Math. Struct. in Comp. Science (2007), *vol.* 17, *pp.* 647–673. © 2007 Cambridge University Press
doi:10.1017/S0960129507006214 Printed in the United Kingdom

Coalgebraic description of generalised binary methods[†]

FURIO HONSELL[‡], MARINA LENISA[‡] and REKHA REDAMALLA[‡§]

[‡] *Dipartimento di Matematica e Informatica, Università di Udine, Via delle Scienze 206, 33100 Udine, Italy*
[§] *B.M. Birla Science Centre, Adarsh Nagar, Hyderabad, 500 063 A.P., India*

Received 16 October 2005; revised 3 November 2006

We extend the coalgebraic account of specification and refinement of objects and classes in object-oriented programming given by Reichel and Jacobs to (*generalised*) *binary methods*. These are methods that take more than one parameter of a class type. Class types include products, sums and powerset type constructors. To allow for class *constructors*, we model classes as *bialgebras*. We study and compare two solutions for modelling generalised binary methods, which use purely covariant functors.
In the first solution, which applies when we already have a class implementation, we reduce the behaviour of a generalised binary method to that of a bunch of unary methods. These are obtained by *freezing* the types of the extra class parameters to constant types. If all parameter types are *finitary*, the *bisimilarity equivalence* induced on objects by this model yields the *greatest congruence* with respect to method application.
In the second solution, we treat binary methods as *graphs* instead of functions, thus turning contravariant occurrences in the functor into covariant ones.
We show the existence of *final coalgebras* in both cases.

1. Introduction

The papers Reichel (1995), Jacobs (1996) and Jacobs (1997) describe a categorical framework based on the notion of *coalgebra* to provide semantics to *objects* and *classes*. The idea underpinning this approach to class specification and refinement is that coalgebras, which are duals to algebras, allow us to focus on the behaviour of objects while abstracting from the concrete representation of the state of the objects. Algebras have 'constructors' (operations packaging information into the underlying carrier set); coalgebras have 'destructors' or 'observers' (operations extracting information from the carrier set), which allow us to detect certain behaviours.

Classes in object-oriented languages are given in terms of *attributes* (*fields*) and *methods*. The values of attributes determine the states of the *class*, that is, the *objects*; methods act on objects.

In the coalgebraic approach of Reichel (1995), Jacobs (1996) and Jacobs (1997), a class is modelled as an F-coalgebra $(A, f : A \to F(A))$ for a suitable functor F. The carrier

[†] Work partially supported by the Project ART(PRIN 2005015824), and by the UE Project TYPES(IST-510996).

A represents the space of objects, and the coalgebra operation f represents the *public* methods of the class, that is, the methods that are accessible from outside the class. Methods are viewed as functions acting on objects. The coalgebraic model, that is, the unique morphism into the *final F-coalgebra*, induces precisely the *behavioural equivalence* on objects, in which two objects are equated if, for each public method, the application of the method to the two objects, for any list of parameters, produces equivalent results. A benefit of this coalgebraic approach is a *coinduction principle* for establishing behavioural equivalence.

So that we can also treat class *constructors*, in this paper, we introduce an algebra part in our model, thus modelling classes as *bialgebras*: Rothe *et al.* (2001) and Cirstea (2000) use a similar approach.

Following Jacobs (1996), we distinguish between class *specifications* and class *implementations* (or simply classes). A class specification is like an abstract class, in which only the signatures of constructors and (public) methods are given, without any actual code. Assertions enforce behavioural constraints on constructors and methods. On the other hand, the implementation of constructors and methods is given in a class implementation. In the bialgebraic approach, a class specification induces a pair of functors, determined by the signatures of constructors and methods, respectively. A class implementation is any bialgebra satisfying the assertions.

Binary methods, that is, methods with more than one class parameter, do not appear to be susceptible to this simple co(bi)algebraic approach[†]. The extra class parameters produce contravariant occurrences in the functor modelling class methods, and hence cannot be handled by a straightforward application of the standard coalgebraic methodology.

In this paper, we extend the Reichel–Jacobs coalgebraic approach to *generalised binary methods*, that is, methods whose type parameters are built over constants and class variables, using products, sums and the powerset type constructor. This is a fairly large collection of methods, including all the methods that are commonly used in object-oriented programming.

Our interest is focused on equivalences for objects that are 'well behaved', that is, are *congruences* with respect to method application. Hence they induce a minimal implementation of the given class specification by considering the quotient of the class through the equivalence.

The principal contribution of this paper is to show that canonical models can also be built for classes with generalised binary methods using purely covariant tools, at least in the case of *finitary binary methods*, that is, methods where type constructors range over *finite* product, sum and powerset. We propose two different solutions. Our first solution applies to the case where we already have a class implementation. It is based on the observation that the behaviour of a generalised binary method can be captured by a bunch of unary methods obtained after a suitable manipulation of the original method. The key step is to *'freeze'* in turn each of the types of the class parameters in the states of the class implementation given at the outset, that is, to view them as constant types.

[†] By a standard abuse of terminology, *binary methods* refer to methods with $n \geqslant 2$ class parameters.

Our second solution is based on a set-theoretic understanding of functions in which binary methods in a class specification can be viewed as *graphs* instead of functions. Thus, contravariant function spaces in the functor are rendered as covariant sets of relations.

We prove that the bisimilarity equivalence induced by the *'freezing approach'* amounts to the *greatest congruence* with respect to method application on the given class, at least for finitary binary methods. As a by-product, we provide a (coalgebraic) coinduction principle for reasoning about such greatest congruence.

As far as the graph model is concerned, the bisimilarity equivalence is not a congruence, in general, even for finitary binary methods. The graph approach, however, yields an equivalence that always includes the freezing equivalence. Therefore, and somewhat remarkably, a necessary and sufficient condition for the graph bisimilarity to be a congruence is that the graph and freezing equivalence coincide. As a consequence, when this is the case, we obtain a spectrum of coinduction principles for reasoning about the greatest congruence.

In this paper, we present various non-trivial examples of class specifications and implementations, where the graph bisimulation is a congruence. *Inter alia*, we consider a class, which generalises the one given in Honsell and Lenisa (1995), for representing terms of the λ-calculus, together with the lazy notion of reduction strategy of Abramksy (1993).

It is natural to ask what the cause is when the freezing and graph approaches do not coincide. We do not have a fully satisfactory answer, but we feel that it may be a symptom of the fact that something is underspecified in the public interface of the class; a similar comment can be made when no maximal congruence exists in the realm of infinitary binary methods.

The interest provided by the graph approach goes beyond coalgebraic semantics, since it suggests a new way for solving the well-known problem of typing binary methods when subclasses are viewed as subtypes, see, for example, Bruce *et al.* (1995). In this paper, we will hint at an intriguing solution based on a new typing system, which utilises a *relation type* for typing (binary) methods when these are viewed as *graphs*, rather than functions. Our solution takes classes seriously, that is, it does not utilise *multiple dispatching*. In concrete object-oriented languages, such as Java, the problem of typing binary methods when subclasses are involved is normally solved by implementing method calls with multiple dispatching, that is, by choosing which method code to activate according to the types of all class parameters, and *not* just the object type.

We emphasise the fact that, in the case of finitary binary methods, we do provide satisfactory canonical models, which can be conveniently understood in terms of final coalgebras, for suitable derived functors. Therefore, a set-theoretical comment on the existence of final coalgebras is in order. The existence of a final coalgebra is important, since it provides a canonical implementation of a given specification. Since we do not want to introduce unnecessary restrictions resulting from the choice of our ambient category, we work in the category of sets and proper classes (Aczel *et al.* 1989; Forti and Honsell 1983), where all covariant functors can be shown to have a final coalgebra, see Cancila *et al.* (2006). Thus, throughout the paper, we fix $\mathscr{C}$ to be a category whose objects are the sets and classes of a (wellfounded or non-wellfounded) set-theoretic universe, and whose morphisms are the functions between them. For basic definitions and results on coalgebras, see Appendix A and Jacobs and Rutten (1996).

Comparison with related work

In this section we describe a number of other approaches in the literature that address the problem of extending the coalgebraic model to binary methods.

In Jacobs (1996a), binary methods are allowed only when they are definable in terms of the unary methods of the class. This means, in particular, that binary methods do not contribute to the definition of the observational equivalence. The same observation applies to the approach in Hennicker and Kurz (1999), where binary methods are defined as algebraic extensions, thus only the case where the resulting type is the class itself is considered. Our approach is more general, since we do not require any *a priori* connection between binary and unary methods in the class.

Binary methods in full generality have been extensively studied in Tews (2002), where various classes of mixed covariant–contravariant functors have been considered, and a theory of coalgebras and bisimulations has been studied for such functors. Tews' approach is very interesting, but quite different from our approach, since from the outset we use purely covariant tools. Nonetheless, there are interesting connections between the two approaches. We include the powerset type constructor, which Tews does not consider, but, apart from this, our generalised binary methods should correspond, essentially, to Tews' class of *extended polynomial functors*[†]. Similarly, our finitary generalised methods should correspond to Tews' *extended cartesian functors*. Tews' bisimulations amount to congruence relations, and do not give rise, in general, to a coinduction principle, since the union of all congruences fails to be a congruence. However, for extended cartesian functors, the union of all congruences is again a congruence (Poll and Zwanenburg 2001; Tews 2002). Our notion of freezing bisimulation is weaker, in the sense that any bisimulation in the sense of Tews is a freezing bisimulation, but the converse is not true. Moreover, our notion of bisimulation, being monotone, always gives rise to a coinduction principle. However, the greatest freezing bisimulation fails, in general, to be a congruence. It is a congruence (the greatest one, in fact) exactly in the case of finitary methods. Thus, in this case, our notion of bisimilarity equivalence coincides with the one by Tews. To conclude this comparison, we make the following two remarks. First, our approach is more elementary. By modelling binary methods using purely covariant functors, we can reuse the standard coalgebraic machinery. On the other hand, Tews develops a theory of coalgebras and bisimulations for mixed functors, which is also interesting in itself. Second, in our setting, final coalgebras always exist, so we have canonical models, while mixed functors in Tews (2002) do not admit final coalgebras.

Yet another approach in the theory of coalgebraic semantics consists of avoiding binary methods altogether by considering a whole system of objects in place of single objects,

[†] Tews' extended polynomial functors appear to cover a wider collection of methods, but we conjecture that the specific cases that appear not to be covered by our approach should be recoverable using manipulations similar to those introduced in Section 6. However, more work needs to be done on this.

for example, by considering a class representing a list of points in place of a class for a single point – see van den Berg *et al.* (1999) and Huisman (2001). This approach is quite different from ours.

Finally, there is an interesting connection between our approach and the approach of *hidden algebras* (Goguen and Malcolm 2000; Rosu 2000), where the focus is on behavioural congruences, rather than on bisimulations. Our freezing model has the positive features of both approaches: the behavioural equivalence that we define is *both* a greatest bisimulation and the greatest congruence with respect to method application.

Synopsis

In Section 2, the notions of class specification and class implementation are presented together with examples. In Section 3, we present the bialgebraic description of objects and classes in the unary case. In Section 4, which contains the main content of the paper, we present, discuss and compare our two bialgebraic accounts of generalised binary methods. In Section 5, we sketch a new solution for typing binary methods, which is inspired by our graph coalgebraic semantics. Final remarks and directions for future work are presented in Section 6. Examples appear throughout the paper.

2. Class specifications and class implementations

Following Jacobs (1996), we distinguish between *class specifications* and *class implementations*. Informally, a class specification consists of constructor and method declarations, and assertions, which regulate the behaviour of objects. A class implementation consists of fields, constructor and method codes.

Before introducing the formal definitions of class specification and implementation, we need to introduce the grammar for the types of fields, and constructor and method parameters. We distinguish between *finitary generalised types*, which corresponds to polynomial types extended with finite powerset, and (*infinitary*) *generalised types*, which extend the previous class of types with possibly infinitary sums, products and powerset constructors.

Definition 2.1 (Generalised parameter types).

— *Finitary generalised types* range over the following grammar:

$$(\mathscr{T}_{\mathscr{F}} \ni)\ T ::= X \mid K \mid T \times T \mid T + T \mid \mathscr{P}_f(T),$$

where $X \in \mathit{TVar}$ is a variable for class types, and K is any constant type. Constant types include Unit, denoted by 1, Boolean, denoted by $\mathbb{B}$, and Integer, denoted by $\mathbb{N}$.

— (*Infinitary*) *Generalised types* range over the following grammar:

$$(\mathscr{T} \ni)\ T ::= X \mid K \mid \prod_{i\in I} T_i \mid \sum_{i\in I} T_i \mid \mathscr{P}(T),$$

where I is a possibly infinite set of indices.

Note that the product type $\prod_{i\in I} T_i$ in Definition 2.1 above subsumes the function space $K \rightarrow T$. That is, we allow functional parameters, where variable types can only appear in *strictly positive* positions.

For simplicity, in this paper we will consider only one class in isolation; there would be no additional conceptual difficulty in dealing with the general case.

Definition 2.2. A *class specification* S is a structure consisting of:

— a finite set of *constructor declarations*

$$c : T_1 \times \ldots \times T_p \rightarrow X$$

— a finite set of *method declarations*

$$m : X \times T_1 \times \ldots \times T_q \rightarrow T_0$$

— a finite set of *assertions* regulating the behaviour of the objects belonging to the class.

The language for assertions is any formal language with constant symbols, function symbols for denoting constructors and methods, and relation symbols for denoting (extensions of) behavioural equivalences at all types. Typical assertions are equations, see, for example, Rothe *et al.* (2001) for more details.

A class (implementation) consists of attributes (fields), constructors and methods. Attributes and methods of a class can be private or public. For simplicity, we assume all attributes to be private and all methods to be public. We do not use a specific programming language to define classes since we are working at a semantic level; any programming language would do. From this perspective: a class is represented by a set (of objects) X; a field f of type T is represented by a function $f : X \rightarrow T$; the code corresponding to a *constructor* declaration $c : \prod_{j=1}^{p} T_j \rightarrow X$ is given by a set-theoretic function $\beta : \prod_{j=1}^{p} T_j \rightarrow X$; and the code corresponding to a *method* declaration $m : X \times \prod_{j=1}^{q} T_j \rightarrow T_0$ is given by a set-theoretic function $\alpha : X \times \prod_{j}^{q} T_j \rightarrow T_0$.

Summarising, we have the following definition.

Definition 2.3. A class $C = \langle X, \{f_i : X \rightarrow T_i\}_{i=1}^{n}, \{c_i : \prod_{j=1}^{p_i} T_{ij} \rightarrow X\}_{i=1}^{h}, \{m_i : X \times \prod_{j=1}^{q_i} T_{ij} \rightarrow T_{i0}\}_{i=1}^{k}\rangle$ is defined by:

— a set of objects/states X;
— functions $f_i : X \rightarrow T_i$ representing fields;
— functions $\beta_i : \prod_{j=1}^{p_i} T_{ij} \rightarrow X$, implementing constructors c_i;
— functions $\alpha_i : X \times \prod_{j=1}^{q_i} T_{ij} \rightarrow T_{i0}$ implementing methods m_i.

Definition 2.4. A class C *implements* a specification S if constructor and method declarations correspond and their implementations satisfy the assertions in S.

Our classes feature a quite general notion of binary methods, which we call *generalised binary methods*, where parameter types are (infinitary) generalised types as defined in Definition 2.1 above. Throughout the paper, we fix the following terminology.

Definition 2.5 (Binary methods). Let $m : X \times T_1 \times \ldots \times T_q \to T_0$ be a method, with $T_0 \in \mathscr{T}$. Then:

— m is *(generalised) binary* if $T_1, \ldots, T_q \in \mathscr{T}$;
— m is *finitary binary* if $T_1, \ldots, T_q \in \mathscr{T}_{\mathscr{F}}$;
— m is *simple binary* if $T_1, \ldots, T_q$ are either constants or the class type X;
— m is *unary* if $T_1, \ldots, T_q$ are all constant types.

Note that our simple binary methods correspond to ordinary binary methods. Throughout the paper, *generalised binary methods* will often simply be called *binary methods.*

2.1. *Examples*

In Table 1, we present four examples of class specifications, together with some examples of class implementations.

The class specification *Stack(A)*, which is taken from Jacobs (1997), specifies the recursive data type of stacks with elements in A. As is customary in object-oriented languages such as Java, we use the same name *new* for all constructors, which differ in the number and type of their parameters. The symbol $\approx$ in the assertions denotes equality on objects of class type. All methods in this example are unary.

Our second example of a class specification, *Register*, features the simple binary method *eq* for comparing the contents of two registers.

The class specification λ-calculus in Table 1 represents the recursive data type of λ-terms under lazy evaluation – see Abramksy (1993). There are three constructors, corresponding to the syntax of λ-terms (namely, variable, application and abstraction):

$$M ::= z \mid MM \mid \lambda z.M$$

for z ranging over an infinite set of variables. The method *isval* tests whether a λ-term converges, that is, it reduces to an abstraction using the leftmost strategy. Lazy convergence, denoted by $\Downarrow$, is defined by

$$\frac{}{\lambda z.M \Downarrow \lambda z.M} \qquad \frac{M[N_1/z]N_2 \ldots N_k \Downarrow P}{(\lambda z.M)N_1 \ldots N_k \Downarrow P}\,.$$

The first assertion in the λ-calculus class specification expresses the fact that method *app* is a simple binary method that behaves like the constructor for application. The second assertion is used to axiomatise the notion of convergence.

As we will see, the λ-calculus class specification is intended to provide the standard notion of *lazy* observational equivalence when restricted to *closed* λ-terms.

Note that the code in the class implementation of Λ is not effective, since it uses the predicate $\Downarrow$ as a primitive, which is only semi-decidable. It could not be otherwise, but it makes the point nevertheless.

A more sophisticated example of a generalised binary method is given by the cell-component of a cellular automaton. In the general case, where neighbourhoods can vary at each generation, they can be best specified using sets of cells.

Table 1. *Examples of class specifications and classes.*

```
class spec : $Stack(A)$
    constructors :
        new : $1 \to X$
        new : $X \times A \to X$
    methods :
        push : $X \times A \to X$
        pop : $X \to X$
        top : $X \to 1 + A$
    assertions :
        s.push($a$).top $= a$
        s.push($a$).pop $\approx$ s
        s.top $= * \Rightarrow$ s.pop $\approx$ s
        new.top $= *$
        new(s, $a$) $\approx$ s.push($a$)
end class spec
```

```
class T
    attributes :
        $first : 1 + A$
        $next : T$
    constructors :
        ....
    methods :
        s.push($a$) = s'
                where s'.$first = a$ and
                      s'.$next$ = s
        s.pop = if s.$first = *$
                      then s
                      else s.$next$
        s.top = s.first
end class
```

```
class spec : $Register$
    constructors :
        new : $1 \to X$
    methods :
        set : $X \times N \to X$
        get : $X \to N$
        eq  : $X \times X \to \mathbb{B}$
    assertions :
        r.set($n$).get $= n$
        $r_1$.get $= r_2$.get $\Leftrightarrow$
            $r_1$.eq($r_2$) $= true$
        new.get = 0
end class spec
```

```
class R
    attributes :
        $val : int$
        constructors :
        ....
    methods :
        r.get = r.$val$
        r.set($n$) = r'
                where r'.$val = n$
        $r_1$.eq($r_2$) = if ($r_1$.get = $r_2$.get)
                        then $true$
                        else $false$
end class
```

```
class spec : $\lambda$-calculus
    constructors :
        new : $X \times X \to X$
        new : $Var \to X$
        new : $Var \times X \to X$
    methods :
        isval  : $X \to \mathbb{B}$
        app  : $X \times X \to X$
    assertions :
        new($M, N$) $\approx M$.app($N$)
        $M$.isval $= true \Leftrightarrow$
                $\exists z N.\ M \approx new(z, N)$
end class spec
```

```
class $\Lambda$
    attributes :
        $term$ : $\lambda$-string
    constructors :
        ....
    methods :
        $M$.isval = if $M.term \Downarrow$
                      then $true$
                      else $false$

        $M$.app($N$) = $P$
            where $P.term = (M.term)(N.term)$
end class
```

```
class spec : $Cell$
    constructors :
        new : $N \times N \times State \to X$
    methods :
        $get_x$ : $X \to N$
        $get_y$ : $X \to N$
        $set_{state}$  : $X \times State \to X$
        $set_{neighborhood}$  : $X \times \mathscr{P}(X) \to X$
        $get_{neighborhood}$  : $X \to \mathscr{P}(X)$
    assertions :
        ...
end class spec
```

```
class C
    attributes :
        $val_x$ : $int$
        $val_y$ : $int$
        neighborhood : $\mathscr{P}(X)$
        state : $State$
    constructors :
        ...
    methods :
        ...

end class
```

3. Bialgebraic description of objects and classes: unary case

In this section we illustrate the bialgebraic description of class specifications and class implementations in the case of unary methods. We extend the coalgebraic description given in Reichel (1995) and Jacobs (1996) with an algebra part modelling class constructors; a similar approach appears in Rothe *et al.* (2001) and Cirstea (2000).

We will begin by explaining how a class specification induces a pair of functors.

Each constructor declaration $c : \prod_{j=1}^{p} T_j \to X$ in a class specification determines a functor $L : \mathscr{C} \to \mathscr{C}$ defined by

$$LX = \prod_{j=1}^{p} T_j \,. \tag{1}$$

In this way, $c : LX \to X$ will induce an L-algebra structure on X.

The treatment of methods is more indirect. By currying the type in a method declaration $m : X \times \prod_{j=1}^{q} T_j \to T_0$, we get the type $X \to [\prod_{j=1}^{q} T_j \to T_0]$. Thus, we define the functor $H : \mathscr{C} \to \mathscr{C}$ induced by m as follows:

$$HX \triangleq \prod_{j=1}^{q} T_j \to T_0 \,. \tag{2}$$

Thus, m will induce an H-coalgebra structure on X.

Note that the functor H is a well-defined covariant functor only if the method m is unary. Binary methods, such as the method *eq* in the class specification *Register*, or *app* in Λ, produce contravariant occurrences of X in the corresponding functor. For example, the functor induced by *eq* would be $H_{eq}X \triangleq X \to \mathbb{B}$. The coalgebraic approach does not apply directly to the case of binary methods; we discuss how to overcome this problem in Section 4. Here we focus on the unary case. In this case, we can immediately associate a pair of functors to a class specification as follows.

Definition 3.1. Let S be a class specification with constructor declarations

$$c_i : \prod_{j=1}^{p_i} T_{ij} \to X, \quad i = 1, \ldots, h$$

and method declarations

$$m_i : X \times \prod_{j=1}^{q_i} T_{ij} \to T_{i0}, \quad i = 1, \ldots, k,$$

where all methods are unary. The constructor declarations in S induce the functor $L : \mathscr{C} \to \mathscr{C}$ defined by

$$L \triangleq \coprod_{i=1}^{h} L_i \,,$$

where $L_i : \mathscr{C} \to \mathscr{C}$ is the functor determined by the constructor declaration c_i defined as in (1). The method declarations in S induce the functor $H : \mathscr{C} \to \mathscr{C}$ defined by

$$H \triangleq \prod_{i=1}^{k} H_i ,$$

where $H_i : \mathscr{C} \to \mathscr{C}$ is the functor determined by the method declaration m_i, defined as in (2).

A class implementation induces a bialgebra for the functors determined by its constructor and method declarations as follows.

Definition 3.2. A *class* $C = \langle X, \{f_i : X \to T_i\}_{i=1}^{n}, \{c_i : \prod_{j=1}^{p_i} T_{ij} \to X\}_{i=1}^{h}, \{m_i : X \times \prod_{j=1}^{q_i} T_{ij} \to T_{i0}\}_{i=1}^{k}\rangle$ induces a bialgebra (X, β, α) (where α and β are defined below) for the functor pair $\langle L, H\rangle$ determined by the declarations of constructors and methods as in Definition 3.1 above:

— the algebra map $\beta : LX \to X$ is defined by $\beta \triangleq [\beta_i]_{i=1}^{h}$, where $\beta_i : L_iX \to X$ is the function implementing the constructor c_i, and $[\ ]$ denotes the standard case function;

— the coalgebra map $\alpha : X \to HX$ is defined by $\alpha \triangleq \langle \alpha_i \rangle_{i=1}^{k}$, where $\alpha_i : X \to H_iX$ is the function implementing the method m_i, and $\langle\ \rangle$ denotes the standard pairing functor.

Thus, the class implementations corresponding to a given specification can be viewed as bialgebras as follows.

Definition 3.3. Let S be a class specification inducing a functor pair $\langle L, H\rangle$. A *class* implementing S is an $\langle L, H\rangle$-bialgebra satisfying the assertions in S.

Note that in Definition 3.3 the classes are taken up to fields because these are private.

3.1. *Coalgebraic behavioural equivalence*

In this section we characterise the behavioural equivalence on objects induced by the coalgebraic part of a class implementation.

A preliminary step in discussing behavioural equivalences and congruences consists of extending the behavioural equivalence on the set of objects X of a class to the whole structure of (sets interpreting) types over X. Such an extension is defined through the following definition, which extends the notion of relational lifting given in Hermida and Jacobs (1998) to the powerset. We abuse notation in the definition below and do not distinguish between types and their usual set-theoretic interpretation.

Definition 3.4 (Relational lifting). Let R^X be a relation on X and $T \in \mathscr{T}$ be such that $Var(T) \subseteq \{X\}$. We define the extension $R^T \subseteq T \times T$ by induction on T as follows:

— If $T = K$, then $R^T = Id_{K \times K}$.

— If $T = \prod_{i \in I} T_i$, then $R^T = \{(\vec{a}, \vec{a}') \mid \forall i \in I. a_i R^{T_i} a_i'\}$.

— If $T = \sum_{i \in I} T_i$, then $R^T = \{((i, a), (i, a')) \mid i \in I \wedge aR^{T_i}a'\}$.

— If $T = \mathscr{P}(T_1)$, then $R^T = \{(u, u') \mid \forall a \in u\ \exists a' \in u'.\ aR^T a' \ \wedge\ \forall a' \in u'\ \exists a \in u.\ aR^T a'\}$.

In the following we will often make an abuse of notation by just using R to denote the lifted relation R^T when its type is clear from the context.

A strong motivation for the coalgebraic account of objects is that the quotient by the bisimilarity equivalence of a given class, when viewed as a coalgebra, can yield, in many cases, such as that of unary methods, a new model of the same class. For this to hold, we at least need that the bisimilarity equivalence is a *congruence with respect to method application.*

Definition 3.5 (Congruence). Let $\approx^X$ be an *equivalence* on the set of objects X of a class C, and let $m : X \times T_1 \times \ldots \times T_q \to T_0$ be a method in C implemented by α. Then, $\approx^X$ is a *congruence* with respect to m if

$$x \approx^X x' \;\wedge\; a_1 \approx^{T_1} a_1' \wedge \ldots \wedge a_q \approx^{T_q} a_q' \implies \alpha(x)(\vec{a}) \approx^{T_0} \alpha(x')(\vec{a}'),$$

where $\approx^{T_i}$ denotes the extension of $\approx^X$ to the type T_i according to the definition above.

Finally, having defined the coalgebraic account of a class in this way, we have that the coalgebraic equivalence in the unary case equates objects with the same behaviour under the application of methods.

Proposition 3.1 (Coalgebraic bisimilarity equivalence). Let S be a class specification and $(X, [\beta_i]_{i=1}^h, \langle\alpha_i\rangle_{i=1}^k)$ be an $\langle L, H\rangle$-bialgebra implementing S. Then:

(i) An H-bisimulation on $(X, \langle\alpha_i\rangle_i)$ is a relation $R \subseteq X \times X$ satisfying

$$x\, R\, x' \implies \forall \alpha_i.\ \forall \vec{a}.\ \alpha_i(x)(\vec{a})\, R\, \alpha_i(x')(\vec{a}).$$

(ii) The *coalgebraic bisimilarity equivalence* $\approx_H$, that is, the greatest H-bisimulation on $(X, \langle\alpha_i\rangle_i)$, can be characterised by

$$x \approx_H x' \iff \forall \alpha_i.\ \forall \vec{a}.\ \alpha_i(x)(\vec{a}) \approx_H \alpha_i(x')(\vec{a}).$$

In particular, the following *coinduction principle* holds:

$$\frac{R \text{ is an } H\text{-bisimulation on } (X, \langle\alpha_i\rangle_i) \qquad x\, R\, x'}{x \approx_H x'}\,.$$

Proof. The proposition follows from the definition of coalgebraic bisimulation (see Definition A.3 of Appendix A) and Theorem A.1 of Appendix A. □

Thus we also have the following theorem.

Theorem 3.1. $\approx_H$ is the *greatest congruence* with respect to methods.

Proof. Since all methods are unary, by the definition of relational lifting on constant types, we immediately have that $\approx_H$ is a congruence with respect to methods. The fact that $\approx_H$ is the greatest congruence follows by observing that any congruence with respect to methods is an H-bisimulation. □

As we mentioned earlier, a major benefit of the coalgebraic approach to classes is that bisimilarity equivalences naturally yield, via quotienting, classes of the same signature as the original class, and, furthermore, preserve various kinds of assertions. This can also be expressed by saying that a suitable subcoalgebra of the final coalgebra still provides an implementation of the specification, in fact, it is the canonical one.

It goes without saying that in dealing with bialgebras we would like to preserve the above important feature of the purely coalgebraic approach. To this end, we require that final bialgebras exist, and, furthermore, that the behavioural equivalence is also a congruence with respect to constructors. Turi and Plotkin (1997) and Corradini *et al.* (2002) investigated general conditions on categories of bialgebras to ensure the above properties.

These results can be extended/adapted to include the collection of functors modelling generalised binary methods considered later in this paper, at least for assertions of a simple equational shape. In this case, if there is a 'tight connection' between the algebraic and the coalgebraic structure for a given bialgebra satisfying the assertions, then the corresponding functors admit final bialgebras still satisfying the assertions, and the behavioural equivalence is a congruence with respect to constructors. We will not elaborate further on this issue here, but instead focus on the coalgebraic part, which is the most problematic one.

4. Coalgebraic description of generalised binary methods

In this section we show how to extend the bialgebraic approach to binary methods. Our first proposal (Section 4.1) applies when a concrete bialgebra (that is, class implementation) is already available. It is based on the observation that the behaviour of a binary method can be simulated by a bunch of unary methods, each determined by 'freezing' all but one of the occurrences of X in the parameter types and object type. The bunch is then obtained after suitable manipulations of the original method. 'Freezing' an occurrence of X means that X is replaced by the carrier, that is, the set of states, of the given class. This allows us to define a covariant *freezing* functor F, where the contravariant occurrences in the original generalised binary method are replaced by a constant type, namely the carrier of the given bialgebra. The freezing procedure is carried out in such a way that, at least in the case of finitary binary methods, the bisimilarity equivalence induced by F turns out to be the greatest congruence with respect to the original binary methods.

In Section 4.2 we present an alternative solution to the freezing functor, which we call a *graph functor*. Here we turn contravariant occurrences in the type of parameters of a generalised binary method m into covariant ones simply by interpreting m as a *graph* instead of a function. To this end, we introduce a new functor G (*graph functor*), where the function space is substituted by the corresponding space of *graph relations*.

The advantage of this latter solution compared with the previous one is that this approach applies directly to specifications. The drawback is that the graph bisimilarity equivalence is not a congruence with respect to method application in general. One may ask why this is the case, but, as yet, there is no general explanation. In many cases it means that the specification is *under-determined*, or, alternatively, there exist class implementations without a common refinement. However, there are many interesting situations where the graph equivalence is a congruence with respect to methods. In these cases a rich spectrum of conceptually independent coinduction principles is available.

We discuss this issue in Section 4.3, together with a comparison of freezing and graph bisimilarity equivalences.

Throughout this section, we assume that S is a class specification with constructors $c_i : \prod_{j=1}^{p_i} T_{ij} \to X$, $i = 1, \ldots, h$, and methods $m_i : X \times \prod_{j=1}^{q_i} T_{ij} \to T_{i0}$, $i = 1, \ldots, k$, and that L is the functor induced by the constructors.

4.1. *The freezing functor*

Given a class implementation C, with carrier $\bar{X}$, we transform C into a class C^* containing only unary methods. The procedure is carried out in two steps:

1 We first reduce each binary method m to a bunch of *simple* binary methods with the same observable behaviour as m. To this end, we process the parameters with complex types as follows:

(a) For each parameter with type $\sum_{i \in I} T_i$ in m, we consider methods $\{m_i\}_{i \in I}$, where the method m_i has a parameter of type T_i.

(b) Each parameter with type $\prod_{i \in I} T_i$ can be viewed as the product of $|I|$ parameters.

(c) The treatment of parameters with type $\mathscr{P}(T')$ is more subtle. If m has a parameter of type $\mathscr{P}(T')$, that is, $m : X \times \ldots \times \mathscr{P}(T') \times \ldots \to T_0$, the behaviour of m can be simulated by a pair of methods $m_1 : X \times \ldots \to T_0$, where the parameter of type $\mathscr{P}(T')$ disappears, and $m_2 : X \times \ldots \times \mathscr{P}(T'[\bar{X}|X]) \times T' \times \ldots \to T_0$, where we 'freeze' the powerset parameter as a constant type and add an extra parameter T'. Intuitively, the method m_1 accounts for the behaviour of m when the powerset parameter is the empty set, while the method m_2 accounts for the case of non-empty sets (the precise definition of m_1 and m_2 will be given in Definition 4.1).

By applying the above transformations to a binary method, we get a (possibly infinite) set of simple binary methods $m : X \times \prod_{j \in J} T_j \to T_0$, where J is a possibly infinite set of indices (if all sum and product types in the original method are finite, the number of simple binary methods together with their parameters are finite).

2 In the second step, we reduce each simple binary method to a bunch of *unary* methods. Let $m : X \times \prod_{j \in J} T_j \to T_0$ be a simple binary method implemented by the function α. In order to capture the observable behaviour of the method m, we need to consider a bunch of unary methods m_l, one for each class parameter, where m_l describes the behaviour of an object when it is used as the l^{th} class parameter.

Formally, steps 1 and 2 are defined in terms of the following method transformation.

Definition 4.1 (Method/class transformation).

(i) Let τ_F be the one-step *method transformation* function that takes a method $m : X \times \prod_{j=1}^{q} T_j \to T_0$, implemented by α, and produces a set of methods, defined by induction on types of m as follows:

— Case: m is *simple binary*.
Let I be the set of indices corresponding to the class parameters of type X in m. We define

$$\tau_F(m) = \{m_l \mid l \in I\}$$

where

$$m_l : X \times \prod_{j=1}^{q}(T_j[\bar{X}/X]) \to T_0$$

is defined by

$$\alpha_l(x)(a_1, \dots, a_q) \triangleq \alpha(a_l)(a_1, \dots, a_{l-1}, x, a_{l+1}, \dots, a_q)\,.$$

— Case: m is non-simple generalised binary.
Assume the leftmost non-constant parameter different from X is T_i. Then:

- Case: $T_i = \sum_{j=1}^{q_i} T_{ij}$.
  We define

  $$\tau_F(m) = \{m_{ij} \mid j = 1, \dots, q_i\}$$

  where

  $$m_{ij} : X \times \dots \times T_{i-1} \times T_{ij} \times T_{i+1} \times \dots \times T_q \to T_0$$

  is defined by

  $$\alpha_{ij}(x)(a_1, \dots, a_{i-1}, a_{ij}, a_{i+1}, \dots, a_q) \triangleq \alpha(x)(a_1, \dots, a_{i-1}, in_j(a_{ij}), \dots, a_q)\,,$$

  where

  $$in_j : T_{ij} \to \sum_{j=1}^{q_i} T_{ij}$$

  is the canonical injection.
- Case: $T_i = \prod_{j=1}^{q_i} T_{ij}$.
  We define

  $$\tau_F(m) = \{m'\}\,,$$

  where

  $$m' : X \times \dots \times T_{i-1} \times T_{i1} \times \dots \times T_{iq_i} \times \dots T_q \to T_0$$

  is defined by

  $$\alpha'(x)(a_1, \dots, a_{i-1}, a_{i1}, \dots, a_{iq_i}, a_{i+1}, \dots, a_q) \triangleq \alpha(x)(a_1, \dots, a_{i-1}, \vec{a}_i, a_{i+1}, \dots a_q)\,.$$
- Case: $T_i = \mathscr{P}(T_i')$.
  We define

  $$\tau_F(m) = \{m_1, m_2\}\,,$$

  where

  $$m_1 : X \times \dots \times T_{i-1} \times T_{i+1} \times \dots \times T_q \to T_0$$

  is defined by

  $$\alpha_1(x)(a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_q) \triangleq \alpha(x)(a_1, \dots, a_{i-1}, \phi, a_{i+1}, \dots a_q)$$

  and

  $$m_2 : X \times \dots \times T_{i-1} \times \mathscr{P}(T_i'[\bar{X}|X]) \times X \times T_{i+1} \times \dots \times T_q \to T_0$$

is defined by

$$\alpha_2(x)(a_1,\ldots,a_{i-1},u,y,a_{i+1},\ldots,a_q) \triangleq \alpha(x)(a_1\ldots a_{i-1},u\cup\{y\},a_{i+1},\ldots,a_q)\,.$$

(ii) Let τ_F^* be the transformation function that takes a method and iteratively applies τ_F defined by

$$\tau_F^*(m) = \begin{cases} \tau(m) & \text{if } m \text{ is simple binary} \\ \bigcup\{\tau_F^*(m_i) | m_i \in \tau_F(m)\} & \text{otherwise.} \end{cases}$$

(iii) Finally, for a class C, let C^* be the class with the same carrier, fields and constructors as C, and with unary methods $\bigcup\{\tau_F^*(m) | m \text{ is a method of } C\}$.

In the following example we apply the above method transformation to a given class.

Example 4.1. Let R' be a class of registers with carrier $\mathbb{N}$, including the method $m : X \times \mathscr{P}_f X \to \mathbb{B}$ defined by:

$$\alpha(x)(u) = \begin{cases} \mathit{true} & \text{if } x \in u \\ \mathit{false} & \text{otherwise.} \end{cases}$$

Then $\tau_F^*(m) = \{\tau_F^*(m_1),\ \tau_F^*(m_2)\}$, where:

— $m_1 : X \to \mathbb{B}$ is defined by $\alpha_1(x) = \alpha(x)(\phi) = \mathit{false}$

— $m_2 : X \times \mathscr{P}_f(\mathbb{N}) \times X \to \mathbb{B}$ is defined by $\alpha_2(x)(u,y) = \alpha(x)(u \cup \{y\})$

— $\tau_F^*(m_1) = \{m_1\}$

— $\tau_F^*(m_2) = \{m_2', m_2''\}$, where

- $m_2' : X \times \mathscr{P}_f(\mathbb{N}) \times \mathbb{N} \to \mathbb{B}$ is defined by $\alpha_2' = \alpha_2$
- $m_2'' : X \times \mathscr{P}_f(\mathbb{N}) \times \mathbb{N} \to \mathbb{B}$ is defined by $\alpha_2''(x)(u,y) = \alpha_2(y)(u,x) = \alpha(y)(u \cup \{x\})$.

Now, given a class C, we can define a coalgebraic model of the transformed class C^* using purely covariant tools, as in Section 3 for unary methods.

Definition 4.2 (Freezing coalgebraic model). Let C be a class and C^* be the class obtained from the transformation τ^* starting from C. We call the functor induced by the methods of C^* according to Definition 3.1 the *freezing functor* F, and the bisimilarity equivalence induced by this coalgebraic model the *freezing equivalence* $\approx_F$.

Finally, we are left to establish the result that motivated our treatment, namely, we need to prove that the freezing bisimilarity equivalence for *finitary* binary methods is the greatest congruence with respect to methods of the *original class* C.

Theorem 4.1. Let C be a class with finitary binary methods. Then the freezing bisimilarity equivalence $\approx_F$ is the greatest congruence on C.

Proof. The theorem follows if we establish:

1. $\approx_F$ is the greatest congruence with respect to methods in C^*.
2. Any equivalence on objects of C is a congruence with respect to the methods of C *if and only if* it is a congruence with respect to the methods of C^*.

Fact 1 is immediate by Lemma 3.1.

For fact 2, it is sufficient to show that an equivalence $\sim$ on a set of objects $\bar{X}$ is a congruence with respect to a method $m : X \times \prod_{j\in J} T_j \to T_0$ *if and only if* it is a congruence with respect to the methods in $\tau_F(m)$. We prove this by induction on the structure of the parameters $\prod_{j\in J} T_j$.

Base case: m is a simple binary method implemented by α.

If $\sim$ is a congruence with respect to m, then $\sim$ is immediately a congruence with respect to the methods in $\tau_F(m)$, by definition.

For the other direction, assume that $\sim$ is a congruence with respect to the methods in $\tau_F(m)$. Let $xa_1 \dots a_q, x'a'_1 \dots a'_q \in X \times \prod_{j=1}^{q} T_j$ be such that $xa_1 \dots a_q \;\sim\; x'a'_1 \dots a'_q$. We prove that $\alpha(x)(\vec{a}) \;\sim\; \alpha(x')(\vec{a}')$ by induction on the number n of different parameters in the lists $xa_1 \dots a_q, x'a'_1 \dots a'_q$.

If $n = 0$, the thesis is immediate from reflexivity of $\sim$.

We now assume that the thesis holds for $n-1$ different parameters. We assume that $xa_1 \dots a_q$ and $x'a'_1 \dots a'_q$ have $n > 0$ different parameters, and let a_j, a'_j be the n^{th} different parameter. By the induction hypothesis,

$$\alpha(x)(a_1, \dots, a_j, \dots a_q) \;\sim\; \alpha(x')(a'_1, \dots, a_j, \dots a'_q)\,.$$

Moreover,

$$\alpha(x')(a'_1, \dots, a_j, \dots a'_q) \;\sim\; \alpha(x')(a'_1, \dots, a'_j, \dots a'_q)\,,$$

by the hypothesis that $\sim$ is a congruence with respect to $\tau_F(m)$. Hence, by transitivity of $\sim$, we get the thesis.

Induction step:

— If the leftmost non-constant parameter in m that is different from X has the shape $\sum_{k\in K} T_{i_K}$ or $\prod_{k\in K} T_{i_K}$, the thesis is immediate from the definitions of $\tau_F(m)$ and relational lifting.

— If the leftmost non-constant parameter in m that is different from X is $T_i = \mathscr{P}_f(T'_i)$, then $\tau_F(m) = \{m_1, m_2\}$, where

$$\begin{aligned} m_1 &: X \times \dots \times T_{i-1} \times T_{i+1} \times \dots \times T_q \to T_0 \\ m_2 &: X \times \dots \times T_{i-1} \times \mathscr{P}_f(T'_i[\bar{X}|X]) \times X \times T_{i+1} \times \dots \times T_q \to T_0\,. \end{aligned}$$

Assume that $\sim$ is a congruence with respect to m, that is,

$$\begin{aligned} & x \sim x' \wedge a_1 \sim a'_1 \wedge \dots \wedge a_{i-1} \sim a'_{i-1} \wedge u \sim u' \wedge a_{i+1} \sim a'_{i+1} \wedge \dots \wedge a_q \sim a'_q \\ & \qquad \Longrightarrow \alpha(x)(a_1, \dots, a_{i-1}, u, a_{i+1}, \dots, a_q) \sim \alpha(x')(a'_1, \dots, a'_{i-1}, u', a'_{i+1}, \dots, a'_q) \end{aligned}$$

Then, in particular, $\sim$ is a congruence with respect to both m_1 and m_2.

For the other direction, assume that $\sim$ is a congruence with respect to m_1, m_2, that is,

$$\begin{aligned} & x \sim x' \wedge a_1 \sim a'_1 \wedge \dots \wedge a_{i-1} \sim a'_{i-1} \wedge a_{i+1} \sim a'_{i+1} \wedge \dots \wedge a_q \sim a'_q \\ & \quad \Longrightarrow \alpha(x)(a_1, \dots, a_{i-1}, \emptyset, a_{i+1}, \dots, a_q) \sim \alpha(x')(a'_1, \dots, a'_{i-1}, \emptyset, a'_{i+1}, \dots, a'_q) \end{aligned} \tag{3}$$

and for all u,

$$x \sim x' \wedge a_1 \sim a'_1 \wedge \ldots \wedge a_{i-1} \sim a'_{i-1} \wedge y \sim y' \wedge a_{i+1} \sim a'_{i+1} \wedge \ldots \wedge a_q \sim a'_q \\ \Longrightarrow \\ \alpha(x)(a_1, \ldots, a_{i-1}, u \cup \{y\}, a_{i+1}, \ldots, a_q) \sim \alpha(x')(a'_1, \ldots, a'_{i-1}, u \cup \{y'\}, a'_{i+1}, \ldots, a'_q) \tag{4}$$

Now let

$$x \sim x' \wedge a_1 \sim a'_1 \wedge \ldots \wedge a_{i-1} \sim a'_{i-1} \wedge u \sim u' \wedge a_{i+1} \sim a'_{i+1} \wedge \ldots \wedge a_q \sim a'_q \,.$$

We have to show that

$$\alpha(x)(a_1, \ldots, a_{i-1}, u, a_{i+1}, \ldots, a_q) \sim \alpha(x')(a'_1, \ldots, a'_{i-1}, u', a'_{i+1}, \ldots, a'_q) \,.$$

We proceed by induction on the number of elements in $(u \setminus u') \cup (u' \setminus u)$:

- If $u = u' = \phi$, the thesis follows immediately by (3).
- If $u = u' \neq \phi$, the thesis follows by (5).
- If $|u \setminus u'| > 0$, let $y \in u \setminus u'$. Since $u \sim u'$, there exists $y' \in u$ such that $y \sim y'$. By (5), we have

  $$\alpha(x)(a_1, \ldots, a_{i-1}, u, a_{i+1}, \ldots, a_q) \sim \alpha(x')(a'_1, \ldots, a'_{i-1}, (u \setminus \{y\}) \cup \{y'\}, a'_{i+1}, \ldots, a'_q) \,.$$

  Now, by the definition of relational lifting, using the fact that $\sim$ is an equivalence, we get $(u \setminus \{y\}) \cup \{y'\} \sim u'$. Then, by the induction hypothesis,

  $$\alpha(x')(a'_1, \ldots, a'_{i-1}, (u \setminus \{y\}) \cup \{y'\}, a'_{i+1}, \ldots, a'_q) \sim \alpha(x')(a'_1, \ldots, a'_{i-1}, u', a'_{i+1}, \ldots, a'_q) \,.$$

  The thesis then follows by the transitivity of $\sim$. □

As a by-product of Theorem 4.1, we get that the greatest congruence with respect to methods always exists for finitary binary methods, that is, we have the following corollary.

Corollary 4.1. Let C be a class with carrier X and whose methods are all finitary binary. Then $\cup\{\sim \subseteq X \times X \mid \sim$ is a congruence with respect to the methods of $C\}$ is a congruence.

Note that, to ensure Theorem 4.1, it is essential to give a coalgebraic description of binary methods that accounts for the behaviour of an object under method application when the object is viewed as *any* of the class parameters of the method. Otherwise, if we observe the behaviour of an object for example, only when it is considered as the target of a method call and not as a generic class parameter, the congruence property of $\approx_F$ fails in general. The following is a counterexample.

Example 4.2. Consider a class R' of registers containing a single method $m : X \times X \to \mathbb{N}$, defined by $\alpha(r_1)(r_2) = r_2.val$.

Now, if in the definition of the freezing functor F we consider only the first component induced by the method m, we have $r_1 \approx_F r_2$, for all r_1, r_2. But then $\approx_F$ is not a congruence with respect to the method m. For example, if we consider r_1, r_2 such that $r_1.val = 1$ and $r_2.val = 2$, then $r_1 \approx_F r_2$, however, for any r_0, $\alpha(r_0)(r_1) = 1$, while $\alpha(r_0)(r_2) = 2$. The problem arises because the result of applying m depends on an unobservable behaviour of the second parameter.

Nevertheless, there are many interesting cases in which it is sufficient to consider only some of the components in the definition of F for the bisimilarity equivalence to be a congruence. An interesting example is that of the λ-calculus (Honsell and Lenisa 1995). In this case, the freezing functor with only the first component for the method *app* induces the *applicative equivalence* on closed λ-terms. On the other hand, if we consider both components in the functor (or just the second one), we get an equivalence that can be viewed as a coinductive characterisation of the *contextual equivalence.* Applicative and contextual equivalences can be proved to coincide. This is not immediate, and many techniques, which apply to various reduction strategies, have been developed to achieve this, see, for example, Honsell and Lenisa (1995) for more details.

4.1.1. *Infinitary binary methods.* Theorem 4.1 does not extend to infinitary binary methods, since the freezing bisimulation equivalence fails, in general, to be a congruence. The following are two counterexamples.

Example 4.3. Let C be a class with carrier $X \triangleq \mathbb{N}$ and a single method $m : X \times \mathscr{P}(X) \to \mathbb{B}$ defined by

$$\alpha(x)(n) = \begin{cases} \mathit{true} & \text{if } |n| < \omega \\ \mathit{false} & \text{otherwise.} \end{cases}$$

One can check that the equivalence $\sim_k = \{(n,m) | n,m \leqslant k\} \cup \{(n,n) | n > k\}$ is a congruence for all k. However, $\bigcup_{k \in \omega} \sim_k = \{(n,m) | n,m \in N\}$, which coincides with the freezing equivalence, is *not* a congruence.

The following example is equivalent to Example 3.5.10 on page 114 of Tews (2002).

Example 4.4. Let C be a class with carrier $X \triangleq \mathbb{N}$ and with just one method $m : X \times [\mathbb{N} \to X] \to \mathbb{B}$ (where $[\mathbb{N} \to X]$ is an alias for the infinite product type $\prod_{i \in \mathbb{N}} X^i$), defined by

$$\alpha(x)(f) = \begin{cases} \mathit{true} & \text{if } f \text{ is bounded} \\ \mathit{false} & \text{otherwise,} \end{cases}$$

where $f : \mathbb{N} \to \mathbb{N}$ is bounded if there exists $k \in \mathbb{N}$ such that $\forall n.f(n) \leqslant k$.

One can check that the equivalence $\sim_k = \{(n,m) | n,m \leqslant k\} \cup \{(n,n) | n > k\}$ is a congruence for all k. However, $\bigcup_{k \in \omega} \sim_k = \{(n,m) | n,m \in \mathbb{N}\}$ coincides with the freezing equivalence and is *not* a congruence.

Clearly, in all cases where the union of all congruences is *not* itself a congruence, the freezing equivalence *cannot* be a congruence, since any congruence is, in particular, a freezing bisimulation. This is not surprising, since in these cases we lack a canonical congruence, so any semantics would be problematic. It is natural to ask whether the other implication holds as well, that is, if the union of all congruences is a congruence, then the freezing equivalence is a congruence. Somewhat surprisingly, this is not the case, as shown by the following counterexample.

Example 4.5. Let C be a class with carrier $X \triangleq \mathbb{N}$ and with just one method $m : X \times [\mathbb{N} \to X] \to \mathbb{B}$ defined by:

$$\alpha(x)(f) = \begin{cases} \mathit{true} & \text{if } f \text{ is definitely constantly } 0 \\ \mathit{false} & \text{otherwise} \end{cases}$$

where $f : \mathbb{N} \to \mathbb{N}$ is definitely constantly 0 if there exists k such that $f(n) = 0$ for all $n \geqslant k$.

One can easily check that the greatest congruence on X is $\{(0,0)\} \cup \{(n,m) | n,m \neq 0\}$. However, the freezing equivalence is $\{(n,m) | n,m \in \mathbb{N}\}$, which is clearly not a congruence.

Nevertheless, there are many situations where the greatest congruence exists and the freezing equivalence captures it. We feel that a situation where the greatest congruence does not exist, or it exists but the freezing equivalence does not capture it, is a situation where the class specification is underspecified. But more work needs to be done to capture this.

Finally, note that the problems with infinitary methods arise because of infinite products and $\mathscr{P}(\ )$. Infinite sums are not problematic. In particular, Theorem 4.1 also holds for the extension of finitary types with infinite sums.

4.2. *The graph functor*

In this section, we introduce an alternative approach to dealing with binary methods, which is satisfactory in most cases, and when it is not, it is a symptom of the fact that the specification is probably underdetermined.

Contravariant occurrences of the type variable in a generalised binary method can be turned into covariant ones by interpreting methods as *graphs* instead of functions. Consequently, the function space appearing in the functor induced by m is turned into a set of relations. For example, for the binary method $eq : X \times X \to \mathbb{B}$ of the class R of registers, we would consider the functor $GX \triangleq \mathscr{P}(X \times \mathbb{B})$.

As with the case of freezing, in order to make the graph bisimilarity equivalence a congruence in a wider spectrum of cases (including Example 4.2), we need to consider multiple copies of the binary method in the definition of the graph functor, in order to account for the behaviour of each class parameter. Thus, the graph functor corresponding to the method eq becomes $GX \triangleq \mathscr{P}(X \times \mathbb{B}) \times \mathscr{P}(X \times \mathbb{B})$. This works straightforwardly for simple binary methods, but requires a preliminary transformation for methods with more complex class parameters. Essentially, this corresponds to the method transformation procedure for the freezing functor, apart from the part that actually freezes the parameters. To deal with the powerset type constructor, we introduce the new symbol $\mathscr{P}\surd$, which is used to denote a powerset type constructor that has already been processed.

Formally, we have the following definitions.

Definition 4.3 (Normal form). A binary method $m : X \times \prod_{j \in J} T_j \to T_0$ is in *normal form* if its parameter types are either constants or X or $\mathscr{P}\surd(T)$, for $T \in \mathscr{T}$.

Definition 4.4 (Graph method/class transformation).

(i) Let τ_G be the one-step method transformation function (which takes a generalised binary method $m : X \times \prod_{j=1}^{q} T_j \to T$ implemented by α and produces a set of methods) defined by:

— m is in normal form.
Let I be the set of indices corresponding to class parameters of type X including the object itself. We define

$$\tau_F(m) = \{m_l \mid l \in I\}$$

where $m_l : X \times \prod_{j \in J} T_j \to T_0$ is defined by

$$\alpha_l(x)(a_1, \ldots, a_q) \triangleq \alpha(a_l)(a_1, \ldots, a_{l-1}, x, a_{l+1}, \ldots, a_q)\,.$$

— m is not in normal form.
Let T_i be the leftmost parameter not in normal form. Then:

– Case: $T_i = \sum_{j=1}^{q_i} T_{ij}$.
We define

$$\tau_G(m) = \{m_{ij} \mid j = 1, \ldots, q_i\}$$

where

$$m_{ij} : X \times \ldots \times T_{i-1} \times T_{ij} \times T_{i+1} \times \ldots \to T_0$$

is defined by

$$\alpha_{ij}(x)(a_1, \ldots, a_{i-1}, a_{ij}, a_{i+1}, \ldots, a_q) \triangleq \alpha(x)(a_1, \ldots a_{i-1}, in_j(a_{ij}), \ldots a_q)\,.$$

– Case: $T_i = \prod_{j=1}^{q_i} T_{ij}$.
We define

$$\tau_G(m) = \{m'\}$$

where

$$m' : X \times \ldots \times T_{i1} \times \ldots \times T_{iq_i} \times \ldots \to T_0$$

is defined by

$$\alpha'(x)(a_1, \ldots, a_{i-1}, a_{i1}, \ldots, a_{iq_i}, a_{i+1}, \ldots, a_q) \triangleq \alpha(x)(a_1, \ldots, a_{i-1}, \vec{a}_i, a_{i+1}, \ldots a_q).$$

– Case: $T_i = \mathscr{P}(T_i')$.
We define

$$\tau_G(m) = \{m_1, m_2\}$$

where

$$m_1 : X \times \ldots \times T_{i-1} \times T_{i+1} \times \times \ldots \to T_0$$

is defined by

$$\alpha_1(x)(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_q) \triangleq \alpha(x)(a_1, \ldots, a_{i-1}, \phi, a_{i+1}, \ldots a_q)$$

and $m_2 : \ldots \times \mathscr{P}^{\surd}(T'_i) \times X \times \ldots \to T_0$ is defined by

$$\alpha_2(x)(a_1, \ldots, a_{i-1}, u, y, a_{i+1}, \ldots, a_q) \triangleq \alpha(x)(a_1, \ldots a_{i-1}, u \cup \{y\}, a_{i+1}, \ldots a_q).$$

(ii) Let τ^*_G be the transformation function that takes a generalised binary method and produces a set of methods in normal form, defined by

$$\tau^*_G(m) = \begin{cases} \tau_G(m) & \text{if } m \text{ is in normal form} \\ \bigcup\{\tau^*_G(m_i) | m_i \in \tau_G(m)\} & \text{otherwise.} \end{cases}$$

(iii) Let S be a class specification. We use S^* to denote the class specification obtained by applying the transformation τ^*_G to all method declarations in S.

(iv) Let C be a class implementation. We use C^* to denote the class implementation obtained by applying the transformation τ^*_G to all methods in C.

Note that there is a precise correspondence between the transformations τ^*_G and τ^*_F. Namely, for any method m, there is a one-to-one correspondence between the methods of $\tau^*_G(m)$ and $\tau^*_F(m)$, mapping each method of $\tau^*_G(m)$ into a method of $\tau^*_F(m)$, which differs from the first one only because of the freezing of some parameter types.

In order to give a graph coalgebraic model to a specification S (implementation C), we consider the corresponding transformed specification S^* (implementation C^*) and define a corresponding graph functor G simply by turning the contravariant function spaces in method declarations into covariant spaces of graphs.

Definition 4.5 (Graph functor). Let S be a class specification. The method declarations in S^* induce the *graph functor* $G : \mathscr{C} \to \mathscr{C}$ defined by

$$G \triangleq \prod_{i=1}^{k} G_i,$$

where, for each method $m_i : X \times \prod_{j \in J} T_{ij} \to T_{i0}$, we have $G_i X \triangleq \mathscr{P}((\prod_{j \in J} T_{ij}) \times T_{i0})$.

Given a class C, the corresponding class C^* immediately induces a coalgebra for the graph functor determined by the method declarations in C^*, according to Definition 3.2 of Section 3, by viewing method codes as graphs instead of functions. However, there is no longer a precise correspondence between classes and coalgebras since not all G-coalgebras correspond to a class – only the functional ones do, that is, those whose coalgebra map is a function.

The graph bisimilarity equivalence on the objects of C^* can be characterised as follows.

Proposition 4.1 (Graph bisimilarity equivalence). Let C be a class, $G = \prod_{i=1}^{k} G_i$ be the functor induced by the method declarations in C^*, and $(X, \langle \alpha_i \rangle_{i=1}^{k})$ be the G-coalgebra induced by the methods of C^*. Then:

(i) A G-bisimulation (graph bisimulation) on $(X, \langle \alpha_i \rangle_{i=1}^{k})$ is a relation $R \subseteq X \times X$ satisfying

$$\begin{aligned} x \, R \, x' \implies \forall \alpha_i. \forall \vec{a}\, \exists \vec{a}'. (\vec{a} \, R \, \vec{a}' \wedge \alpha_i(x)(\vec{a}) \, R \, \alpha_i(x')(\vec{a}')) \ \wedge \\ \forall \vec{a}' \ \exists \vec{a}. (\vec{a} \, R \, \vec{a}' \ \wedge \alpha_i(x)(\vec{a}) \, R \, \alpha_i(x')(\vec{a}')). \end{aligned}$$

(ii) The graph bisimilarity equivalence $\approx_G$ (the greatest G-bisimulation on $(X, \langle \alpha_i \rangle_{i=1}^k)$) can be characterised as follows:

$$x \approx_G x' \iff \forall \alpha_i.\ \forall \vec{a}\ \exists \vec{a}'.(\vec{a} \approx_G \vec{a}' \ \wedge \alpha_i(x)(\vec{a}) \approx_G \alpha_i(x')(\vec{a}')) \ \wedge$$
$$\forall \vec{a}'\ \exists \vec{a}.(\vec{a} \approx_G \vec{a}' \ \wedge \alpha_i(x)(\vec{a}) \approx_G \alpha_i(x')(\vec{a}'))\,.$$

In particular, the following coinduction principle holds:

$$\frac{R \text{ is a graph bisimulation} \qquad x\,R\,x'}{x \approx_G x'}\,.$$

Proof. The proposition follows from the definition of coalgebraic bisimulation (see Definition A.3 of Appendix A) and Theorem A.1 of Appendix A. □

Note the alternation of the $\forall$ and $\exists$ quantifiers in the definition of graph bisimulation, which is due to the presence of the powerset in the graph functor.

The functor G has a final coalgebra, see, for example, Cancila *et al.* (2006). But, in general, it is not functional, and, moreover, the functionality property of a coalgebra is not preserved by the unique morphism into the final coalgebra. Therefore, the image of a class implementation under the final morphism is not guaranteed to be a class implementation. Thus, in general, we lack minimal class implementations. In Section 4.3 we study conditions for the final morphism to preserve the functionality property, thus recovering minimal implementations.

4.3. *Comparing graph and freezing bisimilarity equivalences*

The following is an easy lemma.

Lemma 4.1. $\approx_F \subseteq \approx_G$.

Proof. One can easily check that $\approx_F$ is a graph bisimulation using the reflexivity of $\approx_F$. □

The converse inclusion does not hold in general. For example, it does not hold for the class R' obtained from the class R of registers of Table 1 when we drop methods *get* and *set*, and just consider the method *eq*. Namely, for R', $\approx_G$ equates all pairs of registers, while $\approx_F$ is the identity relation on registers. Moreover, note that in this case $\approx_G$ is *not* a congruence with respect to *eq*.

The following result is a fundamental tool for recovering $\approx_F = \approx_G$.

Theorem 4.2. Let C be a class with finitary generalised binary methods. Then

$$\approx_G = \approx_F \iff \approx_G \quad \text{is a congruence with respect to the methods in the class.}$$

Proof.

($\Rightarrow$)This direction follows from Theorem 4.1.

($\Leftarrow$)By Lemma 4.1, $\approx_F \subseteq \approx_G$. Since $\approx_G$ is a congruence, by Theorem 4.1, $\approx_G \subseteq \approx_F$. Thus $\approx_G = \approx_F$. □

The equality $\approx_G = \approx_F$ on a given class C is equivalent to the fact that the image of the G-coalgebra representing C into the final coalgebra is still a functional coalgebra. Hence, we have the following corollary.

Corollary 4.2. Let C be a class with finitary generalised binary methods. Then the image of the G-coalgebra representing C into the final coalgebra is a functional coalgebra *if and only if* $\approx_G$ is a congruence.

Thus Corollary 4.2 above gives an answer to the problem of minimal class implementations for the graph functor that we raised at the end of Section 4.2.

In principle, Theorem 4.2 is all that we may need. However, in practice, it is useful to have some other alternative sufficient conditions. The following theorem gives a sufficient condition on the freezing equivalence, ensuring that $\approx_G = \approx_F$.

Theorem 4.3. If $\approx_F$ is determined only by the unary methods of the class, that is,

$$x \approx_F x' \iff \forall m \text{ unary implemented by } \alpha.\ \forall \vec{a}.\ \alpha(x)(\vec{a}) \approx_F \alpha(x')(\vec{a'}),$$

then $\approx_G = \approx_F$.

Proof. By Lemma 4.1, $\approx_F \subseteq \approx_G$. For the other direction, we have

$$\approx_F = (\approx_F)_{|\text{unary methods}} = (\approx_G)_{|\text{unary methods}} \supseteq \approx_G. \qquad \square$$

Theorem 4.3 applies to the class R of registers, since the freezing equivalence is already solely determined by the unary methods *get* and *set*.

Remark 4.1. Note that Theorem 4.3 does not apply to the class Λ, where, nevertheless, $\approx_G = \approx_F$. Proving this latter result for the λ-calculus is quite a difficult task, and was addressed in the more general setting of applicative structures in Honsell and Lenisa (1999).

The following corollary is an almost trivial, but useful, application of Theorem 4.3.

Corollary 4.3. If the freezing equivalence restricted to the unary methods of the class is the identity on objects, then $\approx_G = \approx_F$.

5. Relational types

The idea of treating binary methods as graphs, rather than as functions, can be fruitfully pushed further to produce a typing system that overcomes the well-known problem arising when inheritance is combined with subtyping, see, for example, Bruce *et al.* (1995). If we type binary methods only with the usual arrow type, which is contravariant, we lose the property that subclasses are subtypes. We propose to introduce a new type constructor, that is, the *relation type*, and use this to type binary methods in class declarations. Since relation types are purely covariant, the subtyping property is maintained by subclasses. To preserve safety, in contrast to arrow types, we assume that relation types are not 'applicable', that is, there is no relational counterpart to the rule

$$\frac{M : \alpha \to \beta \quad N : \alpha}{MN : \beta}.$$

Table 2. *Typing rules for* Relational Types $\otimes$.

$\dfrac{x : \alpha \quad M : \beta}{\lambda x^{\alpha}.M : \alpha \otimes \beta}$	$\dfrac{\alpha \leqslant \alpha' \quad \beta \leqslant \beta'}{\alpha \otimes \beta \leqslant \alpha' \otimes \beta'}$	$\dfrac{}{\alpha \to \beta \leqslant \alpha \otimes \beta}$

Binary methods can, nevertheless, also be typed with the standard arrow type, which is a subtype of the corresponding relation type, see Table 2. Thus, this typing system is a conservative extension of the usual one. This solution to the problem of typing binary methods is quite simple, and it allows for *single dispatching* in method calls. Moreover, unlike other proposals, our proposal allows for 'future code extensions' without losing the subtyping property of classes. Of course, type uniqueness fails in this system. We will study this proposal in a future paper.

6. Final remarks and directions for future work

Here are some final comments and some potentially fruitful lines of research:

— Without the powerset, our (finitary) generalised binary methods are a subset of the ones handled by Tews using extended polynomial (cartesian) functors. However, we feel that the two collections of methods essentially correspond. Namely, given a method m that has an extended polynomial type, m either corresponds directly (possibly up-to currying) to a generalised binary method or can be cast into a generalised binary type at the price of extending it vacuously. For example, a method $m : X \to ((X \to (N \to X)) + N)$ has an extended polynomial type in the sense of Tews, but does not have a generalised binary type (since the occurrence of $+$ prevents currying). But, the effects of m can be recovered in our setting, since m can be cast into a method of type $X \times X \to ((N \to X) + N)$. Of course, this is just an example, and further investigation is needed to streamline this procedure.

— The existence of final (minimal) models for a given specification is important. To achieve this, as discussed at the end of Section 3, it is crucial that the bisimilarity equivalence is a congruence and, moreover, it preserves assertions. It would be quite interesting to investigate for which kinds of assertions this is the case.

— We plan to use the coalgebraic descriptions of binary methods discussed in this paper to model classes of concrete object-oriented languages. In particular, it would be interesting to extend the coalgebraic model for Java-like classes with unary methods only, which were studied in Honsell *et al.* (2004). Imperative object-oriented languages are more difficult to handle than purely functional languages, since accounting for the store adds extra issues.

— Following Jacobs (1996), one can also define an equivalence between coalgebras implementing the same specification by taking coalgebras to be equivalent when initial objects are bisimilar.

— The grammar for parameter types in Definition 2.1 could be extended to include inductive and coinductive types. However, it appears that it cannot be extended with the (contravariant) arrow type. In particular, there is no 'well-behaved' natural extension of the behavioural equivalence to the function type, since the natural definition $R^{T_1 \to T_2} = \{(f, f') \mid \forall x R^{T_1} x'.\ fx R^{T_2} f'x'\}$ of relational lifting fails to preserve equivalence relations, because, in general, $R^{T_1 \to T_2}$ is not reflexive. A possible remedy for this problem is to include $T \to T$ in the *covariant* space of *binary relations*. The difference compared with the traditional interpretation of the function space arises when we define bisimulation equivalences.

— Finally, we point out that our approach to the bialgebraic description of classes involving binary methods is quite general. It can also be used to model coinductive data types, possibly with binary evolution laws, such as the concurrent process language with process parameters studied in Lenisa (1996).

Appendix A. Algebraic and coalgebraic preliminaries

In this appendix we recall some notions and results for *algebras*, *coalgebras* and *bialgebras*. For more details, see, for example, Jacobs and Rutten (1996), Turi and Plotkin (1997) and Corradini *et al.* (2002). We work in a category $\mathscr{C}$ of sets and proper classes of any wellfounded universe (or possibly non-wellfounded universe (Forti and Honsell 1983; Aczel 1988)), and set-theoretic functions. Throughout this section, we assume H, L to be endofunctors on $\mathscr{C}$.

Definition A.1 (L-algebra, H-coalgebra and $\langle L, H\rangle$-bialgebra).

— An L-*algebra* is a pair (X, β_X), where X is a set (the *carrier* of the algebra) and $\beta_X : LX \to X$ is a function in $\mathscr{C}$ (the *operation* of the algebra).

— Dually, an H-*coalgebra* is a pair (X, α_X), where $\alpha : X \to HX$.

— An $\langle L, H\rangle$-*bialgebra* is a triple (X, β_X, α_X), where (X, β_X) is an L-algebra and (X, α_X) is an H-coalgebra.

L-algebras, H-coalgebras and $\langle L, H\rangle$-bialgebras can all be endowed with a suitable category structure by defining the notions of an L-algebra, H-coalgebra and $\langle L, H\rangle$-bialgebra morphism, as follows.

Definition A.2.

$$\begin{tikzcd} LX \arrow[r, "\beta_X"] \arrow[d, "Lf"'] \arrow[dr, phantom, "(1)"] & X \arrow[r, "\alpha_X"] \arrow[d, "f"] \arrow[dr, phantom, "(2)"] & HX \arrow[d, "Hf"] \\ LY \arrow[r, "\beta_Y"'] & Y \arrow[r, "\alpha_Y"'] & HY \end{tikzcd}$$

A function $f : X \to Y$ is an:

— L-*algebra morphism* from the L-algebra (X, β_X) to the L-algebra (Y, β_Y) if it makes diagram (1) commute.

— H-*coalgebra morphism* from the H-coalgebra (X, α_X) to the H-coalgebra (Y, α_Y) if it makes diagram (2) commute.

— $\langle L, H\rangle$-bialgebra morphism from (X, β_X, α_X) to (Y, β_Y, α_Y) if f makes diagrams (1) *and* (2) commute.

Initial morphisms, that is, algebra morphisms from initial algebras, induce equivalences that are *congruences* with respect to algebra operations. Dually, *final morphisms*, that is, coalgebra morphisms into final coalgebras, induce equivalences that have coinductive characterisations in terms of *bisimulations*.

Definition A.3 (H-bisimulation). Let H be an endofunctor on the category $\mathscr{C}$. A relation $\mathscr{R}$ on objects X, Y is an H-*bisimulation* on the H-coalgebras (X, α_X) and (Y, α_Y) if there exists an arrow $\gamma : \mathscr{R} \to H(\mathscr{R})$ in $\mathscr{C}$ such that the following diagram commutes:

$$\begin{tikzcd}
X \arrow[d, "\alpha_X"'] & \mathscr{R} \arrow[l, "r_1"'] \arrow[r, "r_2"] \arrow[d, "\gamma"] & Y \arrow[d, "\alpha_Y"] \\
H(X) & H(\mathscr{R}) \arrow[l, "H(r_1)"] \arrow[r, "H(r_2)"'] & H(Y)
\end{tikzcd}$$

The following theorem expresses the fact that unique morphisms into final coalgebras induce behavioural equivalences on the starting coalgebras, which can be characterised coinductively as *greatest* H-bisimulations.

Theorem A.1. Suppose that $H : \mathscr{C} \to \mathscr{C}$ preserves weak pullbacks and has a final H-coalgebra $(\Omega_H, \alpha_{\Omega_H})$. Let (X, α_X) be a H-coalgebra and $\mathscr{M} : (X, \alpha_X) \to (\Omega_H, \alpha_{\Omega_H})$ be the unique final morphism. Then

$$\mathscr{M}(x) = \mathscr{M}(x') \iff x \approx_H x'$$

where:

— $\approx_H$ denotes the greatest H-bisimulation on (X, α_X); and

— $\approx_H = \bigcup\{\mathscr{R} \mid \mathscr{R} \text{ is a } H\text{-bisimulation on } (X, \alpha_X)\}$.

In particular, the following *coinduction principle* holds:

$$\frac{\mathscr{R} \text{ is a } H\text{-bisimulation on } (X, \alpha_X) \qquad x \mathscr{R} x'}{x \approx_H x'}.$$

Acknowledgments

The authors are grateful to the anonymous referees for very useful comments.

References

Abramsky, S. and Ong, L. (1993) Full Abstraction in the Lazy Lambda Calculus. *Information and Computation* **105** (2) 159–267.

Aczel, P. (1988) *Non-wellfounded sets*, CSLI Lecture Notes **14**.

Aczel, P. and Mendler, N. (1989) A Final Coalgebra Theorem. In: Pitt, D.H. *et al.* (eds.) Proc. category theory and computer science. *Springer-Verlag Lecture Notes in Computer Science* **389** 357–365.

van den Berg, J., Huisman, M., Jacobs, B. and Poll, E. (1999) A type-theoretic memory model for verification of sequential Java programs. In: Bert *et al.* (eds.) WADT'99. *Springer-Verlag Lecture Notes in Computer Science* **1827** 1–21.

Bruce, K. B., Cardelli, L., Castagna, G., Eifrig, J., Smith, S. F., Trifonov, V., Leavens, G. T. and Pierce, B. C. (1995) On Binary Methods. *TAPOS* **1** (3) 221–242.

Cancila, D., Honsell, F. and Lenisa, M. (2006) Some Properties and Some Problems on Set Functors. CMCS'06. *Electronic Notes in Theoretical Computer Science* (to appear).

Corradini, A., Heckel, R. and Montanari, U. (2002) Compositional SOS and beyond: A coalgebraic view of open systems. *Theoretical Computer Science* **280** 163–192.

Cirstea, C. (2000) *Integrating observations and computations in the specification of state-based, dynamical systems*, Ph.D. thesis, University of Oxford.

Forti, M. and Honsell, F. (1983) Set-theory with free construction principles. *Ann. Scuola Norm. Sup. Pisa*, Cl. Sci. (4) **10** 493–522.

Goguen, J. and Malcolm, G. (2000) A Hidden Agenda. *Theoretical Computer Science* **245** 55–101.

Hennicker, R. and Kurz, A. (1999) (Ω, Ξ)-Logic: On the algebraic extension of coalgebraic specifications. CMCS'1999. *Electronic Notes in Theoretical Computer Science* **19**.

Hermida, C. and Jacobs, B. (1998) Structural induction and coinduction in a fibrational setting. *Information and Computation* **145** (2) 107–152.

Honsell, F. and Lenisa, M. (1995) Final Semantics for Untyped Lambda Calculus. In: Dezani, M. *et al.* (eds.) TLCA'95. *Springer-Verlag Lecture Notes in Computer Science* **902** 249–265.

Honsell, F. and Lenisa, M. (1999) Coinductive Characterizations of Applicative Structures. *Mathematical Structures in Computer Science* **9** 403–435.

Honsell, F., Lenisa, M. and Redamalla, R. (2004) Coalgebraic Semantics and Observational Equivalences of an Imperative Class-based OO-Language. In: Honsell, F. *et al.* (eds.) COMETA'03. *Electronic Notes in Theoretical Computer Science* **104** 163–180.

Huisman, M. (2001) *Reasoning about Java programs in Higher-order logic using PVS and Isabelle*, Ph.D. thesis, University of Nijmegen, The Netherlands.

Jacobs, B. (1996) Objects and Classes, co-algebraically. In: Freitag, B. *et al.* (eds.) *Object-Orientation with Parallelism and Book Persistence*, Kluwer Academic Publishers 83–103.

Jacobs, B. (1996a) Inheritance and cofree constructions. In: Cointe, P. (ed.) *ECOOP'96. Springer-Verlag Lecture Notes in Computer Science* **1098** 210–231.

Jacobs, B. (1997) Behaviour-refinement of object-oriented specifications with coinductive correctness proofs. In: Bidoit, M. *et al.* (eds.) TAPSOFT'97. *Springer-Verlag Lecture Notes in Computer Science* **1214** 787–802.

Jacobs, B. and Rutten, J. (1996) A tutorial on (co)algebras and (co)induction. *Bulletin of the EATCS* **62** 222–259.

Lenisa, M. (1996) Final Semantics for a Higher Order Concurrent Language. In: Kirchner, H. *et. al.* (eds.) CAAP'96. *Springer-Verlag Lecture Notes in Computer Science* **1059** 102–118.

Poll, E. and Zwanenburg, J. (2001) From algebras and coalgebras to dialgebras. In: Corradini, A., Lenisa, M. and Montanari, U. (eds.) CMCS'01. *Electronic Notes in Theoretical Computer Science* **44**.

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

Rosu, G. and Goguen, J. (2000) Hidden Congruent Deduction. FTP'98. *Springer-Verlag Lecture Notes in Artificial Intelligence* **1761** 252–267.

Rothe, J., Tews, H. and Jacobs, B. (2001) The Coalgebraic Class Specification Language CCSL. *Journal of Universal Computer Science* **7** 175–193.

Tews, H. (2002) *Coalgebraic Methods for Object-Oriented Specifications*, Ph.D. thesis, Dresden University of technology, 2002.

Turi, D. and Plotkin, G. (1997) Towards a mathematical operational semantics. 12th *LICS*, IEEE, Computer Science Press 280–291.