Theoretical Informatics and Applications Will be set by the publisher
Informatique Théorique et Applications

A COALGEBRAIC SEMANTICS OF SUBTYPING

Erik Poll[1]

Abstract. Coalgebras have been proposed as formal basis for the semantics of objects in the sense of object-oriented programming. This paper shows that this semantics provides a smooth interpretation for subtyping, a central notion in object-oriented programming. We show that different characterisations of behavioural subtyping found in the literature can conveniently be expressed in coalgebraic terms. We also investigate the subtle difference between behavioural subtyping and refinement.

AMS Subject Classification. 18C50, 68Q70, 68Q85.

1. Introduction

Subtyping is one of the famous buzzwords in object-oriented 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 coalgebras has been proposed as a semantics of objects in [26], an obvious question to ask is if this semantics accounts for subtyping. This paper shows that the coalgebraic view of objects provides a clean semantics for so-called *behavioural subtyping*. Moreover, different characterisations of behavioural subtyping found in the literature can conveniently be expressed in coalgebraic terms, and proved to be equivalent.

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

One should be aware that there are important limitations to coalgebras as semantics of objects. The coalgebraic semantics of objects only explains objects in a purely functional – as opposed to imperative – setting, and, because states of individual objects are completely independent of one another, it does not account

[1] Department of Computer Science, University of Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.

for aliasing and sharing, which are major complications in the imperative OO setting. The work in this paper inherits these limitations of the coalgebraic view of objects.

This paper is organised as follows. Section 2 gives an informal explanation of subtyping in object-oriented programming languages. 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 considers the extension of classes with constructors. Section 8 discusses the relation between subtyping and refinement, and we conclude in Section 9.

2. Subtyping

Different notions of subtyping exist. There is a purely syntactical notion of subtyping, which we call *signature subtyping*, and a stronger, semantical, notion of subtyping, usually called *behavioural subtyping* [3, 19].

2.1. Signature Subtyping

Signature subtyping concerns the signatures – or interfaces – of classes, i.e. the collection of methods a class provides together with their in- and output types. A class A' is a signature subtype of another class A if the subclass A' provides all the methods that the superclass A provides, with "compatible" types[1]. This notion of subtyping is extensively studied in type theory, e.g. see [1,6,10]. Signature subtyping can be mechanically checked by type checking algorithms, ensuring that no type errors (of the form "method not found") can occur at run-time.

2.2. Behavioural Subtyping

Behavioural subtyping is a stronger notion than signature subtyping. It not only concerns the signatures of the methods, but also their semantics. 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: 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

[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.

`Vehicle`. Signature subtyping is a necessary – but not a sufficient – condition for this.

Many definitions of behavioural subtyping have been proposed in the literature, e.g. [2–4, 17–21, 25].

One approach to define behavioural subtyping 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 [21], and is widely used in the literature, e.g. [2, 17, 19].

Another well-known characterisation of behavioural subtyping is by the principle of *substitutability* [18]: "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 give definitions of behavioural subtyping in the coalgebraic setting in both of the ways mentioned above, and relate the two.

2.3. Subtyping vs Inheritance

In the OO literature 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 [2, 8, 29]. 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". Sometimes subtyping is called "interface inheritance" and inheritance "implementation inheritance".

Inheritance allows a (sub)class A' of a (super)class A to be constructed 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[2].

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 more important than the code reuse made possible by behavioural subtyping, i.e. the reuse of client code. The programming language C++ offers a

[2] In the presence of so-called binary methods just adding methods may also break behavioural subtyping, even when no methods are overridden (see [5, 8]).

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 [15] or [27]).

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

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

where C ranges over constant sets. Throughout this paper, the variable name X is used to denote some (hidden) state space. 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. For functions $f_i : A_i \rightarrow C$ and $g_i : C \rightarrow A_i$, the functions $[f_1, f_2] : A_1 + A_2 \rightarrow C$ and $\langle g_1, g_2 \rangle : C \rightarrow A_1 \times A_2$ are defined as usual.

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

For every polynomial functor F, 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 $\mathit{behaviour}_m$. The final coalgebra can be viewed as the collection of all the possible behaviours of objects with interface F; the function $\mathit{behaviour}_m$ then maps every state $s \in S$ to its observable behaviour $\mathit{behaviour}_m(s) \in \nu F$.

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). Invariants are closed under union, so given a predicate $P \subseteq S$ we can define the strongest invariant contained in P, written $\underline{P}$, as the union of all invariants contained in P.

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

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

Bisimulations are closed under union; we write $\underline{\leftrightarrow}$ for *bisimilarity*, the largest bisimulation, 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 $s \underline{\leftrightarrow} s'$.* □

4. Classes and Class Specifications

Our format of class specification is based on that used in the experimental specification language CCSL [11, 14]. An example of such a class specification is given in Figure 1. This example specifies a class with two methods, `getcount` and `count`. 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. Later, in Section 7, we will consider classes that not only have methods but also have constructors. Note that coalgebras provide a purely functional – as opposed to imperative – view of objects: Invoking a method such as `count` doesn't mutate a object, leaving its “identity” intact, but simply provides a new object with a mutated state.

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

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

Figure 1. The class specification `Counter`

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,

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), with S giving a representation of the state space and m giving an implementation of the methods that satisfies the assertions. For example, the obvious implementation for the specification `Counter` above would be $(\mathbb{N}, \langle id, Succ\rangle)$

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

A way to impose these restrictions would be to give a precise syntax for assertions. 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. To keep things simple here we will not go to all this trouble; instead we simply require that assertions are predicates on the final coalgebra, i.e. predicates on object behaviours:

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

For example, if $M(X) = Int \times X$ and the final M-coalgebra $(\nu M, \alpha_M) = (\nu M, \langle getcount, count\rangle)$, then the predicate Φ expressing the assertions of the specification `Counter` is simply $\Phi(x) = (getcount(count(x)) = getcount(x)))$.

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 $\mathcal{A} = (M, \Phi)$ is a coalgebra (S, m), such that $\forall s : S.\ \Phi(behaviour_m(s))$, i.e. $behaviour_m(S) \subseteq \Phi$.

The fact that (S, m) is a model of $\mathcal{A}$ is denoted by $(S, m) \models \mathcal{A}$, and we write $(S, m) \models \Phi$ for $behaviour_m(S) \subseteq \Phi$.

For example, $(\mathbb{N}, \langle id, S\rangle) \models$ `Counter`. The restrictions on the shape of assertions give some nice properties. Any submodel of a model is also a model, and specifications do not distinguish between observationally equal models:

Lemma 4.3. *Let (S', m) and (S, m) be M-coalgebras.*

1. *If (S', m) is a subcoalgebra of (S, m), then $(S, m) \models \mathcal{A} \Rightarrow (S', m) \models \mathcal{A}$.*
2. *If there exists a total bisimulation $\sim$ between (S, m) and (S, m') then $(S, m) \models \mathcal{A} \Leftrightarrow (S', m') \models \mathcal{A}$.*

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

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.

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
```

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

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 (see for instance [6]), 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,

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

namely 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 ([27], 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 [24]:

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

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

Subscripts of η are typically omitted when they are clear from the context, e.g. in $\eta_{\nu M'} \circ \alpha_{M'}$ above. 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. By definition $behaviour_{m'} : (S', m') \to (\nu M', \alpha_{M'})$ is an M'-coalgebra homomorphism, so by Theorem 5.2 $behaviour_{m'} : (S', \eta \circ m') \to (\nu M', \eta \circ \alpha_{M'})$ is an M-coalgebra homomorphism. Since $(S', \eta \circ m')$ is an M-coalgebra, by definition $behaviour_{\eta \circ m'} : (S', m') \to (\nu M', \alpha_{M'})$ is an M-coalgebra homomorphism. So in the category of M-coalgebras

$$\begin{tikzcd}
(S', \eta \circ m') \arrow[rr, "behaviour_{m'}"] \arrow[dr, "behaviour_{\eta \circ m'}"'] & & (\nu M', \eta \circ \alpha_{M'}) \arrow[dl, "cast_\eta"] \\
& (\nu M, \alpha_M) &
\end{tikzcd}$$

and then $cast_\eta \circ behaviour_{m'} = behaviour_{\eta \circ m'}$ by the finality of $(\nu M, \alpha_M)$. □

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

$$\begin{tikzcd}
\nu M' \arrow[r, "\eta_{\nu M'} \circ \alpha_{M'}"] \arrow[d, "cast_\eta"'] & M(\nu M') \arrow[d, "M(cast_\eta)"] \\
\nu M \arrow[r, "\alpha_M"] & M(\nu M)
\end{tikzcd}$$

commutes. This diagram expresses precisely the condition that subsumption – the implicit cast from sub- to superclass – does not introduce any ambiguities. For example, let $(\nu M, \langle \mathit{getcount}, \mathit{count}\rangle)$ and $(\nu M', \langle \mathit{getcount}', \mathit{count}', \mathit{reset}'\rangle)$ be the final M- and M'-coalgebras, with $M(X) = \mathit{Int} \times X$ and $M'(X) = \mathit{Int} \times X \times X$. Then for the $\mathit{cast}_{\langle \pi_1, \pi_2\rangle} : \nu M' \to \nu M$ we have

$$\begin{tikzcd}
\nu M' \arrow[r, "\langle \mathit{getcount}', \mathit{count}', \mathit{reset}'\rangle"] \arrow[d, "\mathit{cast}_{\langle \pi_1, \pi_2\rangle}"'] & \mathit{Int} \times \nu M' \times \nu M' \arrow[d, "\langle \pi_1, \mathit{cast}_{\langle \pi_1, \pi_2\rangle} \circ \pi_2\rangle"] \\
\nu M \arrow[r, "\langle \mathit{getcount}, \mathit{count}\rangle"] & \mathit{Int} \times \nu M
\end{tikzcd}$$

So, for instance, 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 [13, 22, 24].

6. Behavioural subtyping

We define two notions of behavioural subtyping. In Subsection 6.1, we define behavioural subtyping between coalgebras, and in Subsection 6.2 we define behavioural subtyping between class specifications. These two definitions correspond to the two ways of characterising behavioural subtyping found in the literature that were discussed in Section 2. We will prove that the latter notion of behavioural subtyping is sound and complete with respect to the former.

6.1. Behavioural subtyping between coalgebras

We already mentioned Liskov's substitution principle [18]: "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. [17, 20, 25].

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

Lemma 6.2. 1. $(S, m) \leq_{id} (S, m)$.

```
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
```

Figure 2. The class specification `RCounter`

2. *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*

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

Proof. It follows immediately from Lemma 3.1 that there exists an M-bisimulation $\sim \subseteq S' \times S$ between $(S', \eta \circ m')$ and (S, m) such that $\forall s' \in S'.\ \exists s \in S.\ s' \sim s$ iff $\mathit{behaviour}_{\eta \circ m'}(S') \subseteq \mathit{behaviour}_m(S)$. □

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 a subset of the set of possible behaviours of objects in C. We believe that this accurately captures the intuition behind behavioural subtyping. Informally subtypes are often explained as subsets; the result above makes it precise in what way subtypes can be viewed as subsets.

In our opinion the coalgebraic characterisation of behavioural subtyping given by Lemma 6.3 above is a lot simpler and more elegant than algebraic characterisations of behavioural subtyping that have been proposed in the literature, e.g. in [17] or [16].

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. This is the way behavioural subtyping is defined in for instance [2,3,9,19,21]. 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

```
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
```

FIGURE 3. The class specification `AlternativeCounter`

specification `Counter`, since `RCounter` includes all the methods and assertions of `Counter`. So `RCounter` can be regarded as a behavioural subtype of `Counter`.

In the OO literature, specifications are typically broken down into invariants, preconditions, and postconditions, and behavioural subtypes are required to have stronger invariants, stronger postconditions, but weaker preconditions[3].

Definition 6.4. Let $\mathcal{A} = (M, \Phi)$ and $\mathcal{A}' = (M', \Phi')$ be class specifications.
$\mathcal{A}'$ is a *behavioural subtype* of $\mathcal{A}$ – written $\mathcal{A}' \leq \mathcal{A}$ – iff

1. $\mathcal{A}'$ is a signature subtype of $\mathcal{A}$, and
2. 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 (2) in the definition above, one could simply require that Φ' implies Φ, suitably translated via η, i.e. $cast_\eta(\Phi') \subseteq \Phi$. This condition is 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 by Lemma 5.4 $behaviour_{\eta \circ m'}(S') = cast_\eta(behaviour_{m'}(S')) \subseteq cast_\eta(\Phi') \subseteq \Phi$. □

The class specification `AlternativeCounter` in Figure 3 illustrates why we have chosen condition (2) instead of the stronger condition $cast_\eta(\Phi') \subseteq \Phi$ in the definition of $\leq$. It is not hard to see that the specification `AlternativeCounter` is equivalent to the specification `Counter` given earlier. Any model for `Counter` will also be a model for `AlternativeCounter` (and vice versa), and `Counter` $\leq$ `AlternativeCounter`. Still, the assertions of `Counter` do not imply those of `AlternativeCounter`.

A condition equivalent to (2) in Definition 6.4 is given by the lemma below:

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

[3] As noted in [7, 9], the requirement that postconditions are stronger and preconditions are weaker is stronger than necessary: the postcondition only has to be stronger in the cases where the stronger precondition of the superclass holds.

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

To prove ($\Leftarrow$), 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`.

We have the following relation between $\leq$, subtyping between class specifications, and $\leq_\eta$, subtyping between class implementations:

Theorem 6.7 (Soundness and Completeness)**.** *Let* $\mathcal{A}' = (M', \Phi')$ *and* $\mathcal{A} = (M, \Phi)$, *and* $\eta : M' \to M$ *given by the signature subtyping between* M' *and* M*. Then*

$$\mathcal{A}' \leq \mathcal{A} \iff \forall (S', m') \models \mathcal{A}'.\ \exists (S, m) \models \mathcal{A}.\ (S', m') \leq_\eta (S, m)$$

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, after casting.

Proof. To prove ($\Rightarrow$), assume $\mathcal{A}' \leq \mathcal{A}$ and $(S', m') \models \mathcal{A}'$. Then by the definition of $\mathcal{A}' \leq \mathcal{A}$ it follows that $(S', \eta \circ m') \models \Phi$, and clearly $(S', \eta \circ m') \leq_\eta (S, m)$.

To prove ($\Leftarrow$), assume $\forall (S', m') \models \mathcal{A}'.\ \exists (S, m) \models \mathcal{A}.\ (S', m') \leq_\eta (S, m)$. We must prove $\mathcal{A}' \leq \mathcal{A}$, i.e. $(S', m') \models \Phi' \Rightarrow (S', \eta \circ m') \models \Phi$ for any M'-coalgebra (S', m'). Let $(S', m') \models \Phi'$. Now by assumption there exists some $(S, m) \models \mathcal{A}$ such that $(S', m) \leq_\eta (S, m)$, and then

$$\begin{array}{lcll} behaviour_{\eta\circ m'}(S') & \subseteq & behaviour_m(S) & \text{since } (S', m') \leq_\eta (S, m) \\ & \subseteq & \Phi & \text{since } (S, m) \models \mathcal{A} \end{array}$$

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

The basic idea behind the combination of existential and universal quantification in the theorem above – that for *every* model of the subtype there must exist *some* corresponding model of the supertype – is also used in the definition of 'correct behavioural subtyping' (Def. 5.1) in [16] and the definition of 'legal subtype relation' (Def. 4.1) in [17].

One might expect a stronger property than given by the theorem above, namely that if two specifications are related by $\leq$, then any implementations will be related by $\leq_\eta$, i.e.

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

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. Clearly $\mathcal{A}' \leq \mathcal{A}$. 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.)

7. Classes with Constructors

We now consider classes that not only provide methods that can be invoked on objects, but also *constructors* with which to create objects.

For simplicity, we assume that such classes have a single parameterless constructor called `new`. This restriction is not essential, and one could easily allow classes with several constructors or constructors that take arguments, as is done in CCSL [11, 14].

In addition to assertions specifying properties of the methods, class specifications will now also include *creation conditions* that specify properties of the constructor `new`. An example of such a specification is given in Figure 4.

```
CLASS Counter0
  METHODS getcount : X -> Int
             count : X -> X

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

  CREATION CONDITIONS
     getcount(new Counter0) = 0
```

Figure 4. The class specification `Counter0`

As for the assertions, we want to impose some restrictions on the creation conditions that are allowed. Like assertions, creation conditions should not distinguish observationally equal implementations. Also, creation conditions should only specify properties of the initial object `new`. As for the assertions, we will not go to the trouble of defining a precise syntax for creation conditions, but we just take a predicate on the final coalgebra to specify the creation conditions:

Definition 7.1. A *specification of a class with a constructor* $\mathcal{A}$ 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 assertions and the creation conditions, respectively. A model of a class specification with a constructor is a coalgebra together an initial state $c : S$ giving the implementation of the constructor `new`:

Definition 7.2. A *model* of a class specification $\mathcal{A} = (M, \Phi, \Psi)$ is a triple (S, m, c) with (S, m) a coalgebra such that $(S, m) \models \Phi$, and $c : S$ such that $\Psi(\mathit{behaviour}_m(c))$.

The fact that (S, m, c) is a model of $\mathcal{A}$ is denoted by $(S, m, c) \models \mathcal{A}$.

In other words, all objects satisfy the assertions and the initial object satisfies the creation conditions. A minor difference with the notion of model in [14] is that we explicitly include the initial state c as part of the model.

Note that there may be elements in the state space S which are not "reachable" from the constructor c using the methods m. For example $(\mathbb{Z}, \langle id, Succ \rangle, 0) \models$ `Counter0`, even though the negative elements in $\mathbb{Z}$ cannot be reached from the initial state 0.

7.1. Behavioural subtyping for classes with constructors

Constructors do not play any role as far as signature or behavioural subtyping is concerned. Subtyping for specifications with creation conditions is simply defined as follows:

Definition 7.3. $(M', \Phi', \Psi') \leq (M, \Phi, \Psi)$ iff $(M', \Phi') \leq (M, \Phi)$.

In other words, $(M', \Phi', \Psi') \leq (M, \Phi, \Psi)$ iff, for all M'-coalgebras (S', m'), $(S', m') \models (M', \Phi')$ implies $(S', \eta \circ m') \models (M, \Phi)$. Note that this definition of subtyping only depends on the assertions (i.e. Φ and Φ'), and not on the creation conditions (i.e. Ψ and Ψ').

The fact that constructors and creation conditions do not play any role in subtyping may need some explanation. As an example, consider the class specification `Counter1` in Figure 5. It only differs from `Counter0` in Figure 4 in its

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

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

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

Figure 5. The class specification `Counter1`

creation condition. By the definition above we have `Counter1` $\leq$ `Counter0`. This may seem strange, because there appears to be an observable difference between the two specifications: their initial objects have different `getcount`s. 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 `Counter0`, there is no way this person can observe that this object is not from class `Counter0`, 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 should 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 8, when we discuss refinement.

Something else which may appear counterintuitive is that not only `Counter1` $\leq$ `Counter0`, but also `Counter0` $\leq$ `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 `Counter0` that has a `getcount0` equal to 0 (namely, the initial object), so that would mean that we can distinguish this object in class `Counter0` 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.)

For classes with constructors we have the same relation between $\leq$ and $\leq_\eta$ as before. Proving this requires the lemma below that says that, using a coproduct, we can take the "union" of models:

Lemma 7.4. *Let* (S, m) *and* (T, n) *be* M*-coalgebras, and* $\mathcal{A} = (M, \Phi, \Psi)$ *some class specification. Then 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 $\mathit{behaviour}_p(S + T) = \mathit{behaviour}_m(S) \cup \mathit{behaviour}_n(T)$, and $\mathit{behaviour}_p(\mathsf{inr}(d)) = \mathit{behaviour}_n(d)$. □

Theorem 7.5 (Soundness)**.** *Let $\mathcal{A}$ be a consistent specification (i.e. one with at least one model). Then*

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

with η given by the signature subtyping between the specifications.

Proof. Assume $\mathcal{A}' \leq \mathcal{A}$ 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 $\mathcal{A}' \leq \mathcal{A}$ 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 7.4 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, so that $\mathcal{A}$ is inconsistent. Then $\mathcal{A}' \leq \mathcal{A}$, since $\mathcal{A}'$ has stronger assertions, but clearly the right-hand side of the implication in the theorem above does not hold, since $\mathcal{A}$ has no models.

Theorem 7.6 (Completeness)**.** *Let $\mathcal{A}'$ be a consistent specification. Then*

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

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 $\mathcal{A}' \leq \mathcal{A}$, 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 7.4 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

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

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 $\mathcal{A}' \leq \mathcal{A}$ will not hold for any $\mathcal{A}$.

```
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 RCounter0) = 0
```

FIGURE 6. The class specification `RCounter0`

8. Refinement

Refinement is a central notion in specification languages. Intuitively, a specification $\mathcal{A}'$ is a refinement of another specification $\mathcal{A}$ if any implementation of $\mathcal{A}'$ can be "turned into" an implementation of $\mathcal{A}$. There is a variety of definitions of refinement in the literature, which differ as to what "turned into" is taken to mean. For example, just in the field of algebraic specification there are already different notions of refinement, e.g. see [28]. In [14] refinement for coalgebraic class specifications is defined as follows:

Definition 8.1 (Refinement [14]). Let $\mathcal{A}'$ and $\mathcal{A}$ be class specifications. $\mathcal{A}'$ *is a refinement* of $\mathcal{A}$ iff

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

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, consider the specifications `RCounter0` in Figure 6 and `Counter0` given earlier in Figure 4: `RCounter0` is a refinement of `Counter0`. Note that we don't really have to do anything to turn an implementation of `RCounter0` into an implementation of `Counter0`, because `RCounter0` is also a behavioural subtype of `Counter0`.

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 7. `DCounter` is not a behavioural subtype of `Counter0`, because there are objects in this class for which the `Counter0` assertion

```
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
```

FIGURE 7. The class specification DCounter

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

does not hold. Still, DCounter is a refinement of Counter0, and it is obvious how we can implement Counter0 given any implementation of DCounter.

For an example of a behavioural subtype that is not a refinement, consider Counter1 and Counter0. As explained earlier, the specification Counter1 is a behavioural subtype of Counter0. However, it is not a refinement, because there are models (S', m', c') of Counter1 that do not provide a suitable initial state $c \in S'$ for Counter0, i.e. one with a getcount equal to 0. 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 behavioural subtypes have less:

Lemma 8.2. 1. *If $\mathcal{A}'$ is a refinement of $\mathcal{A}$ then*

$$\forall (S', m', c') \models \mathcal{A}'.\ \exists (S, m, c) \models \mathcal{A}.\ \mathit{behaviour}_{\eta \circ m'}(S') \supseteq \mathit{behaviour}_m(S).$$

2. *If $\mathcal{A}'$ is a behavioural subtype of $\mathcal{A}$ and $\mathcal{A}$ is consistent then*

$$\forall (S', m', c') \models \mathcal{A}'.\ \exists (S, m, c) \models \mathcal{A}.\ \mathit{behaviour}_{\eta \circ m'}(S') \subseteq \mathit{behaviour}_m(S).$$

Proof. To prove (1), it suffice to observe that if $S \subseteq S'$ and $m = \eta \circ m'$ then $\mathit{behaviour}_{\eta \circ m'}(S') \supseteq \mathit{behaviour}_m(S)$. Part (2) follows immediately from soundness, Theorem 7.5. □

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

Definition 8.3. 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 $\mathcal{A}' \Rightarrow \mathcal{A}$, iff

1. $\mathcal{A}'$ is a signature subtype of $\mathcal{A}$, and
2. if $(S', m', c') \models \mathcal{A}'$ then $(S', \eta \circ m', c') \models \mathcal{A}$,

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

Note the subtle difference between $(M', \Phi', \Psi') \leq (M, \Phi, \Psi)$ and $(M', \Phi', \Psi') \Rightarrow (M, \Phi, \Psi)$: the latter requires that if $(S', m', c') \models (M', \Phi', \Psi')$ then $(S', \eta \circ m', c') \models (M, \Phi, \Psi)$, the former only requires that if $(S', m') \models (M', \Phi')$ then $(S', \eta \circ m') \models (M, \Phi)$.

Lemma 8.4. *If $\mathcal{A}' \Rightarrow \mathcal{A}$, then $\mathcal{A}'$ is a refinement of $\mathcal{A}$ and $\mathcal{A}' \leq \mathcal{A}$.*

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

9. Conclusions

We have shown that the coalgebraic view of objects provides a natural interpretation of behavioural subtyping, both for coalgebras (i.e. implementations of classes) and for coalgebraic specifications (i.e. specifications of classes). The 'model-theoretic' subtyping relation between coalgebras is based on the notion of substitutability, and uses the notion of bisimulation. The 'proof-theoretic' 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. In the coalgebraic setting we can formalize both and prove the correspondence between them.

Some definitions of behavioural subtyping between specifications in the literature involve the use of abstractions functions. For example, in [2,19] specifications make use of abstraction functions that map objects to the abstract values they represent. (The use of such abstraction functions dates back to [12]. Note that every abstraction function defines a bisimulation relation, namely objects are related iff they have the same abstract value.) In our setting abstraction functions are also used: the mappings $behaviour_m$ to the final coalgebras are the abstraction functions.

It is interesting to compare our work with [16], which also investigates soundness and completeness of different characterisations of behavioural subtyping, but in an algebraic rather than a coalgebraic setting.

The setting of [16] is more general than ours. E.g. we only have one hidden type X, whereas [16] allows an arbitrary number of invisible or hidden sorts. Also, our coalgebraic setting only allows unary methods, whereas the algebraic setting of [16] also allows binary methods in the sense of [5] (though their main result only applies to cases with only unary methods).

By restricting ourselves to the slightly less general coalgebraic setting, our characterisations and results become a lot simpler and more elegant than those in [16].

In the algebraic setting of [16] complicated definitions are needed to describe notions which are essentially for free in the coalgebraic setting. In particular, the existence of final coalgebras, which characterise observable behaviour, and the existence of the canonical mappings $behaviour_m$ to these final coalgebras make it possible to give the very simple model-theoretic characterisation of behavioural subtyping of Lemma 6.3.

This paper started out with the question whether the coalgebraic semantics of objects can explain subtyping. Another obvious question to ask is whether the coalgebraic semantics can explain inheritance. But whereas the coalgebraic semantics easily explains (behavioural) subtyping, we believe that it is does not easily explain inheritance. As mentioned in Section 2, a crucial 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, it is a limitation of the coalgebraic view of objects that it does not directly account for the self-references in method definitions, i.e. invocations of methods on "self". Modelling these would require similar tricks as used in [23]. As 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.

Acknowledgments

I would like to thank the anonymous referees for many valuable suggestions for improvements.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

[2] P. 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.

[3] P. America. Designing an Object-Oriented Languages with Behavioural Subtyping. In J.W. de Bakker e.a., editor, *Foundations of Object Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer, 1991.

[4] H. Bowman, C. Briscoe-Smith, J. Derrick, and B. Strulo. On Behavioural Subtyping in LOTOS. In H. Bowman and J. Derrick, editors, *FMOODS'97, Second IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, pages 335–351. Chapman and Hall, 1997.

[5] K.B. Bruce, L. Cardelli, G. Castagna, the Hopkins Objects Group (J. Eifrig, S. Smith, V. Trifonov), G.T. Leavens, and B.C. Pierce. On Binary Methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

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

[7] Y. Chen and B. H.C. Cheng. A semantic foundation for specification matching. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 5, pages 91–109. Cambridge University Press, 2000.

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

[9] K.K. Dhara and G.T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE, 1996.

[10] C.A. Gunter and J.C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* The MIT Press, 1994.

[11] U. Hensel, M. Huisman, B. Jacobs, and H. 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, 1998.

[12] C.A.R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1:271–281, 1972.

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

[14] B. 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.

[15] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, June 1997.

[16] G.T. Leavens and D. Pigozzi. A complete algebraic characterization of behavioral subtyping. *Acta Informatica*, 36:617–663, 2000.

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

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

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

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

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

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

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

[24] E. 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.

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

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

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

[28] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997.

[29] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. *ACM SIGPLAN*, 21(11):38–45, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

Communicated by (The editor will be set by the publisher).

...