A Coalgebraic Semantics of Subtyping

Erik Poll

Department of Computer Science, University of Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
erikpoll@cs.kun.nl

Abstract

Subtyping is a central notion in object-oriented programming. In this paper we investigate how the coalgebraic semantics of objects accounts for subtyping. We show that different characterisations of so-called *behavioural subtyping* found in the literature can conveniently be expressed in coalgebraic terms. We define subtyping between coalgebras and subtyping between coalgebraic specifications, and show that the latter is sound and complete w.r.t. the former. We also illustrate the subtle difference between the notions of subtyping and refinement.

1 Introduction

Subtyping is one of the famous buzzwords in object-oriented (OO) programming. However, the precise meaning of subtyping, and more in particular the question whether subtyping is the same as inheritance, has been the subject of a lot of debate (more on that in Section 2). Given that the notion of (terminal) coalgebra can be used to describe the semantics of objects [Rei95], an obvious question to ask is how this semantics accounts for subtyping. We will show that the coalgebraic view of objects provides a clean semantics for so-called *behavioural subtyping*.

Refinement is an important notion in specification languages. At first sight it seems to be closely related, if not identical, to the notion of subtyping. However, we will show that refinement and subtyping are really different notions.

This paper is organised as follows. Section 2 gives an informal explanation of the notion of behavioural subtyping and discusses the difference with the notion of inheritance. Section 3 defines some basic coalgebraic notions and Section 4 explains the format of class specifications we use. Sections 5 and 6 then consider the coalgebraic semantics of signature subtyping and behavioural subtyping, respectively. Section 7 discusses the relation between subtyping and refinement, and we conclude in Section 8.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

2 Behavioural subtyping

Behavioural subtyping captures the idea that objects in one class (the subclass) behave like objects in another class (the superclass). For example, classes `Car` and `Truck` could be behavioural subtypes of a class `Vehicle`. Behavioural subtyping is sometimes referred to as the "is a" relation: a car "is a" vehicle.

Behavioural subtyping guarantees that any code written for objects in the superclass, i.e. vehicles, will behave as expected when applied to objects in the subclasses, i.e. cars or trucks. So behavioural subtyping allows the reuse of so-called client code: a piece of code written for vehicles will also work for cars and trucks. This is the justification of the implicit *casting* of objects from sub- to superclasses, also known as *subsumption*, by which for example any object of type `Car` is also of type `Vehicle`.

There is a weaker relation than behavioural subtyping, known as *signature subtyping*, which only concerns the signatures of objects. A class A' is a signature subtype of another class A if objects in A' have all the methods that objects in A have, with compatible types[1]. For a signature subtype to also be a behavioural subtype, these methods must also have the same semantics. Unlike behavioural subtyping, signature subtyping is a purely syntactic notion, defined by standard contra/covariant rules (e.g. [CW85]). Most of the type-theoretical literature on OO is concerned with this notion of subtyping. Signature subtyping can be mechanically checked by type checking algorithms, and can be used to ensure that no type errors (of the form "method not found") can occur at run-time.

Different ways of defining behavioural subtyping have been proposed in the literature. One way is to say that behavioural subtypes correspond to *stronger specifications*. Usually, this is expressed in terms of pre- and post-conditions of methods: methods in a behavioural subtype are then required to have weaker pre-conditions and stronger post-conditions than the corresponding methods in the supertype. This characterisation of behavioural subtyping is used in the programming language Eiffel and the "Design by Contract" approach [Mey97], and is widely used in the literature, e.g. [Ame87] [LW94] [LW95].

Another well-known characterisation of behavioural subtyping is the principle of *substitutability* [Lis88]: "A' is a behavioural subtype of A iff for every object a' of type A' there is an object a of type A such that for all programs p that use a, the behaviour of p is unchanged when a is replaced with a'".

In Section 6 we will give definitions of behavioural subtyping in the coalgebraic setting in both of the ways discussed above, and relate the two.

[1] A word about notation: throughout this paper we stick to the convention that a primed letter such as A' refers to a subtype of the unprimed one.

Behavioural Subtyping vs Inheritance
In the world of OO there has been a lot of discussion on the precise meaning of inheritance and subtyping, and the difference, if any, between them. It is now generally recognised that one can distinguish (at least) two different notions. Beware that a lot of the literature on OO treats the terms inheritance and subtyping as synonyms! This is why, to avoid any confusion, we use the term "behavioural subtyping" instead of just "subtyping" [2].

Inheritance allows us to make a (sub)class A' of a (super)class A by adding new methods and new fields to the class, and by overriding existing methods. In many cases this will lead to behavioural subtyping, i.e. A' will be a behavioural subtype of A. However, this is not always the case. It should be clear that if methods are *overridden* in a subclass, then objects in this subclass may behave quite differently from objects in the superclass [3].

So the only relation between inheritance and behavioural subtyping is that inheritance *may* result in behavioural subtyping. Although ideally one uses inheritance to produce behavioural subtypes, there may be good reasons to use inheritance even if it does not produce behavioural subtypes. Like behavioural subtyping, inheritance makes it possible to reuse code, namely the code of class definitions. The code reuse by inheritance, i.e. the reuse of class definitions, may well be a more important source of code reuse than the code reuse made possible by behavioural subtyping, i.e. the reuse of client code. The programming language C++ offers a distinction between private and public inheritance for this purpose: public inheritance should be used when inheritance produces a behavioural subtype, otherwise private inheritance should be used. Objects of a subclass can then only be cast to the superclass when public inheritance has been used.

Note that there can be behavioural subtyping between two classes even though there is no inheritance between them. This is because behavioural subtyping, unlike inheritance, concerns the observable behaviour of objects, and not their implementation. Classes with completely different implementations, which are not in the inheritance relation, may well provide objects with identical behaviour, and can thus be behavioural subtypes.

3 Coalgebraic Preliminaries

We will only need the very basics of the theory of coalgebras (see for instance [Rut96] or [JR97]).

We work in the category Set. Polynomial functors are of the form

$$F(X) ::= X \mid C \mid F_1(X) + F_2(X) \mid F_1(X) \times F_2(X) \mid C \to F(X).$$

[2] Sometimes subtyping is called "interface inheritance" and inheritance "implementation inheritance".

[3] In the presence of so-called binary methods just adding methods may also break behavioural subtyping, even when no methods are overridden (see [CHC90] or [BCC+96]).

We write π_1 and π_2 for the projections from the Cartesian product, and inl and inr for the injections into the disjoint sum.

An *F-coalgebra* is a pair (S, m) with S a set – called the state space – and $m : S \to F(X)$. An *F-coalgebra homomorphism* $f : (S, m) \to (S', m')$ is a function $f : S \to S'$ such that $m \circ f = F(f) \circ m$.

For every polynomial functor there exists a *final coalgebra*, which is unique up to isomorphism. We fix particular final coalgebras, denoted $(\nu F, \alpha_F)$. The unique homomorphism from a coalgebra (S, m) to the final coalgebra is denoted by $behaviour_m$. The final coalgebra can be viewed as the collection of all the possible behaviours of F-coalgebras; the function $behaviour_m$ then maps every state $s \in S$ to its observable behaviour $behaviour_m(s) \in \nu F$.

An *invariant* on a coalgebra (S, m) is a subset $S' \subseteq S$ such that (S', m) is also a coalgebra; (S', m) is then called a subcoalgebra of (S, m).

To define the notion of bisimulation, we first define relation lifting. For a relation $\sim \subseteq X \times Y$ the relation $F^{rel}(\sim) \subseteq F(X) \times F(Y)$ is defined by induction on the structure of F, as follows:

- if $F(X) = X$ then $F^{rel}(\sim) = \sim$,
- if $F(X) = C$ then $F^{rel}(\sim) = eq_C$, the equality relation on C,
- if $F(X) = F_1(X) + F_2(X)$ then $F^{rel}(\sim) =$
  $\{(\mathsf{inl}(x), \mathsf{inl}(y)) \mid (x, y) \in F_1^{rel}(\sim)\} \cup \{(\mathsf{inr}(x), \mathsf{inr}(y)) \mid (x, y) \in F_2^{rel}(\sim)\}$,
- if $F(X) = F_1(X) \times F_2(X)$ then
  $F^{rel}(\sim) = \{(x, y) \mid (\pi_1(x), \pi_1(y)) \in F_1^{rel}(\sim) \wedge (\pi_2(x), \pi_2(y)) \in F_2^{rel}(\sim)\}$,
- if $F(X) = C \to F_1(X)$ then
  $F^{rel}(\sim) = \{(f, g) \mid \forall x \in C.(f(x), g(x)) \in F_1^{rel}(\sim)\}$.

A *bisimulation* $\sim$ between F-coalgebras (S, m) and (S', m') is a relation $\sim \subseteq S \times S'$ such that

$$\forall x : S, x' : S'.\ x \sim x' \Rightarrow (m(x), m'(x')) \in F^{rel}(\sim).$$

We write $\overset{\leftrightarrow}{=}$ for *bisimilarity*, the largest bisimulation (i.e. the union of a bisimulations) between two coalgebras. Bisimilar elements have the same behaviour, and the unique homomorphisms to the final F-coalgebra identify precisely these elements:

Lemma 3.1 *Let (S, m) and (S', m') be F-coalgebras. Then $behaviour_m(s) = behaviour_{m'}(s')$ iff there is some bisimulation $\sim$ between (S, m) and (S', m') such that $s \sim s'$, i.e. iff $s \overset{\leftrightarrow}{=} s'$.* □

4 Class Specifications

Our format of class specification is based on that used in the experimental specification language CCSL [Jac97] [HHJT98]. An example of such a class specification is given in Figure 1. This example specifies a class with two methods, `getcount` and `count`. All classes have a single constructor called

```
CLASS Counter
  METHODS getcount : X -> Int
              count : X -> X

  ASSERTIONS
   ∀ x:X.  getcount(count(x)) = getcount(x)+1

  CREATION CONDITIONS
     getcount(new Counter) = 0
END
```

Fig. 1. The class specification `Counter`

`new`. Methods always act on an object. This argument of a method, often referred to as "this" or "self", and usually left implicit, is made explicit here: all methods get an argument of type `X`. This type `X` stands for the state space of objects.

In general, a class specification consists of:

- a signature of *methods* $m_i : X \to M_i(X)$ with the M_i polynomial functors,
- a collection of *assertions*, properties of the methods,
- a collection of *creation conditions*, properties of the constructor `new`.

Of course, the M_i can be combined into a single functor $M(X) = M_1(X) \times \ldots \times M_n(X)$. An implementation – or model – of a class specification then consists of an M-coalgebra (S, m) giving a representation of the state space and an implementation of the methods, plus an initial state $c : S$ giving the implementation of the constructor `new`. They should satisfy the assertions and creation conditions.

We want to impose some restrictions on the assertions and creation conditions that are allowed. First, the assertions should be universal quantifications giving properties that all objects have, and the creation conditions should only specify properties of the initial object `new`. Second, we do not want assertions or creation conditions to distinguish observationally equal implementations. Here implementations are observationally equal if there exist a total bisimulation between the respective coalgebras that relates the initial states. In particular, this requirement means that a class specifications may *not* refer to the notion of equality on the state space, but should use the notion of bisimilarity $\overset{\leftrightarrow}{=}$ instead. For example, we do not want to allow $\forall$`x:X.count(x)`$\neq$`x` as an assertion; this should be written as $\forall$`x:X.count(x)`$\overset{\leftrightarrow}{\neq}$`x` instead.

A way to impose these restrictions would be to give a precise syntax for assertions and creation conditions. Such a syntax could rule out the use of `=` as relation on the state space, and only allow $\overset{\leftrightarrow}{=}$ to be used instead. Alternatively, we could allow the use of `=` but define its interpretation to be bisimilarity. For reasons of space we will not go to all this trouble here; instead we simply

require that assertions and creation conditions are predicates on the final coalgebra, i.e. predicates on object behaviours:

Definition 4.1 A *class specification* $\mathcal{A}$ is a triple (M, Φ, Ψ) with M a polynomial functor and Φ and Ψ predicates on νM, the state space of the final coalgebra. □

The predicates Φ and Ψ are the conjunction of the assertions and the conjunction of the creation conditions, respectively. We now use the mappings $behaviour_m$ from M-coalgebras (S, m) to the final M-coalgebra to define the notion of model:

Definition 4.2 A *model* of a class specification $\mathcal{A} = (M, \Phi, \Psi)$ is a triple (S, m, c) with

- (S, m) a coalgebra, with S giving the state space X and m giving the interpretation of the methods, and
- $c : S$, giving the interpretation of the constructor,

such that

- $\forall s : S.\ \Phi(behaviour_m(s))$, i.e. $behaviour_m(S) \subseteq \Phi$,
- $\Psi(behaviour_m(c))$.

In other words, all objects satisfy the assertions and the initial object satisfies the creation conditions. The fact that (S, m, c) is a model of $\mathcal{A}$ is denoted by $(S, m, c) \models \mathcal{A}$, and we write $(S, m) \models \Phi$ for $behaviour_m(S) \subseteq \Phi$. □

A minor difference with the notion of model in [Jac97] is that we explicitly include the initial state c as part of the model. Note that there may be elements in S which are not "reachable" from the constructor c using the methods m.

The restrictions on the shape of assertions and creation conditions give some nice properties:

Lemma 4.3 *Let* $(S, m, c) \models \mathcal{A}$.

(i) *If* (S', m) *is a subcoalgebra of* (S, m) *then* $(S', m, c) \models \mathcal{A}$.

(ii) *If* (T, n) *is a coalgebra,* $d : T$*, and there exists a total bisimulation* $\sim$ *between* (S, m) *and* (T, n) *such that* $c \sim d$*, then* $(T, n, d) \models \mathcal{A}$.

Proof. Follows immediately from the definition of $\models$, using Lemma 3.1. □

So any submodel of a model is also a model, and specifications do not distinguish between observationally equal models. The first property would not hold if there could, for instance, be existential quantifications over the state space in class specifications. The second would not hold if specifications could state (in)equalities on the state space.

A construction needed later is that, using a coproduct, we can take the "union" of models:

Lemma 4.4 *Let* (S, m) *and* (T, n) *be* M*-coalgebras, and* $\mathcal{A} = (M, \Phi, \Psi)$ *some class specification. Then*

(i) *if* $(S, m) \models \Phi$ *and* $(T, n) \models \Phi$ *then* $(S + T, p) \models \Phi$,

(ii) *if* $(S, m) \models \Phi$ *and* $(T, n, d) \models \mathcal{A}$ *then* $(S + T, p, \mathsf{inr}(d)) \models \mathcal{A}$,

where $p = [M(\mathsf{inl}) \circ m, M(\mathsf{inr}) \circ n] : S + T \to M(S + T)$.

Proof. Follows easily from the fact that $behaviour_{[M(\mathsf{inl})\circ m, M(\mathsf{inr})\circ n]}(S + T) = behaviour_m(S) \cup behaviour_n(T)$. □

5 Signature Subtyping

A necessary condition for behavioural subtyping between classes is signature subtyping: objects in a subclass should at least have all the methods that objects in the superclass have.

Definition 5.1 Let $\mathcal{A}'$ and $\mathcal{A}$ be class specifications. $\mathcal{A}'$ is a *signature subtype* of $\mathcal{A}$ iff $\mathcal{A}'$ has at least all the methods that $\mathcal{A}$ has, with the same types □

For example, the class specification `RCounter` of "resetable' counters below is a signature subtype of the specification `Counter` given earlier:

```
CLASS RCounter
  METHODS getcount : X -> Int
          count : X -> X
          reset : X -> X

  ASSERTIONS
          ⋮
  CREATION CONDITIONS
          ⋮
END
```

We have omitted assertions and creation conditions, because these do not play a role in signature subtyping.

If $\mathcal{A}'$ is a signature subtype of $\mathcal{A}$, then giving an $\mathcal{A}'$-object in a context where an $\mathcal{A}$-object is expected will not cause any "method not found" errors. The definition of signature subtyping above is stronger than strictly necessary: we could weaken it by only requiring that the type of a method in $\mathcal{A}'$ is a *subtype* of the type that this method has in $\mathcal{A}$. In particular, by the standard contra/covariant subtyping rule for function types (e.g. see [CW85]), the input types of a method in the subclass could be supertypes of the input types this method has in the superclass. For simplicity we use the stronger definition above. Most existing object-oriented languages use such a simple definition.

Semantics of Signature Subtyping
Semantically, signature subtyping between specifications results in a natural transformation between their method signatures. Let $M'(X) = \prod_{i \in I'} M_i(X)$ and $M(X) = \prod_{i \in I} M_i(X)$ be the method signatures of specifications $\mathcal{A}'$ and $\mathcal{A}$, with $\mathcal{A}'$ a signature subtype of $\mathcal{A}$. Then $I' \supseteq I$, and there is an obvious natural transformation

$$\eta = \langle \pi_i \mid i \in I \rangle : M' \to M,$$

i.e. the mapping that drops all components in $M'(X)$ that are not in $M(X)$. This natural transformation provides a way of turning any M'-coalgebra into an M-coalgebra:

Theorem 5.2 ([Rut96], Thm. 14.1)
A natural transformation $\eta : M' \to M$ induces a functor from the category of M'-coalgebras to the category of M-coalgebras, which maps an M'-coalgebra (S', m') to the M-coalgebra $(S', \eta_{S'} \circ m')$, and an M'-coalgebra homomorphism f simply to the M-coalgebra homomorphism f. This functor preserves bisimulations. □

There are two interesting points to note about the construction above.

First, because the functor induced by η preserves bisimulations, elements bisimilar in (S', m') are also bisimilar in $(S', \eta_{S'} \circ m')$. This is of course what you would expect: on the signature subclass we may have *more* methods and hence a *stronger* notion of "observational equivalence". It is a useful property when comparing class specifications that are signature subtypes and that therefore use different notions of bisimilarity.

Second, the construction in the theorem above provides a semantics of the implicit cast from sub- to superclass, as in [Pol97]:

Definition 5.3 For $\eta : M' \to M$,

$$cast_\eta =_{def} behaviour_{\eta \circ \alpha_{M'}} : (\nu M', \alpha_{M'}) \to (\nu M, \alpha_M).$$

A basic property of $cast_\eta$ needed later is:

Lemma 5.4 *For any M'-coalgebra (S', m') and $\eta : M' \to M$*

$$cast_\eta \circ behaviour_{m'} = behaviour_{\eta \circ m'} .$$

Proof. Trivial. □

The function $cast_\eta$ is the unique function such that the diagram

$$\begin{array}{ccc} \nu M' & \xrightarrow{\eta_{\nu M'} \circ \alpha_{M'}} & M(\nu M') \\ {\scriptstyle cast}\downarrow & & \downarrow{\scriptstyle M(cast)} \\ \nu M & \xrightarrow{\alpha_M} & M(\nu M) \end{array}$$

commutes. This diagram expresses precisely the condition that subsumption – the implicit cast from sub- to superclass – does not introduce any ambiguities. For example, let $(\nu M, \langle getcount, count \rangle)$ and $(\nu M', \langle getcount', count', reset' \rangle)$

be the final M- and M'-coalgebras, with $M(X) = Nat \times X$ and $M'(X) = Nat \times X \times X$. Then for the $cast_{\langle \pi_1, \pi_2 \rangle} : \nu M' \to \nu M$ we have

$$\begin{array}{ccc} \nu M' & \xrightarrow{\langle getcount', count', reset' \rangle} & Nat \times \nu M' \times \nu M' \\ {\scriptstyle cast}\downarrow & & \downarrow{\scriptstyle \langle \pi_1, cast \circ \pi_2 \rangle} \\ \nu M & \xrightarrow{\langle getcount, count \rangle} & Nat \times \nu M \end{array}$$

This shows for instance that invoking the method *count'* on a resetable counter and then casting to the superclass gives the same result as first casting to the superclass and then invoking the method *count*. In other words, leaving *cast* implicit and not distinguishing between the subclass methods and the superclass methods, e.g. between *count* and *count'*, in the syntax – as is done in all OO languages – does not cause any ambiguities. The diagrams above express exactly the so-called *coherence conditions* for subsumption discussed in [Mit90] [HP95] [Pol97].

6 Behavioural subtyping

We define two notions of behavioural subtyping. First, in Subsection 6.1, we define behavioural subtyping between coalgebras, and then, in Subsection 6.2, we define behavioural subtyping between coalgebraic class specifications. In Subsection 6.3 we prove that the latter notion of subtyping is sound and complete with respect to the former.

6.1 Behavioural subtyping between coalgebras

We already mentioned Liskov's substitution principle [Lis88]: "A' is a behavioural subtype of A iff for every object a' of type A' there is an object a of type A such that for all programs p that use a, the behaviour of p is unchanged when a is replaced with a'". This principle immediately translates to a definition of behavioural subtyping between coalgebras, using the notion of bisimulation to express that objects have the same behaviour:

Definition 6.1 Let (S, m) be an M-coalgebra, (S', m') an M'-coalgebra, and $\eta : M' \to M$.

(S', m') is a *behavioural subtype* of (S, m), written $(S', m') \leq_\eta (S, m)$, iff there exists an M-bisimulation $\sim \subseteq S' \times S$ between $(S', \eta \circ m')$ and (S, m) such that $\forall s' \in S'.\ \exists s \in S.\ s' \sim s$. □

Definitions of behavioural subtyping that use the notion of (bi)simulation can already be found in the literature, e.g. [LW95] [Mau95] [Pol98].

Basic properties of $\leq_\eta$ are "reflexivity" and "transitivity":

Lemma 6.2 (i) $(S, m) \leq_{id} (S, m)$.

(ii) *If* $(S_1, m_1) \leq_{\eta_1} (S_2, m_2)$ *and* $(S_2, m_2) \leq_{\eta_2} (S_3, m_3)$ *then* $(S_1, m_1) \leq_{\eta_2 \circ \eta_1} (S_3, m_3)$.

Proof. Trivial. □

An alternative definition of $\leq_\eta$ is given by the lemma below.

Lemma 6.3 *Let (S', m') be an M'-coalgebra, (S, m) an M-coalgebra, and $\eta : M' \to M$. Then*

$$(S', m') \leq_\eta (S, m) \iff \mathit{behaviour}_{\eta \circ m'}(S') \subseteq \mathit{behaviour}_m(S).$$

Proof. To prove $\Rightarrow$, assume $(S', m') \leq_\eta (S, m)$ and $x \in \mathit{behaviour}_{\eta \circ m'}(S')$, i.e. $x = \mathit{behaviour}_{\eta \circ m'}(s')$ for some $s' \in S'$. By the definition of $\leq_\eta$ there exists a bisimulation $\sim$ between $(S', \eta \circ m')$ and (S, m) such that there is an $s \in S$ with $s' \sim s$. Then $\mathit{behaviour}_m(s) = \mathit{behaviour}_{\eta \circ m'}(s') = x \in \mathit{behaviour}_m(S)$.

To prove $\Leftarrow$, assume $\mathit{behaviour}_{\eta \circ m'}(S') \subseteq \mathit{behaviour}_m(S)$. Define $\sim \subseteq S' \times S$ as $\{(s', s') \mid \mathit{behaviour}_{\eta \circ m'}(s') = \mathit{behaviour}_m(s)\}$. This is an M-bisimulation and clearly $\forall s' \in S'.\ \exists s \in S.\ s' \sim s$, so $(S', m') \leq_\eta (S, m)$. □

Recall that, by Lemma 5.4, $\mathit{behaviour}_{\eta \circ m'}(S') = \mathit{cast}_\eta(\mathit{behaviour}_{m'}(S'))$. So the lemma above states that a coalgebra C' is a subtype of another coalgebra C iff the set of possible behaviours of the objects in C', viewed as objects with signature M, i.e. after casting, is included in the set of possible behaviours of objects in C. We believe that this accurately captures the intuition behind behavioural subtyping.

6.2 Behavioural subtyping between class specifications

Many definitions of behavioural subtyping in the literature are given in terms of specifications: behavioural subtypes then simply correspond to stronger specifications (e.g. see [Ame87] [Mey97] [LW94] [LW95] [4]). For example, consider the class specification `RCounter` given in Figure 2. It is easy to see that any object that meets the specification `RCounter` also meets the weaker specification `Counter`, since `RCounter` includes all the methods and assertions of `Counter`. So `RCounter` can be regarded as a behavioural subtype of `Counter`.

Definition 6.4
Let $\mathcal{A} = (M, \Phi, \Psi)$ and $\mathcal{A}' = (M', \Phi', \Psi')$ be class specifications.

$\mathcal{A}'$ is a *behavioural subtype* of $\mathcal{A}$ – written $\mathcal{A}' \leq_{assert} \mathcal{A}$ – iff

(i) $\mathcal{A}'$ is a signature subtype of $\mathcal{A}$, and

(ii) if $(S', m') \models \Phi'$ then $(S', \eta \circ m') \models \Phi$, for any M'-coalgebra (S', m'),

where $\eta : M' \to M$ is given by the signature subtyping between the specifications. □

Instead of condition (ii) in the definition above, one could simply require that Φ' implies Φ, suitably translated via η, i.e. $\mathit{cast}_\eta(\Phi') \subseteq \Phi$. This condition is

[4] In all these references, as is common in the OO literature, specifications are broken down into invariants, preconditions and postconditions; stronger specifications then have stronger invariants, stronger postconditions, but weaker preconditions.

```
CLASS RCounter
  METHODS getcount : X -> Int
              count : X -> X
              reset : X -> X

  ASSERTIONS
    ∀x:X.  getcount(count(x)) = getcount(x)+1
    ∀x:X.  getcount(reset(x)) = 0

  CREATION CONDITIONS
     getcount(new RCounter) = 0
END
```

Fig. 2. The class specification `RCounter`

```
CLASS AlternativeCounter
  METHODS getcount : X -> Int
              count : X -> X

  ASSERTIONS
   ∀ x:X.  getcount(count(x)) = getcount(x)+1
   ∀ x:X.  getcount(count(count(x))) = getcount(x)+2

  CREATION CONDITIONS
     getcount(new Counter) = 0
END
```

Fig. 3. The class specification `AlternativeCounter`

in fact strictly stronger:

Lemma 6.5
If $cast_\eta(\Phi') \subseteq \Phi$ *then* $\forall (S', m').\ (S', m') \models \Phi' \Rightarrow (S', \eta \circ m') \models \Phi$.

Proof. Assume $cast_\eta(\Phi') \subseteq \Phi$, and $(S', m') \models \Phi'$ i.e. $behaviour_{m'}(S') \subseteq \Phi'$. Then $behaviour_{\eta \circ m'}(S') = cast_\eta(behaviour_{m'}(S')) \subseteq cast_\eta(\Phi') \subseteq \Phi$. □

The class specification below illustrates why we have chosen the condition (ii) instead of the stronger condition $cast_\eta(\Phi') \subseteq \Phi$ in the definition of $\leq_{assert}$. Consider the class specification `AlternativeCounter` in Figure 3. It is not hard to see that this specification is equivalent to the specification `Counter` given earlier. Any model for `AlternativeCounter` will also be a model for `Counter` (and vice versa), and `AlternativeCounter` $\leq_{assert}$ `Counter`. Still, the assertions of `AlternativeCounter` do not imply those of `Counter`.

A condition equivalent to (ii) in Definition 6.4 is given by the lemma below:

Lemma 6.6 *Let* $\underline{\Phi'} \subseteq \nu M'$ *be the strongest invariant contained in* Φ'.
Then $cast_\eta(\underline{\Phi'}) \subseteq \Phi$ *iff* $\forall (S', m').\ (S', m') \models \Phi' \Rightarrow (S', \eta \circ m'a) \models \Phi$.

Proof. For the if-part, assume $cast_\eta(\underline{\Phi'}) \subseteq \Phi$, and let $(S', m') \models \Phi'$ for some M'-coalgebra (S', m'). As $behaviour_{m'}(S')$ is an invariant, it follows that $behaviour_{m'}(S') \subseteq \underline{\Phi'}$ and hence $behaviour_{\eta \circ m'}(S') = cast_\eta(behaviour_{m'}(S') \subseteq cast_\eta(\underline{\Phi'}) \subseteq \Phi$.

For the fi-part, assume $\forall (S', m').\ S', m') \models \Phi' \Rightarrow (S', \eta \circ m') \models \Phi$. As $(\underline{\Phi'}, \alpha_{M'})$ is an M'-coalgebra and $(\underline{\Phi'}, \alpha_{M'}) \models \Phi'$, it then follows by the assumption that $(\underline{\Phi'}, \eta \circ \alpha_{M'}) \models \Phi$, i.e. $behaviour_{\eta \circ \alpha_{M'}}(\underline{\Phi'}) = cast_\eta(\underline{\Phi'}) \subseteq \Phi$. □

For specifications that use the notion of bisimilarity $\overset{\leftrightarrow}{=}$ we have to be careful, because different specifications may use different notions of bisimilarity. By theorem 5.2, if $\mathcal{A}'$ is a signature subtype of $\mathcal{A}$, then the notion of bisimilarity used in $\mathcal{A}'$ is at least as strong as that used in $\mathcal{A}$. This means that an assertion stating a bisimilarity in $\mathcal{A}'$ specifies a stronger property than the same assertion in $\mathcal{A}$. But the same does not hold for an assertion stating that certain elements are not bisimilar. For example, an assertion $\forall$`x:X.count(x)`$\overset{\leftrightarrow}{\neq}$`x` in a specification with the signature of `Rcounter` does not imply the same assertion in a specification with the signature of `Counter`.

The creation conditions do not play any role in the definition of $\leq_{assert}$. This has some consequences which may appear counterintuitive at first sight. For example, consider the class specification `Counter1` in Figure 4.

```
CLASS Counter1
  METHODS getcount : X -> Int
              count : X -> X

  ASSERTIONS
    ∀x:X.  getcount(count(x)) = getcount(x)+1

  CREATION CONDITIONS
     getcount(new Counter1) = 1
END
```

Fig. 4. The class specification `Counter1`

It only differs from `Counter` in its creation condition, which for `Counter` was `getcount(new Counter) = 0`. Clearly `Counter1` $\leq_{assert}$ `Counter`. This may seem strange, because there appears to be an observable difference between the two specifications: their initial objects have different `getcounts`. But this is not an observable difference between individual objects: if you give an object from class `Counter1` to someone who is expecting to receive an object of class `Counter`, there is no way this person can observe that this object is not from class `Counter`, because the only possible observations are invocations of the methods, `count` and `getcount`. Unlike the methods, the constructor is not invoked on objects and is not an observation.

Another way of explaining that creation conditions do not play a role in

behavioural subtyping is that behavioural subtyping is about substitutability of individual objects, i.e. substituting objects of one class by objects of another class, and not about substitutability of classes.

One can consider a stronger notion of behavioural subtyping, which also requires that the creation conditions of the subclass imply those of the superclass. We will do this in Section 7, when we discuss refinement.

Something else which may appear counterintuitive is that we do not only have `Counter1` $\leq_{assert}$ `Counter`, but also `Counter` $\leq_{assert}$ `Counter1`. One might be tempted to conclude from the specification of `Counter1` that all objects in this class have a `count` of at least 1. There is an object in class `Counter` that has a `getcount` equal to 0 (namely, the initial object), so that would mean that we can distinguish this object in class `Counter` from all objects in class `Counter1`. However, we may not use this form of inductive reasoning, sometimes called *data induction*, to reason about classes. The motivation for this is that a class specification should offer as much freedom as possible for further extensions of the class. E.g. we want to leave open the possibility of adding a method `negate` with the specification

```
∀x:X.  getcount(negate(x)) = -getcount(x)
```

unless this is explicitly ruled out by the assertions. So, if we want all objects in class `Counter1` to have a `getcount` of at least 1, it should be specified explicitly as one of the assertions. (The semantic counterpart of this argument is that for models (S, m, c) we do not require that all elements of S are "reachable" from c by m.)

6.3 Soundness and Completeness

It is not immediately obvious what the connection between the subtype relation $\leq_{assert}$ between class specifications and the subtype relation $\leq_\eta$ between coalgebras should be. As a first guess, one might expect that if two specifications are related by $\leq_{assert}$, then their implementations will be related by $\leq_\eta$, i.e.

$$\mathcal{A}' \leq_{assert} \mathcal{A} \;\Rightarrow\; \forall (S', m', c') \models \mathcal{A}'. \forall (S, m, c) \models \mathcal{A}.\ (S', m') \leq_\eta (S, m).$$

However, we cannot expect this property to hold. For example, take $\mathcal{A}' \equiv \mathcal{A}$ some trivially true specification, e.g. one with without any assertions or creation conditions. Clearly $\mathcal{A}' \leq_{assert} \mathcal{A}$. However, there are lots of models of this specification that are not related by $\leq_\eta$ in any way, so the right-hand side of the implication above will not hold. In fact, if $\mathcal{A}' \equiv \mathcal{A}$ and $\mathcal{A}'$ is any specification weak enough to allow observably different implementations, then we cannot expect the property above to hold.

We will prove a weaker relation, namely

$$\mathcal{A}' \leq_{assert} \mathcal{A} \;\Longleftrightarrow\; \forall (S', m', c') \models \mathcal{A}'. \exists (S, m, c) \models \mathcal{A}.\ (S', m') \leq_\eta (S, m).$$

The right-hand side says that however we implement $\mathcal{A}'$, there is an implementation of $\mathcal{A}$ that includes all the behaviour of this implementation of $\mathcal{A}'$, after casting. The property crucially relies on the restriction imposed on assertions in Section 4. In particular, we use Lemma 4.4.

Theorem 6.7 (Soundness)
Let $\mathcal{A}$ be a consistent specification (i.e. one with at least one model). Then

$$\mathcal{A}' \leq_{assert} \mathcal{A} \quad \Rightarrow \quad \forall (S', m', c') \models \mathcal{A}'.\ \exists (S, m, c) \models \mathcal{A}.\ (S', m') \leq_\eta (S, m)$$

with η given by the signature subtyping between the specifications.

Proof. Assume $\mathcal{A}' \leq_{assert} \mathcal{A}$ and $(S', m', c') \models \mathcal{A}'$, with $\mathcal{A}' = (M', \Phi', \Psi')$ and $\mathcal{A} = (M, \Phi, \Psi)$. To prove: $\exists (S, m, c) \models \mathcal{A}.\ (S', m') \leq_\eta (S, m)$.

We know that $(S', m') \models \Phi'$, and hence by $\mathcal{A}' \leq_{assert} \mathcal{A}$ it follows that $(S', \eta \circ m') \models \Phi$. Since $\mathcal{A}$ is a consistent specification, we may assume there is some model (T, n, d) of $\mathcal{A}$. By Lemma 4.4 now $(S' + T, p, \mathsf{inr}(d)) \models \mathcal{A}$, where $p = [M(\mathsf{inl}) \circ \eta \circ m', M(\mathsf{inr}) \circ n]$, and it is simple to prove $(S', m') \leq_\eta (S' + T, p)$. □

The restriction to consistent specifications in the theorem above is really necessary. For example, let $\mathcal{A}'$ be a consistent specification and $\mathcal{A}$ a specification with weaker assertions but an inconsistent creation condition. Then $\mathcal{A}' \leq_{assert} \mathcal{A}$, but clearly the right-hand side of the implication in the theorem above does not hold, since $\mathcal{A}$ has no models.

Theorem 6.8 (Completeness)
Let $\mathcal{A}'$ be a consistent specification. Then

$$(\forall (S', m', c') \models \mathcal{A}'.\ \exists (S, m, c) \models \mathcal{A}.\ (S', m') \leq_\eta (S, m)) \quad \Rightarrow \quad \mathcal{A}' \leq_{assert} \mathcal{A}$$

with η given by the signature subtyping between the specifications.

Proof. Let $\mathcal{A} = (M, \Phi, \Psi)$ and $\mathcal{A}' = (M', \Phi', \Psi')$ and suppose

$$\forall (S', m', c') \models \mathcal{A}'.\ \exists (S, m, c) \models \mathcal{A}.\ (S', m') \leq_\eta (S, m). \quad \text{(i)}$$

We must prove $\mathcal{A}' \leq_{assert} \mathcal{A}$, i.e. $(S', m') \models \Phi' \Rightarrow (S', \eta \circ m') \models \Phi$ for any M'-coalgebra (S', m').

Let $(S', m') \models \Phi'$. Since $\mathcal{A}'$ is a consistent specification, we may assume some model (T, n, d) of $\mathcal{A}'$. By Lemma 4.4 then $(S' + T, p, \mathsf{inr}(d)) \models \mathcal{A}'$, where $p = [M'(\mathsf{inl}) \circ m', M'(\mathsf{inr}) \circ n]$. Now by (i) there exists some $(S, m, c) \models \mathcal{A}$ such that $(S' + T, p) \leq_\eta (S, m)$, and then

$$\begin{aligned} \mathit{behaviour}_{\eta \circ m'}(S') &\subseteq \mathit{behaviour}_{\eta \circ p}(S' + T) && \text{by def. } p \\ &\subseteq \mathit{behaviour}_m(S) && \text{since } (S' + T, p) \leq_\eta (S, m) \\ &\subseteq \Phi && \text{since } (S, m, c) \models \mathcal{A} \end{aligned}$$

i.e. $(S', \eta \circ m') \models \Phi$. □

Again, the restriction to consistent specifications in the theorem above is really necessary. For example, suppose $\mathcal{A}'$ is inconsistent because its creation

condition is in contradiction with its assertions. Then the left-hand side of the implication in the theorem above is trivially true for any specification $\mathcal{A}$, but clearly $\mathcal{A}' \leq_{assert} \mathcal{A}$ will not hold for any $\mathcal{A}$.

7 Refinement

Refinement is a central notion in specification languages. A specification $\mathcal{A}'$ is a refinement of another specification $\mathcal{A}$ if given any implementation of $\mathcal{A}'$ we can construct an implementation of $\mathcal{A}$. In [Jac97] refinement for class specifications is defined as follows:

Definition 7.1 [Refinement [Jac97]]
Let $\mathcal{A}'$ and $\mathcal{A}$ be class specifications. Let $\eta : M' \to M$.

$\mathcal{A}'$ *is a refinement* of $\mathcal{A}$ iff

$$\forall (S', m', c') \models \mathcal{A}'.\ \exists (S, \eta \circ m', c) \models \mathcal{A}.\ \ S \subseteq S' \wedge\ c \in S\ \ ,$$

with η given by the signature subtyping between the specifications. □

The natural transformation η gives the definition of the abstract operations of $\mathcal{A}$ in terms of the concrete ones of $\mathcal{A}'$. Note that for refinement these natural transformations can be much wilder than the very simple natural transformation that occur in subtyping, which simply drop some components from a n-tuple.

For an example for refinement, we can use the specifications `Rcounter` and `Counter` given earlier: `Rcounter` is a refinement of `Counter`. Because `Rcounter` is also a behavioural subtype of `Counter`, this example suggests a close connection between refinement and behavioural subtyping. However, such a connection does not exist: there are refinements that are not behavioural subtypes, and behavioural subtypes that are not refinements. For example, consider the class specification `DCounter` in Figure 5. `DCounter` is not a behavioural subtype of `Counter`, because there are objects in this class for which the `Counter` assertion

```
∀ x:X.  getcount(count(x)) = getcount(x)+1
```

does not hold. Still, `DCounter` is a refinement of `Counter`, and it is obvious how we can implement `Counter` given any implementation of `DCounter`.

For an example of a behavioural subtype that is not a refinement, consider `Counter1` and `Counter`. As explained earlier, the specification `Counter1` is a behavioural subtype of `Counter`. However, it is not a refinement, because there are models of `Counter1` that do not provide a suitable initial state for `Counter`, i.e. one with a `getcount` equal to 0. Note that this shows that for refinement, unlike subtyping, the creation conditions do play a role.

The lemma below illustrates a fundamental difference between refinement and subtyping. Refinements have more behaviour, whereas subtypes have less:

```
CLASS DCounter
  METHODS getcount : X -> Int
             count : X -> X
          getdelta : X -> Int
          setdelta : X × Int -> X

  ASSERTIONS
    ∀x:X.  getcount(count(x)) = getcount(x)+getdelta(x)
    ∀x:X.  getdelta(count(x)) = getdelta(x)
    ∀x:X,n:Int.  getdelta(setdelta(x,n)) = n
    ∀x:X,n:Int.  getcount(setdelta(x,n)) = getcount(x)

  CREATION CONDITIONS
     getcount(new DCounter) = 0
     getdelta(new DCounter) = 1
END
```

Fig. 5. The class specification `DCounter`

Lemma 7.2 (i) *If $\mathcal{A}'$ is a refinement of $\mathcal{A}$ then*

$$\forall (S', m', c') \models \mathcal{A}'.\ \exists (S, m, c) \models \mathcal{A}.\ behaviour_{\eta \circ m'}(S') \supseteq behaviour_m(S).$$

(ii) *If $\mathcal{A}'$ is a subtype of $\mathcal{A}$ and $\mathcal{A}$ is consistent then*

$$\forall (S', m', c') \models \mathcal{A}'.\ \exists (S, m, c) \models \mathcal{A}.\ behaviour_{\eta \circ m'}(S') \subseteq behaviour_m(S).$$

Proof. To prove (i), simply note that if $S \subseteq S'$ and $m = \eta \circ m'$ then $behaviour_{\eta \circ m'}(S') \supseteq behaviour_m(S)$. Part (ii) follows immediately from soundness, Theorem 6.7. □

For a stronger relation than behavioural subtyping, that does take the creation conditions into account, we can define a clear relation with refinement:

Definition 7.3 Let $\mathcal{A}' = (M', \Phi', \Psi')$ and $\mathcal{A} = (M, \Phi, \Psi)$ be class specifications.

Then *$\mathcal{A}'$ is a stronger specification than $\mathcal{A}$* – written $\mathcal{A}' \leq_{assert,create} \mathcal{A}$ – iff $\mathcal{A}' \leq_{assert} \mathcal{A}$ and $cast_\eta(\Psi') \subseteq \Psi$, with η given by the signature subtyping between the specifications. □

Lemma 7.4 *If $\mathcal{A}' \leq_{assert,create} \mathcal{A}$ then $\mathcal{A}'$ is a refinement of $\mathcal{A}$.*

Proof. Easy: if $\mathcal{A}' \leq_{assert,create} \mathcal{A}$ and $(S', m', c') \models \mathcal{A}'$ then by the definition of $\leq_{assert,create}$ it immediately follows that $(S', \eta \circ m', c') \models \mathcal{A}$. □

8 Conclusions

We have shown that the coalgebraic view of objects also provides a natural interpretation of (behavioural) subtyping, both for coalgebras themselves and for coalgebraic specifications. The subtyping relation between coalgebras is

based on the notion of substitutability, and uses the notion of bisimulation. The subtyping relation between coalgebraic specifications is based on the idea that behavioural subtypes correspond to stronger specifications. Both ways of characterizing behavioural subtyping already exist in the OO literature.

It is interesting to note that some definitions of behavioural subtyping in the literature combine the use of specifications with the use of bisimulations. For example in [Ame87] and [LW94], specifications make use of a so-called abstraction function that maps objects to the abstract values they represent. Such an abstraction function in fact defines a bisimulation relation, namely objects are related iff they have the same abstract value.

As mentioned in Section 2, one difference between inheritance and behavioural subtyping is that the former concerns implementations of classes, whereas the latter only concerns the observable behaviour of these implementations. This already suggests that in the coalgebraic setting, where observability plays such a central role, it is easier to explain subtyping than inheritance. Indeed, a limitation of the coalgebraic view of objects is that is does *not* directly account for the self-references in method definitions, i.e. invocations of methods on "self". As these self-references play a central role in inheritance with late binding, we cannot expect the coalgebraic object model to easily account for inheritance with late binding.

References

[Ame87] Pierre America. Inheritance and Subtyping in a Parallel Object-Oriented Language. In J. Bezivin et al., editor, *ECOOP'87*, volume 276 of *Lecture Notes in Computer Science*, pages 232–242. Springer, 1987.

[BCC+96] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Gary T. Leavens, and Benjamin Pierce. On Binary Methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

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

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[HHJT98] Ulrich Hensel, Marieke Huisman, Bart Jacobs, and Hendrik Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In Ch. Hankin, editor, *European Symposium on Programming (ESOP)*, number 1381 in Lecture Notes in Computer Science, pages 105–121. Springer, Berlin, 1998.

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

[Jac97] Bart Jacobs. Invariants, bisimulations and the correctness of coalgebraic refinements. In M. Johnson, editor, *Algebraic Methodology and Software Technology (AMAST'97)*, LNCS, pages 276–291. Springer, 1997.

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

[Lis88] Barbara H. Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, 23(3), 1988.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *TOPLAS*, 16(6):1811–1841, November 1994.

[LW95] Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, November 1995.

[Mau95] Ian Maung. On simulation, subtyping and substitutability in sequential object systems. *Formal Aspects of Computing*, 7(6):620–651, 1995.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2^{nd} rev. edition, 1997.

[Mit90] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Principles of Programming Languages (POPL)*, pages 109–124. ACM, 1990.

[Pol97] Erik Poll. Subtyping and Inheritance for Categorical Datatypes. In *Theories of Types and Proofs (TTP) - Kyoto*, RIMS Lecture Notes 1023, pages 112–125. Kyoto University Research Insitute for Mathematical Sciences, 1997.

[Pol98] Erik Poll. Behavioural subtyping for a type-theoretic model of objects. In *Foundations of Object-Oriented Languages (FOOL5)*, 1998.

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

[Rut96] Jan Rutten. Universal co-algebra: a theory of systems. CWI report 9652, CWI, 1996.