Inheritance and Cofree Constructions*

Bart Jacobs
CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands.
Email: bjacobs@cwi.nl

Abstract. *The coalgebraic view on classes and objects is elaborated to include inheritance. Inheritance in coalgebraic specification (of classes) will be understood dually to parametrization in algebraic specification. That is, inheritance involves restriction (specialization), where parametrization involves extension. And cofree constructions are "best" restrictions, like free constructions are "best" extensions. To make this view on inheritance precise we need a suitable notion of behaviour preserving morphism between classes, which will be defined as a "coalgebra map up-to-bisimulation".*

1 Introduction

Two basic relations in object-oriented languages are: object o belongs to class C, and: class C inherits from class C' (see e.g. [23]). Class membership yields what is sometimes called a "first order" classification of objects by classes, whereas inheritance provides a "second order" classification of classes by their superclasses (ancestors). According to Cardelli [2, p. 139]: "... a theory of object-oriented programming should first of all focus on the meaning of inheritance". The first of these relations (class membership) is interpreted in [12] (following [19], and also [10, 11]): briefly, a class is a coalgebra, and an object belonging to a class is an element of the underlying state space of the class, as a coalgebra. This will be used as a basis for an interpretation of the second (inheritance) relation in the present paper: inheritance will involve a behaviour preserving coercion function between classes.

Inheritance in object-oriented programming is used primarily for two purposes: reuse and conceptual modeling (i.e. classification). In the first case inheritance is useful in implementation, and in the second case its advantages come up mainly in design: it allows suitable representations of the data domain, giving the "is-a" relation between classes (see e.g. [22] for an elaborate discussion). We think that inheritance is intuitively a clear and useful notion: for example, it is convenient to have a class of students inheriting from a class of humans, so that all operations acting on humans can directly be applied to students, without reimplementation. And because inheritance is intuitively clear, it should admit a simple set-theoretic semantics (without complicated fixed points, like for example in [23, 3]).

In our approach the aspect of conceptual modeling gets more attention than the aspect of reuse. We make a clear separation between class specifications (also called "abstract" classes) and class implementations (or, "concrete" classes), where the latter are models of the former. We shall put more emphasis on specification, than on actual implementa-

*This paper appeared in: P. Cointe (ed.) *European Conference on Object-Oriented Programming*, Springer LNCS 1098, 1996, p. 210–231.

tion. Class implementations are (non-deferred) classes as used in object-oriented languages. They will be interpreted as so-called coalgebras, consisting of a state space (the interpretation of the class as a type), together with a collection of functions (the interpretation of the methods) acting on the state space. Coalgebras may be understood as general dynamical systems, consisting of a state space with a transition function. Objects belonging to such a class are elements of the state space (i.e. of the carrier of the coalgebra), see [12]. A class implementation gives the method interpretations on a state space, and an object belonging to that class contains specific data values. A class specification gives a behavioural description of classes. The format of class specifications is "coalgebraic", as opposed to the more traditional "algebraic" format (see below).

Two ideas in particular are elaborated in this paper.

(1) In a class specification we distinguish a "core" part and a "definition" part. The definition part may contain definitions of functions (possibly non-unary), in terms of unary methods in the core part. Models of the specification are models of the core part, in which the defined functions receive their interpretation via their definitions and the interpretations of the core part. The definition part does not contribute to the semantics. It may be altered freely in descendants. But the core part may only become more specific in descendants, ensuring monotony. Thus we essentially model what is sometimes called "strict" inheritance, but we do have some flexibility in the definition part.

In fact, the distinction between core and definition part provides a criterion for when it is appropriate to redefine in descendant classes.

(2) Inheritance in coalgebraic specification is similar, but dual, to parametrization in algebraic specification. Both are mechanisms for the stepwise construction of datastructures, but the paradigm for algebraic specification is *extension* (with "unit" morphism as "extension" map), and in coalgebraic specification the paradigm is *restriction* (with "counit" morphism as "restriction" or "coercion" map). Accordingly, one has *free* constructions in algebraic specification where one has *cofree* constructions in coalgebraic specification. We shall use some elementary category theory—involving categories and functors only—to make this duality explicit.

We illustrate this duality between parametrization and inheritance in a simple example, using some ad hoc notation. Consider an algebraic specification NELIST of non-empty lists (of elements of some fixed data set A), as below. It is imported (or, used as a parameter) in a subsequent parametrized specification LIST of possibly empty lists. Coalgebraically we first specify an elementary bank account BANK, and then describe the inheriting specification NBANK with an additional name attribute. The crucial difference between the algebraic and the coalgebraic specification techniques is that in the first case we only have "constructors" pointing into the unknown type X that we are specifying, whereas in the second coalgebraic case we have "destructors" or "observers" pointing out of X (see also the difference between abstract data types and procedural abstraction in [4], and between functional modules and object modules in [9] going back to [8]; the unknown X is a (single) hidden sort in the latter approach). Our use of the terminology of constructors and destructors comes from data type theory, and is different from their use in C++, see [21]. A typical constructor has the form $A \times X \times \cdots \times X \longrightarrow X$ where A is a constant set, whereas typical destructors are $X \longrightarrow A$ and $X \longrightarrow X^B$. The latter can equivalently be written as $X \times B \longrightarrow X$, so that it is also a constructor. Hence constructors and destructors form non-disjoint sets of function symbols.

Here, then, are the specifications: the algebraic ones on the left, and the coalgebraic ones on the right.

Alg spec: NELIST
  operations:
    $\mathsf{el}: A \longrightarrow X$
    $\mathsf{conc}: X \times X \longrightarrow X$
  assertions:
    $\mathsf{conc}(x, \mathsf{conc}(y, z))$
      $= \mathsf{conc}(\mathsf{conc}(x, y), z))$

Coalg spec: BANK
  operations:
    $\mathsf{bal}: X \longrightarrow \mathbb{Z}$
    $\mathsf{ch\_bal}: X \times \mathbb{Z} \longrightarrow X$
  assertions:
    $\mathsf{bal}(\mathsf{ch\_bal}(s, x)) = \mathsf{bal}(s) + x$

Alg spec: LIST
  imports:
    NELIST
  operations:
    $\mathsf{empty}: 1 \longrightarrow X$
  assertions:
    $\mathsf{conc}(x, \mathsf{empty}) = x$
    $\mathsf{conc}(\mathsf{empty}, x) = x$

Coalg spec: NBANK
  imports:
    BANK
  operations:
    $\mathsf{name}: X \longrightarrow \mathsf{String}$
  assertions:
    $\mathsf{name}(\mathsf{ch\_bal}(s, x)) = \mathsf{name}(s)$

A model of such a (algebraic or coalgebraic) specification consists of a "carrier" set $U = [\![X]\!]$ interpreting the type X, together with interpretations of the specified operations (as suitable functions) satisfying the assertions. (In the algebraic case these functions form an *algebra* $T(U) \to U$ on U, and in the coalgebraic case they form a *coalgebra* $U \to S(U)$ on U, for suitable functors $T, S: \mathbf{Sets} \rightrightarrows \mathbf{Sets}$ describing the signatures.)

The import clause in the LIST and NBANK specifications tells us that all the operations and assertions are copied from the imported specification. This means that every model of the LIST specification is also a model of the NELIST specification, and every model of the NBANK specification is also a model of the BANK specification: we have "forget" operations $\mathcal{U}: \mathbf{Models}(\text{LIST}) \to \mathbf{Models}(\text{NELIST})$ and $\mathcal{V}: \mathbf{Models}(\text{NBANK}) \to \mathbf{Models}(\text{BANK})$, which respectively, forget the interpretations of the empty operation, and of the name operation (but keep the carrier sets unaltered). At this point the difference in interpretation of the import clause starts: algebraically one thinks of every non-empty list as a list, whereas coalgebraically every bank account with name is seen as a bank account. Notice the reversal of direction. Thus, (algebraic) parametrization is about *extension*, whereas (coalgebraic) inheritance is about *restriction* (or *specialization*). For example, we can take as model of NELIST the set A^+ of non-empty finite sequences of A's, and as model of LIST the set $A^\star$ of finite sequences of A's, including the empty one. There is then an obvious "extension" map $\eta: A^+ \to \mathcal{U}(A^\star)$, commuting with the interpretations of the NELIST-operations. For the coalgebraic specifications we can take as bank account model the set $\mathbb{Z}$ of integers (with identity as interpretation for bal and addition for $\mathsf{ch\_bal}$). And as model of a bank account with name we can take the set $\mathbb{Z} \times \mathsf{String}$, with obvious interpretations of the operations. There is then a "restriction" or "coercion" map $\varepsilon: \mathcal{V}(\mathbb{Z} \times \mathsf{String}) \to \mathbb{Z}$ given by first projection, which commutes with the interpretations of the BANK-operations.

This difference between parametrization and inheritance results from the difference between the use of constructors in algebraic specification and of destructors in coalgebraic specification. All the constructors of the imported (algebraic) specification also construct elements of the importing specification, so that we have extension. And all destructors (or observers) of the imported (coalgebraic) specification also act on the importing specification, but in this case we have restriction. This difference is crucial.

In the preliminary Sections 2, 3 and 4 we explain the essentials of coalgebraic specification, of free and cofree constructions, and of bisimilarity on classes. The latter means indistinguishability of objects via attributes, and plays an important role for our notion of morphism between classes, involving "coalgebra maps up-to-bisimulation". The rest of this paper is essentially devoted to examples, explaining the coalgebraic view on classes and inheritance. Examples will be given of single inheritance, of multiple inheritance (both with and without common ancestor) and of repeated inheritance. We are not so concerned about specific syntactic details of the language that we use, because we start from a clear semantics, and see language as derived.

2 Class specifications and implementations

In this section we recall the essentials from [12], which forms the basis for what follows. We distinguish between class specifications and class implementations. These class implementations are what are usually simply called classes in object-oriented languages. Class specifications are linguistic entities consisting of three parts describing (1) the methods (operations), (2) the logical assertions which these methods should satisfy, and (3) the conditions which should hold for newly created objects. A class specification may be understood as a class in Eiffel (see [15]) in which all methods (or, features, in Eiffel-speak) are deferred (i.e. not yet interpreted) and in which pre- and post-conditions and invariants specify[1] the behaviour of the methods. In C++ one can also have classes with deferred methods (or, virtual data/member functions, in C++-speak), but assertions do not form part of the language.

As mathematical model of class implementations we use *coalgebras*. These are the formal duals of algebras. They consist of a carrier set (or local state space) U together with a (transition) function $U \to T(U)$ acting on this set U, with as codomain $T(U)$ an expression, possibly containing U, denoting a set. Formally, T is a functor $\mathbf{Sets} \to \mathbf{Sets}$; it describes the signature of function symbols. The state space U gives an interpretation $U = [\![X]\!]$ of the type X occurring in class specifications, and the function $U \to T(U)$ interprets the methods. Objects belonging to a class with operations $U \to T(U)$ are elements $u \in U$ of this state space. An object evaluates a method via function application (to itself). Especially, we require that each class comes with a distinguished element (or initial state) $u_0 \in U$ serving as interpretation of newly created objects. Below we shall use class specifications with methods having one of the following two forms (like in [19]):

$$\mathsf{at}\colon X \longrightarrow A \qquad \text{or} \qquad \mathsf{proc}\colon X \times B \longrightarrow X$$

where A and B are constant sets, not depending on the "unknown" type X (of self). In the first case we have an *attribute* giving for a "local state" $\mathsf{s} \in X$ an (observable) attribute value $\mathsf{s.at} = \mathsf{at}(\mathsf{s}) \in A$. One can only observe the state space X via such attributes. In the second case we have a *procedure* proc which has an effect on the local state space X: it yields for a local state $\mathsf{s} \in X$ and a parameter value $b \in B$ a new state $\mathsf{s.proc}(b) = \mathsf{proc}(\mathsf{s}, b) \in X$. The effect of such a procedure call may be visible via the attributes. Attributes are like instance variables in object-oriented languages; procedures may be used to change the values of these instance variables, see the example below. When the parameter set B is a singleton set $1 = \{*\}$, then we write $X \to X$ instead of $X \times 1 \to X$. Also, B may consist of a product $B_1 \times \cdots \times B_n$. For simplicity we here restrict ourselves to these two forms of methods. Functions $X \times A \to B$ are seen as special instances of attributes using function spaces, in $X \to B^A$. In [12] a more general form of method $X \times A \to B + C \times X$ is used,

[1] Assertions in Eiffel are used not only for specification but also for run-time monitoring.

giving additional expressive power. But this is not needed to describe inheritance, and only distracts from the essentials.

Two methods $X \to A$ and $X \times B \to X$ may be combined into a single "destructor" map $X \to A \times X^B$, giving us a coalgebra on X, pointing *out of* X. Dually, algebras are "constructor" maps of the form $T(X) \to X$ pointing *into* X. Algebraically, one constructs where coalgebraically one observes (or, destructs). See [12] for more details. Multiple attributes $X \longrightarrow A_1, \ldots, X \longrightarrow A_n$ may be combined into a single attribute $X \longrightarrow A_1 \times \cdots \times A_n$. And multiple procedures $X \times B_1 \longrightarrow X, \ldots, X \times B_m \longrightarrow X$ may be combined into a single one $X \times (B_1 + \cdots + B_m) \longrightarrow X$, where $+$ is disjoint union.

A typical example of a class specification is as follows. It describes an unknown type X behaving like a set of locations in a plane.

class spec: LOC
  methods:
    $\mathsf{fst}: X \longrightarrow \mathbb{R}$
    $\mathsf{snd}: X \longrightarrow \mathbb{R}$
    $\mathsf{move}: X \times \mathbb{R} \times \mathbb{R} \longrightarrow X$
  assertions:
    $\mathsf{s.move}(dx, dy).\mathsf{fst} = \mathsf{s.fst} + dx$
    $\mathsf{s.move}(dx, dy).\mathsf{snd} = \mathsf{s.snd} + dy$
  creation:
    $\mathsf{new.fst} = 0$
    $\mathsf{new.snd} = 0$
end class spec

Here we specify classes of locations with first and second coordinate attributes fst and snd yielding real numbers, and with a move procedure yielding a new state. In the assertion clause we have the obvious conditions that after a move with change parameters dx and dy the first coordinate is incremented by dx and the second one by dy. In such specifications we use 's' for 'self' or 'state' as pseudovariable describing an arbitrary inhabitant of X. We shall use the object-oriented dot (.) notation, instead of the functional notation, so that we write $\mathsf{s.move}(dx, dy).\mathsf{fst}$ for what would functionally be written as $\mathsf{fst}(\mathsf{move}(\mathsf{s}, dx, dy))$. Finally in the creation clause we stipulate that newly created objects must have first and second coordinate equal to $0 \in \mathbb{R}$. This is coalgebraic (behavioural) specification since we prescribe nothing about what should be inside the local state space X or about how the methods should be implemented, but only what the observable behaviour should be. Typically, one cannot construct inhabitants of X via methods. This X is best seen as a black box to which we have limited access via the specified methods. In fact, we do not really care about what is inside X as long as X comes with operations as specified. Proper implementation is a local responsibility.

A class (implementation) satisfying such a specification is a (coalgebraic) model of the specification. In the example it consists of an interpretation $U = [\![X]\!]$ of the local state space X, together with interpretations $[\![\mathsf{fst}]\!]: U \to \mathbb{R}$, $[\![\mathsf{snd}]\!]: U \to \mathbb{R}$, $[\![\mathsf{move}]\!]: U \times \mathbb{R} \times \mathbb{R} \to U$ of the methods in such a way that the equations are satisfied. Also a class should contain a distinguished element $u_0 \in U$ satisfying the creation conditions: $[\![\mathsf{fst}]\!](u_0) = 0 = [\![\mathsf{snd}]\!](u_0)$. These interpretations of the methods correspond to a single function $U \to \mathbb{R} \times \mathbb{R} \times U^{(\mathbb{R} \times \mathbb{R})}$ forming a coalgebra of the functor $X \mapsto \mathbb{R} \times \mathbb{R} \times X^{(\mathbb{R} \times \mathbb{R})}$.

Formally, a class (implementation) is a 3-tuple $\langle U, U \to T(U), u_0 \in U \rangle$, consisting of a state space U, a coalgebra $U \to T(U)$ on this set, and an initial state $u_0 \in U$. When part of this structure is understood from the context, we often refer to a class simply by mentioning its state space U.

An obvious example of a class implementation is obtained by taking Cartesian coordinates $[\![X]\!] = \mathbb{R}^2$ as local states, with operations:

$$[\![\mathsf{fst}]\!] = \pi\colon \mathbb{R}^2 \longrightarrow \mathbb{R}, \quad \text{i.e. } (x,y) \mapsto x, \qquad\qquad [\![\mathsf{snd}]\!] = \pi'\colon \mathbb{R}^2 \longrightarrow \mathbb{R} \quad \text{i.e. } (x,y) \mapsto y.$$

And

$$[\![\mathsf{move}]\!]\colon \mathbb{R}^2 \times \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}^2 \qquad \text{is} \qquad (x,y,dx,dy) \mapsto (x+dx, y+dy).$$

Obviously, the assertions in the specification hold for this interpretation. As initial state we take the element $(0,0) \in \mathbb{R}^2$. Another class can be obtained with polar coordinates $[\![X]\!] = [0,\infty) \times [0,2\pi)$, but this complicates the definition of the (interpretations of the) methods. A totally different class implementation has as state space the set $(\mathbb{R}^2)^\star$ of finite sequences of Cartesian coordinates. Such a sequence as object may be seen as the sequence of consecutive changes in the lifetime of the object. We can interpret the operations as:

$$\begin{array}{rcl} [\![\mathsf{fst}]\!]\colon (\mathbb{R}^2)^\star & \longrightarrow & \mathbb{R} \\ ((x_1,y_1),\ldots,(x_n,y_n)) & \longmapsto & x_1 + \cdots + x_n \\ [\![\mathsf{snd}]\!]\colon (\mathbb{R}^2)^\star & \longrightarrow & \mathbb{R} \\ ((x_1,y_1),\ldots,(x_n,y_n)) & \longmapsto & y_1 + \cdots + y_n \\ [\![\mathsf{move}]\!]\colon (\mathbb{R}^2)^\star \times \mathbb{R} \times \mathbb{R} & \longrightarrow & (\mathbb{R}^2)^\star \\ \langle((x_1,y_1),\ldots,(x_n,y_n)), dx, dy\rangle & \longmapsto & ((x_1,y_1),\ldots,(x_n,y_n),(dx,dy)). \end{array}$$

where the latter involves concatenation of the parameter (dx,dy). It is not hard to see that the equations hold in this model. The empty sequence $() \in (\mathbb{R}^2)^\star$ can serve as initial state. But one can also take the singleton sequence $(0,0) \in (\mathbb{R}^2)^\star$ as initial state, or $((0,0),(0,0))$ etcetera. (These are all "bisimilar" (or indistinguishable), see Section 3 below.) Thus we have another example of a class (implementation). Notice that although these three examples give quite different interpretations, a client cannot see these differences, since a client can only use the specified methods. Implementation is not a client's concern. We achieve this encapsulation by separating specification (including the interface) from implementation.

In the remainder of this text we shall omit the interpretation braces $[\![-]\!]$. When we write a method, the context should make clear whether it is meant as a function symbol in some specification, or as an interpretation thereof in some model.

2.1 Class specifications with definitions

We now extend our class specification format with an extra clause for definable functions. This extension does not yet occur in [12]. It will help us avoid some of the anomalies usually associated with inheritance, see [1] for a discussion. Such an extended class specification may contain, besides a "core" part as described above, an additional part describing some function definitions. These functions may have types of the form $X^n \to A$ or $X^n \times B \to X$, for $n \geq 1$, where X is the local state space (the type of self). Notice that these definable functions may thus be binary (or ternary etcetera). But the function definitions may only use the unary methods described in the core specification. This core will determine the meaning of the specification, and within a particular model the definable functions will receive their meaning via their definitions. Thus, in every specific model, we have specific interpretations of the definable functions. For example, we may write a variation LOC+

on the above specification LOC as:

> **class spec:** LOC+
> (methods, assertions and creation as for LOC)
> **definitions:**
> $\mathsf{dist}\colon X \longrightarrow \mathbb{R}$
> $\mathsf{dist(s)} = \mathsf{sqrt}((\mathsf{s.fst})^2 + (\mathsf{s.snd})^2)$
> $\mathsf{eq}\colon X \times X \longrightarrow \mathrm{Bool}$
> $\mathsf{eq}(\mathsf{s}_1, \mathsf{s}_2) = (\mathsf{s}_1.\mathsf{fst} = \mathsf{s}_2.\mathsf{fst}) \wedge (\mathsf{s}_1.\mathsf{snd} = \mathsf{s}_2.\mathsf{snd})$
> **end class spec**

Hence by dist we mean distance to the origin. These defined functions dist and eq do not contribute to the meaning of the specification. Thus any model of the LOC specification is also a model of the LOC+ specification. But in different models the interpretations of dist and eq will be different, as a result of the different interpretations of the fst and snd attributes. For example, in the above LOC model with state space $\mathbb{R}^2$ we have

$$\mathsf{dist}(x, y) = \sqrt{x^2 + y^2}$$

whereas in the LOC model with state space $(\mathbb{R}^2)^\star$ it will be

$$\mathsf{dist}((x_1, y_1), \ldots, (x_n, y_n)) = \sqrt{(x_1 + \cdots + x_n)^2 + (y_1 + \cdots + y_n)^2}.$$

There are similarly different interpretations of the equality function eq, determined by the different interpretations of fst and snd. We shall use the function notation for these definable functions, since for multiple state arguments there is in general no preferred component which should be mentioned first: it seems more natural to write $\mathsf{eq}(\mathsf{s}_1, \mathsf{s}_2)$ than $\mathsf{s}_1.\mathsf{eq}(\mathsf{s}_2)$ or $\mathsf{s}_2.\mathsf{eq}(\mathsf{s}_1)$. For unary methods in the core part, the dot-notation s.method does make sense.

Since these definable functions do not contribute to the meaning of specifications, we may freely alter them in descendants without affecting monotonicity (or "strictness") for the interpretations of the core part. This is the main point of separating the core part and the definition part. The alterations that we allow are removal of definitions and overriding of definitions, for which we shall use ad hoc syntax. An example will be presented in the next section, consisting of a specification of circles inheriting from locations by extension with an extra radius attribute, in the core part. For circles we shall redefine the equality function eq, in the definition part.

Definability is a language dependent notion, but what definability means in a specific programming language will be unproblematic. We shall use elementary language constructs only, meant as illustration.

Within this framework one must choose in advance which methods of a class specifications are essential and belong to the core part, and which to the definition part. But (good) class design is the hardest part of object-oriented programming anyway.

3 Bisimulation and morphisms of classes

Consider the class specification LOC of locations from the previous section, with the implementation (class) on the set $(\mathbb{R}^2)^\star$ of finite sequences of pairs of reals. A client of this class cannot distinguish between the locations $((2,3),(1,1)) \in (\mathbb{R}^2)^\star$ and $((3,0),(0,4)) \in (\mathbb{R}^2)^\star$: in both cases the first coordinate is equal to 3, and the second to 4, and by moving these points around we cannot create a difference between them. These locations (or states) are indistinguishable by the methods in the LOC-specification, and are called **bisimilar**. Here is the general notion. (We can restrict ourselves to class specifications with a single attribute and procedure only, by combination of attributes and methods, as mentioned in Section 2.)

3.1. Definition. Consider a functor $X \mapsto A \times X^B$ and a coalgebra $\varphi = \langle \varphi_1, \varphi_2 \rangle : U \to A \times U^B$ of this functor, giving us interpretations of an attribute φ_1 and a procedure φ_2 acting on a set U.

(i) A **bisimulation** on φ is a relation $R \subseteq U \times U$ on its state space which satisfies for each pair $x, y \in U$:

$$R(x, y) \;\Rightarrow\; \big[\varphi_1(x) = \varphi_1(y) \quad \text{and} \quad \text{for all } b \in B,\; R(\varphi_2(x)(b), \varphi_2(y)(b))\big].$$

(ii) Two elements $x, y \in U$ are called **bisimilar** (with respect to the coalgebra structure φ) if there is a bisimulation $R \subseteq U \times U$ with $R(x, y)$. We then write $x \mathrel{\underline{\leftrightarrow}} y$.

It is not hard to see that bisimilarity $\underline{\leftrightarrow}$ is itself a bisimulation: it is the greatest bisimulation. And it is an equivalence relation, since the identity relation, the opposite $\underline{\leftrightarrow}^{\mathrm{op}}$ and the composite $\underline{\leftrightarrow} \circ \underline{\leftrightarrow}$ are bisimulations, and are thus contained in $\underline{\leftrightarrow}$. Bisimilarity $\underline{\leftrightarrow}$ formalizes behavioural indistinguishability. It is a standard notion in process theory (see e.g. [17]) and in coalgebra.

Bisimilarity on the above LOC-class $(\mathbb{R}^2)^\star$ is given by

$$\begin{array}{c} ((x_1, y_1), \ldots, (x_n, y_n)) \mathrel{\underline{\leftrightarrow}} ((x'_1, y'_1), \ldots, (x'_m, y'_m)) \\ \Leftrightarrow \\ x_1 + \cdots + x_n = x'_1 + \cdots + x'_m \qquad \text{and} \qquad y_1 + \cdots + y_n = y'_1 + \cdots + y'_m. \end{array}$$

States in this relation $\underline{\leftrightarrow}$ are indeed indistinguishable by the LOC-methods. Bisimilarity on the LOC-class $\mathbb{R}^2$ is simply the identity relation. This is because states are simply given by their first and second coordinate. (The class is based on a terminal coalgebra, see [12].)

A client of a class can only see objects (inhabitants of a state space) up-to-bisimulation. This will be reflected in the notion of morphism of classes that we introduce below.

3.2. Definition. Consider a class specification S with its signature of methods described by the functor $\mathcal{S}(X) = A \times X^B$. We define a category **Class**(S) of classes satisfying this specification in the following manner.

objects pairs $\langle U \stackrel{\varphi}{\to} \mathcal{S}(U), u_0 \in U\rangle$ consisting of a coalgebra $\varphi = \langle \varphi_1, \varphi_2 \rangle$ with local state space U, giving an interpretation of the methods in S which satisfies the assertions in S, together with an initial state $u_0 \in U$ satisfying the creation conditions in S.

morphisms $\langle U \stackrel{\varphi}{\to} \mathcal{S}(U), u_0 \in U\rangle \longrightarrow \langle V \stackrel{\psi}{\to} \mathcal{S}(V), v_0 \in V\rangle$ consist of a function $f: U \to V$ between the underlying state spaces satisfying the requirements:

(i) f preserves bisimilarity: $u \mathrel{\underline{\leftrightarrow}} u'$ implies $f(u) \mathrel{\underline{\leftrightarrow}} f(u')$;
(ii) $\psi_1 \circ f = \varphi_1 : U \to A$;
(iii) for each $u \in U$ and $b \in B$ one has $\psi_2(f(u))(b) \mathrel{\underline{\leftrightarrow}} f(\varphi_2(u)(b))$;
(iv) $f(u_0) \mathrel{\underline{\leftrightarrow}} v_0$.

The first condition (i) is actually derivable from (ii) and (iii)—see the lemma below—but is convenient to have explicit in the definition, for example to see that these maps are closed under composition.

What is traditionally called a "morphism of coalgebras" from $U \xrightarrow{\varphi} \mathcal{S}(U)$ to $V \xrightarrow{\psi} \mathcal{S}(V)$ is a function $f: U \to V$ satisfying (ii) as above but (iii) with bisimilarity $\underline{\leftrightarrow}$ replaced by equality $=$. The conditions (ii) and (iii) in this definition describe what may be called a "morphism of coalgebras up-to-bisimulation" (like one has "bisimilarity up-to-bisimulation", see [17]). Since bisimilarity on terminal coalgebras is equality, changing the notion of morphism between coalgebras in this way does not affect terminality.

For example, in the category **Class**(LOC) of classes of the LOC-specification we have morphisms

$$(\mathbb{R}^2)^\star \xrightarrow{\quad f \quad} \mathbb{R}^2 \qquad \text{and} \qquad \mathbb{R}^2 \xrightarrow{\quad g \quad} (\mathbb{R}^2)^\star$$

given by

$$f((x_1, y_1), \ldots, (x_n, y_n)) = (x_1 + \cdots + x_n, y_1 + \cdots + y_n) \quad \text{and} \quad g(x, y) = ((x, 0), (0, y)).$$

We show that g commutes up-to-bisimulation with the move-interpretations:

$$\begin{array}{rcl} \mathsf{move}(g((x,y), dx, dy) & = & \mathsf{move}(((x,0),(0,y)), dx, dy) \\ & = & ((x,0),(0,y),(dx,dy)) \\ & \underline{\leftrightarrow} & ((x+dx,0),(y+dy,0)) \\ & = & g(x+dx, y+dy) \\ & = & g(\mathsf{move}((x,y), dx, xy)). \end{array}$$

3.3. Lemma. *The first condition (i) for morphisms in* **Class**(S) *in Definition 3.2 is derivable from conditions (ii) and (iii).*

Proof. Assume coalgebras φ, ψ as in the definition, and a function $f: U \to V$ between their state spaces, satisfying conditions (ii) and (iii). For an element $u \in U$ and a sequence $\beta \in B^\star$, define $u_\beta \in U$ by induction on the length of β as:

$$u_{\langle\rangle} = u \qquad \text{and} \qquad u_{\beta \cdot b} = \varphi_2(u_\beta)(b).$$

We claim that for $u, u' \in U$ with $u \underline{\leftrightarrow} u'$ and for $\beta \in B^\star$ the following holds.
(a) $u_\beta \underline{\leftrightarrow} u'_\beta$;
(b) $f(u_\beta) \underline{\leftrightarrow} f(u'_\beta)$.
Notice that (b) gives the required result, for $\beta = \langle\rangle$. Statement (a) follows directly by induction on β from the fact that $\underline{\leftrightarrow}$ is itself a bisimulation. For (b) we have to do some work. Define relations $R, S \subseteq V \times V$ by

$$R = \{\langle f(u_\beta), f(u'_\beta)\rangle \mid u, u' \in U \text{ with } u \underline{\leftrightarrow} u', \text{ and } \beta \in B^\star\} \qquad \text{and} \qquad S = \underline{\leftrightarrow} \circ R \circ \underline{\leftrightarrow}\,.$$

Our aim is to show that S is a bisimulation. This yields that R is also a bisimulation (since $\underline{\leftrightarrow}$ is reflexive), and thus that $R \subseteq \underline{\leftrightarrow}$, as required.

Assume therefore $\langle v, v'\rangle \in S$, say with $v \underline{\leftrightarrow} f(u_\beta) R f(u'_\beta) \underline{\leftrightarrow} v'$, where $u \underline{\leftrightarrow} u'$ and $\beta \in B^\star$. Then

- $\psi_1(v) = \psi_1(f(u_\beta)) \overset{\text{(ii)}}{=} \varphi_1(u_\beta) \overset{\text{(a)}}{=} \varphi_1(u'_\beta) \overset{\text{(ii)}}{=} \psi_1(f(u'_\beta)) = \psi_1(v')$.

- $\psi_2(v)(b) \underline{\leftrightarrow} \psi_2(f(u_\beta))(b) \overset{\text{(iii)}}{\underline{\leftrightarrow}} f(\varphi_2(u_\beta)(b)) = f(u_{\beta \cdot b}) R f(u'_{\beta \cdot b}) = f(\varphi_2(u'_\beta)(b)) \overset{\text{(iii)}}{\underline{\leftrightarrow}} \psi_2(f(u'_\beta))(b) \underline{\leftrightarrow} \psi_2(v')(b)$. Hence $\langle \psi_2(v)(b), \psi_2(v')(b)\rangle \in S$. □

The relation R used in this proof is what Milner [17] calls a "bisimulation up-to-bisimilarity", since $\underline{\leftrightarrow} \circ R \circ \underline{\leftrightarrow}$ is a bisimulation.

4 Cofree constructions

"Cofree" constructions are the formal duals of "free" constructions. These free constructions are well-known in mathematics, and also in computer science in the theory of algebraic specifications. The starting point consists of two notions where one naturally gives rise to the other by forgetting part of the structure. As paradigmatic example we take monoids and sets. A monoid consists of a set with a unary and binary operation satisfying some equations. Every monoid gives us a set, simply by forgetting its operations. In this situation we can say that the **free monoid** on a given set A consists of a monoid $(M, u, \cdot)$ together with a "unit" function $\eta\colon A \to M$ such that for every monoid $(N, v, \bullet)$ with a function $f\colon A \to N$ there is a unique homomorphism $g\colon (M, u, \cdot) \to (N, v, \bullet)$ of monoids[2] with $f = g \circ \eta$. This monoid $(M, e, \cdot)$ is called the "free" monoid on A. It can intuitively be understood as the "smallest" monoid which "contains" the set A via $\eta\colon A \to M$. It is the "best possible" monoid into which one can map A. Free monoids on a set exist: it is not hard to see that the set $A^\star$ of finite sequences of elements of A with the empty sequence and concatenation as unary and binary operation, is the free monoid on A. The required unit map $\eta\colon A \to A^\star$ sends an element $a \in A$ to the singleton sequence $\langle a \rangle \in A^\star$.

Free constructions are used in algebraic specification to give meaning to parametrized specifications, see e.g. [5]. For example, consider a specification ABMON of Abelian monoids, with signature $e\colon 1 \longrightarrow X$, $m\colon X \times X \longrightarrow X$ and equations $m(x, e) = x$, $m(x, y) = m(y, x)$, $m(m(x, y), z) = m(x, m(y, z))$. If we now wish to write a specification ABGR of Abelian groups, we can extend the specification of monoids with an extra function symbol $i\colon X \longrightarrow X$ for inverse with equation $m(x, i(x)) = e$. One says that the specification ABGR is parametrized by ABMON. And one thinks of ABGR as an extension of ABMON, which can be expressed formally via an inclusion ABMON $\hookrightarrow$ ABGR of specifications. Semantically, every Abelian group yields an Abelian monoid by forgetting the inverse operation. This gives us a forget operation **Models**(ABGR) $\to$ **Models**(ABMON) induced by the inclusion ABMON $\hookrightarrow$ ABGR. And if we have a model of the ABMON specification, consisting of an Abelian monoid $(M, u, \cdot)$. then the **free Abelian group** on this monoid gives us a canonical model for the specification ABGR. Also this free construction exists, and can be described via a quotient of the free Abelian group on the underlying set, see the "Grothendieck group" example in [14]. One can think of this free construction as adding to the given Abelian monoid as little as necessary to obtain an Abelian group. One does not build an Abelian group from scratch, but one starts from an already given Abelian monoid. Such mechanisms are important in the stepwise construction of (algebraic) data-structures.

The general situation is the following. Suppose we have two categories $\mathbb{C}$ and $\mathbb{D}$ and a forgetful functor $\mathcal{U}\colon \mathbb{C} \to \mathbb{D}$. One can think of $\mathcal{U}$ as the forgetful functor from monoids to sets, or from Abelian groups to Abelian monoids. A **free construction** (also called **universal arrow**) on an object $A \in \mathbb{D}$ (with respect to this functor $\mathcal{U}$) consists of an object $B \in \mathbb{C}$ together with an arrow $\eta\colon A \to \mathcal{U}(B)$ in $\mathbb{D}$ which is universal in the following sense: for each object $B' \in \mathbb{C}$ with a map $f\colon A \to \mathcal{U}(B')$ in $\mathbb{D}$ there is a unique map $g\colon B \to B'$ in $\mathbb{C}$ such that $f = \mathcal{U}(g) \circ \eta$. In a diagram:

$$\text{for} \quad \begin{array}{c} A \\ \big\downarrow f \\ \mathcal{U}(B') \end{array} \quad \text{in } \mathbb{D} \text{ we get} \quad \begin{array}{c} B \\ \big\downarrow g \\ B' \end{array} \quad \text{in } \mathbb{C} \text{ with} \quad \begin{tikzcd} A \arrow[r, "\eta"] \arrow[dr, "f"'] & \mathcal{U}(B) \arrow[d, "\mathcal{U}(g)"] \\ & \mathcal{U}(B') \end{tikzcd} \quad \text{in } \mathbb{D}.$$

[2] This means that g is a function $g\colon M \to N$ between the underlying sets with $g(u) = v$ and $g(x \cdot y) = g(x) \bullet g(y)$.

Such a free construction, if it exists, is determined up-to isomorphism. And if a free construction exists for each object $A \in \mathbb{D}$, then we can define a functor $\mathcal{F}: \mathbb{D} \to \mathbb{C}$, *left* adjoint to the forgetful functor $\mathcal{U}$, see [13, IV] for details.

A **cofree construction** with respect to a functor $\mathcal{U}: \mathbb{C} \to \mathbb{D}$ can now simply be defined by duality as a free construction with respect to the associated functor $\mathcal{U}^{\mathrm{op}}: \mathbb{C}^{\mathrm{op}} \to \mathbb{D}^{\mathrm{op}}$ between opposite categories (with arrows reversed). Explicitly, a cofree construction on an object $A \in \mathbb{D}$ consists of an object $B \in \mathbb{C}$ together with a "counit" arrow $\varepsilon: \mathcal{U}(B) \to A$ in $\mathbb{D}$ which is universal: for every $B' \in \mathbb{C}$ and map $f: \mathcal{U}(B') \to A$ in $\mathbb{D}$ there is a unique map $g: B' \to B$ in $\mathbb{C}$ with $\varepsilon \circ \mathcal{U}(g) = f$, like in:

$$\text{for} \quad \begin{tikzcd} A \\ \mathcal{U}(B') \arrow[u, "f"'] \end{tikzcd} \quad \text{in } \mathbb{D} \text{ there is} \quad \begin{tikzcd} B \\ B' \arrow[u, dashed, "g"'] \end{tikzcd} \quad \text{in } \mathbb{C} \text{ with} \quad \begin{tikzcd} \mathcal{U}(B) \arrow[r, "\varepsilon"] & A \\ \mathcal{U}(B') \arrow[u, "\mathcal{U}(g)"] \arrow[ur, "f"'] & \end{tikzcd} \quad \text{in } \mathbb{D}.$$

Thus every map into A out of an object coming from $\mathbb{C}$ must factor uniquely through the counit ε. If we have such a cofree construction for each object $A \in \mathbb{D}$, then we get a *right* adjoint to the forgetful functor $\mathcal{U}$.

Cofree constructions (right adjoints to forgetful functors) are more rare in mathematics. Here is a simple example. Consider the forgetful functor $\mathcal{U}$: **PreOrd** $\to$ **Sets** from the category of preorders (with monotone functions) to sets. The cofree construction on a set A yields the "indiscrete" preorder $\langle A, A \times A \rangle$ on A, where $A \times A$ is the order relation on A relating all elements. The identity function $\mathcal{U}(A, A \times A) \to A$ is then the universal map ε. As an aside, the *free* construction with respect to this functor assigns to the set A the "discrete" preorder $\langle A, = \rangle$ in which only equal elements are related. Similarly, with respect to the forgetful functor **Top** $\to$ **Sets** from topological spaces to sets, the free construction puts the discrete topology on a set (everything open), and the cofree construction imposes the indiscrete topology (only $\emptyset$ and the set itself are open).

The main point of this paper is that cofree constructions arise naturally in the semantics of inheritance of object-oriented languages. The paradigm underlying inheritance is restriction, instead of extension: groups extend monoids and lorries inherit from vehicles (i.e. form a restricted class of vehicles). This is because the (algebraic) operations for constructing elements of a monoid also yield elements of a group, and dually, the (coalgebraic) operations which act on (or, destruct) vehicles also act on lorries. Free constructions are minimal extensions, and similarly, cofree constructions are minimal restrictions. This minimality of restriction is called "minimal realization", see e.g. [6, 8], but also [7, 5.3].

5 Main definitions, and examples

Class specifications have been introduced above as a means of describing the methods and behaviour of classes (their models, or implementations). We shall now describe inheritance both between class specifications and between class implementations (so that we get "specification and implementation hierarchies", as discussed in [22, 1.1]). A class specification S **inherits from** a class specification T if the text of S mentions "inherits from: T" (instead of the more neutral "imports: T" as used in the introduction). Then it is understood that all the methods, assertions, creation conditions and definitions of T form part of S. But S may contain more, namely:

(1) S may have additional methods.

(2) S may have additional assertions; moreover, the assertions of T may be strengthened.

(3) S may have additional creation conditions; moreover, the creation conditions of T may be strengthened.

(4) The output type A of an attribute $X \longrightarrow A$ in T may be restricted to a subtype $A' \hookrightarrow A$. And the input type B of a procedure $X \times B \longrightarrow X$ in T may be extended to a supertype $B' \hookleftarrow B$.

(5) In the definition section of S, function definitions from T may be removed or redefined, and new function definitions may be added.

These five points ensure that models of the child specification S are also models of the parent specification T. Formally, they ensure that there is a forgetful functor

$$\mathbf{Class}(S) \xrightarrow{\quad \mathcal{F} \quad} \mathbf{Class}(T)$$

between the corresponding categories of classes. This expresses the monotonicity (or strictness) of inheritance.

(We sketch some details of this forgetful functor $\mathcal{F}$. Suppose the specification T has an attribute $X \longrightarrow A_1$ and a procedure $X \times B_1 \longrightarrow X$, so that a model of these methods is a coalgebra $U \to A_1 \times U^{B_1}$ of the functor $\mathcal{T}(X) = A_1 \times X^{B_1}$. Assume the inheriting specification S adds a new attribute $X \longrightarrow A_2$ and procedure $X \times B_2 \longrightarrow X$, and further restricts the attribute of T to $i: A_1' \hookrightarrow A_1$, and extends the input of the procedure of T to $j: B_1 \hookrightarrow B_1'$. The functor associated with S is then $\mathcal{S}(X) = (A_1' \times A_2) \times X^{(B_1'+B_2)}$. It is not hard to see that an $\mathcal{S}$-coalgebra $\varphi = \langle \varphi_1, \varphi_2 \rangle: U \to (A_1' \times A_2) \times U^{(B_1'+B_2)}$ can be mapped to a $\mathcal{T}$-coalgebra, namely to the composite $\mathcal{F}(\varphi) = (i \circ \pi) \times U^{(\mathsf{inl} \circ j)} \circ \varphi = \lambda u \in U. \langle i(\pi\varphi_1(u)), \varphi_2(u)(j(\mathsf{inl}\, b))\rangle: U \to A_1 \times U^{B_1}$. In going from φ to $\mathcal{F}(\varphi)$ the interpretations of the additional attribute and procedure in S are forgotten, and the input and output types are restored. This operation $\varphi \mapsto \mathcal{F}(\varphi)$ yields a functor $\mathbf{Class}(S) \to \mathbf{Class}(T)$ between categories of classes since the assertion and creation conditions in S imply those of T. On morphisms $\mathcal{F}$ is simply the identity.)

Two further remarks are in order. First, the monotonicity mentioned above exists because the function definitions do not contribute to the meaning of classes. Hence one can modify these definitions as one wishes. In fact, from a semantical perspective, the above point (5) is totally irrelevant. We shall see an example in Subsection 5.2. Secondly, in the examples below we shall not see instances of the fourth point. Therefore we can describe inheritance in these examples as an inclusion $T \hookrightarrow S$ of specifications, giving rise to the forgetful functor $\mathcal{F}: \mathbf{Class}(S) \to \mathbf{Class}(T)$.

We have described inheritance between class specifications as a syntactic notation for incremental specification. We now turn to inheritance between class implementations. This will be semantic in nature.

5.1. Definition. Consider a class specification S inheriting from a class specification T as above, together with the resulting forgetful functor

$$\mathbf{Class}(S) \xrightarrow{\quad \mathcal{F} \quad} \mathbf{Class}(T)$$

(i) In this situation we say that a class $B \in \mathbf{Class}(S)$ **inherits from** a class $A \in \mathbf{Class}(T)$ if there is a morphism of classes $f: \mathcal{F}(B) \to A$ in the category $\mathbf{Class}(T)$. This means that the local states of B are mapped by f to the local states of A in such a way that f commutes (up-to-bisimulation) with the interpretations of the methods in T, and preserves the initial state (again, up-to-bisimulation).

We shall then call B a **subclass** of A, and $f: \mathcal{F}(B) \to A$ a **coercion map** (from B to A). This coercion map turns objects of B into objects of A, in such a way that T-behaviour is preserved.

(ii) The **cofree subclass** on $A \in \mathbf{Class}(T)$ is the cofree construction on A with respect to the forgetful functor $\mathcal{F}$. It consists of a subclass B with a universal coercion $\varepsilon: \mathcal{F}(B) \to A$: for each subclass B' with coercion $f: \mathcal{F}(B') \to A$ there is a unique map $g: B' \dashrightarrow B$ of classes with $\varepsilon \circ \mathcal{F}(g) = f$.

The intuition is that the cofree subclass on A is the "best possible" implementation of S, starting from the already given implementation A of the parent T.

The following result asserts that if class a B inherits from a class A, then, elements of B with the same B-behaviour, have, in A, the same A-behaviour. Thus, objects with are indistinguishable in a subclass are also indistinguishable in the parent. This is because morphisms of classes preserve behaviour.

5.2. Lemma. *Let $B \in \mathbf{Class}(S)$ inherit from $A \in \mathbf{Class}(T)$, say via $f: \mathcal{F}(B) \to A$ as above. Then*

$$x \underline{\leftrightarrow}_B \, y \;\Rightarrow\; f(x) \underline{\leftrightarrow}_A \, f(y).$$

Proof. It is not hard to see that the composite relation

$$R = \underline{\leftrightarrow}_A \circ \, \{\langle f(x), f(y)\rangle \mid x \underline{\leftrightarrow}_B \, y\} \circ \underline{\leftrightarrow}_A$$

is a bisimulation on A. Hence $R \subseteq \underline{\leftrightarrow}_A$, and thus $\{\langle f(x), f(y)\rangle \mid x \underline{\leftrightarrow}_B \, y\} \subseteq \underline{\leftrightarrow}_A$, as required. $\square$

The rest of this paper is devoted to examples illustrating these concepts for toy class specifications. With multiple and repeated inheritance one does not have one class (specification) inheriting from another, so a slightly different functor $\mathcal{F}$ will be used. But the main points of the definition remain the same.

5.1 Single inheritance, without definitions

We shall elaborate the bank account example from the introduction. We first specify classes of elementary bank accounts with a balance attribute, and a change procedure (using the object-oriented dot notation, instead of the functional notation as in the introduction). Then we extend this specification with a name attribute, together with an associated procedure for setting the name (of the holder of the bank account; note that such a name may change—e.g. through marriage).

class spec: BANK
    methods:
        $\mathsf{bal}: X \longrightarrow \mathbb{Z}$
        $\mathsf{ch\_bal}: X \times \mathbb{Z} \longrightarrow X$
    assertions:
        $\mathsf{s.ch\_bal}(x)\mathsf{.bal} = \mathsf{s.bal} + x$
    creation:
        $\mathsf{new.bal} = 0$
end class spec

class spec: NBANK
    inherits from:
        BANK
    methods:
        $\mathsf{name}: X \longrightarrow \mathsf{String}$
        $\mathsf{ch\_name}: X \times \mathsf{String} \longrightarrow X$
    assertions:
        $\mathsf{s.ch\_bal}(x)\mathsf{.name} = \mathsf{s.name}$
        $\mathsf{s.ch\_name}(y)\mathsf{.bal} = \mathsf{s.bal}$
        $\mathsf{s.ch\_name}(y)\mathsf{.name} = y$
    creation:
        $\mathsf{new.name} =$ ""
end class spec

where “” is the empty string. The idea is that the specification BANK is extended with an additional attribute name and procedure $\mathsf{ch\_name}$ for telling and changing the name. Thus NBANK contains all the methods of BANK. Also the specification NBANK is extended with some extra assertions and conditions at creation. The first two assertions tell us that by changing the balance the name does not change, and by changing the name the balance remains the same. These assertions make sure that after a change of name we still have a balance, and that after changing the balance we still have a name. This corresponds to what is called “capture” in [18].

Let us now assume that we have a class implementation $A \in \mathbf{Class}(\text{BANK})$ of this specification BANK with as state space the set $\mathbb{Z}^\star$ of finite sequences of integers. The “balance” and “change-balance” operations of A are interpreted as:

$$\begin{cases} \mathsf{bal}\colon \mathbb{Z}^\star \longrightarrow \mathbb{Z} & \text{is} \quad (x_1, \ldots, x_n) \mapsto x_1 + \cdots + x_n \\ \mathsf{ch\_bal}\colon \mathbb{Z}^\star \times \mathbb{Z} \longrightarrow \mathbb{Z}^\star & \text{is} \quad \langle (x_1, \ldots, x_n), x\rangle \mapsto (x_1, \ldots, x_n, x). \end{cases}$$

As initial state of A we take the empty sequence $() \in \mathbb{Z}^\star$.

The cofree subclass B on A gives the most efficient implementation of the extended specification NBANK, given the implementation A of the parent BANK. Its state space simply has an extra string field with respect to A, to accomodate for the extra name information. That is, the state space of B is $\mathbb{Z}^\star \times \mathsf{String}$ with operations

$$\begin{array}{cc} \mathsf{bal}((x_1, \ldots, x_n), \alpha) = x_1 + \cdots + x_n, & \mathsf{ch\_bal}((x_1, \ldots, x_n), \alpha, x) = ((x_1, \ldots, x_n, x), \alpha), \\ \mathsf{name}((x_1, \ldots, x_n), \alpha) = \alpha, & \mathsf{ch\_name}((x_1, \ldots, x_n), \alpha, \beta) = ((x_1, \ldots, x_n), \beta) \end{array}$$

The initial state of B is $(\langle\rangle, \text{“”}) \in \mathbb{Z}^\star \times \mathsf{String}$. The first projection $\pi\colon \mathbb{Z}^\star \times \mathsf{String} \to \mathbb{Z}^\star$ is the appropriate universal coercion map from B to A. This will be shown in some detail.

First, we have that bisimilarity on $\mathbb{Z}^\star$ is given by

$$\begin{aligned} (x_1, \ldots, x_n) \underline{\leftrightarrow} (y_1, \ldots, y_m) \quad &\Leftrightarrow \quad \mathsf{bal}(x_1, \ldots, x_n) = \mathsf{bal}(y_1, \ldots, y_m) \\ &\Leftrightarrow \quad x_1 + \cdots + x_n = y_1 + \cdots + y_m. \end{aligned}$$

And similarly bisimilarity on $\mathbb{Z}^\star \times \mathsf{String}$ is

$$\langle (x_1, \ldots, x_n), \alpha\rangle \underline{\leftrightarrow} \langle (y_1, \ldots, y_m), \beta\rangle \;\Leftrightarrow\; (x_1 + \cdots + x_n = y_1 + \cdots + y_m) \wedge (\alpha = \beta).$$

It is then not hard to check that the first projection $\pi\colon \mathbb{Z}^\star \times \mathsf{String} \to \mathbb{Z}^\star$ is a morphism $\mathcal{F}(B) \to A$ in the category $\mathbf{Class}(\text{BANK})$. That is, $\mathsf{bal} \circ \pi = \mathsf{bal}$, $\mathsf{ch\_bal} \circ \pi \times id \underline{\leftrightarrow} \pi \circ \mathsf{ch\_bal}$ (pointwise), and $\pi(\langle\rangle, \text{“”}) \underline{\leftrightarrow} \langle\rangle$.

If we assume another class $C \in \mathbf{Class}(\text{NBANK})$ implementing a bank account with name, together with a morphism $f\colon \mathcal{F}(C) \to A$ in $\mathbf{Class}(\text{BANK})$, then we get a map $g = \langle f, \mathsf{name}\rangle\colon C \to \mathbb{Z}^\star \times \mathsf{String}$. We shall show that g is a morphism of classes $C \to B$ in $\mathbf{Class}(\text{NBANK})$ by checking conditions (ii)–(iv) in Definition 3.2.

(ii) We have $(\mathsf{bal} \circ g)(c) = \mathsf{bal}(f(c), \mathsf{name}(c)) = \mathsf{bal}(f(c)) = \mathsf{bal}(c)$ since f commutes with the BANK-operations. And $(\mathsf{name} \circ g)(c) = \mathsf{name}(f(c), \mathsf{name}(c)) = \mathsf{name}(c)$. Hence g commutes with the NBANK-attributes.

(iii) With respect to the procedures, we compute:

$$\begin{aligned} (\mathsf{ch\_bal} \circ g \times id)(c, x) \quad &= \quad \mathsf{ch\_bal}(f(c), \mathsf{name}(c), x) \\ &= \quad (\mathsf{ch\_bal}(f(c), x), \mathsf{name}(c)) \\ &\underline{\leftrightarrow} \quad (f(\mathsf{ch\_bal}(c, x)), \mathsf{name}(\mathsf{ch\_bal}(c, x))) \\ &= \quad (g \circ \mathsf{ch\_bal})(c, x). \end{aligned}$$

For commutation of the function g with the "change-name" procedure we first have to establish that $f(\mathsf{ch\_name}(c,\beta)) \underline{\leftrightarrow} f(c)$ in $\mathbb{Z}^\star$. This follows from

$$\mathsf{bal}(f(\mathsf{ch\_name}(c,\beta))) = \mathsf{bal}(\mathsf{ch\_name}(c,\beta)) = \mathsf{bal}(c) = \mathsf{bal}(f(c)).$$

Now we get

$$\begin{array}{rcl}(\mathsf{ch\_name} \circ g \times id)(c,\beta) & = & \mathsf{ch\_name}(f(c), \mathsf{name}(c), \beta) \\ & = & (f(c), \beta) \\ & \underline{\leftrightarrow} & (f(\mathsf{ch\_name}(c,\beta)), \mathsf{name}(\mathsf{ch\_name}(c,\beta))) \\ & = & (g \circ \mathsf{ch\_name})(c,\beta).\end{array}$$

(iv) Finally, the initial state is preserved: $g(c_0) = (f(c_0), \mathsf{name}(c_0)) \underline{\leftrightarrow} (\langle\rangle, \text{``''})$, since $f(c_0) \underline{\leftrightarrow} \langle\rangle$.

Obviously $\pi \circ g = f$. And if there is another morphism of classes $h\colon C \to \mathbb{Z}^\star \times \mathsf{String}$ with $\pi \circ h = f$, then $\pi' \circ h = \mathsf{name} \circ h = \mathsf{name}$, so that $h = \langle \pi \circ h, \pi' \circ h\rangle = \langle f, \mathsf{name}\rangle = g$. This concludes the argument.

At the end of this subsection we notice how code is reused under inheritance: the implementations of the operations in the base class A are wrapped inside the descendant class B, where one has an extra field. In this way there is no coercion necessary when one calls a method from the parent class for an object of the child class.

5.2 Single inheritance, with definitions

We shall describe an example of inheritance between class specifications with definitions (see Subsection 2.1). We will start from the class specification LOC+ of locations with defined functions dist and eq, and extend the specification with an extra radius attribute so that we can describe circles (like in [3]). We keep the dist definition as it is, so that the distance of a circle to the origin is the distance of its center to the origin, and redefine the equality function; further, we add two new function definitions perim and surf for the perimeter and surface of a circle.

class spec: CIRC
- **inherits from:**
  - LOC+
- **methods:**
  - $\mathsf{rad}\colon X \longrightarrow \mathbb{R}_{\geq 0}$
  - $\mathsf{magn}\colon X \times \mathbb{R}_{\geq 0} \longrightarrow X$
- **assertions:**
  - $\mathsf{s.move}(dx, dy).\mathsf{rad} = \mathsf{s.rad}$
  - $\mathsf{s.magn}(a).\mathsf{fst} = \mathsf{s.fst}$
  - $\mathsf{s.magn}(a).\mathsf{snd} = \mathsf{s.snd}$
  - $\mathsf{s.magn}(a).\mathsf{rad} = a \cdot (\mathsf{s.rad})$
- **creation:**
  - $\mathsf{new.rad} = 1$
- **definitions:**
  - $\mathsf{perim}\colon X \longrightarrow \mathbb{R}_{\geq 0}$
    - $\mathsf{perim(s)} = 2 \cdot \pi \cdot (\mathsf{s.rad})$
  - $\mathsf{surf}\colon X \longrightarrow \mathbb{R}_{\geq 0}$
    - $\mathsf{surf(s)} = \pi \cdot (\mathsf{s.rad})^2$

redefine:

$\mathsf{eq}: X \times X \longrightarrow \mathrm{Bool}$

$\mathsf{eq}(\mathsf{s}_1, \mathsf{s}_2) = (\mathsf{s}_1.\mathsf{fst} = \mathsf{s}_2.\mathsf{fst}) \wedge (\mathsf{s}_1.\mathsf{snd} = \mathsf{s}_2.\mathsf{snd}) \wedge (\mathsf{s}_1.\mathsf{rad} = \mathsf{s}_2.\mathsf{rad})$

end class spec

Hence the magn procedure magnifies the radius of the circle by a certain factor, which is given as parameter. A class implementation (model) of this specification CIRC is an implementation of the core part of the specification (the part without the definitions). It consists of a model of the LOC-specification for which we have additional radius and magnification operations satisfying the above assertions. We thus have a forgetful functor

$$\mathbf{Class}(\mathrm{CIRC}) \xrightarrow{\;\;\mathcal{F}\;\;} \mathbf{Class}(\mathrm{LOC+}) = \mathbf{Class}(\mathrm{LOC})$$

so that a class $B \in \mathbf{Class}(\mathrm{CIRC})$ inherits from $A \in \mathbf{Class}(\mathrm{LOC})$ if there is a map of classes $\mathcal{F}(B) \to A$. For example, taking A to be the class of locations on $\mathbb{R}^2$, the cofree subclass of circles on A has $\mathbb{R}^2 \times \mathbb{R}_{\geq 0}$ as state space with operations

$$\mathsf{fst}(x, y, z) = x, \qquad \mathsf{snd}(x, y, z) = y, \qquad \mathsf{rad}(x, y, z) = z,$$
$$\mathsf{move}(x, y, z, dx, dy) = (x + dx, y + dy, z), \qquad \mathsf{magn}(x, y, z, a) = (x, y, a \cdot z),$$

and $(0, 0, 1) \in \mathbb{R}^2 \times \mathbb{R}_{\geq 0}$ as initial state. In this class the defined functions of CIRC take the form

$$\mathsf{dist}(x, y, z) = \sqrt{x^2 + y^2}, \qquad \mathsf{perim}(x, y, z) = 2 \cdot \pi \cdot z$$
$$\mathsf{surf}(x, y, z) = \pi \cdot z^2, \qquad \mathsf{eq}((x, y, z), (x', y', z')) = (x = x') \wedge (y = y') \wedge (z = z').$$

There is an obvious coercion map $\varepsilon: \mathbb{R}^2 \times \mathbb{R}_{\geq 0} \to \mathbb{R}^2$, namely $\varepsilon(x, y, z) = (x, y)$. It commutes with the (core) LOC-methods, but not with the defined functions, since we have separate equality functions for locations and for circles. We further stipulate (operationally) that for a location s and a circle t the expressions $\mathsf{eq}(\mathsf{s}, \mathsf{t})$ and $\mathsf{eq}(\mathsf{t}, \mathsf{s})$ will result in calling the equality function for locations. Thus, in the mixed case a coercion to the ancestor class takes place. Denotationally, this requires the composite functions

$$\begin{tikzcd}
\mathbb{R}^2 \times (\mathbb{R}^2 \times \mathbb{R}_{\geq 0}) \arrow[dr, "id \times \varepsilon"] & & \\
& \mathbb{R}^2 \times \mathbb{R}^2 \arrow[r, "\mathsf{eq}"] & \mathrm{Bool} \\
(\mathbb{R}^2 \times \mathbb{R}_{\geq 0}) \times \mathbb{R}^2 \arrow[ur, "\varepsilon \times id"'] & &
\end{tikzcd}$$

Due to our restriction that redefinition can only be applied to functions in the definition clause of a specification, certain inappropriate (non-monotonic) uses of inheritance are excluded under this coalgebraic interpretation. For example, if the core part of a specification contains certain methods which are characteristic for fish, then we can never get a subclass of birds by redefinition.

Since these definable functions are peripheral and present no complications in our description of inheritance, they will be omitted from the examples below.

5.3 Multiple inheritance, without common ancestor

Multiple inheritance means inheritance with multiple ancestors. It exists in Eiffel and in C++, but not in Smalltalk. We shall present an example in which we combine a class specification of flip-flops with the earlier class specification of locations in a class specification of flip-flops on location:

$$
\begin{tikzcd}
\text{flip-flops (FF)} & & \text{locations (LOC)} \\
& \text{flip-flops on location (FF\_on\_LOC)} \arrow[ul] \arrow[ur] &
\end{tikzcd}
$$

Such "flip-flops on location" may be used as movable pixels on a black-and-white screen.

The class specification LOC of locations is as in Section 2. The specifications FF of flip-flops and FF_on_LOC of flip-flops on locations will be given below:

class spec: FF
  methods:
    val: $X \longrightarrow \{0,1\}$
    on: $X \longrightarrow X$
    off: $X \longrightarrow X$
  assertions:
    s.on.val $= 1$
    s.off.val $= 0$
  creation:
    new.val $= 0$
end class spec

class spec: FF_on_LOC
  inherits from:
    FF
    LOC
  assertions:
    s.move(dx, dy).val = s.val
    s.on.fst = s.fst
    s.on.snd = s.snd
    s.off.fst = s.fst
    s.off.snd = s.snd
end class spec

In the class specification FF_on_LOC we do not add any new methods: we only inherit the methods from both the two parent classes FF and LOC, and specify how the attributes of the one act on the procedures of the other. There is no need to further specify the initial state. This gives us an example of multiple inheritance without common ancestors, because the class specifications FF and LOC do not have a specification from which they both inherit.

The idea is that a class implementing the FF_on_LOC specification implements both the specifications FF and LOC and additionally satisfies the conditions mentioned in FF_on_LOC. In this situation we have two forgetful functors $\mathcal{F}_1 \colon \mathbf{Class}(\text{FF\_on\_LOC}) \to \mathbf{Class}(\text{FF})$ and $\mathcal{F}_2 \colon \mathbf{Class}(\text{FF\_on\_LOC}) \to \mathbf{Class}(\text{LOC})$. They can be combined into a single functor

$$\mathbf{Class}(\text{FF\_on\_LOC}) \xrightarrow{\mathcal{F} = \langle \mathcal{F}_1, \mathcal{F}_2 \rangle} \mathbf{Class}(\text{FF}) \times \mathbf{Class}(\text{LOC})$$

Inheritance and cofreeness will be described with respect to this forgetful functor $\mathcal{F} = \langle \mathcal{F}_1, \mathcal{F}_2 \rangle$. We can say that a class $B \in \mathbf{Class}(\text{FF\_on\_LOC})$ inherits from $A_1 \in \mathbf{Class}(\text{FF})$ and $A_2 \in \mathbf{Class}(\text{LOC})$ if there are maps of classes $\mathcal{F}_1(B) \to A_1$ and $\mathcal{F}_2(B) \to A_2$—or equivalently, if there is a single map $\mathcal{F}(B) \to (A_1, A_2)$.

An obvious class implementation A_1 of the flip-flop specification FF is obtained by taking the set $\{0,1\}$ of attribute values as state space. The val attribute $\{0,1\} \to \{0,1\}$ is then simply the identity functor. The on and off procedures are interpreted as the functions $\{0,1\} \rightrightarrows \{0,1\}$ given by $\mathsf{on}(x) = 1$ and $\mathsf{off}(x) = 0$. As initial state we take of course $0 \in \{0,1\}$. (Instead of $\{0,1\}$ one can take any set with at least two elements as state space.)

As class implementation A_2 of the locations specification LOC we choose the one from Section 2 with $(\mathbb{R}^2)^\star$ as state space. This gives us a pair of classes $(A_1, A_2) \in \mathbf{Class}(\mathrm{FF}) \times \mathbf{Class}(\mathrm{LOC})$. We claim that the cofree construction on (A_1, A_2) gives us a class with state space $\{0,1\} \times (\mathbb{R}^2)^\star$ and with operations:

$$\begin{array}{rclrcl} \mathsf{val}(z,\alpha) & = & z & \mathsf{move}(z,\alpha,dx,dy) & = & (z, \alpha \cdot (dx,dy)) \\ \mathsf{on}(z,\alpha) & = & (1,\alpha) & \mathsf{fst}(z,((x_1,y_1),\ldots,(x_n,y_n))) & = & x_1 + \cdots + x_n \\ \mathsf{off}(z,\alpha) & = & (0,\alpha) & \mathsf{snd}(z,((x_1,y_1),\ldots,(x_n,y_n))) & = & y_1 + \cdots + y_n. \end{array}$$

where $\alpha \in (\mathbb{R}^2)^\star$ and $\alpha \cdot (dx, dy)$ is the result of concatenating (dx, dy) at the end of α. There are obvious coercion maps $\{0,1\} \times (\mathbb{R}^2)^\star \to \{0,1\}$ and $\{0,1\} \times (\mathbb{R}^2)^\star \to (\mathbb{R}^2)^\star$ given by first and second projection. For any class $B \in \mathbf{Class}(\mathrm{FF\_on\_LOC})$ with coercion maps $f_1 : \mathcal{F}_1(B) \to A_1$ and $f_2 : \mathcal{F}_2(B) \to A_2$ we get a unique map of classes $B \dashrightarrow \{0,1\} \times (\mathbb{R}^2)^\star$, namely the tuple $\langle f_1, f_2 \rangle$.

5.4 Multiple inheritance, with common ancestor

We slightly modify the flip-flops on location from the previous subsection to flip-flops on circles in a situation:

$$\begin{tikzcd} & \mathrm{LOC} & \\ \mathrm{FF\_on\_LOC} \arrow[ur] & & \mathrm{CIRC} \arrow[ul] \\ & \mathrm{FF\_on\_CIRC} \arrow[ul] \arrow[ur] & \end{tikzcd}$$

The extra 0/1 information on circles may be used to indicate whether a circle is filled (i.e. a disk) or open (e.g. when displayed).

The specification FF_on_LOC for flip-flops on circles is as follows.

class spec: FF_on_CIRC
  inherits from:
    FF_on_LOC
    CIRC
  assertions:
    s.on.rad = s.rad
    s.off.rad = s.rad
    s.magn(a).val = s.val
end class spec

The set of methods in this specification FF_on_CIRC is the (ordinary, non-disjoint) union of the sets of methods in FF_on_LOC and in CIRC. A model (class implementation) of flip-flops on circles is thus at the same time a model of flip-flops on locations and of circles, and the underlying model of locations is the same. This means that we have the following commuting diagram of forgetful functors between the categories of classes of these specifications.

$$\begin{tikzcd}[column sep=small]
 & \mathbf{Class}(\mathrm{LOC}) & \\
\mathbf{Class}(\mathrm{FF_on_LOC}) \arrow[ur, "\mathcal{H}_1"] & & \mathbf{Class}(\mathrm{CIRC}) \arrow[ul, "\mathcal{H}_2"'] \\
 & \mathbf{Class}(\mathrm{FF_on_CIRC}) \arrow[ul, "\mathcal{F}_1"] \arrow[ur, "\mathcal{F}_2"'] &
\end{tikzcd}$$

where $\mathcal{H}_1 \circ \mathcal{F}_1 = \mathcal{H}_2 \circ \mathcal{F}_2 = \mathcal{K}$, say. Then we can form the comma category $(\mathcal{H}_1 \times \mathcal{H}_2 \downarrow \Delta)$ of the two functors

$$\begin{tikzcd}[column sep=small]
\mathbf{Class}(\mathrm{FF_on_LOC}) \times \mathbf{Class}(\mathrm{CIRC}) \arrow[dr, "\mathcal{H}_1 \times \mathcal{H}_2"'] & & \mathbf{Class}(\mathrm{LOC}) \arrow[dl, "\Delta = \langle id, id \rangle"] \\
 & \mathbf{Class}(\mathrm{LOC}) \times \mathbf{Class}(\mathrm{LOC}) &
\end{tikzcd}$$

(see [13]), and define a functor

$$\mathbf{Class}(\mathrm{FF\_on\_CIRC}) \xrightarrow{\quad \mathcal{F} \quad} (\mathcal{H}_1 \times \mathcal{H}_2 \downarrow \Delta)$$

which send a class $B \in \mathbf{Class}(\mathrm{FF\_on\_CIRC})$ to the pair of identities

$$\begin{tikzcd}[column sep=small]
\mathcal{H}_1(\mathcal{F}_1(B)) \arrow[dr, equal] & & \mathcal{H}_2(\mathcal{F}_2(B)) \arrow[dl, equal] \\
 & \mathcal{K}(B) &
\end{tikzcd}$$

We shall describe inheritance and cofreeness with respect to this functor $\mathcal{F}$.

Assume classes $A_1 \in \mathbf{Class}(\mathrm{FF\_on\_LOC})$ and $A_2 \in \mathbf{Class}(\mathrm{CIRC})$ with a common ancestor class $A \in \mathbf{Class}(\mathrm{LOC})$ via coercions $f_1 : \mathcal{H}_1(A_1) \to A$ and $f_2 : \mathcal{H}_2(A_2) \to A$. We say that $B \in \mathbf{Class}(\mathrm{FF\_on\_CIRC})$ inherits from $\mathcal{H}_1(A_1) \xrightarrow{f_1} A \xleftarrow{f_2} \mathcal{H}_2(A_2)$ if there is a morphism

$$\mathcal{F}(B) \longrightarrow \left(\begin{tikzcd}[column sep=small] \mathcal{H}_1(A_1) \arrow[dr, "f_1"'] & & \mathcal{H}_2(A_2) \arrow[dl, "f_2"] \\ & A & \end{tikzcd} \right) \qquad \text{in the category } (\mathcal{H}_1 \times \mathcal{H}_2 \downarrow \Delta)$$

consisting of coercion maps $g_1 : \mathcal{F}_1(B) \to A_1$ and $g_2 : \mathcal{F}_2(B) \to A_2$ with $f_1 \circ \mathcal{H}_1(g_1) = f_2 \circ \mathcal{H}_2(g_2)$. And this B is the cofree subclass inheriting from $\mathcal{H}_1(A_1) \to A \leftarrow \mathcal{H}_2(A_2)$ if every such subclass $B' \in \mathbf{Class}(\mathrm{FF\_on\_CIRC})$ is a subclass of B via a unique morphism $B' \dashrightarrow B$ making appropriate diagrams commute.

We present one example. Assume we have implementations of FF_on_LOC on $\{0,1\} \times \mathbb{R}^2$, and of CIRC on $\mathbb{R}^2 \times \mathbb{R}_{\geq 0}$, with $\mathbb{R}^2$ as common implementation of the specification LOC of locations, via projection morphisms

$$\{0,1\} \times \mathbb{R}^2 \xrightarrow{\quad \pi' \quad} \mathbb{R}^2 \xleftarrow{\quad \pi \quad} \mathbb{R}^2 \times \mathbb{R}_{\geq 0}$$

Then the cofree subclass on these data has as state space the set $\{0,1\} \times \mathbb{R}^2 \times \mathbb{R}_{\geq 0}$. The definition of the operations on this state space is left to the reader. We only mention that there is an obvious commuting square of coercion maps:

$$\begin{tikzcd}[column sep=small] & \mathbb{R}^2 & \\ \{0,1\}\times\mathbb{R}^2 \arrow[ur, "\pi'"] & & \mathbb{R}^2\times\mathbb{R}_{\geq 0} \arrow[ul, "\pi"'] \\ & \{0,1\}\times\mathbb{R}^2\times\mathbb{R}_{\geq 0} \arrow[ul] \arrow[ur] & \end{tikzcd}$$

In the end, notice that multiple inheritance without common ancestor in the previous subsection may be fitted in the present framework, by taking the empty specification as common ancestor. The above comma category then becomes the cartesian product of categories of classes, as used in the previous subsection.

5.5 Repeated inheritance

Repeated inheritance occurs when a class (specification) inherits from the same ancestor more than once (via different inclusions). Naively this leads to name clashes. But these clashes can be avoided by appropriate renameings of methods (like in Eiffel, see [16, 20]). As an example, suppose we wish to specify two coupled flip-flops (CFFs), which can be switched on independently, but can only be switched off simultaneously.

class spec: CFF
  inherits from:
    FF **rename:**
      val **as** left_val
      on **as** left_on
      off **as** left_off
    FF **rename:**
      val **as** right_val
      on **as** right_on
      off **as** right_off
  assertions:
    s.left_on.right_val = s.right_val
    s.left_off.right_val = 0
    s.right_on.left_val = s.left_val
    s.right_off.left_val = 0
end class spec

The point of this renameing is that the specification FF of flip-flops is incorporated twice. The set of methods of the specification CFF of coupled flip-flops is the disjoint union with itself of the set of methods of the specification FF. Disjointness is achieved via this renaming. Thus we have two inclusions of specifications FF $\rightrightarrows$ CFF, and correspondingly two forgetful functors

$$\begin{tikzcd} \mathbf{Class}(\mathrm{CFF}) \arrow[r, shift left, "\mathcal{L}"] \arrow[r, shift right, "\mathcal{R}"'] & \mathbf{Class}(\mathrm{FF}) \end{tikzcd}$$

mapping a class $B \in \mathbf{Class}(\mathrm{CFF})$ to its interpretations of the "left" and "right" part of the specification.

There is something more going on in our understanding of repeated inheritance, which is not expressed by the pair of functors $\mathcal{L}, \mathcal{R}$. In constructing models of the specification CFF of coupled flip-flops from a model B of flip-flops we wish to use this same model B

twice; we do not seek to construct a CFF-model from two arbitrary models B, B' of flip-flops. This idea of using the same interpretation for an ancestor occurring twice occurs also for multiple inheritance in the previous subsection. The approach that we propose here to understand repeated inheritance is similar, except that we now use a comma category as a domain. We restrict ourselves to those models $B \in \mathbf{Class}(\mathrm{CFF})$ which inherit twice from a single FF-class, i.e. to those B with maps $\ell : \mathcal{L}(B) \to A$, $r : \mathcal{R}(B) \to A$ to a class $A \in \mathbf{Class}(\mathrm{FF})$. Such B's occur in the comma category $(\langle \mathcal{L}, \mathcal{R} \rangle \downarrow \Delta)$ of the functors

$$\begin{tikzcd}
\mathbf{Class}(\mathrm{CFF}) \arrow[dr, "\langle \mathcal{L}, \mathcal{R} \rangle"'] & & \mathbf{Class}(\mathrm{FF}) \arrow[dl, "\Delta = \langle id, id \rangle"] \\
& \mathbf{Class}(\mathrm{FF}) \times \mathbf{Class}(\mathrm{FF}) &
\end{tikzcd}$$

There is an associated "second projection" functor

$$(\langle \mathcal{L}, \mathcal{R} \rangle \downarrow \Delta) \xrightarrow{\quad \mathcal{F} \quad} \mathbf{Class}(\mathrm{FF})$$

which we use to describe inheritance and cofreeness in this situation. Explicitly, $B \in \mathbf{Class}(\mathrm{CFF})$ together with $\mathcal{L}(B) \xrightarrow{\ell} A \xleftarrow{r} \mathcal{R}(B)$ inherits from $C \in \mathbf{Class}(\mathrm{FF})$ if there is a coercion $f : A \to C$. And this $\mathcal{L}(B) \xrightarrow{\ell} A \xleftarrow{r} \mathcal{R}(B)$ is the cofree subclass inheriting from $C \in \mathbf{Class}(\mathrm{FF})$ if for every B' with maps $\mathcal{L}(B') \xrightarrow{\ell'} A' \xleftarrow{r'} \mathcal{R}(B')$ there is a unique pair of maps $g : B' \dashrightarrow B$, $h : A' \dashrightarrow A$ making the following diagrams commute.

$$\begin{tikzcd}
\mathcal{L}(B) \arrow[r, "\ell"] & A & \mathcal{R}(B) \arrow[l, "r"'] \\
\mathcal{L}(B') \arrow[u, "\mathcal{L}(g)"] \arrow[r, "\ell'"'] & A' \arrow[u, "h"'] & \mathcal{R}(B) \arrow[u, "\mathcal{L}(g)"'] \arrow[l, "r'"]
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "f"] & C \\
A' \arrow[u, "h"] \arrow[ur, "f'"'] &
\end{tikzcd}$$

In our example, if $C \in \mathbf{Class}(\mathrm{FF})$ is the implementation of flip-flops with $\{0, 1\}$ as state space, then the cofree subclass on C has $\{0, 1\} \times \{0, 1\}$ as state space, and operations

$$\begin{array}{rclrclrcl}
\mathsf{left_val}(x, y) & = & x, & \mathsf{left_on}(x, y) & = & (1, y), & \mathsf{left_off}(x, y) & = & (0, 0), \\
\mathsf{right_val}(x, y) & = & x, & \mathsf{right_on}(x, y) & = & (x, 1), & \mathsf{right_off}(x, y) & = & (0, 0).
\end{array}$$

Obviously there are coercion maps $\{0, 1\} \times \{0, 1\} \rightrightarrows \{0, 1\}$, namely first and second projection. They commute with the flip-flop operations. And the above map $f : A \to C$ is simply the identity $\{0, 1\} \to \{0, 1\}$. If we have another subclass implementation $\mathcal{L}(B') \xrightarrow{\ell'} A' \xleftarrow{r'} \mathcal{R}(B')$ with common ancestor class A', then the required unique maps are $\langle \mathsf{left\_val}, \mathsf{right\_val} \rangle : B' \dashrightarrow \{0, 1\} \times \{0, 1\}$, $\mathsf{val} : A' \dashrightarrow \{0, 1\}$.

6 Conclusions and further work

We have presented some paradigmatic examples of inheritance within the framework of coalgebraic specification and implementation. Of course, these example do not cover all possibilities. For instance, one can have multiple bank accounts on the same name via maps of specifications NBBANK $\rightrightarrows$ NBANK where the "balance" and "change-balance" methods are renamed, but the "name" and "change-name" methods are shared. This may be described by a combination of the above techniques.

In later work we shall have more to say about the existence of cofree constructions (of the kind used above). The logical aspects (completeness, conservativity) still have to be investigated. In the end we should emphasize that we have described examples of inheritance without genericity. The latter would require suitably indexed versions (via free type variables) of the above descriptions.

Acknowledgement

Thanks are due to Jan Rutten for helpful discussions.

References

1. K. Bruce, L. Cardelli, G. Castagna, The Hopkins Objects Group, G. Leavens, and B. Pierce. On binary methods. *Theory & Practice of Object Systems*, 1(3), 1996.
2. L. Cardelli. A semantics of multiple inheritance. *Inf. & Comp.*, 76(2/3):138–164, 1988.
3. W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *Inf. & Comp.*, 114(2):329–350, 1995.
4. W.R. Cook. Object-oriented programming versus abstract data types. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, number 489 in Lect. Notes Comp. Sci., pages 151–178. Springer, Berlin, 1990.
5. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*. Number 6 in EATCS Monographs. Springer, Berlin, 1985.
6. J.A. Goguen. Realization is universal. *Math. Syst. Theor.*, 6(4):359–374, 1973.
7. J.A. Goguen. A categorical manifesto. *Math. Struct. Comp. Sci.*, 1(1):49–67, 1991.
8. J.A Goguen and J. Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In M. Nielsen and E.M. Schmidt, editors, *Automata, Languages and Programming (ICALP'82)*, number 140 in Lect. Notes Comp. Sci., pages 263–281. Springer, Berlin, 1982.
9. J.A Goguen and J. Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. The MIT Press series in computer systems, 1987.
10. B. Jacobs. Mongruences and cofree coalgebras. In V.S. Alagar and M. Nivat, editors, *Algebraic Methods and Software Technology*, number 936 in Lect. Notes Comp. Sci., pages 245–260. Springer, Berlin, 1995.
11. B. Jacobs. Coalgebraic specifications and models of deterministic hybrid systems. In M. Wirsing and M. Nivat, editors, *Algebraic Methods and Software Technology*, number 1101 in Lect. Notes Comp. Sci., pages 520–535. Springer, Berlin, 1996.
12. B. Jacobs. Objects and classes, co-algebraically. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*. Kluwer Acad. Publ., 1996, to appear.
13. S. Mac Lane. *Categories for the Working Mathematician*. Springer, Berlin, 1971.
14. S. Lang. *Algebra*. Addison Wesley, 2^{nd} rev. edition, 1984.
15. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.
16. B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
17. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
18. J. Palsberg and M.I. Schwartzbach. *Object-Oriented Type Systems*. Wiley, 1994.
19. H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. Comp. Sci.*, 5:129–152, 1995.
20. R. Rist and R. Terwilliger. *Object-Oriented Programming in Eiffel*. Prentice Hall, 1995.
21. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2^{nd} rev. edition, 1994.
22. P. Wegner. The object-oriented classification paradigm. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 479–560. The MIT Press series in computer systems, 1987.
23. P. Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, 1990.