Under consideration for publication in Math. Struct. in Comp. Science

Coalgebraic Description of Generalized Binary Methods[†]

Furio Honsell[1], Marina Lenisa[1‡], Rekha Redamalla[1,2]

[1] *Dipartimento di Matematica e Informatica, Università di Udine,*
Via delle Scienze 206, 33100 Udine, ITALY.
[2] *B.M. Birla Science Centre,*
Adarsh Nagar, Hyderabad, 500 063 A.P., INDIA.

Received 2 November 2006

We extend Reichel-Jacobs coalgebraic account of specification and refinement of objects and classes in Object Oriented Programming to (*generalized*) *binary methods*. These are methods that take more than one parameter of a class type. Class types include products, sums and powerset type constructors. In order to take care of class *constructors*, we model classes as *bialgebras*. We study and compare two solutions for modeling generalized 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 generalized 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*, then 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.
In both cases, *final coalgebras* are shown to exist.

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

Contents

1. Introduction

In (Reichel(1995); Jacobs(1996); Jacobs(1997)), a categorical framework, based on the notion of *coalgebra*, is given for providing semantics to *objects* and *classes*. The idea underpinning this approach to class specification and refinement is that coalgebras, duals to algebras, allow 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 out of the carrier set), which allow 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*, *i.e.* the *objects*; methods act on objects.

In the coalgebraic approach of (Reichel(1995); Jacobs(1996); Jacobs(1997)), a class is modeled as an F-coalgebra $(A, f : A \to F(A))$ for a suitable functor F. The carrier A represents the space of objects, and the coalgebra operation f represents the *public* methods of the class, *i.e.* the methods which are accessible from outside the class. Methods are viewed as functions acting on objects. The coalgebraic model, *i.e.* the unique morphism into the *final F-coalgebra*, induces precisely the *behavioural equivalence* on objects, whereby 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.

In order to account also for class *constructors*, in this paper, we introduce an algebra part in our model, thus modelling classes as *bialgebras*. A similar move appears in (Rothe et al.(2001); Cirstea(2000)).

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. 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, *i.e.* methods with more than one class parameter, apparently escape this simple co(bi)algebraic approach[†]. The extra class parameters produce contravariant occurrences in the functor modelling class methods, and hence cannot be dealt with by a straightforward application of the standard coalgebraic methodology.

In this paper, we extend the Reichel-Jacobs coalgebraic approach to *generalized 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 quite large

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

collection of methods, including all the methods which are commonly used in Object Oriented Programming.

Our focus of interest are equivalences on objects which are "well-behaved", *i.e.* are *congruences w.r.t.* 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 that of showing that canonical models can be built also for classes with generalized binary methods using purely covariant tools, at least in the case of *finitary binary methods*, *i.e.* 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 generalized binary method can be captured by a bunch of unary methods obtained after a suitable manipulation of the original method. The key step is that of *"freezing"*, in turn, the types of the class parameters to the states of the class implementation given at the outset, *i.e.* by viewing them as constant types.

Our second solution is based on a set-theoretic understanding of functions, whereby 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 w.r.t.* 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 which always includes the freezing equivalence. Therefore, but 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 on 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, generalizing the one in (Honsell et al.(1995)), for representing terms of the λ-calculus, together with the lazy notion of reduction strategy of (Abramksy(1993)).

A natural question to ask when the freezing approach does not coincide with the graph approach is why is it the case. We do not have a fully satisfactory answer, but we feel that this is a telltale alarm. 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 of 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, *e.g.* see (Bruce et al.(1995)). In this paper, we will hint to an intriguing solution based on a new typing system, which utilizes a *relation type* for typing (binary) methods when these are viewed as *graphs*, rather than functions. Our solution takes classes seriously, *i.e.* it does not utilize *multiple dispatching.* In concrete

OO languages, such as Java, the problem of typing binary methods when subclasses are involved is normally solved by implementing method calls with multiple dispatching, *i.e.* by choosing the method code to activate according to the types of all class parameters, and *not* only the object type.

We emphasize 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 due to the choice of our ambient category, we work in the category of sets and proper classes (Aczel et al.(1989); Forti et al.(1983)), where all covariant functors can be shown to have a final coalgebra, see (Cancila et al.(2006)). Thus, throughout the paper, we fix $\mathcal{C}$ 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 we refer to Appendix A and (Jacobs et al.(1996)).

Comparison with Related Work

There are other approaches in the literature which address the problem of extending the coalgebraic model to binary methods. We offer the following analysis.

In (Jacobs(1996a)), binary methods are allowed only when they are definable in terms of the unary methods of the class. Hence, in particular binary methods do not contribute to the definition of the observational equivalence. The same observation applies to the approach of (Hennicker et al.(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 connection *a priori* 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 we use only covariant tools, from the very outset. Nonetheless, there are interesting connections between the two approaches. We consider also the powerset type constructor, which Tews does not include, but, apart from this, our generalized binary methods should correspond, essentially, to the class of *extended polynomial functors* of Tews[‡]. Similarly, our finitary generalized 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 et al.(2001);

[‡] Apparently, Tews' extended polynomial functors cover a wider collection of methods, but we conjecture that also those particular cases which appear to escape from our approach should be recovered using manipulations similar to those introduced in Section 6. However, more work needs to be done.

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 not vice versa. Moreover, our notion of bisimulation, being monotone, gives always 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 modeling binary methods with 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 has interest also in itself. Second, in our setting, final coalgebras always exist, and hence we have canonical models, while in (Tews(2002)) mixed functors do not admit final coalgebras.

Yet another approach in the theory of coalgebraic semantics consists in avoiding binary methods altogether by considering a whole system of objects in place of single objects, e.g. 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); Huisman(2001)). This approach is quite different from ours.

Finally, there is an interesting connection between our approach and the approach of *hidden algebras*, see (Goguen et al.(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 *w.r.t.* 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 is the main section of the paper, we present, discuss, and compare two bialgebraic accounts of generalized binary methods. In Section 5, we sketch a new solution for typing binary methods, 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 generalized types*, which corresponds to polynomial types extended with finite powerset, and *(infinitary) generalized types*, which extend the previous class of types with possibly infinitary sums, products and powerset constructors.

Definition 2.1 (Generalized Parameter Types).

- *Finitary Generalized Types* range over the following grammar:

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

  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 $\mathbb{B}$, Integer, denoted by $\mathbb{N}$.
- *(Infinitary) Generalized Types* range over the following grammar:

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

  where I is a possibly infinite set of indices.

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

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

- a finite set of *method declarations*

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

- 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 and function symbols for denoting constructors and methods, and relation symbols for denoting (extensions of) behavioural equivalences at all types. Typical assertions are equations, *e.g.* see (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 either 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. In this perspective, a class is represented by a set (of objects) X; a field f of type T is represented by a function $f : X \to T$; the code corresponding to a constructor declaration $c : \prod_{j=1}^{p} T_j \to X$ is given by a set-theoretic function $\beta : \prod_{j=1}^{p} T_j \to X$, while the code corresponding to a method declaration $m : X \times \prod_{j=1}^{q} T_j \to T_0$ is given by a set-theoretic function $\alpha : X \times \prod_{j}^{q} T_j \to T_0$.

Summarizing:

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

- a set of objects/states X;
- functions $f_i : X \to T_i$ representing fields;
- functions $\beta_i : \prod_{j=1}^{p_i} T_{ij} \to X$, implementing constructors c_i;
- functions $\alpha_i : X \times \prod_{j=1}^{q_i} T_{ij} \to T_{i0}$ 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 *generalized binary methods*, where parameter types are (infinitary) generalized 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 $T_0 \in \mathcal{T}$. Then

- m is *(generalized) binary* if $T_1, \ldots, T_q \in \mathcal{T}$;
- m is *finitary binary* if $T_1, \ldots, T_q \in \mathcal{T}_{\mathcal{F}}$;
- 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.

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

2.1. *Examples*

In Table 1, we present four examples of class specifications, together with 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 it is customary in OO-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 class specification, *Register*, features the simple binary method *eq* for comparing the content 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: variable, application and abstraction, namely

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

for z ranging over an infinite set of variables. Method *isval* tests whether a λ-term converges, *i.e.* it reduces to an abstraction, according to the leftmost strategy. Lazy convergence, denoted by $\Downarrow$, is defined as follows:

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

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

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

Notice 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 couldn't be otherwise, but it makes nevertheless the point.

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

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 of (Reichel(1995); Jacobs(1996)) with an algebra part modelling class constructors. A similar move appears also in (Rothe et al.(2001); Cirstea(2000)).

We start 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 $L : \mathcal{C} \to \mathcal{C}$ 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 $H : \mathcal{C} \to \mathcal{C}$ induced by m as follows:

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

Thus m will induce a H-coalgebra structure on X.

Notice that the functor H is a welldefined 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. In Section 4, we discuss how to overcome this problem. 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 $c_i : \prod_{j=1}^{p_i} T_{ij} \to X$, $i = 1, \dots, h$ and with method declarations $m_i : X \times \prod_{j=1}^{q_i} T_{ij} \to T_{i0}$, $i = 1, \dots, k$, where all methods are unary. The constructor declarations in S induce the functor

```
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 :
        $\mathrm{s.push}(a).\mathrm{top} = a$
        $\mathrm{s.push}(a).\mathrm{pop} \approx \mathrm{s}$
        $\mathrm{s.top} = * \Rightarrow \mathrm{s.pop} \approx \mathrm{s}$
        $\mathrm{new.top} = *$
        $\mathrm{new}(\mathrm{s}, a) \approx \mathrm{s.push}(a)$
end class spec
```

```
class T
    attributes :
        $first : 1 + A$
        $next : T$
    constructors :
        ....
    methods :
        $\mathrm{s.push}(a) = \mathrm{s}'$
                where $\mathrm{s}'.first = a$ and
                      $\mathrm{s}'.next = \mathrm{s}$
        $\mathrm{s.pop} =$ if $\mathrm{s}.first = *$
                   then s
                   else $\mathrm{s}.next$
        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 :
        $\mathrm{r.set}(n).\mathrm{get} = n$
        $\mathrm{r}_1.\mathrm{get} = \mathrm{r}_2.\mathrm{get} \Leftrightarrow$
            $\mathrm{r}_1.\mathrm{eq}(\mathrm{r}_2) = true$
        $\mathrm{new.get} = 0$
end class spec
```

```
class R
    attributes :
        $val : int$
        constructors :
        ....
    methods :
        $\mathrm{r.get} = \mathrm{r}.val$
        $\mathrm{r.set}(n) = \mathrm{r}'$
              where $\mathrm{r}'.val = n$
        $\mathrm{r}_1.\mathrm{eq}(\mathrm{r}_2) =$ if $(\mathrm{r}_1.\mathrm{get} = \mathrm{r}_2.\mathrm{get})$
                        then $true$
                        else  $false$
end class
```

```
class spec : λ-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 :
        $\mathrm{new}(M, N) \approx M.\mathrm{app}(N)$
        $M.\mathrm{isval} = true \Leftrightarrow$
              $\exists z N.\ M \approx new(z, N)$
end class spec
```

```
class Λ
    attributes :
        $term : \lambda\text{-}string$
    constructors :
        ....
    methods :
        $M.\mathrm{isval} =$ if $M.term \Downarrow$
                      then $true$
                      else $false$

        $M.\mathrm{app}(N) = P$
          where $P.term = (M.term)(N.term)$
end class
```

```
class spec : Cell
    constructors :
        new : $N \times N \times State \to X$
    methods :
        $\mathrm{get}_x : X \to N$
        $\mathrm{get}_y : X \to N$
        $\mathrm{set}_{\mathrm{state}} : X \times State \to X$
        $\mathrm{set}_{\mathrm{neighborhood}} : X \times \mathcal{P}(X) \to X$
        $\mathrm{get}_{\mathrm{neighborhood}} : X \to \mathcal{P}(X)$
    assertions :
    ...
end class spec
```

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

Table 1. *Examples of Class Specifications and Classes.*

$L : \mathcal{C} \to \mathcal{C}$ defined by

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

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

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

where $H_i : \mathcal{C} \to \mathcal{C}$ 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 $\beta_i : L_i X \to X$ 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 $\alpha_i : X \to H_i X$ is the function implementing the method m_i, and $\langle\ \rangle$ denotes the standard pairing functor.

Thus, 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.

Notice that, in Definition 3.3 above, classes are taken up to fields, because these are private.

3.1. *Coalgebraic Behavioural Equivalence*

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

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

Definition 3.4 (Relational Lifting). Let R^X be a relation on X, let $T \in \mathcal{T}$ 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 $T = \mathcal{P}(T_1)$, 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 what follows, by abuse of notation, we will often denote the lifted relation R^T simply by R, 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 need at least that the bisimilarity equivalence is a *congruence w.r.t. method application, i.e.:*

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 w.r.t. 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 the way we did, we have that the coalgebraic equivalence in the unary case equates objects with the same behaviour under application of methods:

Proposition 3.1 (Coalgebraic Bisimilarity Equivalence). Let S be a class specification and let $(X, [\beta_i]_{i=1}^h, \langle\alpha_i\rangle_{i=1}^k)$ 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* $\approx_H$, *i.e.* the greatest H-bisimulation on $(X, \langle\alpha_i\rangle_i)$, can be characterized as follows:

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

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. By definition of coalgebraic bisimulation (see Definition A.3 of Appendix A), and by Theorem A.1 of Appendix A. □

Thus we have also:

Theorem 3.1. $\approx_H$ is the *greatest congruence w.r.t.* methods.

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

As we remarked earlier, a strong point 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 be expressed also by saying that a suitable subcoalgebra of the final coalgebra still provides an implementation of the specification, in fact 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 aim, we need that final bialgebras exist, and furthermore that the behavioural equivalence is a congruence also *w.r.t.* constructors. In (Turi et al.(1997); Corradini et al.(2002)), general conditions on categories of bialgebras are studied in order to ensure the above properties.

These results can be extended/adapted also to the collection of functors modelling generalized binary methods considered later in this paper, at least for assertions of a simple equational shape. In this case, if for a given bialgebra satisfying the assertions there is a "tight connection" between the algebraic and the coalgebraic structure, then the corresponding functors admit final bialgebras still satisfying the assertions, and the behavioural equivalence is a congruence *w.r.t.* constructors. Here we do not elaborate more on this issue, but we rather focus on the coalgebraic part, which is the most problematic one.

4. Coalgebraic Description of Generalized 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 (*i.e.* 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 one determined by "freezing" all the occurrences of X in the parameter types and object type, but one. The bunch being obtained after suitable manipulations of the original method. "Freezing" an occurrence of X means that X is replaced by the carrier, *i.e.* 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 generalized 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 *w.r.t.* the original binary methods.

In Section 4.2, we present an alternative solution to the freezing functor, which we call *graph functor*. Here we turn contravariant occurrences in the type of parameters of a generalized binary method m into covariant ones simply by interpreting m as a *graph* instead of a function. To this aim, 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 with respect to the previous one is that this approach directly applies to specifications. The drawback is that the graph bisimilarity equivalence is not a congruence *w.r.t.* method application in general. One may wonder as to why this is the case. There is as yet no general explanation. Often this 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 *w.r.t.* methods. In these cases a rich spectrum of conceptually independent coinduction principles is available. We discuss this issue in Section 4.3, together with the comparison of freezing and graph bisimilarity equivalences.

Throughout this section, let S be a class specification with constructors $c_i : \prod_{j=1}^{p_i} T_{ij} \to X,\ i = 1, \ldots, h$, and methods $m_i : X \times \prod_{j=1}^{q_i} T_{ij} \to T_{i0},\ i = 1, \ldots, k$. Moreover, let L be 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. To this aim, we proceed in two steps.

First, we reduce each binary method m to a bunch of *simple* binary methods with the same observable behaviour of m. To this aim, we proceed by processing parameters of complex types as follows. For each parameter of 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. Each parameter of type $\prod_{i \in I} T_i$ can be viewed as the product of $|I|$ parameters. More subtle is the treatment of parameters of type $\mathcal{P}(T')$. If m has a parameter of type $\mathcal{P}(T')$, *i.e.* $m : X \times \ldots \times \mathcal{P}(T') \times \ldots \to T_0$, then 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 $\mathcal{P}(T')$ disappears, and $m_2 : X \times \ldots \times \mathcal{P}(T'[\bar{X}|X]) \times T' \times \ldots \to T_0$, where we “freeze” the powerset parameter to a constant type and we 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 $m_1,\ m_2$ will be given in Definition 4.1 below).

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, then the number of simple binary methods together with their parameters are finite).

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 l^{th} class parameter.

Formally, step 1 and step 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, which 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:

- if m is *simple binary*, then 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

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

- if m is non-simple generalized binary and its leftmost non-constant parameter different from X is T_i, then
  - if $T_i = \sum_{j=1}^{q_i} T_{ij}$, then $\tau_F(m) = \{m_{ij} \mid j = 1, \ldots, q_i\}$ and $m_{ij} : X \times \ldots \times T_{i-1} \times T_{ij} \times T_{i+1} \times \ldots \times T_q \to T_0$ is defined by $\alpha_{ij}(x)(a_1, \ldots, a_{i-1}, a_{ij}, a_{i+1}, \ldots, a_q) \triangleq \alpha(x)(a_1, \ldots, a_{i-1}, in_j(a_{ij}), \ldots, a_q)$, where $in_j : T_{ij} \to \sum_{j=1}^{q_i} T_{ij}$ is the canonical injection.
  - if $T_i = \prod_{j=1}^{q_i} T_{ij}$, then $\tau_F(m) = \{m'\}$, where $m' : X \times \ldots \times T_{i-1} \times T_{i1} \times \ldots \times T_{iq_i} \times \ldots T_q \to T_0$ is defined by $\alpha'(x)(a_1, \ldots, a_{i-1}, a_{i1}, \ldots, a_{iq_i}, a_{i+1}, \ldots, a_q) \triangleq \alpha(x)(a_1, \ldots, a_{i-1}, \vec{a}_i, a_{i+1}, \ldots a_q)$.
  - if $T_i = \mathcal{P}(T_i')$, then $\tau_F(m) = \{m_1, m_2\}$, where $m_1 : X \times \ldots \times T_{i-1} \times T_{i+1} \times \ldots \times T_q \to T_0$ 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 $m_2 : X \times \ldots \times T_{i-1} \times \mathcal{P}(T_i'[\bar{X}|X]) \times X \times T_{i+1} \times \ldots \times T_q \to T_0$ 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 which 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 same carrier, fields and constructors of C, and with unary methods $\bigcup\{\tau_F^*(m) | m \text{ is a method of } C\}$.

Here 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 $m : X \times \mathcal{P}_f X \to \mathbb{B}$ defined by:*

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

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

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

— *$m_2 : X \times \mathcal{P}_f(\mathbb{N}) \times X \to \mathbb{B}$ 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*

- *$m_2' : X \times \mathcal{P}_f(\mathbb{N}) \times \mathbb{N} \to \mathbb{B}$ is defined by $\alpha_2' = \alpha_2$*
- *$m_2'' : X \times \mathcal{P}_f(\mathbb{N}) \times \mathbb{N} \to \mathbb{B}$ 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 let C^* be the class obtained from the transformation τ^* starting from C. We call *freezing functor* F the functor induced by the methods of C^* according to Definition 3.1, and *freezing equivalence* $\approx_F$ the bisimilarity equivalence induced by this coalgebraic model.

Finally, we are left to establish the result which motivates our treatment, namely, for *finitary* binary methods, we can prove that the freezing bisimilarity equivalence is the greatest congruence *w.r.t.* 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 proof follows from the following facts:

1. $\approx_F$ is the greatest congruence *w.r.t.* methods in C^*.
2. Any equivalence on objects of C is a congruence *w.r.t.* the methods of C *iff* it is a congruence *w.r.t.* the methods of C^* .

Fact 1 above is immediate by Lemma 3.1. In order to prove fact 2 above, it is sufficient to show that an equivalence $\sim$ on a set of objects $\bar{X}$ is a congruence *w.r.t.* a method $m : X \times \prod_{j\in J} T_j \to T_0$ *iff* it is a congruence *w.r.t.* the methods in $\tau_F(m)$. This latter fact is proved 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 *w.r.t.* m, then $\sim$ is immediately a congruence *w.r.t.* the methods in $\tau_F(m)$, by definition. Vice versa, assume that $\sim$ is a congruence *w.r.t.* 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 $n = 0$, then the thesis is immediate from reflexivity of $\sim$. Let us assume that the thesis holds for $n-1$ different parameters. Now 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 parameters. By 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 *w.r.t.* $\tau_F(m)$. Hence, by transitivity of $\sim$, we get the thesis.
Induction step: If the leftmost non-constant parameter in m different from X is of the shape $\sum_{k\in K} T_{i_K}$ or $\prod_{k\in K} T_{i_K}$, then the thesis is immediate from the definition of $\tau_F(m)$ and that of relational lifting. If the leftmost non-constant parameter in m different from X is $T_i = \mathcal{P}_f(T'_i)$, then $\tau_F(m) = \{m_1, m_2\}$, where $m_1 : X \times \dots \times T_{i-1} \times T_{i+1} \times \dots \times T_q \to T_0$, $\;m_2 : X \times \dots \times T_{i-1} \times \mathcal{P}_f(T'_i[\bar{X}|X]) \times X \times T_{i+1} \times \dots \times T_q \to T_0$. Assume that $\sim$ is a congruence *w.r.t.* m, *i.e.*

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

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

Then in particular $\sim$ is a congruence *w.r.t.* both m_1 and m_2.
Vice versa assume that $\sim$ is a congruence *w.r.t.* m_1, m_2, *i.e.*

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

and for all u,

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

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 $u = u' = \phi$ then the thesis follows immediately by (3), if $u = u' \neq \phi$, then the thesis follows by (4).
If $|u \setminus u'| > 0$, then let $y \in u \setminus u'$. Since $u \sim u'$, there exists $y' \in u$ such that $y \sim y'$. By (4) 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 definition of relational lifting, using the fact that $\sim$ is an equivalence, we get $(u \setminus \{y\}) \cup \{y'\} \sim u'$. Then, by induction hypothesis, $\alpha(x')(a'_1, \ldots, a'_{i-1}, (u \setminus \{y\}) \cup \{y'\}, a'_{i+1}, \ldots, a'_q) \sim \alpha(x')(a'_1, \ldots, a'_{i-1}, u', a'_{i+1}, \ldots, a'_q)$.
Then the thesis follows by transitivity of $\sim$. □

As a by-product of Theorem 4.1 above, we get that for finitary binary methods the greatest congruence *w.r.t.* methods always exists, *i.e.*

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 *w.r.t.* the methods of $C\}$ is a congruence.

Notice that, in order to ensure Theorem 4.1 above, it is essential to give a coalgebraic description of binary methods which 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 *e.g.* 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. *Let us consider a class R' of registers containing just a 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 w.r.t. the method m. E.g., 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 since 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 components in the definition of F for the bisimilarity equivalence to be a congruence. An interesting example is that of the λ-calculus, (Honsell et al.(1995)). In this case, the freezing functor with only first component for the method *app* induces the *applicative equivalence* on closed λ-terms. While, if we consider both components in the functor (or only the second one), we get an equivalence which can be viewed as a coinductive characterization 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 aim, *e.g.* see (Honsell et al.(1995)) for more details.

4.1.1. *Infinitary Binary Methods.* Theorem 4.1 above 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 with just one method $m : X \times \mathcal{P}(X) \to \mathbb{B}$ 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 $\sim_k = \{(n,m) | n,m \leq k\} \cup \{(n,n) | n > k\}$ 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 amounts to Example 3.5.10 of (Tews(2002)), page 114.

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 $\forall n. f(n) \leq k$.

One can check that the equivalence $\sim_k = \{(n,m) | n,m \leq k\} \cup \{(n,n) | n > k\}$ 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* a congruence itself, 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, thus any semantics would be problematic. A natural question which arises is whether also the other implication holds, *i.e.* if the union all congruences is a congruence, then the freezing equivalence is a congruence. Somehow surprisingly, this is not the case, the following being a 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 $n \geq k$.

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 in order to capture this.

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

4.2. *The Graph Functor*

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

Contravariant occurrences of the type variable in a generalized 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 $GX \triangleq \mathcal{P}(X \times \mathbb{B})$.

Similarly to 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 $GX \triangleq \mathcal{P}(X \times \mathbb{B}) \times \mathcal{P}(X \times \mathbb{B})$. This works straightforwardly for simple binary methods, but it requires a preliminary transformation for methods with more complex class parameters. This essentially corresponds to the method transformation procedure for the freezing functor, apart from the part which actually freezes the parameters. For dealing with the powerset type constructor, we introduce the new symbol $\mathcal{P}^{\surd}$, which is used to denote a powerset type constructor which has been already processed.

Formally, we define:

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 $\mathcal{P}^{\surd}(T)$, for $T \in \mathcal{T}$.

Definition 4.4 (Graph Method/Class Transformation).
i) Let τ_G be the one-step method transformation function, which takes a generalized binary method $m : X \times \prod_{j=1}^{q} T_j \to T$ implemented by α and produces a set of methods, defined by:

— if m is in normal form, then 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

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

— if m is not in normal form, let T_i be the leftmost parameter not in normal form, then

- if $T_i = \sum_{j=1}^{q_i} T_{ij}$, then $\tau_G(m) = \{m_{ij} \mid j = 1, \ldots, q_i\}$ 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 $\alpha_{ij}(x)(a_1, \ldots, a_{i-1}, a_{ij}, a_{i+1}, \ldots, a_q) \triangleq \alpha(x)(a_1, \ldots a_{i-1},\ in_j(a_{ij}), \ldots a_q)$.
- if $T_i = \prod_{j=1}^{q_i} T_{ij}$, then $\tau_G(m) = \{m'\}$, where $m' : X \times \ldots \times T_{i1} \times \ldots \times T_{iq_i} \times \ldots \to T_0$ is defined by

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

- if $T_i = \mathcal{P}(T'_i)$, then $\tau_G(m) = \{m_1, m_2\}$, where $m_1 : X \times \ldots \times T_{i-1} \times T_{i+1} \times \times \ldots \to T_0$, $\alpha_1(x)(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_q) \triangleq \alpha(x)(a_1, \ldots,\ a_{i-1}, \phi, a_{i+1}, \ldots a_q)$ and $m_2 : \ldots \times \mathcal{P}^{\surd}(T'_i) \times X \times \ldots \to T_0$ 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 which takes a generalized 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 denote by S^* the class specification obtained by applying the transformation τ^*_G to all method declarations in S.
iv) Let C be a class implementation, we denote by C^* the class implementation obtained by applying the transformation τ^*_G to all methods in C.

Notice 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 we 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* $G : \mathcal{C} \to \mathcal{C}$ defined by

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

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

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 not a precise correspondence between classes and coalgebras anymore, since not all G-coalgebras correspond to a class, but only the functional ones, *i.e.* those whose coalgebra map is a function.

The graph bisimilarity equivalence on the objects of C^* can be characterized as follows:

Proposition 4.1 (Graph Bisimilarity Equivalence). Let C be a class, let $G = \prod_{i=1}^k G_i$ be the functor induced by the method declarations in C^*, and let $(X, \langle \alpha_i \rangle_{i=1}^k)$ 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

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

(ii) The graph bisimilarity equivalence $\approx_G$, *i.e.* the greatest G-bisimulation on $(X, \langle \alpha_i \rangle_{i=1}^k)$, can be characterized 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. By definition of coalgebraic bisimulation (see Definition A.3 of Appendix A), and by Theorem A.1 of Appendix A. □

Notice the alternation of quantifiers $\forall\exists$ in the definition of graph bisimulation, due to the presence of the powerset in the graph functor.

The functor G has a final coalgebra, see *e.g.* (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 we lack minimal class implementations, in general. 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 reflexivity of $\approx_F$. □

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

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

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

$$\approx_G\ =\ \approx_F \iff \approx_G \textit{ is a congruence w.r.t. the methods in the class.}$$

Proof. ($\Rightarrow$) By 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 $\approx_G = \approx_F$. $\square$

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:

Corollary 4.2. Let C be a class with finitary generalized 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, raised at the end of Section 4.2.

Theorem 4.2 above is all that we might want. However, in practice, it is useful to have also 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, *i.e.*

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

then $\approx_G\ =\ \approx_F$.

Proof. By Lemma 4.1, $\approx_F \subseteq \approx_G$. Vice versa, we have $\approx_F = (\approx_F)_{|\text{unary methods}} = (\approx_G)_{|\text{ unary methods}} \supseteq \approx_G$. $\square$

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

Remark 4.1. Notice 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. This problem has been addressed in the more general setting of applicative structures in (Honsell et al.(1999)).

An almost trivial, but useful application of Theorem 4.3 is the following:

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

$$\frac{x:\alpha \quad M:\beta}{\lambda x^{\alpha}.M:\alpha\otimes\beta} \qquad \frac{\alpha\leq\alpha' \quad \beta\leq\beta'}{\alpha\otimes\beta\leq\alpha'\otimes\beta'} \qquad \frac{}{\alpha\to\beta\leq\alpha\otimes\beta}$$

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

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 which overcomes the well-known problem arising when inheritance is combined with subtyping, see *e.g.* (Bruce et al.(1995)). If we type binary methods only with the usual arrow type, which is contravariant, we loose the property that subclasses are subtypes. We propose to introduce a new type constructor, *i.e.* 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, contrary to arrow types, we assume relation types not to be "applicable" *i.e.* there is no relational counterpart to the rule : $\frac{M:\alpha\to\beta \quad N:\alpha}{MN:\beta}$. Binary methods can nevertheless be typed also 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, contrary to other proposals, our proposal allows for "future code extensions" without loosing the subtyping property of classes. In this system, type uniqueness fails, of course. 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) generalized 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 which has an extended polynomial type, either m corresponds directly (possibly up-to currying) to a generalized binary method, or m can be cast into a generalized 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 not a generalized 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 this aim, 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

for modelling classes of concrete OO-languages. In particular, it would be interesting to extend the coalgebraic model for Java-like classes with only unary methods, studied in (Honsell et al.(2004)). Imperative OO-languages are more difficult to deal with 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, apparently, it cannot be extended with the (contravariant) arrow type. Namely, there is no “well-behaved” natural extension of the behavioural equivalence to the function type, since the natural definition $R^{T_1 \to T_2} = \{(f, f') \mid \forall x R^{T_1} x'.\ f x R^{T_2} f' x'\}$ of relational lifting fails to preserve equivalence relations, because $R^{T_1 \to T_2}$ is not reflexive, in general. A possible remedy to this problem is that of including $T \to T$ in the *covariant* space of *binary relations*. The difference *w.r.t.* 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 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)).

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

References

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

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

P. Aczel, N. Mendler. A Final Coalgebra Theorem, *CTCS*, D.H.Pitt et al. eds., Springer LNCS **389**, 1989, 357–365.

J. van den Berg, M. Huisman, B. Jacobs, E. Poll. A type-theoretic memory model for verification of sequential Java programs, *WADT'99*, Bert et al. eds., Springer LNCS **1827**, 1999, 1–21.

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

D. Cancila, F. Honsell, M. Lenisa. Some Properties and Some Problems on Set Functors, *CMCS'06*, ENTCS, to appear.

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

C. Cirstea. Integrating observations and computations in the specification of state-based, dynamical systems, PhD thesis, University of Oxford, 2000.

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

J. Goguen, G. Malcolm. A Hidden Agenda, TCS **245**, 2000, 55-101.

R. Hennicker, A. Kurz. (Ω, Ξ)-Logic: On the algebraic extension of coalgebraic specifications, *CMCS'1999*, ENTCS **19**, 1999.

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

F. Honsell, M. Lenisa. Final Semantics for Untyped Lambda Calculus, *TLCA'95*, M.Dezani et al.eds., Springer LNCS **902**, Berlin 1995, 249–265.

F. Honsell, M. Lenisa. Coinductive Characterizations of Applicative Structures, *Math. Struct. in Comp. Science* **9**, 1999.

F. Honsell, M. Lenisa, R. Redamalla. Coalgebraic Semantics and Observational Equivalences of an Imperative Class-based OO-Language, *COMETA'03*, F.Honsell et al. eds., ENTCS, **104**, 2004, 163-180.

M. Huisman. Reasoning about Java programs in Higher-order logic using PVS and Isabelle, Phd thesis, University of Nijmegen, The Netherlands.

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

B. Jacobs. Inheritance and cofree constructions, *ECOOP'96*, P.Cointe ed., Springer LNCS **1098**, 1996, 210–231.

B. Jacobs. Behaviour-refinement of object-oriented specifications with coinductive correctness proofs, *TAPSOFT'97*, M.Bidoit et. al. eds., Springer LNCS **1214**, 1997, 787–802.

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

M. Lenisa. Final Semantics for a Higher Order Concurrent Language, *CAAP'96*, H.Kirchner et. al. eds., LNCS **1059**, 1996, 102–118.

E. Poll, J. Zwanenburg. From algebras and coalgebras to dialgebras, *CMCS'01*, Corradini et al. eds., ENTCS **44**, 2001.

H. Reichel. An approach to object semantics based on terminal co-algebras, *MSCS* **5**, 1995, 129-152.

G. Rosu, J. Goguen. Hidden Congruent Deduction, *FTP'98*, LNAI **1761**, 2000, 252-267.

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

H. Tews. Coalgebraic Methods for Object-Oriented Specifications, Ph.D. thesis, Dresden Univ. of technology, 2002.

D. Turi, G. Plotkin. Towards a mathematical operational semantics, 12^{th} *LICS*, IEEE, Computer Science Press, 1997, 280–291.

Appendix A. Algebraic and Coalgebraic Preliminaries

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

Definition A.1 (L-algebra, H-coalgebra, $\langle L, H\rangle$-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 $\mathcal{C}$ (the *operation* of the algebra). Dually, an H-*coalgebra* is a pair (X, α_X), where $\alpha : X \to HX$. A $\langle L, H\rangle$-*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 structure of category by definining the notions of L-algebra, H-coalgebra, $\langle L, H\rangle$-bialgebra morphism, as follows:

Definition A.2. 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. A function $f : X \to Y$ is an *H-coalgebra morphism* from the H-coalgebra (X, α_X) to the H-coalgebra (Y, α_Y) if it makes diagram (2) commute. A function $f : X \to Y$ is a $\langle L, H\rangle$-bialgebra morphism from (X, β_X, α_X) to (Y, β_Y, α_Y), if f makes both diagrams (1) and (2) commute.

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

Initial morphisms, i.e. algebra morphisms from initial algebras, induce equivalences which are *congruences* w.r.t. algebra operations. Dually, *final morphisms*, i.e. coalgebra morphisms into final coalgebras, induce equivalences which have coinductive characterizations in terms of *bisimulations*.

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

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

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

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

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

where

— $\approx_H$ denotes the greatest H-bisimulation on (X, α_X), and
— $\approx_H = \bigcup\{\mathcal{R} \mid \mathcal{R}$ is a H-bisimulation on $(X, \alpha_X)\}$.

In particular, the following *coinduction principle* holds:

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