Subtyping and Inheritance for Categorical Datatypes

Preliminary Report

Erik Poll
University of Kent, Canterbury, UK
E.Poll@ukc.ac.uk

Abstract

We extend Hagino’s categorical datatypes with subtyping and a limited form of inheritance. The view of objects as coalgebras provides the inspiration for subtyping and inheritance for coalgebraic (or coinductive) types. Exploiting the duality between coalgebras and algebras then yields notions of subtyping and inheritance for algebraic (or inductive) types.

1 Introduction

Category theory is a very convenient formalism for describing datatypes. In particular, the dual notions of initial algebra and final coalgebra provide an interesting class of datatypes. This possibility was first exploited by Hagino in [Hag87a][Hag87b], and the categorical datatypes introduced there have since been used as the basis of the functional programming language Charity [CS92].

Initial algebras – or term algebras – provide algebraic or inductive datatypes, such as natural numbers, lists and trees. Final coalgebras provide coalgebraic or coinductive types containing (possibly) infinite data, such as infinite lists (or streams) and infinite trees. Final coalgebras can also be used as object types, as is described in [Rei95]. This observation is also made in [HP95], where it is noted that the encoding of object types given in [PT94] uses (weakly) final coalgebras. Here we use the word “object” in the OO sense, not the categorical sense, and by an "object type” we mean the type of all objects that provide a given set of methods, or, in other words, the type of all objects with a given interface.

This view of final coalgebras as object types provides the starting point for this paper. Two important features of object-oriented languages are subtyping and inheritance. If coalgebras can be used to model objects, an obvious question to ask is then:

\textit{Can we explain subtyping and inheritance in the coalgebraic setting?}

And, given that initial algebras are the duals of final coalgebras, another obvious question to ask is:

\textit{What are the duals of subtyping and inheritance?}
This paper tries to answer these questions. We show how subtyping can be explained in terms of coalgebras, and that this notion has an interesting dual for algebras. The dual of subtyping turns out to be supertyping, which is related to subtyping in the obvious way: \( A \) is a subtype of \( B \) if \( B \) is a supertype of \( A \). We also show that a limited form a inheritance can be explained in terms of coalgebras, and that this has an interesting dual for algebras, providing a form of code reuse for functions on algebraic datatypes. (In [Po97] we described these notions of subtyping and inheritance for algebraic datatypes in the setting of a functional programming language, without any reference to category theory or coalgebras.)

We begin Section 2 by defining initial algebras and final coalgebras. We then introduce some syntax for categorical datatypes that denote initial algebras and final coalgebras, and illustrate how final coalgebras can be used to model objects. Section 3 introduces a notion of subtyping for categorical datatypes and its interpretation as coercions between (co)algebras. Section 4 introduces a simple form notation of inheritance for categorical datatypes. Inspiration for subtyping and inheritance for coalgebras are subtyping and inheritance as found in object-oriented languages. Dualising these produces the corresponding notions for algebras.

## 2 Categorical Datatypes

In 2.1 we briefly review the notion of initial algebra and final coalgebra with (co)iteration. For a gentle introduction to algebras and coalgebras see [JR97]. In 2.2 we then introduce a syntax for declaring algebraic and coalgebraic datatypes that denote initial algebras and final coalgebras, and for defining iterative and coiterative functions We use this syntax to explain the coalgebraic view of objects in 2.3.

### 2.1 Algebras and Coalgebras

Let \( \mathbb{C} \) be a category with products and coproducts, and a terminal object \( \mathbb{1} \).

**Definition 2.1 (Initial Algebra)** Let \( F \) be a functor on \( \mathbb{C} \). Then

- An \( F \)-algebra is a pair \( (A, f) \) consisting of an object \( A \) and an arrow \( f : FA \rightarrow A \).

- If \( (A, f) \) and \( (B, g) \) are \( F \)-algebras, then an \( F \)-algebra homomorphism from \( (A, f) \) to \( (B, g) \) is an arrow \( h : A \rightarrow B \) such that the following diagram commutes

\[
\begin{array}{ccc}
FA & \xrightarrow{f} & A \\
\downarrow{Fh} & & \downarrow{h} \\
FB & \xrightarrow{g} & B
\end{array}
\]

- An \( F \)-algebra \( (\mu F, \text{in}_{\mu F}) \) is initial if for every \( F \)-algebra \( (B, g) \) there is a unique \( F \)-algebra homomorphism from \( (\mu F, \text{in}_{\mu F}) \) to \( (B, g) \).
The initial $F$-algebra, if it exists, is unique up to isomorphism.

Typically, we are interested in $F$-algebras where the functor $F$ is of the form $F(X) = F_1(X) + \ldots + F_n(X)$. $F$-algebras are then of the form $(A, [f_1, \ldots, f_n])$ with each $f_i : F_i(A) \to A$, and the unique algebra homomorphism from the initial algebra $(\mu F, [i_1, \ldots, i_n])$ to another algebra $(B, [g_1, \ldots, g_n])$ is the unique $h : \mu F \to B$ such that

\[
F_i \mu F \xrightarrow{i_i} \mu F \xrightarrow{\mu h} \mu F \xrightarrow{\mu g} B
\]

commutes for all $i$.

**Example 2.2 (Natural Numbers)**

Let $NatF$ be the functor $NatF(X) = 1 + X$. Then the initial $NatF$-algebra $(Nat, [\text{zero}, \text{succ}])$ is a natural numbers object. The arrows $\text{zero} : 1 \to Nat$ and $\text{succ} : Nat \to Nat$ are called the constructors of $Nat$. Initiality guarantees that for every $NatF$-algebra $(B, [b, g])$, i.e. for every $b : 1 \to B$ and $g : B \to B$, there exist a unique $h : Nat \to B$ such that

\[
h \circ \text{zero} = b \quad \text{and} \quad h \circ \text{succ} = g \circ h
\]

This recursion scheme above is known as *iteration*.

**Example 2.3 (Lists)**

Let $ListF$ be the functor $ListF(X) = 1 + Nat \times X$. The initial $ListF$-algebra $(List, [\text{nil}, \text{cons}])$, with $\text{nil} : 1 \to List$ and $\text{cons} : Nat \times List \to List$, is a object of finite lists of natural numbers. Initiality of guarantees that for every $ListF$-algebra $(B, [b, g])$, i.e. for every $b : 1 \to B$ and $g : Nat \times B \to B$, there exists a unique $h : List \to B$ such that

\[
h \circ \text{nil} = b \quad \text{and} \quad h \circ \text{cons} = g \circ (id_A \times h)
\]

An example of such an arrow is $\text{length} : List \to Nat$ that satisfies

\[
\text{length} \circ \text{nil} = \text{zero} \quad \text{and} \quad \text{length} \circ \text{cons} = \text{succ} \circ \pi_2 \circ (id_A \times \text{length})
\]

Dualising the definition of (initial) algebra yields the definition of (final) coalgebra:

**Definition 2.4 ((Final) Coalgebra)** Let $F$ be a functor on $C$. Then

- An $F$-coalgebra is a pair $(A, f)$ consisting of an object $A$ and an arrow $f : A \to F(A)$.
• If \((A, f)\) and \((B, g)\) are \(F\)-coalgebras, then an \(F\)-coalgebra homomorphism from \((B, g)\) to \((A, f)\) is an arrow \(h : B \to A\) such that the following diagram commutes

\[
\begin{array}{c}
FA \xrightarrow{f} A \\
\downarrow h \quad \downarrow h \\
FB \xrightarrow{g} B
\end{array}
\]

• An \(F\)-coalgebra \((\nu F, \text{out}_F)\) is final or terminal if for every \(F\)-coalgebra \((B, g)\) there is a unique morphism to \((\nu F, \text{out}_F)\) from \((B, g)\).

Typically, we are interested in \(F\)-coalgebras where \(F\) is a functor. For example, if \(F(X) = F_1(X) \times \ldots \times F_n(X)\). \(F\)-coalgebras are then of the form \((A, (f_1, \ldots, f_n))\) with each \(f_i : A \to F_i(A)A\), and the unique coalgebra homomorphism from a coalgebra \((B, (g_1, \ldots, g_n))\) to the final coalgebra \((\nu F, (i_{n_1}, \ldots, i_{n_n}))\) is then the unique \(h : \nu F \to B\) such that

\[
\begin{array}{c}
F_i \nu F \xrightarrow{\text{out}_i} \nu F \\
\downarrow h_i \\
F_i B \xrightarrow{g_i} B
\end{array}
\]

commutes for all \(i\).

Standard examples of final coalgebras are infinite data structures, such as infinite lists:

**Example 2.5 (Streams)**

Let \(\text{Stream} F\) be the functor \(\text{Stream} F(X) = \text{Nat} \times X\). A final \(\text{Stream} F\)-coalgebra \((\text{Stream}, \langle\text{head, tail}\rangle)\) is an object of infinite lists or streams of natural numbers. The arrows \(\text{head} : \text{Stream} \to \text{Nat}\) and \(\text{tail} : \text{Stream} \to \text{Stream}\) are called destructors.

Let \((B, [\text{head}, \text{tail}])\) be another \(\text{Stream} F\)-algebra, i.e. \(\text{head} : B \to \text{Nat}\) and \(\text{tail} : B \to B\). Terminality guarantees then that there exists a unique \(h : B \to \text{Stream}\) such that

\[
\begin{align*}
\text{head} \circ t &= g_{\text{head}} \\
\text{tail} \circ h &= h \circ g_{\text{tail}}
\end{align*}
\]

This scheme is known as co-iteration. For any \(b : \mathbb{1} \to B\) we can think of \(h \circ b : \mathbb{1} \to \text{Stream}\) as the infinite list of natural numbers with \(b\) as its "seed" and with \(g_{\text{head}}\) and \(g_{\text{tail}}\) telling us how to compute the head and (the seed of) the tail for a given seed.

An example of an coiterative arrow is \(\text{from} : \text{Nat} \to \text{List}\) defined by

\[
\begin{align*}
\text{head} \circ \text{from} &= \text{id}_{\text{Nat}} \\
\text{tail} \circ \text{from} &= \text{from} \circ \text{succ}
\end{align*}
\]

For any \(n : \mathbb{1} \to \text{Nat}\), the arrow \(\text{from} \circ n : \mathbb{1} \to \text{Stream}\) then represents the infinite list \(n, \text{succ} \circ n, \text{succ}^2 \circ n, \ldots\). \(\square\)
2.2 Syntax for Categorical Datatypes

We introduce some syntax for declaring categorical datatypes that denote initial algebras and final coalgebras. An algebraic (or inductive) type is declared by listing its constructors and their types, e.g.

```haskell
data Nat =
    zero : Nat
    succ : Nat -> Nat
```

data List =
    nil : List
    cons : Nat × List -> List

and a coalgebraic (or coinductive) type is declared by listing its destructors and their types, e.g.

```haskell
codata Stream =
    head : Stream -> Nat
    tail : Stream -> Stream
```

Iterative functions on algebraic types are defined in the pattern-matching style used in functional programming, e.g.

```haskell
length : List -> Nat
    length nil = zero
    length (cons (a,l)) = succ (length l)
```

and co-iterative functions to coalgebraic types are defined in the dual way, e.g.

```haskell
from : Nat -> Stream
    head (from n) = n
    tail (from n) = from (succ n)
```

The interpretation of this syntax in the category C should be obvious, provided the required initial algebras and final coalgebras exist in C. We will not give a formal definition of the syntax and its interpretation. Our only reason for introducing a syntax at all is that it introduces names for constructors and destructors, which will be needed for subtyping.

Coalgebraic datatypes can be seen as recursive labelled products or records, for example

```haskell
Stream = Record{head : Nat, tail : Stream}.
```

Dually, algebraic datatypes can be seen as recursive labelled sums or variants, for example

```haskell
List = Variant{nil : List, cons : A × List}.
```

2.3 Coalgebraic Types as Object Types

As noted in [Rei95] and [HP95], a coalgebra can be viewed as an object type, the type of all objects with a certain interface. The only difference between object types and infinite datatypes is in the interpretation: we now think of the destructors as methods. For example, we can think of a stream as an object with methods head and tail. Another example of an object type is given below:
Example 2.6 (Counters)

The type

codata Counter with
  getcount : Counter -> Nat
  count   : Counter -> Counter

can be regarded as the type of all counter objects that have methods count and getcount. Applying the destructor getcount or count to a counter is then regarded as invoking getcount- or count-method of that counter. Counter does not specify anything about the way in which counters might be implemented, but only specifies their interface, i.e. lists the methods they should provide.

(Note that we are in a functional setting, so invoking count does not increase the count as side-effect, but produces a new counter. Of course, the type Counter is just the type Stream in disguise: head is called getcount and tail is called count.)

Suppose getcountimp:B->Nat and countimp:B->Nat for some type B. These provide a way to implement counters. Define

\[
\begin{align*}
h : B \to Stream & \quad \\
getcount (h \ b) &= \text{getcountimp} \ b \\
count (h \ b) &= h (\text{countimp} \ b)
\end{align*}
\]

Intuitively, h b is the counter object with a hidden state b:B and a method table containing getcountimp and countimp as implementations of the methods getcount and count. The first equation above says that invoking the method getcount of (h b) results in the application of getcountimp - the implementation of getcount given by the method table - to the hidden state b. The second equation says that the result of invoking the method count of (h b) is obtained by first applying the implementation of count - i.e. countimp - to the hidden state to produce a new state (countimp b) and then applying newCounter to produce a new counter object with this new state (countimp b) as its state.

The obvious implementation of counters is of course to have a state of type Nat and implementing getcount and count as the identity and succ, respectively:

\[
\begin{align*}
\text{newCounter} : \text{Nat} \to \text{Stream} & \\
\text{getcount} (\text{newCounter} \ n) &= n \\
\text{count} (\text{newCounter} \ n) &= \text{newCounter} (\text{succ} \ n)
\end{align*}
\]

The coiterative function above is a class definition in the sense of [PT94]. Coiteration allows only a very limited form of class definition, because methods cannot call other methods. (A more general form of class definition is provided in [PT94].)

Note that if in the definition above the type \text{Nat} is replaced with a one-field record type \text{Record}\{x:\text{Nat}\}, i.e.

\[
\begin{align*}
\text{newCounter}' : \text{Record}\{x:\text{Nat}\} \to \text{Stream} & \\
\text{getcount} (\text{newCounter}' \ n) &= n.x \\
\text{count} (\text{newCounter}' \ n) &= \text{newCounter} \{x=\text{succ} \ n.x\}
\end{align*}
\]

then the field x can be regarded as an instance variable. \qed
3 Subtyping

We now consider a subtyping relation on algebraic and coalgebraic types, and show how this subtyping can be understood as coercions between the corresponding initial algebras and final coalgebras.

Subtyping tries to capture a natural inclusion relation between types. Record types provide the standard example: there is an natural inclusion between the record types $\text{Record}\{x:\text{Nat}, y:\text{Nat}\}$ and $\text{Record}\{x:\text{Nat}\}$, since any record with an $x$- and a $y$-field of type $\text{Nat}$ is also a record an $x$-field of type $\text{Nat}$. This is usually written as

$$\text{Record}\{x : \text{Nat}, y : \text{Nat}\} \leq \text{Record}\{x : \text{Nat}\}.$$ 

By the so-called subsumption rule any term of type $\text{Record}\{x : \text{Nat}, y : \text{Nat}\}$ then also has type $\text{Record}\{x : \text{Nat}\}$. Sometimes subtyping can be understood as a set-theoretic inclusion between two types. But a more general way to understand subtyping is as an implicit coercion, i.e. $A \leq B$ means that there is a coercion function from $A$ to $B$ which is left implicit. For example, the coercion from $\text{Record}\{x : \text{Nat}, y : \text{Nat}\}$ to $\text{Record}\{x : \text{Nat}\}$ should of course be the function that maps a record $\{x = \text{N}, y = \text{M}\}$ to the record $\{x = \text{N}\}$.

Not any function will do as a coercion: coercions need to satisfy some properties to guarantee that leaving them implicit does not cause any ambiguity. For example, the coercion $\text{coerce} : \text{Record}\{x : \text{Nat}, y : \text{Nat}\} \rightarrow \text{Record}\{x : \text{Nat}\}$ should be such that $r.x = (\text{coerce } r).x$ (i). If this does not hold, e.g. if $\text{coerce}$ maps the record $\{x = \text{N}, y = \text{M}\}$ to the record $\{x = \text{M}\}$, then leaving the coercion implicit would introduce ambiguities. The absence of ambiguity in the presence of implicit coercions is known as coherence, and properties such as (i) are known as coherence conditions. Coherence conditions are naturally expressed as commuting diagrams, e.g.

$$\begin{array}{ccc}
\text{Record}\{x : \text{Nat}, y : \text{Nat}\} & \xrightarrow{x} & \text{Nat} \\
\downarrow \text{coerce} & & \downarrow \text{id} \\
\text{Record}\{x : \text{Nat}\} & \xrightarrow{x} & \text{Nat}
\end{array}$$

We now consider subtyping for (co)algebraic and coherence conditions for the coercions between (co)algebraic that provide the interpretation for this subtyping. We will not give complete proofs of coherence here. (Doing so would require a formal definition of a syntax and type system.) A formal definition of a type system providing algebraic and coalgebraic datatypes with subtyping, and the coherence of its categorical interpretation is left as future work.

3.1 Subtyping for Coalgebras

The subtyping found in object-oriented languages suggests how we can define a notion of subtyping for final coalgebras. In an object-oriented language a subclass typically has more methods than its superclass. In our setting, this corresponds to a coalgebra having more destructors. As an example, we will consider
the type of "resetable" counters, that in addition to count and getcount also have a destructor reset:

codata RCounter =
  count : RCounter -> RCounter
  getcount : RCounter -> Nat
  reset : RCounter -> RCounter

We would like RCounter to be a subtype of Counter:

RCounter ≤ Counter.

Informally, this subtyping may be justified by the observation that a RCounter-object is also a Counter-object, since it provides all the methods that an Counter-object does. Or, viewing coalgebraic types as record types, we can see that this subtyping is a special case of the usual subtyping on record types

RCounter = Record{count : RCounter, getcount : Nat, reset : RCounter}
≤ Record{count : Counter, getcount : Nat}
= Counter

It is a special case of subtyping on record types because these are recursive record types.

To interpret the subtyping between RCounter and Counter we need an implicit coercion between the final coalgebras they denote. This coercion is in fact definable in the syntax as a function coerce:RCounter->Counter. The definition of the coercion is suggested by the properties - coherence conditions - it has to satisfy for there is to be no ambiguity. We now consider what these coherence conditions are.

If o:RCounter then there are two ways to interpret (count o):Counter, namely

- the application of the coercion, yielding o:Counter, followed by the application of the Counter destructor count, or
- the application of the RCounter destructor count, giving as result (count o):Counter, followed by the application of the coercion to get a Counter.

Similarly, there are two ways to interpret (getcount o):Nat for o:RCounter, namely

- the application of the coercion, yielding o:Counter, followed by the application of the Counter destructor getcount, or
- the application of the Counter destructor getcount.

These two scenarios suggest the following coherence conditions for the coercion coerce:RCounter->Counter:

Nat getcount RCounter × RCounter count RCounter
id coerce coerce coerce

Nat getcount Counter × Counter count Counter

But these two coherence conditions provide a definition of coerce, namely
coerce : RCounter -> Counter
    count (coerce o) = count o
    getcount (coerce o) = coerce (getcount o)

This is a co-iterative definition of a function to Counter like the ones we have seen before. It uses count:Counter->Counter and getcount:Counter->Nat in the right-hand sides, and in the left-hand sides it uses getcount:RCounter->Nat and count:RCounter->RCounter.

In [BCGS89] it is observed that coercion functions needed to interpret subtyping in Ffun, a second-order lambda calculus with records, are already definable in the syntax. Here we see that this extends to coalgebraic datatypes with coiteration.

The fact that the coherence conditions completely determine the coercion is not really surprising: it can even be regarded as an essential requirement. If would be unsatisfactory if there were several coercions satisfying the coherence conditions and we chose a particular one. Indeed, we would no longer be justified in leaving such a coercion implicit. The whole justification for leaving coercions implicit is that "no information is lost".

The coalgebraic types Counter and RCounter denote two different final coalgebras and coerce:RCounter->Counter above denotes a mapping between these two final coalgebras. This mapping can of course also be defined in the semantics directly:

**Lemma 3.1** Let (νF,outF) be the final F-coalgebra and (νG,outG) the final G-coalgebra. Then given a natural transformation η : G → F there is a unique arrow coerce : νG → νF such that

```
   F(νF) ----> G(νG) ----> νG
             |                   |
             v                   v
   F(coerce) ----> νG
             |                   |
             v                   v
   F(νF) ----> νF
```

commutes.

**Proof** Follows directly by terminality of (νF,outF).

We now verify that instantiating this lemma does indeed produce the coercion denoted by coerce:Counter->Counter. Let CounterF and RCounterF be the functors

```
   CounterF(X) = Nat × X
   RCounterF(X) = Nat × X × X
```

Let (C,(getcount,count)) be the final CounterF-coalgebra, giving the interpretation of Counter and its constructors. Let (C,(getcount',count',reset')) be the final RCounterF-coalgebra, giving the interpretation of RCounter and its constructors. There is a natural transformation between these functors, namely

```
   π₁,π₂ : RCounterF → CounterF.
```
This natural transformation provides the coercion corresponding to
\[
\text{Record}\{\text{getcount} : \text{Nat}, \text{count} : X, \text{reset} : X\} \\
\leq \text{Record}\{\text{getcount} : \text{Nat}, \text{count} : X\}
\]
for any \(X\). By the lemma above there then is a unique \(\text{coerce} : R\text{Counter} \to C\text{ounter}\) such that

\[
\begin{array}{ccc}
\text{Nat} \times R\text{C} & \xrightarrow{\langle \pi_1, \pi_2 \rangle} & \text{Nat} \times R\text{C} \\
\downarrow \text{id} \times \text{coerce} & & \downarrow \text{coerce} \\
\text{Nat} \times C & \xrightarrow{[\text{getcount}, \text{count}]} & C
\end{array}
\]

commutes, i.e. such that the following two diagrams commute

\[
\begin{array}{ccc}
\text{Nat} & \xrightarrow{\text{getcount}'} & R\text{C} \\
\downarrow \text{id} & & \downarrow \text{coerce} \\
\text{Nat} & \xrightarrow{\text{getcount}} & C
\end{array}
\quad \begin{array}{ccc}
R\text{C} & \xrightarrow{\text{count}'} & R\text{C} \\
\downarrow \text{coerce} & & \downarrow \text{coerce} \\
R\text{C} & \xrightarrow{\text{count}} & C
\end{array}
\]

These are indeed the coherence conditions we came up with earlier.

### 3.2 Subtyping for Algebras

The subtype relation on coalgebraic types immediately suggests a subtype relation for their duals:

Consider the type \(\text{CList}\) of Cons-Snoc lists that not only provides an operation \(\text{cons}\) to add an element at the front of a list, but also provides an operation \(\text{snoc}\) to add an element at the end of a list:

\[
\text{data CList with} \\
\text{nil : 1 }\to\text{ CList} \\
\text{cons : A }\times\text{ CList }\to\text{ CList} \\
\text{snoc : CList }\times\text{ A }\to\text{ CList}
\]

Clearly \(\text{CList}\) can be regarded as a subtype of \(\text{List}\), i.e.

\[
\text{List} \leq \text{CList}.
\]

Intuitively, all the elements of \(\text{List}\) can be constructed using \(\text{nil}\) and \(\text{cons}\), which means that they are also elements of \(\text{CList}\). Indeed, in the category \text{Set} it is not hard to give interpretations of \(\text{List}\) and \(\text{CList}\) that are subsets.

Note the duality between algebraic and coalgebraic datatypes here: adding the destructor \(\text{reset}\) to \(\text{Counter}\) produced a subtype \(R\text{Counter}\), adding the constructor \(\text{snoc}\) to \(\text{List}\) produces a supertype \(\text{CList}\). By supertyping we here just mean the inverse of subtyping: \(B\) is a supertype of \(A\) – written \(B \geq A\)–iff \(A\) is a subtype of \(B\).
Just like \texttt{RCounter} \leq \texttt{Counter} can be regarded as a special case of subtyping between labelled products, \texttt{CList} \geq \texttt{List} can be regarded as a special case of subtyping between labelled sums:
\[
\text{CList} = \text{Variant}\{\text{nil} : \text{CList}, \text{cons} : A \times \text{CList}, \text{snoc} : \text{CList} \ast A\}
\geq \text{Variant}\{\text{nil} : \text{List},\text{cons} : A \times \text{List}\}
= \text{List}
\]
The coercion from \texttt{List} to \texttt{CList} that we want is of course
\[
\text{list coerce} : \text{List} \to \text{CList}
\]
\[
\text{list coerce nil} = \text{nil}
\]
\[
\text{list coerce (cons (a,1))} = \text{cons (a, (list coerce 1))}
\]

Dualising Lemma 3.1 yields

\textbf{Lemma 3.2} Let \((\mu F, \text{out}_{\mu F})\) be the initial \(F\)-algebra and \((\mu G, \text{out}_{\mu G})\) the initial \(G\)-algebra. Then given a natural transformation \(\eta : F \to G\) there is a unique arrow \textit{coerce} : \(\mu F \to \mu G\) such that

\[
\begin{array}{c}
F(\mu G) \xrightarrow{\eta_{\mu G}} G(\mu G) \xrightarrow{i_{\mu G}} \mu G
\end{array}
\]

\[
\begin{array}{c}
F(\text{coerce})
\end{array}
\]

\[
\begin{array}{c}
F(\mu F) \xrightarrow{i_{\mu F}} \mu F
\end{array}
\]

commutes.

Instantiating this lemma for the initial algebras denoted by \texttt{List} and \texttt{CList} does indeed provide the expected coercion. Recall that \((\text{List}, [\text{nil}, \text{cons}])\) was an initial \texttt{ListF}-algebra. Let \((\text{CList}, [\text{nil}', \text{cons}', \text{snoc}'])\) be an initial \texttt{CListF}-algebra, where \texttt{CListF}(X) = \(A + A \times X + X \times X\). There is a natural transformation between these two functors:

\[
[\text{in}_{1}, \text{in}_{2}] : \text{ListF} \to \text{CListF}.
\]

By the lemma above there is then a unique arrow \texttt{list coerce} : \texttt{List} \to \texttt{CList} such that

\[
\begin{array}{c}
1 + A \times \text{CList} \xrightarrow{[\text{in}_{1}, \text{in}_{2}]} \text{CListF}(\text{CList}) \xrightarrow{[\text{nd}', \text{cons}', \text{snoc}']} \text{CList}
\end{array}
\]

\[
\begin{array}{c}
id_{1} + id_{A} \times \text{list coerce}
\end{array}
\]

\[
\begin{array}{c}
1 + A \times \text{List} \xrightarrow{[\text{nil}, \text{cons}]} \text{List}
\end{array}
\]

commutes, i.e. such that the following two diagrams commute

\[
\begin{array}{c}
\text{id}_{1}
\end{array}
\]

\[
\begin{array}{c}
\text{list coerce}
\end{array}
\]

\[
\begin{array}{c}
id_{A} \times \text{list coerce}
\end{array}
\]

\[
\begin{array}{c}
\text{list coerce}
\end{array}
\]

\[
\begin{array}{c}
\text{nil}
\end{array}
\]

\[
\begin{array}{c}
\text{List}
\end{array}
\]

\[
\begin{array}{c}
A \times \text{List}
\end{array}
\]

\[
\begin{array}{c}
\text{cons}
\end{array}
\]

\[
\begin{array}{c}
\text{List}
\end{array}
\]

\[
\begin{array}{c}
\text{nil}'
\end{array}
\]

\[
\begin{array}{c}
\text{CList}
\end{array}
\]

\[
\begin{array}{c}
A \times \text{List}
\end{array}
\]

\[
\begin{array}{c}
\text{cons}'
\end{array}
\]

\[
\begin{array}{c}
\text{CList}
\end{array}
\]
which are the obvious coherence conditions for an implicit coercion from List to CSList.

As we said earlier, in Set we can chose List and CSList such that List ⊆ CSList and listcoerce:List→CSList is the associated injection. In this way the question of the coherence can be be avoided subtyping on algebraic types. However, the coherence problem for coalgebraic types can not be avoided in this same way, as the coercions between coalgebraic datatypes are not injective and cannot be given by inclusions between sets.

4 Inheritance

In object-oriented languages, inheritance allows class definitions to be re-used: new (sub)classes can be defined by modifying and/or extending existing class definitions. For example, an implementation of resetable counters could be defined by inheriting an implementation a class of counters.

We have seen how class definitions in the sense of [PT94] correspond to the coiterative functions in our setting. It turns out that there is an obvious way in which definitions of coiterative functions can be re-used:

Example 4.1

The obvious way to implement resetable counters is given by

\[
\begin{align*}
\text{newRCounter} & : \text{Nat} \rightarrow \text{RCounter} \\
\text{getcount (newRCounter n)} & = n \\
\text{count (newRCounter n)} & = \text{newRCounter (succ n)} \\
\text{reset (newRCounter n)} & = \text{newRCounter zero}
\end{align*}
\]

This implementation extends the implementation of counters given earlier by the function newCounter. The definition above just adds a single line to the definition of newCounter, namely the last one. We could introduce some syntax abbreviate the definition of newRCounter, for example as follows

\[
\begin{align*}
\text{newRCounter} & : \text{Nat} \rightarrow \text{RCounter} \\
\text{inherits newCounter : Nat} & \rightarrow \text{Counter} \\
\text{reset (newRCounter n)} & = \text{newCounter zero}
\end{align*}
\]

The definition of newRCounter above would be the same as the one obtained by copying the two defining clauses of newCounter and replacing all occurrences of newCounter by newRCounter.

Note that this is only a very limited form of inheritance. For instance, there is no way to define a new method in terms of old methods (e.g. define a method doublecount as count . count). Also, the same type — viz. Nat — is used by newCounter and newRCounter to represent the states of counters. There is no way to introduce extra instance variables, which in our setting would correspond to moving from a record type Record\{vari:A, vari2:B\} to the "wider" record type Record\{vari:A, vari2:B, vari3:C\} as representation type. (For the more complicated class definitions considered in [PT94], more powerful forms of inheritance are possible.)

The limited form of inheritance comes with a dual. Consider the definition of length function for CSList’s given below:
cslen**: CSList -> Nat
    cslen nil = zero
    cslen (cons (a,l)) = succ (length l)
    cslen (snoc (1,a)) = succ (length l)

It is clear that this definition extends the definition of length: List->Nat given earlier, i.e.

length : List -> Nat
    length nil = zero
    length (cons (a, l)) = succ (length l)

in exactly the same way as the definition of newRCounter extended the definition of newRCounter. And in the same way, we could introduce some syntax to abbreviate the definition of cslen, e.g.

    cslen : CSList -> Nat
    inherits length : List -> Nat
    cslen (snoc (1,a)) = succ (length l)

An obvious thing to want is then to be able to use the same name for length and cslen. The fact that the following diagram commutes

```
    List       Nat
      \arrow{length}
        \arrow{listCoerce}
            \arrow{cslen}
                \arrow{CSList}
```

suggest that it would be safe to do so.

Of course, in exactly the same way the diagram

```
    Counter    Nat
      \arrow{newCounter}
        \arrow{coerce}
            \arrow{newRCounter}
                \arrow{RCounter}
```

commutes, so we could use the same name for newCounter and newRCounter. (Even though there does not appear to be a good reason to do so, unlike for cslen, where using the same name length is clearly convenient.)

One could think of ways of making this inheritance mechanism more general. For instance, instead of just adding clauses to the definition, we could also allow *overriding*, for instance

    newRCounter2 : Nat -> RCounter
    inherits newCounter : Nat -> Counter
        count (newCounter2 n) = newRCounter2 zero
        reset (newCounter2 n) = newRCounter2 (succ n)

Then newRCounter2 produces counters with a count-method that reset them and a reset-method that counts.
5 Conclusion

We have described a notion of subtyping and a simple form of inheritance for Hagino's categorical datatypes, and indicated how subtyping can be interpreted as implicit coercions between (co)algebras. We have not given a formal definition of a type system providing algebraic and coalgebraic datatypes with subtyping and a complete proof of coherence of the categorical interpretation of such a language. This is left as future work.

We believe that a type theory with coalgebraic types and subtyping would be useful as a target calculus for encodings of objects. Indeed, in [HP95] the notion of coalgebra is already used to relate the encodings based on object as recursive records [Car88][CHC90][KR94] and the encodings based on existential types [PT94]; still, coalgebraic types are not used to express these encodings, because the target type theory does not provide them.

References


