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:

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:

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 *super*typing, which is related to subtyping in the obvious way: `A` is a subtype of `B` iff `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 [Pol97] 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 notion 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 $\mathbf{C}$ be a category with products and coproducts, and a terminal object $\mathbb{1}$.

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

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

$$\begin{tikzcd} FA \arrow[r, "f"] \arrow[d, "Fh"'] & A \arrow[d, "h"] \\ FB \arrow[r, "g"] & B \end{tikzcd}$$

- An F-algebra $(\mu F, in_{\mu F})$ is *initial* if for every F-algebra (B, g) there is a unique F-algebra homomorphism from $(\mu F, in_{\mu F})$ 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 $(\mu F, [in_1, \ldots, in_n])$ to another algebra $(B, [g_1, \ldots, g_n])$ is the unique $h : \mu F \to B$ such that

$$\begin{tikzcd} F_i\mu F \arrow[r, "in_i"] \arrow[d, dashed, "Fh"'] & \mu F \arrow[d, dashed, "h"] \\ F_i B \arrow[r, "g_i"] & B \end{tikzcd}$$

commutes for all i.

Example 2.2 (Natural Numbers)
Let $NatF$ be the functor $NatF(X) = \mathbb{1} + X$. Then the initial $NatF$-algebra $(Nat, [zero, succ])$ is a natural numbers object. The arrows $zero : \mathbb{1} \to Nat$ and $succ : Nat \to Nat$ are called the *constructors* of Nat. Initiality guarantees that for every $NatF$-algebra $(B, [b, g])$, i.e. for every $b : \mathbb{1} \to B$ and $g : B \to B$, there exist a unique $h : Nat \to B$ such that

$$\begin{aligned} h \circ zero &= b \\ h \circ succ &= g \circ h \end{aligned}$$

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

$$\begin{aligned} h \circ nil &= b \\ h \circ cons &= g \circ (id_A \times h) \end{aligned}$$

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

$$\begin{aligned} length \circ nil &= zero \\ length \circ cons &= succ \circ \pi_2 \circ (id_A \times length) \end{aligned}$$

□

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

Definition 2.4 ((Final) Coalgebra) Let F be a functor on $\mathbf{C}$. 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

$$\begin{tikzcd} FA & A \arrow[l, "f"'] \\ FB \arrow[u, "F_i h"] & B \arrow[l, "g"] \arrow[u, "h"'] \end{tikzcd}$$

- An F-coalgebra $(\nu F, out_{\nu F})$ is *final* or *terminal* if for every F-coalgebra (B, g) there is a unique morphism to $(\nu F, out_{\nu F})$ from (B, g).

Typically, we are interested in F-coalgebras where F is a functor of the form $F(X) = F_1(X) \times \ldots \times F_n(X)$. F-coalgebras are then of the form $(A, \langle f_1, \ldots, f_n \rangle)$ with each $f_i : A \to F_i(A)A$, and the unique coalgebra homomorphism from a coalgebra $(B, \langle g_1, \ldots, g_n \rangle)$ to the final coalgebra $(\nu F, \langle in_1, \ldots, in_n \rangle)$ is then the unique $h : \nu F \to B$ such that

$$\begin{tikzcd} F_i \nu F & \nu F \arrow[l, "out_i"'] \\ F_i B \arrow[u, dashed, "Fh"] & B \arrow[l, "g_i"] \arrow[u, dashed, "h"'] \end{tikzcd}$$

commutes for all i.

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

Example 2.5 (Streams)
Let $StreamF$ be the functor $StreamF(X) = Nat \times X$. A final $StreamF$-coalgebra $(Stream, \langle head, tail \rangle)$ is an object of infinite lists or streams of natural numbers. The arrows *head:Stream* $\to$*Nat* and *tail:Stream* $\to$*Stream* are called *destructors*.

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

$$\begin{aligned} head \circ t &= g_{head} \\ tail \circ h &= h \circ g_{tail} \end{aligned}$$

This scheme is known as *co-iteration*. For any $b : \mathbb{1} \to B$ we can think of $h \circ b : \mathbb{1} \to Stream$ as the infinite list of natural numbers with b as its "seed" and with g_{head} and g_{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 $from : Nat \to List$ defined by

$$\begin{aligned} head \circ from &= id_{Nat} \\ tail \circ from &= from \circ succ \end{aligned}$$

For any $n : \mathbb{1} \to Nat$, the arrow $from \circ n : \mathbb{1} \to Stream$ then represents the infinite list $n, succ \circ n, succ^2 \circ n, \ldots$. □

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.

```
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.

```
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.

```
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.

```
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

$$\mathtt{Stream} = \mathtt{Record}\{\mathtt{head} : \mathtt{Nat}, \mathtt{tail} : \mathtt{Stream}\}.$$

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

$$\mathtt{List} = \mathtt{Variant}\{\mathtt{nil} : \mathtt{List}, \mathtt{cons} : \mathtt{A} \times \mathtt{List}\}.$$

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

```
 h : B -> Stream
   getcount (h b) = getcountimp b
      count (h b) = h (countimp b)
```

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:

```
 newCounter : Nat -> Stream
   getcount (newCounter n) = n
      count (newCounter n) = newCounter (succ n)
```

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 `Nat` is replaced with a one-field record type `Record{x:Nat}`, i.e.

```
 newCounter' :  Record{x:Nat} -> Stream
   getcount (newCounter n) = n.x
      count (newCounter n) = newCounter {x=succ n.x}
```

then the field `x` can be regarded as an instance variable. □

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 `Record{x:Nat,y:Nat}` and `Record{x:Nat}`, since any record with an `x`- and a `y`-field of type `Nat` is also a record an `x`-field of type `Nat`. This is usually written as

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

By the so-called *subsumption rule* any term of type `Record{x:Nat,y:Nat}` then also has type `Record{x: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. $\mathtt{A} \leq \mathtt{B}$ means that there is a coercion function from `A` to `B` which is left implicit. For example, the coercion from `Record{x:Nat,y:Nat}` to `Record{x:Nat}` should of course be the function that maps a record `{x=N,y=M}` to the record `{x=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 `coerce:Record{x:Nat,y:Nat}->Record{x:Nat}` should be such that $\mathtt{r.x} = (\mathtt{coerce}\ \mathtt{r}).\mathtt{x}$ (i). If this does not hold, e.g. if `coerce` maps the record `{x=N,y=M}` to the record `{x=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.

$$\begin{tikzcd}
\mathtt{Record}\{\mathtt{x} : \mathtt{Nat}, \mathtt{y} : \mathtt{Nat}\} \arrow[r, ".\mathtt{x}"] \arrow[d, "\mathtt{coerce}"'] & \mathtt{Nat} \arrow[d, "\mathtt{id}"] \\
\mathtt{Record}\{\mathtt{x} : \mathtt{Nat}\} \arrow[r, ".\mathtt{x}"] & \mathtt{Nat}
\end{tikzcd}$$

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`:

$$\texttt{RCounter} \leq \texttt{Counter}.$$

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

$$\begin{array}{lcl} \texttt{RCounter} & = & \texttt{Record}\{\texttt{count} : \texttt{RCounter}, \texttt{getcount} : \texttt{Nat}, \texttt{reset} : \texttt{RCounter}\} \\ & \leq & \texttt{Record}\{\texttt{count} : \texttt{Counter}, \texttt{getcount} : \texttt{Nat}\} \\ & = & \texttt{Counter} \end{array}$$

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`:

$$\begin{tikzcd} \texttt{Nat} \arrow[d, "\texttt{id}"'] & \texttt{RCounter} \arrow[l, "\texttt{getcount}"'] \arrow[d, "\texttt{coerce}"'] & \texttt{RCounter} \arrow[d, "\texttt{coerce}"'] & \texttt{RCounter} \arrow[l, "\texttt{count}"'] \arrow[d, "\texttt{coerce}"'] \\ \texttt{Nat} & \texttt{Counter} \arrow[l, "\texttt{getcount}"'] & \texttt{Counter} & \texttt{Counter} \arrow[l, "\texttt{count}"'] \end{tikzcd}$$

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 an 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 *Fun*, 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 $(\nu F, out_{\nu F})$ is the final F-coalgebra and $(\nu G, out_{\nu G})$ the final G-coalgebra. Then given a natural transformation $\eta : G \stackrel{\bullet}{\to} F$ there is a unique arrow $coerce : \nu G \to \nu F$ such that

$$
\begin{tikzcd}
F(\nu G) \arrow[d, "F(coerce)"'] & G(\nu G) \arrow[l, "\eta_{\nu G}"'] & \nu G \arrow[l, "out_{\nu G}"'] \arrow[d, "coerce"] \\
F(\nu F) & & \nu F \arrow[ll, "out_{\nu F}"']
\end{tikzcd}
$$

commutes.

PROOF Follows directly by terminality of $(\nu F, out_{\nu F})$.

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

$$
\begin{aligned}
CounterF(X) &= Nat \times X \\
RCounterF(X) &= Nat \times X \times X
\end{aligned}
$$

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

$$\langle \pi_1, \pi_2 \rangle : RCounterF \stackrel{\bullet}{\to} CounterF.$$

This natural transformation provides the coercion corresponding to

$$\begin{aligned}&\mathtt{Record}\{\mathtt{getcount} : \mathtt{Nat}, \mathtt{count} : \mathtt{X}, \mathtt{reset} : \mathtt{X}\}\\ &\leq \mathtt{Record}\{\mathtt{getcount} : \mathtt{Nat}, \mathtt{count} : \mathtt{X}\}\end{aligned}$$

for any X. By the lemma above there then is a unique $coerce : RCounter \to Counter$ such that

$$\begin{tikzcd}
Nat \times RC \arrow[d, "id \times coerce"] & Nat \times RC \times RC \arrow[l, "\langle \pi_1, \pi_2 \rangle"'] & RC \arrow[l, "{[getcount', count', reset']}"'] \arrow[d, "coerce"'] \\
Nat \times C & & C \arrow[ll, "{[getcount, count]}"']
\end{tikzcd}$$

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

$$\begin{tikzcd}
Nat \arrow[d, "id"'] & RC \arrow[l, "getcount'"'] \arrow[d, "coerce"] \\
Nat & C \arrow[l, "getcount"']
\end{tikzcd}
\qquad
\begin{tikzcd}
RC \arrow[d, "coerce"'] & RC \arrow[l, "count'"'] \arrow[d, "coerce"] \\
C & C \arrow[l, "count"']
\end{tikzcd}$$

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 `CSList` of Cons-Snoc lists that not only provides an operation `cons` to add an element at the front of a list, but also provides an operation `snoc` to add an element at the end of a list:

```
data CSList with
     nil : 1 -> CSList
    cons : A × CSList -> CSList
    snoc : CSList × A ->  CSList
```

Clearly `CSList` can be regarded as a subtype of `List`, i.e.

$$\mathtt{List} \leq \mathtt{CSList}.$$

Intuitively, all the elements of `List` can be constructed using `nil` and `cons`, which means that they are also elements of `CSList`. Indeed, in the category **Set** it is not hard to give interpretations of `List` and `CSList` that are subsets.

Note the duality between algebraic and coalgebraic datatypes here: adding the destructor `reset` to `Counter` produced a *sub*type `RCounter`, adding the constructor `snoc` to `List` produces a *super*type `CSList`. 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 `RCounter` $\leq$`Counter` can be regarded as a special case of subtyping between labelled products, `CSList` $\geq$`List` can be regarded as a special case of subtyping between labelled sums:

$$\begin{array}{lcl} \texttt{CSList} & = & \texttt{Variant}\{\texttt{nil}:\texttt{CSList}, \texttt{cons}:\texttt{A}\times\texttt{CSList}, \texttt{snoc}:\texttt{CSList}*\texttt{A}\} \\ & \geq & \texttt{Variant}\{\texttt{nil}:\texttt{List}, \texttt{cons}:\texttt{A}\times\texttt{List}\} \\ & = & \texttt{List} \end{array}$$

The coercion from `List` to `CSList` that we want is of course

```
listcoerce : List -> CSList
  listcoerce nil          = nil
  listcoerce (cons (a,l)) = cons (a,(listcoerce l))
```

Dualising Lemma 3.1 yields

LEMMA 3.2 Let $(\mu F, out_{\mu F})$ be the initial F-algebra and $(\mu G, out_{\mu G})$ the initial G-algebra. Then given a natural transformation $\eta : F \overset{\bullet}{\to} G$ there is a unique arrow $coerce : \mu F \to \mu G$ such that

$$\begin{tikzcd} F(\mu G) \arrow[r, "\eta_{\mu G}"] & G(\mu G) \arrow[r, "in_{\mu G}"] & \mu G \\ F(\mu F) \arrow[u, "F(coerce)"] \arrow[rr, "in_{\mu F}"] & & \mu F \arrow[u, "coerce"'] \end{tikzcd}$$

commutes. □

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

$$[in_1, in_2] : ListF \overset{\bullet}{\to} CSListF.$$

By the lemma above there is then a unique arrow $listcoerce : List \to CSList$ such that

$$\begin{tikzcd} 1 + A\times CSList \arrow[r, "{[in_1, in_2]}"] & CSListF(CSList) \arrow[r, "{[nil', cons', snoc']}"] & CSList \\ 1 + A \times List \arrow[u, "id_1 + id_A \times listcoerce"] \arrow[rr, "{[nil, cons]}"] & & List \arrow[u, "listcoerce"'] \end{tikzcd}$$

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

$$\begin{tikzcd} \mathbb{1} \arrow[r, "nil"] \arrow[d, "id_1"'] & List \arrow[d, "listcoerce"] \\ \mathbb{1} \arrow[r, "nil'"] & CSList \end{tikzcd} \qquad \begin{tikzcd} A \times List \arrow[r, "cons"] \arrow[d, "id_A \times listcoerce"'] & List \arrow[d, "listcoerce"'] \\ A \times List \arrow[r, "cons'"] & CSList \end{tikzcd}$$

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 $List \subseteq CSList$ and $listcoerce{:}List{\rightarrow}CSList$ 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

```
newRCounter : Nat → RCounter
   getcount (newRCounter n) = n
      count (newRCounter n) = newRCounter (succ n)
      reset (newRCounter n) = newRCounter zero
```

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

```
newRCounter : Nat → RCounter
  inherits newCounter : Nat → Counter
      reset (newRCounter n) = newCounter zero
```

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{var1:A,var2:B}` to the "wider" record type `Record{var1:A,var2:B,var3: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:

```
cslength : CSList -> Nat
  cslength nil          = zero
  cslength (cons (a,l)) = succ (length l)
  cslength (snoc (l,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 `cslength`, e.g.

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

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

$$\begin{tikzcd}
\texttt{List} \arrow[r, "\texttt{length}"] \arrow[d, "\texttt{listcoerce}"'] & \texttt{Nat} \\
\texttt{CSList} \arrow[ur, "\texttt{cslength}"'] &
\end{tikzcd}$$

suggest that it would be safe to do so.

Of course, in exactly the same way the diagram

$$\begin{tikzcd}
\texttt{Counter} & \texttt{Nat} \arrow[l, "\texttt{newCounter}"'] \arrow[dl, "\texttt{newRCounter}"] \\
\texttt{RCounter} \arrow[u, "\texttt{coerce}"] &
\end{tikzcd}$$

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 `cslength`, 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

[BCGS89] V. Breazu-Tannen, Th. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. In *Logic in Computer Science*, pages 112–129. IEEE, 1989.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Principles of Programming Languages*, pages 125–135. ACM, 1990.

[CS92] Robin Cockett and Dwight Spencer. Strong categorical datatypes I. In R. A. G. Seely, editor, *International Meeting on Category Theory 1991*, Canadian Mathematical Society Proceedings. AMS, 1992.

[Hag87a] Tatsuya Hagino. *A categorical programming language*. PhD thesis, University of Edinburgh, 1987.

[Hag87b] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In D.H. Pitt, A Poigné, and D.E. Rydeheard, editors, *Category and Computer Science*, pages 140–157. Springer, September 1987.

[HP95] Martin Hofmann and Benjamin C. Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, 1995.

[JR97] Bart Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. In *EATCS Bulletin*. june 1997.

[KR94] Samuel N. Kamin and Uday S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. The MIT Press, 1994.

[Pol97] Erik Poll. Subtyping and Inheritance for Inductive Types. In *Informal proceedings of the 1994 TYPES Workshop, Durham, UK*, August 1997.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[Rei95] Horst Reichel. An approach to object semantics based on terminal coalgebras. *Mathematical Structures in Computer Science*, 5:129–152, 1995.