      ite (eq? (ir2(y n)) (pred n))
        (succ(π₁(y n)), 0)
        (π₁(y n), succ(π₂(y n))))

div ≡ λm : Nat.λn : Nat.π₁(divrem m n)
rem ≡ λn : Nat.λm : Nat.π₂(divrem m n)

diff ≡ prNat[Nat → Nat] IdNat
  ite (zero? n) (succ(π₁x), (π₂x) (pred n)))

Filtering and accumulating, on a list:

filter ≡ λp : A → Bool.
  R[ListA](λx : 1 + A × ListA.
    case(x, λu : 1.inl *,
      λal : A × ListA.ite (p(π₁ al)) (al, π₂ al))

accumulate ≡ R[((A × B) → B) → B → B]
(λx : 1 + A × (((A × B) → B) → B → B).
  case(x, λu : 1.λf : (A × B) → B.IdB,
    λy : A × (((A × B) → B) → B → B).
      λf : (A × B) → B.λb : B.
        (π₂y f (f(π₁ y, b))))

The equivalent functions on streams are not total and, therefore, not definable in the calculus.

Merging sorted infinite streams, allowing duplicates:
mergeIS ≡ λA.G[IS_A × IS_A]  
(λss : IS_A × IS_A.
  
it {\text{ite (leq? (headIStr}[A](π_1 ss)) (headIStr}[A](π_2 ss))}
  ((headIStr}[A](π_1 ss), (tailIStr}[A](π_1 ss), (tailIStr}[A](π_2 ss)),
  (headIStr}[A](π_2 ss), (π_1 ss, tailIStr}[A](π_2 ss))))

The type \text{Unit} ≡ ν(λX.X) is the dual to \text{Void} and thus isomorphic to 1.
Adapting examples from the isomorphic type InfStream₁,

\text{unit} ≡ G[1]Id^₁ *  
\text{Id}^{\text{Unit}} ≅ \text{out}^{-1} ≅ G[\text{Unit}]\text{out} ≅ \text{out}

Many more inductive type examples can be adapted from [3].

B Translation to \text{F}_2

Since \text{F}_2 has figured prominently in the work on (co-)inductive types, we give a translation _−_ from \text{λMMI\nu}_2 to \text{F}_2. Type and term variables occurring only on the right-hand side of the equations are assumed to be fresh.

\[
\begin{align*}
  X & \equiv X \\
  1 & \equiv \forall X.X \rightarrow X \\
  \sigma_1 \times \sigma_2 & \equiv \forall X.(\sigma_1 \rightarrow \sigma_2 \rightarrow) \rightarrow X \\
  \sigma_1 + \sigma_2 & \equiv \forall X.((\sigma_1 \rightarrow X) \rightarrow (\sigma_2 \rightarrow X) \rightarrow X \\
  \sigma \rightarrow \sigma' & \equiv \sigma \rightarrow \sigma' \\
  \mu(\lambda X.r) & \equiv \forall X.(\tau \rightarrow X) \rightarrow X \\
  \nu(\lambda X.r) & \equiv \forall Y.((\forall X.(X \rightarrow \tau) \rightarrow X \rightarrow Y) \rightarrow Y \\
  \forall X.\sigma & \equiv \forall X.\sigma
\end{align*}
\]

\[
\begin{align*}
  z & \equiv z \\
  x & \equiv \Lambda X.\lambda x : X.x \\
  (t_1, t_2) & \equiv \Lambda X.\lambda p : \sigma_1 \rightarrow \sigma_2 \rightarrow X.p t_1 t_2 \quad \text{(if} \; t_i : \sigma_i) \\
  \tau_1 t & \equiv t[\sigma_1](\lambda l : \sigma_1, \lambda r : \sigma_2.l) \quad \text{(if} \; t : \sigma_1 \times \sigma_2) \\
  \tau_2 t & \equiv t[\sigma_2](\lambda l : \sigma_1, \lambda r : \sigma_3.r) \quad \text{(if} \; t : \sigma_1 \times \sigma_2) \\
  \text{inl}^\tau t & \equiv \Lambda X.\lambda l : \sigma_1 \rightarrow X.\lambda r : \sigma_2 \rightarrow X.l \; t \quad \text{(if} \; t : \sigma_1) \\
  \text{inr}^\tau t & \equiv \Lambda X.\lambda l : \sigma_1 \rightarrow X.\lambda r : \sigma_2 \rightarrow X.r \; t \quad \text{(if} \; t : \sigma_2) \\
  \text{case}(t, t_1, t_2) & \equiv t[\sigma]t_1 t_2 \quad \text{(if} \; t_i : \sigma_i \rightarrow \sigma) \\
  \lambda x : \sigma. t & \equiv \lambda x : \sigma. t \\
  \Lambda X. t & \equiv \Lambda X. t
\end{align*}
\]
\[\begin{aligned}
\lambda x.\lambda y.\mu u.\lambda Y.\lambda f : u Y \rightarrow Y.
&f(\Phi[u]\mu(u)[Y] (R\{u\}[Y]f) h) \\
\text{out}^{-1}\{u\} &\equiv \lambda t : u(\mu(u)).G\{u\}[u \nu(u)](\Phi\{u\}[\nu(u)]u \nu(u)) \text{ out}\{u\} t \\
\text{out}\{u\} &\equiv \lambda t : u \nu(u).G\{u\}[u \nu(u)](\Phi\{u\}[\nu(u)]u \nu(u)) \text{ out}\{u\} t \\
\text{G}\{u\}[r]/f &\equiv \lambda x : z.\lambda z.\lambda h : (\forall W.(W \rightarrow u W) \rightarrow W \rightarrow Z.h[r]/f x \\
\end{aligned}\]

This translation maps \(\lambda^{MM\mu\nu}\) types into \(F_2\) types which are closely related to the standard \(F_2\) encodings of these types. For example,

\[\text{Nat} \equiv \forall Z.((\forall X.X \rightarrow X) \rightarrow Z) \rightarrow (Z \rightarrow Z) \rightarrow Z\]

which is isomorphic to \(\forall Z.Z \rightarrow (Z \rightarrow Z) \rightarrow Z\), the normal definition.

Naturally, using the definition of \(\text{pred}\) in Example 6, \(\text{pred}\) is similar to the standard \(F_2\) definition. However, translating Example 11 (and simplifying with isomorphisms for readability) results in an alternative:

\[\begin{aligned}
\text{pred} &\equiv_{F_2} \lambda n : \text{Nat}.n(\forall Z.Z \rightarrow (\text{Nat} \rightarrow Z) \rightarrow Z] \\
& (\lambda Z.\lambda x : Z.\lambda s : \text{Nat} \rightarrow Z.s) \\
& (\lambda y : (\forall Z.Z \rightarrow (\text{Nat} \rightarrow Z) \rightarrow Z). \\
& (\lambda Z.\lambda x : Z.\lambda s : \text{Nat} \rightarrow Z. \\
& (s(y(\forall Z.Z \rightarrow (\text{Nat} \rightarrow Z) \rightarrow Z] 0 \text{ succ)\}))) \\
&\text{Nat} 0 \text{ Id}_{\text{Nat}} \\
\end{aligned}\]

However, even using this definition, it requires linear-time to evaluate \(\text{pred} n\), since \(F_2\) does not have an one-step equivalent of \(\text{in}^{-1}(\text{in} t) \rightarrow t\).

References


32


