Inheritance in Higher Order Logic: Modeling and Reasoning

M. Huisman, B.P.F. Jacobs

Computing Science Institute/

CSI-R0004 February

Inheritance in Higher Order Logic: Modeling and Reasoning

Marieke Huisman, Bart Jacobs

Computing Science Institute, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
{marieke,bart}@cs.kun.nl

29 February 2000

Abstract. This paper describes a way of modeling inheritance (in object-oriented programming languages) in higher order logic. This particular approach is used in the LOOP project for reasoning about JAVA classes, with the proof tools PVS and ISABELLE. It relies on nested interface types to capture the superclasses, fields, methods, and constructors of classes, together with suitable casting functions incorporating the difference between hiding of fields and overriding of methods. This leads to the proper handling of late binding, as illustrated in several examples.
Keywords: Inheritance, Java, program verification
Classification: 68Q55, 68Q60, 68Q65 (AMS'91); D.1.5, D.2.4, F.3.1, F.4.1 (CR'98).

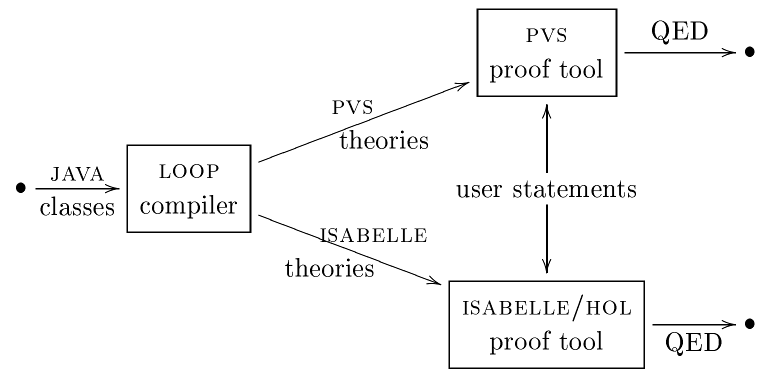
1 Introduction

This paper reports on a particular aspect of the semantics of object-oriented languages, like JAVA, used in the "LOOP" verification project [19]. It concentrates on inheritance. A companion paper [3] explains the underlying memory model.

Inheritance is a key feature of object-oriented languages. It allows a programmer to model his/her application domain according to a natural "is-a" relationship between classes of objects. One can use inheritance, for instance, to say that a lorry is-a vehicle by making a class of lorries a subclass (or "child" or "descendant") of a superclass (or "ancestor") of vehicles. Important aspects of inheritance are re-use of code, and polymorphism. The latter is sometimes called subtype polymorphism (to distinguish it for example from parametric polymorphism). Its effect is that the particular implementation that is used in a method call is determined by the actual (run-time) type of the receiving object, telling to which class the object belongs. This mechanism is often referred to as dynamic method look-up or late binding. It is precisely this dynamic aspect of object-oriented languages which is difficult to capture in a static logical setting. Therefore, the semantics of inheritance—as a basis for reasoning about classes—is a real challenge, see *e.g.* [5, 15, 6, 11, 8, 16]. There is a whole body of research on encodings of classes using recursive or existential types, in a suitably rich polymorphic type theory (like $F^{\omega}_{<:}$ or $F_{<:}$). Four such (functional) encodings are formulated and compared in a common notational framework in [4]. But they all use quantification or recursion over type variables, which is not available in the higher order logic (comparable to the logics of PVS and ISABELLE/HOL) that will be used here. The setting of the encoding in [16] is higher order logic with "extensible records". This framework is closest to what we use (but is still stronger). Also, an experimental *functional* object-oriented language, without references and object identity is studied there. This greatly simplifies matters, because the subtle late binding issues involving run-time types of objects (which may change through assignments, see Section 7) do not occur. Indeed, it is a crucial aspect of *imperative* object-oriented programming languages that the declared type of a variable may be different from—but must be a supertype of—the actual, run-time type of an object to which it refers. Our semantics of inheritance works for an existing object-oriented language, namely JAVA, with all such semantical complications.

The explanations below only describe a small part of the semantics of JAVA used in the LOOP project. Due to space limitations, many aspects have to be left unexplained. We intend to concentrate on the main ideas underlying the handling of inheritance, using paradigmatic examples. Many related issues, like inheritance from multiple interfaces (which mainly involves proper name handling), method overloading, or object creation via constructor chaining, are not covered in the present paper.

For our JAVA verification work a special purpose compiler, called LOOP, for *Logic of Object-Oriented Programming*, has been developed. It is used as a front-end to a proof tool, for which we can use both PVS [17] and ISABELLE [18], as follows.



The LOOP tool translates JAVA classes into logical theories, containing definitions (embodying the semantics of the classes) plus special lemmas that are used for automatic rewriting. These logical theories can be loaded into the proof tool, together with the so-called semantical prelude, which contains basic definitions, like in Section 3 below. Subsequently, the user can state desired properties about the original JAVA classes and prove these on basis of the semantical prelude and the generated theories. For example, a user may want to prove that a method terminates, returning a certain value; see Section 8 for several examples.

The semantics that is used is based on so-called coalgebras (see [13, 12]). In this paper, coalgebras are only used to conveniently combine all the ingredients of a class in a single function. Specifically, n functions $f_1 \colon \mathsf{Self} \to \sigma_1$, $\ldots$, $f_n \colon \mathsf{Self} \to \sigma_n$ with a common domain can be combined in one function $\mathsf{Self} \to [\, f_1 \colon \sigma_1, \ldots, f_n \colon \sigma_n \,]$ with a labeled product type as codomain[1], forming a coalgebra. Coalgebras give rise to a general theory of behaviour for dynamical systems, involving useful notions like invariance and bisimilarity, but we shall not make use of it here. Therefore, the use of coalgebras remains fairly superficial, and is not essential for what happens[2].

The paper is organised as follows. It starts with two preliminary sections: one on the type-theoretic notation that will be used, and one about some basic aspects of JAVA semantics. Then, Section 4 introduces interfaces types as labeled products to capture the ingredients of classes, and shows how these are nested to incorporate superclasses. Section 5 discusses hiding and overriding at the level of these interface types, via special cast functions, and Sections 6 and 7 show how these functions realise the appropriate late binding behaviour. Finally, Section 8 describes two example verifications, one in PVS and one in ISABELLE/HOL, involving small but non-trivial JAVA programs.

2 Higher order logic

The actual verifications of JAVA programs in the LOOP project are done using either PVS or ISABELLE/HOL, see Section 8. In this paper we shall abstract away from the specific syntax for the higher order logic of PVS or ISABELLE/HOL, and use a (hopefully more generally accessible) type-theoretic language. It involves types which are built up from: type variables $\alpha, \beta, \ldots$, type constants nat, bool, string (and some more), exponent types $\sigma \to \tau$, labeled product (or record) types $[\, \mathsf{lab}_1 \colon \sigma_1, \ldots, \mathsf{lab}_n \colon \sigma_n \,]$ and labeled coproduct (or variant) types $\{\, \mathsf{lab}_1 \colon \sigma_1 \mid \ldots \mid \mathsf{lab}_n \colon \sigma_n \,\}$, for given types $\sigma, \tau, \sigma_1, \ldots, \sigma_n$. New types or type constructors can be introduced via definitions, as in:

$$\mathsf{lift}[\alpha] \,:\, \mathsf{TYPE} \overset{\text{def}}{=} \{\, \mathsf{bot} \colon \mathsf{unit} \mid \mathsf{up} \colon \alpha \,\}$$

where unit is the empty labeled product type $[\,]$. This lift type constructor adds a bottom element to an arbitrary type, given as type variable α.

For exponent types we shall use the standard lambda abstraction $\lambda x \colon \sigma.\, M$ and application $N \cdot L$ notation. For terms $M_i \colon \sigma_i$, we have a labeled tuple $(\, \mathsf{lab}_1 = M_1, \ldots, \mathsf{lab}_n = M_n \,)$ inhabiting the labeled product type $[\, \mathsf{lab}_1 \colon \sigma_1, \ldots, \mathsf{lab}_n \colon \sigma_n \,]$. For a term $N \colon [\, \mathsf{lab}_1 \colon \sigma_1, \ldots, \mathsf{lab}_n \colon \sigma_n \,]$ in this product, we write $N.\mathsf{lab}_i$ for the selection term of type σ_i. Dually, for a term $M \colon \sigma_i$ there is a labeled or tagged term $\mathsf{lab}_i\, M$ in the labeled coproduct type $\{\, \mathsf{lab}_1 \colon \sigma_1 \mid \ldots \mid \mathsf{lab}_n \colon \sigma_n \,\}$. And for a term $N \colon \{\, \mathsf{lab}_1 \colon \sigma_1 \mid \ldots \mid \mathsf{lab}_n \colon \sigma_n \,\}$ in this coproduct type, together with n

[1] Alternatively, one can combine these n functions into elements of a "trait type" $[\, f_1 \colon \mathsf{Self} \to \sigma_1, \ldots, f_n \colon \mathsf{Self} \to \sigma_n \,]$, like in [1, §§8.5.2].

[2] As a side-remark, all the encodings discussed in [4] implicitly also use coalgebras.

terms $L_i(x_i)\colon \tau$, possibly containing a free variable $x_i\colon \sigma_i$, there is a case term $\mathsf{CASES}\ N\ \mathsf{OF}\ \{\, \mathsf{lab}_1\, x_1 \mapsto L_1(x_1) \mid \ldots \mid \mathsf{lab}_n\, x_n \mapsto L_n(x_n)\,\}$ of type τ. These introduction and elimination constructions for exponents and labeled (co)products are required to satisfy standard (β)- and (η)-conversions.

In this paper we do not use any formulas in higher order logic—which are of course terms of type bool—and work exclusively in the underlying type theory. This is only possible because we describe a limited part of the semantics of JAVA.

3 Semantics of Java statements and expressions

In the remainder of this paper we shall use Self as a type variable for a state space. JAVA statements and expressions will be modeled as state transformer functions (or coalgebras) acting on Self. Statements and expressions in JAVA may either hang, terminate normally, or terminate abruptly. These different output options are captured by two output types StatResult[Self], and ExprResult[Self, α], in:

$$\mathsf{Self} \longrightarrow \mathsf{StatResult}[\mathsf{Self}] \qquad\qquad \mathsf{Self} \longrightarrow \mathsf{ExprResult}[\mathsf{Self}, \alpha]$$

where α is a type variable for the result type of the JAVA expression. These output types are defined as labeled coproducts:

$$\begin{array}{ll}
\mathsf{StatResult}[\mathsf{Self}] : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} & \mathsf{ExprResult}[\mathsf{Self}, \alpha] : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\
\quad \{\, \mathsf{hang}\colon \mathsf{unit} & \quad \{\, \mathsf{hang}\colon \mathsf{unit} \\
\quad \mid \mathsf{norm}\colon \mathsf{Self} & \quad \mid \mathsf{norm}\colon [\, \mathsf{ns}\colon \mathsf{Self}, \mathsf{res}\colon \alpha\,] \\
\quad \mid \mathsf{abnorm}\colon \mathsf{StatAbn}[\mathsf{Self}]\,\} & \quad \mid \mathsf{abnorm}\colon \mathsf{ExprAbn}[\mathsf{Self}]\,\}
\end{array}$$

The types StatAbn[Self] and ExprAbn[Self] capture the various abnormalities that can occur for JAVA statements and expressions (like exceptions, returns, breaks and continues). Their precise structure is not relevant for this paper—but can be found in [10, 3, 9].

On the basis of this representation, the semantics of all of JAVA's language constructs, like `while`, `catch` etc., can be defined. For instance, the composition $s\,;t$ of two statements $s, t\colon \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}]$ is defined as:

$$\begin{array}{l}
s\,;t \,:\, \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}] \stackrel{\mathrm{def}}{=} \lambda x\colon \mathsf{Self}.\ \mathsf{CASES}\ s \cdot x\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{hang} \mapsto \mathsf{hang} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{norm}\, y \mapsto t \cdot y \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}\, a\,\}
\end{array}$$

We need not describe all these details here to understand inheritance. What we do need in the sequel is a special type RefType for references. It is defined as either a null-reference null or a non-null-reference ref x, where x consists of a pointer to a memory location, where the object that is being referred to resides (see [3] for details), a string, indicating the run-time type of the object, and a third field that is used to give the dimension and length for arrays:

$$\begin{array}{l}\mathsf{RefType} : \mathsf{TYPE} \overset{\mathrm{def}}{=} \\ \qquad \{\, \mathsf{null} \colon \mathsf{unit} \mid \mathsf{ref} \colon [\, \mathsf{objpos} \colon \mathsf{MemLoc}, \\ \qquad\qquad\qquad\quad \mathsf{clname} \colon \mathsf{string}, \\ \qquad\qquad\qquad\quad \mathsf{dimlen} \colon \mathsf{lift}[\,[\, \mathsf{dim} \colon \mathsf{nat}, \mathsf{len} \colon \mathsf{nat} \,]\,]\,] \,\}\end{array}$$

All references in JAVA (both to objects and to arrays) are translated in type theory to values of type RefType. Thus, if we have an object `a` in a class `A` and an object `b` in a subclass `B` of `A`, then the translation of an assignment `a = b` involves a replacement of the reference to `a` by the reference to `b`. Since both are inhabitants of RefType, this is well-typed. If `b` has run-time type `B`, then so will `a` after the assignment.

4 Nested labeled product types for interfaces

JAVA has classes and interfaces. Interfaces only contain the headers of methods (their names and the types of their parameters and results, if any), but not their implementations (given by what are often called method bodies). The latter can only occur in classes. In this section we do not make a distinction between classes and interfaces, because method bodies do not play a rôle. Therefore, class can also mean interface at this stage. What we describe is how certain labeled product types (in type theory) are extracted from JAVA classes. These product types describe the superclasses, fields (or instance variables) with associated assignment operations, methods, and constructors of a class. They form the basis for the type-theoretic formalisation of JAVA classes.

It is easiest to proceed via an example of a JAVA class:

– JAVA

```
class MyClass {
  int i;
  int k = 3;
  void m (byte a, int b) { if (a > b) i = a; else i = b; }
  MyClass () { i = 6; }
}
```

Ignoring the implicit superclass `Object`, we extract the following interface type.

– TYPE THEORY

$$
\begin{array}{l}
\mathsf{MyClassIFace}[\mathsf{Self}] : \mathsf{TYPE} \stackrel{\text{def}}{=} \\
\quad [\ \ldots \qquad // \text{ for the superclass, see below} \\
\quad\ \ \mathsf{i} : \mathsf{int}, \\
\quad\ \ \mathsf{i\_becomes} : \mathsf{int} \to \mathsf{Self}, \\
\quad\ \ \mathsf{k} : \mathsf{int}, \\
\quad\ \ \mathsf{k\_becomes} : \mathsf{int} \to \mathsf{Self}, \\
\quad\ \ \mathsf{m} : \mathsf{byte} \to \mathsf{int} \to \mathsf{StatResult}[\mathsf{Self}], \\
\quad\ \ \mathsf{MyClass} : \mathsf{ExprResult}[\mathsf{Self}, \mathsf{RefType}]\ ]
\end{array}
$$

There are several things worth noticing here.

– The field declaration `int i` gives rise not only to a label i: int in the product type but also to an associated assignment operation, with label i_becomes. This assignment operation takes an integer as input, and produces a new state in Self, in which the i field is changed to the argument of the assignment operation (and the rest is unchanged). Similarly for `k`. Variable initialisers (like `k = 3`) are ignored at this stage, since they are irrelevant for the interface type (just like method bodies).
– The method `m`, which is a void method, is modeled as an entry m in the labeled product of type StatResult[Self][3]. Similarly, methods with a return value are modeled with ExprResult, *e.g.* `int n () {return 3;}` would give rise to a label n with type ExprResult[Self, int].
– The type of the constructor `MyClass` is implicit in the JAVA code, but is made explicit in the type-theoretic formalisation. Since a constructor returns a reference to a newly created object, it is modeled as an entry with type ExprResult[Self, RefType]. Constructors are often left implicit in JAVA code, as so-called default constructors. These are added explicitly to interface types.

The types occurring in the interface type MyClassIFace above describe the "visible" signatures of the fields, methods and constructors in the JAVA class `MyClass`. But in object-oriented programs there is always an implicit argument to a field/method/constructor, namely the current state of the object on which the field/method/constructor is invoked. This is made explicit by modeling classes as *coalgebras* for interface types, *i.e.* as functions of the form:

$$\mathsf{Self} \longrightarrow \mathsf{MyClassIFace}[\mathsf{Self}]$$

Such a coalgebra actually combines the fields, methods and constructors of the class in a single function. These are made explicit, using the isomorphism $\mathsf{Self} \to [\, f_1 : \sigma_1, \ldots, f_n : \sigma_n \,] \cong [\, f_1 : \mathsf{Self} \to \sigma_1, \ldots, f_n : \mathsf{Self} \to \sigma_n \,]$, via what we call "extraction" functions:

[3] To prevent name clashes, the LOOP compiler does more. For example, overloading of labels usually is not allowed in labeled product types. Therefore the LOOP compiler does not use m but m_byte_int as translation of `m` in JAVA. Here we shall ignore such bureaucratic aspects, and simply assume that no such name clashes occur.

– TYPE THEORY

Assuming a variable $c\colon \mathsf{Self} \to \mathsf{MyClassIFace}[\mathsf{Self}]$,

$$\vdash\ \mathsf{i}(c)\colon \mathsf{Self} \to \mathsf{int} \overset{\text{def}}{=} \lambda x\colon \mathsf{Self}.\,(c \cdot x).\mathsf{i}$$

$$\vdash \mathsf{i\_becomes}(c)\colon \mathsf{Self} \to \mathsf{int} \to \mathsf{Self} \overset{\text{def}}{=} \lambda x\colon \mathsf{Self}.\,(c \cdot x).\mathsf{i\_becomes}$$

$$\vdash\ \mathsf{k}(c)\colon \mathsf{Self} \to \mathsf{int} \overset{\text{def}}{=} \lambda x\colon \mathsf{Self}.\,(c \cdot x).\mathsf{k}$$

$$\vdash \mathsf{k\_becomes}(c)\colon \mathsf{Self} \to \mathsf{int} \to \mathsf{Self} \overset{\text{def}}{=} \lambda x\colon \mathsf{Self}.\,(c \cdot x).\mathsf{k\_becomes}$$

$$b\colon \mathsf{byte}, j\colon \mathsf{int}\ \vdash\ \mathsf{m}(b)(j)(c)\colon \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}] \overset{\text{def}}{=} \lambda x\colon \mathsf{Self}.\,((c \cdot x).\mathsf{m}) \cdot b \cdot j$$

$$\vdash\ \mathsf{MyClass}(c)\colon \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{RefType}] \overset{\text{def}}{=} \lambda x\colon \mathsf{Self}.\,(c \cdot x).\mathsf{MyClass}$$

The coalgebra $c\colon \mathsf{Self} \to \mathsf{MyClassIFace}[\mathsf{Self}]$ above thus combines all the operations of the class `MyClass`. In the remainder of this paper, we shall always describe operations—fields (with their assignments), methods, constructors—of a class, say `A`, using extraction functions as above, with respect to a coalgebra of type AIFace,

4.1 Inheritance and nested interface types

Now that we have seen the basic idea of how to build an interface type from the fields, methods and constructors of a JAVA class, we proceed to incorporate superclasses. This will be done via nesting of interface types. Again, it is easiest to use an example.

– JAVA

```
class MySubClass extends MyClass {
  int j;
  int n (byte b) { m(b, 3); return i; }
}
```

This new class `MySubClass` inherits the field `i` and method `m` of `MyClass`, and it declares its own field `j` and method `n`. As can be seen in the body of the method `n`, the methods and fields from the super class are immediately available, *i.e.* the method `m` and field `i` are called without any further reference to `MyClass`. This should also be possible in our formalisation.

This class gives rise to the following interface type in type theory, in which the interface type $\mathsf{MyClassIFace}$ defined earlier for the class `MyClass` appears as the first entry of the labeled product, thus formalising the inheritance relationship. In a similar way, $\mathsf{MyClassIFace}$ contains a field $\mathsf{super\_Object}\colon \mathsf{ObjectIFace}[\mathsf{Self}]$, formalising the implicit inheritance from `Object` by `MyClass`.

– TYPE THEORY

$$\begin{array}{l}\mathsf{MySubClassIFace[Self]} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\ \quad [\,\mathsf{super\_MyClass}\colon \mathsf{MyClassIFace[Self]}, \\ \quad\; \mathsf{j}\colon \mathsf{int}, \\ \quad\; \mathsf{j\_becomes}\colon \mathsf{int} \to \mathsf{Self}, \\ \quad\; \mathsf{n}\colon \mathsf{byte} \to \mathsf{ExprResult[Self, int]}, \\ \quad\; \mathsf{MySubClass}\colon \mathsf{ExprResult[Self, RefType]}\,]\end{array}$$

As before, we shall consider a coalgebra $c\colon \mathsf{Self} \to \mathsf{MySubClassIFace[Self]}$ as representation of the class `MySubClass`. For such a coalgebra we can again define extraction functions, giving us access to all ingredients of `MySubClass`, but also of `MyClass`, via the nesting of interfaces. This goes as follows.

– TYPE THEORY

Assuming a variable $c\colon \mathsf{Self} \to \mathsf{MySubClassIFace[Self]}$,

$$\vdash\ \mathsf{j}(c)\colon \mathsf{Self} \to \mathsf{int} \stackrel{\mathrm{def}}{=} \lambda x\colon \mathsf{Self}.\,(c\cdot x).\mathsf{j}$$

$$\vdash \mathsf{j\_becomes}(c)\colon \mathsf{Self} \to \mathsf{int} \to \mathsf{Self} \stackrel{\mathrm{def}}{=} \lambda x\colon \mathsf{Self}.\,(c\cdot x).\mathsf{j\_becomes}$$

$$b\colon \mathsf{byte}\ \vdash\ \mathsf{n}(b)(c)\colon \mathsf{Self} \to \mathsf{StatResult[Self]} \stackrel{\mathrm{def}}{=} \lambda x\colon \mathsf{Self}.\,((c\cdot x).\mathsf{n})\cdot b$$

$$\begin{array}{l}\vdash\ \mathsf{MySubClass}(c)\colon \mathsf{Self} \to \mathsf{ExprResult[Self, RefType]} \stackrel{\mathrm{def}}{=} \\ \quad \lambda x\colon \mathsf{Self}.\,(c\cdot x).\mathsf{MySubClass}\end{array}$$

// continue with the superclass `MyClass`

$$\vdash\ \mathsf{i}(c)\colon \mathsf{Self} \to \mathsf{int} \stackrel{\mathrm{def}}{=} \lambda x\colon \mathsf{Self}.\,(c\cdot x).\mathsf{super\_MyClass.i}$$

$$\vdash \mathsf{i\_becomes}(c)\colon \mathsf{Self} \to \mathsf{int} \to \mathsf{Self} \stackrel{\mathrm{def}}{=} \lambda x\colon \mathsf{Self}.\,(c\cdot x).\mathsf{super\_MyClass.i\_becomes}$$

$$\begin{array}{l}b\colon \mathsf{byte}, j\colon \mathsf{int}\ \vdash\ \mathsf{m}(b)(j)(c)\colon \mathsf{Self} \to \mathsf{StatResult[Self]} \stackrel{\mathrm{def}}{=} \\ \qquad\qquad \lambda x\colon \mathsf{Self}.\,((c\cdot x).\mathsf{super\_MyClass.m})\cdot b\cdot j\end{array}$$

// etcetera

The repeated extraction functions, *e.g.* i, $\mathsf{i\_becomes}$ and m, thus give immediate access to all ingredients of superclasses. Note how this involves overloading, because for instance $\mathsf{i}(c)$ is defined both for coalgebras $c\colon \mathsf{Self} \to \mathsf{MyClassIFace[Self]}$ and for coalgebras $c\colon \mathsf{Self} \to \mathsf{MySubClassIFace[Self]}$ representing the classes `MyClass` and `MySubClass`.

5 Overriding and hiding

In the previous section we have seen an example of inheritance where the subclass `MySubClass` simply adds an extra field and method to the superclass. But the same fields and methods may also be repeated in subclasses. In JAVA this is called *hiding* of fields, and *overriding* of methods. Different names are used, because the mechanisms are different: field selection is based on the static type of receiving objects, whereas method selection is based on the dynamic (or run-time) type of an object. The latter mechanism is often referred to as dynamic method lookup, or late binding. Consider the following example.

– JAVA

```
class A {
  int i = 1;
  int m() { return i * 100; }
}
class B extends A {
  int i = 10;
  int m() { return i * 1000; }
}


class Test {
  int test1() { A[] ar = { new A(), new B() };
    return ar[0].i + ar[0].m() + ar[1].i + ar[1].m(); }
}
```

The field `i` in the subclass `B` hides the field `i` in the superclass `A`, and similarly, the method `m` in `B` overrides the method `m` in `A`. In the `test1` method of class `Test` a local variable `ar` of type 'array of `A`s' is declared and initialised with length 2 containing a new `A` object at position 0, and a new `B` object at position 1. Note that at position 1 there is an implicit conversion from `B` to `A` to make the new `B` object fit into the array of `A`s. Interestingly, the `test1` method will return `ar[0].i + ar[0].m() + ar[1].i + ar[1].m()`, which is `1 + 1 * 100 + 1 + 10 * 1000 = 10102`, because: when `new B()` is converted to type `A` the hidden field becomes visible again—so that the field `ar[1].i` refers to `i` in `A`—but the overriding method replaces the original method—so that the method `ar[1].m()` leads to execution of `m` in `B` (which uses the field `i` from `B`). See [2, §§3.4], or also [7, §§8.4.6.1]:

> Note that a qualified name or a cast to a superclass is not effective in attempting to access an overridden method; in this respect, overriding of methods differs from hiding of fields.

It is a challenge to provide a semantics for this behaviour. We do so by using a special cast function between coalgebras, which performs appropriate replacements of methods and fields. We shall illustrate this in the above JAVA example. The interface types for classes `A` and `B` are defined as follows.

– TYPE THEORY

$$\begin{array}{ll}
\mathsf{AIFace[Self]} : \mathsf{TYPE} \stackrel{\text{def}}{=} & \mathsf{BIFace[Self]} : \mathsf{TYPE} \stackrel{\text{def}}{=} \\
\quad [\,\mathsf{super\_Object}\colon \mathsf{ObjectIFace[Self]}, & \quad [\,\mathsf{super\_A}\colon \mathsf{AIFace[Self]}, \\
\quad\ \mathsf{i}\colon \mathsf{int}, & \quad\ \mathsf{i}\colon \mathsf{int}, \\
\quad\ \mathsf{i\_becomes}\colon \mathsf{int} \to \mathsf{Self}, & \quad\ \mathsf{i\_becomes}\colon \mathsf{int} \to \mathsf{Self}, \\
\quad\ \mathsf{m}\colon \mathsf{ExprResult[Self, int]}, & \quad\ \mathsf{m}\colon \mathsf{ExprResult[Self, int]}, \\
\quad\ \mathsf{A}\colon \mathsf{ExprResult[Self, RefType]}\,] & \quad\ \mathsf{B}\colon \mathsf{ExprResult[Self, RefType]}\,]
\end{array}$$

Notice that the interface type BIFace[Self] contains m and i twice: once directly, and once inside the nested interface type AIFace[Self]. Thus we define two extraction functions for each of them:

– TYPE THEORY

$$\begin{array}{c}
\text{Assuming a variable } c\colon \mathsf{Self} \to \mathsf{BIFace[Self]}, \\
\vdash\ \mathsf{i}(c)\colon \mathsf{Self} \to \mathsf{int} \stackrel{\text{def}}{=} \lambda x\colon \mathsf{Self}.\,(c \cdot x).\mathsf{i} \\
\vdash\ \mathsf{A\_i}(c)\colon \mathsf{Self} \to \mathsf{int} \stackrel{\text{def}}{=} \lambda x\colon \mathsf{Self}.\,(c \cdot x).\mathsf{super\_A.i} \\
\vdash\ \mathsf{m}(c)\colon \mathsf{Self} \to \mathsf{ExprResult[Self, int]} \stackrel{\text{def}}{=} \lambda x\colon \mathsf{Self}.\,(c \cdot x).\mathsf{m} \\
\vdash\ \mathsf{A\_m}(c)\colon \mathsf{Self} \to \mathsf{ExprResult[Self, int]} \stackrel{\text{def}}{=} \lambda x\colon \mathsf{Self}.\,(c \cdot x).\mathsf{super\_A.m}
\end{array}$$

The extraction functions A_i and A_m are used for `super` invocations.

What we want is a way of "casting" a `B` coalgebra $c\colon \mathsf{Self} \to \mathsf{BIFace[Self]}$ to a `A` coalgebra $\mathsf{B2A}(c)\colon \mathsf{Self} \to \mathsf{AIFace[Self]}$ which incorporates the differences between hiding and overriding. Just taking the super_A entry is not good enough: we need additional updates, which select the fields of the superclass `A`, but the methods of the subclass `B`. Therefore, we use a record update on the entry super_A, which updates the method entries to the methods of `B`, defining:

– TYPE THEORY

$$\begin{array}{l}
c\colon \mathsf{Self} \to \mathsf{BIFace[Self]}\ \vdash \\
\quad \mathsf{B2A}(c)\ :\ \mathsf{Self} \to \mathsf{AIFace[Self]} \stackrel{\text{def}}{=} \\
\quad\quad \lambda x\colon \mathsf{Self}.\,(c \cdot x).\mathsf{super\_A}\ \mathsf{WITH}\ (\mathsf{m} := (c \cdot x).\mathsf{m})
\end{array}$$

As a result, $\mathsf{m}(\mathsf{B2A}(c)) = \mathsf{m}(c)$, and $\mathsf{i}(\mathsf{B2A}(c)) = \mathsf{i}(\mathsf{super\_A}(c))$.

In general, all overriding methods from a subclass replace the methods from its superclass. Hidden fields reappear in such casting because they are not replaced.

6 Handling late binding

The example in the previous section involved late binding: `ar[1]` has static type `A`, but invoking `ar[1].m()` results in the execution of `m` from `B`, because `ar[1]` has *run-time* type `B`. We shall study this mechanism in more detail. First, in this section we concentrate on late binding within the current object (on `this` if you like), and later, in the next section, we concentrate on method invocations on different objects.

Suppose for now that the class `A` from the previous section also contains a method `n` which simply calls `m`, and is used in `B`, as in:

– JAVA

```
class A {
  ... // as before
  int n() { return i +  m(); }
}

class B extends A {
  ... // as before
  int test2() { return n(); }
}
```

Again due to late binding, `test2` returns the value of the field `i` from `A` plus the result from the method `m` in `B`, since, as explained earlier, field selection is based on the static type and method selection is based on dynamic types. Since the run-time type of the object in which `test2` is executed is `B`, late binding ensures the execution of `m` from `B`. Thus, `test2` returns the value of `i` from `A` + 1000 × the value of `i` from `B`.

This behaviour is realised in our semantics by using the method bodies of `A`, in particular the body of `n`, with appropriate casts from a `B` coalgebra to an `A` coalgebra. Before we can see how this works, we need to know a bit more about method bodies.

6.1 Formalisation of method bodies

Space restrictions prevent us from explaining the details about the translation of JAVA method bodies into type theory, as performed by the LOOP tool. Therefore we concentrate on what is relevant here, necessarily leaving many things unexplained. More details may be found in [14, 10, 3, 9]. So far we have used a type variable Self for the state space. In the actual translation, a fixed type OM is used. It represents the underlying memory model, see [3]. It consists of three parts: a heap, a stack, and a static part, each with an infinite series of memory cells and a 'top' position indicating the next unused cell. Several 'put' and 'get' operations are defined for writing and reading from this memory, at various locations.

The type-theoretic translation of the body of the method `n` from `A` looks as follows.

– TYPE THEORY

$$
\begin{array}{l}
c\colon \mathsf{OM} \to \mathsf{AIFace}[\mathsf{OM}] \ \vdash \\
\qquad \mathsf{nbody}(c) \ : \ \mathsf{OM} \to \mathsf{ExprResult}[\mathsf{OM}, \mathsf{int}] \ \stackrel{\text{def}}{=} \\
\qquad\qquad \lambda x\colon \mathsf{Self}.\ \mathsf{LET}\, \mathsf{ret\_n}\colon \mathsf{OM} \to \mathsf{int} = \mathsf{get\_int}(\mathsf{stack}(\mathsf{stacktop}(x), 0)), \\
\qquad\qquad\qquad\qquad \mathsf{ret\_n\_becomes}\colon \mathsf{OM} \to \mathsf{int} \to \mathsf{OM} = \\
\qquad\qquad\qquad\qquad\quad \mathsf{put\_int}(\mathsf{stack}(\mathsf{stacktop}(x), 0)) \\
\qquad\qquad\qquad \mathsf{IN}\ \big(\mathsf{CATCH\text{-}EXPR\text{-}RETURN}(\mathsf{stacktop\_inc}\,; \\
\qquad\qquad\qquad\qquad \mathsf{E2S}(\mathsf{A2E}(\mathsf{ret\_n\_becomes}(\mathsf{F2E}(\mathsf{i}(c)) + \mathsf{m}(c))))\,; \\
\qquad\qquad\qquad\qquad \mathsf{RETURN})(\mathsf{ret\_n})\ @@\, \mathsf{stacktop\_dec}\big)\,(x)
\end{array}
$$

We explain the basics: first, a special local variable ret_n is declared, together with an associated assignment operation, and is bound to a particular position on the stack. This variable ret_n is used to temporarily hold the result of the computation. At the end it is read by the CATCH-EXPR-RETURN function. But first, the stacktop is incremented, so that later method calls do not interfere with the values in the cell used for this method (where the value for ret_n is stored). The actual body `return i + m()` gets translated into an assignment of $\mathsf{F2E}(\mathsf{i}(c)) + \mathsf{m}(c)$ to the return variable ret_n_becomes, followed by the return statement RETURN—where F2E is an auxiliary function used to turn a function into an expression, just like E2S and A2E. At the very end, the stacktop is decremented again, freeing the used cell at the stack. The reader is not expected to understand all details about the translation of this method body, but that is not really needed at this stage. Hopefully, it does convey the main idea of what is going on, namely:

$$\mathsf{nbody}(c) = \boxed{\ \cdots\ \mathsf{i}(c) + \mathsf{m}(c)\ \cdots\ }$$

The important thing to note is that the definition of nbody is parameterised by an `A` coalgebra $c\colon \mathsf{OM} \to \mathsf{AIFace}[\mathsf{OM}]$. In the translation of `A`, the call $\mathsf{n}(c)$ rewrites to $\mathsf{nbody}(c)$. The whole trick in getting late binding to work correctly is to have the (repeated) extraction function $\mathsf{n}(d)$ for a `B` coalgebra $d\colon \mathsf{OM} \to \mathsf{BIFace}[\mathsf{OM}]$ rewrite to the method body $\mathsf{nbody}(\mathsf{B2A}(d))$, which is the body as in `A`, but with a casted coalgebra. The effect is summarised in the following table.

Class	binding	m in nbody	i in nbody
`A` with coalgebra $c\colon \mathsf{OM} \to \mathsf{AIFace}[\mathsf{OM}]$	$\mathsf{n}(c)$ to $\mathsf{nbody}(c)$	$\mathsf{m}(c)$	$\mathsf{i}(c)$
`B` with coalgebra $d\colon \mathsf{OM} \to \mathsf{BIFace}[\mathsf{OM}]$	$\mathsf{n}(d)$ to $\mathsf{nbody}(\mathsf{B2A}(d))$	$\mathsf{m}(\mathsf{B2A}(d)) = \mathsf{m}(d)$	$\mathsf{i}(\mathsf{B2A}(d)) = \mathsf{i}(\mathsf{super\_A}(d))$

This is precisely what we want, namely that the method call to m in n in B, *i.e.* m in nbody(B2A(d)), is m from B, whereas i in nbody(B2A(d)) is i from A. The coalgebra by which nbody is parametrised thus formalises the method table of the current object.

In conclusion, late binding is realised by binding in subclasses the repeated extraction functions of methods from superclasses to the bodies from the superclasses, but with casted coalgebras.

7 Method calls to other objects

In this section we consider method calls of the form `o.m()`, where `o` is a "receiving" or "component" object. Field access `o.i` is not discussed explicitly, but is handled similarly. Examples occurred in Section 5, where `o` was an array access `ar[0]` or `ar[1]`.

So far we have been using coalgebras with type $\mathsf{OM} \to \mathsf{AIFace}[\mathsf{OM}]$ to capture the ingredients of a class A. These coalgebras actually have two more parameters, namely a memory position, of type MemLoc, and a string. The memory position points to the location in memory where the values of the fields of the object are stored. The string can be the name of the class that the coalgebra itself represents (like "A"), or the name of one of its subclasses, representing the run-time type of an object. Thus, we use coalgebras of type $\mathsf{string} \to \mathsf{MemLoc} \to \mathsf{OM} \to \mathsf{AIFace}[\mathsf{OM}]$.

For each class, say A, a specific coalgebra $\mathsf{A\_clg}\colon \mathsf{string} \to \mathsf{MemLoc} \to \mathsf{OM} \to \mathsf{AIFace}[\mathsf{OM}]$ is assumed, with requirement:

$$\mathsf{A\_clg}(\text{“A”}) \text{ implements } \mathtt{A}.$$

This implementation requirement expresses that fields, methods and constructors in A_clg("A") behave as described, for example, in their method bodies. If A has a subclass B, then additional requirements are imposed, namely,

$$\mathsf{A\_clg}(\text{“B”})(p) = \mathsf{B2A}(\mathsf{B\_clg}(\text{“B”})(p)) \qquad \text{and} \qquad \mathsf{B\_clg}(\text{“B”}) \text{ implements } \mathtt{B}$$

(And similarly, for further subclasses.) The first of these requirements expresses that the implementation of A on an object with run-time type B behaves like the implementation of B, casted to A.

Why is this relevant? Consider a JAVA method invocation expression `o.m()`, where the receiving object `o` has static type A, say, and `m` is non-void (*i.e.* has a return type). This expression is translated via an auxiliary function CE2E[4], namely as $[\![\mathtt{o.m()}]\!] = \mathsf{CE2E}(\mathsf{A\_clg})([\![\mathtt{o}]\!])(\mathsf{m})$. This function CE2E first evaluates $[\![\mathtt{o}]\!]$; if $[\![\mathtt{o}]\!]$ terminates normally, this produces a value in RefType, see the end of Section 3. In case this result is a null-reference, a `NullPointerException` will be thrown; if it is a non-null-reference, it contains a memory location p and a string s (describing o's run-time type). The method $\mathsf{m}(\mathsf{A\_clg} \cdot s \cdot p)$ will then be evaluated,

[4] CE2E stands for "Component-Expression-to-Expression".

corresponding to execution of `m` by the receiving object `o`—stored at location p in OM—with the run-time type of `o` determining the implementation of `m` that is chosen. Using the implementation requirements on coalgebras from the previous paragraph and the coalgebra-cast functions, the appropriate body for m is found. All this is in accordance with the explanation of method invocation in the JAVA language specification [7, §§15.11].

The function $\mathsf{CE2E}$ is defined as follows.

– TYPE THEORY

$$
\begin{array}{l}
c\colon \mathsf{string} \to \mathsf{MemLoc} \to \mathsf{OM} \to \mathsf{IFace},\\
o\colon \mathsf{OM} \to \mathsf{ExprResult}[\mathsf{OM}, \mathsf{RefType}],\\
m\colon (\mathsf{OM} \to \mathsf{IFace}) \to \mathsf{OM} \to \mathsf{ExprResult}[\mathsf{OM}, \alpha] \ \vdash\\
\qquad \mathsf{CE2E}(c)(o)(m) \ : \ \mathsf{OM} \to \mathsf{ExprResult}[\mathsf{OM}, \alpha] \ \stackrel{\text{def}}{=}\\
\qquad\qquad \lambda x\colon \mathsf{OM}.\ \mathsf{CASES}\ o \cdot x\ \mathsf{OF}\ \{\\
\qquad\qquad\qquad\qquad\quad \mid \mathsf{hang} \mapsto \mathsf{hang}\\
\qquad\qquad\qquad\qquad\quad \mid \mathsf{norm}\, y \mapsto\\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{CASES}\ y.\mathsf{res}\ \mathsf{OF}\ \{\\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{null} \mapsto \text{“}\texttt{NullPointerException}\text{”}\\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{ref}\, r \mapsto m\big(c \cdot (r.\mathsf{clname}) \cdot (r.\mathsf{objpos})\big) \cdot (y.\mathsf{ns})\ \}\\
\qquad\qquad\qquad\qquad\quad \mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}\, a\ \}
\end{array}
$$

Notice that such a method invocation hangs, or terminates abruptly if the receiving object o does, and also that the possible side-effect of evaluating o is passed on to the method m, via the state $y.\mathsf{ns}$. The details of how exceptions are thrown are not relevant here, and are omitted.

The main point is: if we have an object `a` in a class `A` and an object `b` in a subclass `B` of `A`, both with a method `m`, then after an assignment `a = b` the run-time type of `a` (given by the clname label) is equal to the run-time type of `b`, and so a method invocation `a.m()` will have the same effect as `b.m()`, since

$$\mathsf{m}\big(\mathsf{A\_clg}(\text{“B”})(p)\big) = \mathsf{m}\big(\mathsf{B2A}(\mathsf{B\_clg}(\text{“B”})(p))\big) = \mathsf{m}\big(\mathsf{B\_clg}(\text{“B”})(p)\big).$$

where p is the memory location of `a` (and `b`). But note that a field access expression `a.i` may yield a different result from `b.i`!

8 Example verifications

Next will be described how the semantics, as sketched in the previous sections, is used for tool-supported reasoning about JAVA classes. Actually, no explicit reasoning principles are needed for handling inheritance, because automatic rewriting takes care of proper method selection. Therefore, inheritance requires no special attention in verification—but remains difficult in specification. This is of course very convenient, and a good reason for using this particular semantics.

We shall describe two example verifications, based on translations of JAVA programs by the LOOP tool. The first verification is in PVS, and the second one in ISABELLE/HOL. Here we shall no longer use the type-theoretic syntax of earlier sections, but use PVS and ISABELLE syntax. The first verification is about the JAVA classes in Section 5, and establishes the properties mentioned there. The PVS statements that have been proved are:

– PVS

```
IMPORTING ...  % code generated by the LOOP tool is loaded

test1 : LEMMA  p < heap?top(x) IMPLIES
    norm??(test1?(Test?clg("Test")(p))(x))
      AND
    res?(test1?(Test?clg("Test")(p))(x)) = 10102

test2 : LEMMA  p < heap?top(x) IMPLIES
    norm??(test2?(B?clg("B")(p))(x))
      AND
    res?(test2?(B?clg("B")(p))(x)) =
      i(B?2?A(B?clg("B")(p)))(x) + i(B?clg("B")(p))(x) * 1000
```

The first lemma `test1` states that evaluation of `test1` terminates normally, returning `10102`. The second lemma states that evaluation of `test2` also terminates normally, and the return value equals the value of `i` from `A`, plus 1000 times the value of `i` from `B`.

The PVS code contains lots of question marks '`?`', which are there only to prevent possible name clashes with JAVA identifiers (which cannot contain '`?`'). Both lemmas have a technical assumption `p < heap?top(x)` requiring that the position `p` of the receiving object is in the allocated part of the heap memory. The proofs of both these lemmas proceed entirely by automatic rewriting[5], and the user only has to tell PVS to load appropriate rewrite rules, and to start reducing. The functions CE2E and B2A play a crucial rôle in this verification. Hopefully the reader appreciates the semantic intricacies involved in the proof of the first lemma: array creation and access, local variables, object creation, implicit casting, and late binding.

The second verification deals with the following JAVA program.

[5] To give an impression, the proof of `test1` involves 790 rewrite steps, taking about 67 sec., on a 450 Mhz. Pentium III with 128 MB RAM under Linux.

— JAVA

```
class C {
  void m() throws Exception { m(); }
}
class D extends C {
  void m() throws Exception { throw new Exception(); }
  void test() throws Exception { super.m(); }
}
```

At a first glance, one might think that evaluation of the method `test` will not terminate, but on the contrary, it throws an exception. In the body of `test` the method `m` of `C` is called. This method calls `m` again, but—due to late binding—this results in execution of `m` in `D`. However, if `m` is called on an instance of class `C` directly, this will not terminate. The ISABELLE/HOL statements that have been proved are the following.

— ISABELLE

```
(* Code generated by the LOOP tool is loaded *)
Goal "p < heap_top x ==> \
\     case DInterface.test_ (D_clg ''D'' p) x of \
\       Hang     => False\
\      |Norm y   => False\
\      |Abnorm a => True";
(* Simplifier *)
qed "m_in_D_Abnorm";

Goal "p < heap_top x ==> \
\     case CInterface.m_ (C_clg ''C'' p) x of \
\       Hang     => True\
\      |Norm y   => False\
\      |Abnorm a => False";
(* Proof *)
qed "m_in_C_hangs";
```

These lemmas state that evaluation of `m` on an object with run-time type `D` will terminate abnormally, while evaluation of `m` on an object with run-time type `C` will not terminate, *i.e.* will hang.

In the ISABELLE code the full name (including the theory name) is used for the extraction functions. This is to prevent name clashes, due to overloading. Again, the technical assumption `p < heap_top x` is used. The proof of the first lemma proceeds entirely by automatic rewriting[6], after the user has added appropriate rewrite rules to the simplifier. The crucial point in this verification is the binding

[6] On a Pentium II 266 Mhz with 96 MB RAM, running Linux, this takes about 71 sec, involving 5070 rewrite steps—including rewriting of conditions.

of the extraction function for `super.m` on a `D` coalgebra $d\colon \mathsf{OM} \to \mathsf{DIFace}[\mathsf{OM}]$ to the method body $\mathsf{C\_mbody}(\mathsf{D2C}(d))$.

The verification of the second lemma requires some more care, since it can not be done via automatic rewriting (as this would loop). To prove non-termination, several unfoldings and an appropriate induction are necessary.

9 Conclusions

We have described the main ideas of the inheritance semantics in the LOOP project for reasoning about JAVA classes, and shown the practical usability of this semantics in two example verifications, where late binding was handled by automatic rewriting, both in PVS and in ISABELLE/HOL.

Acknowledgements

Thanks are due to to Hans Meijer and Erik Poll for their useful feedback on the manuscript, and to Joachim van den Berg for suggesting the JAVA example in the second verification in Section 8.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects.* Monographs in Comp. Sci. Springer, 1996.
2. K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.
3. J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. Techn. Rep. CSI-R9924, Comput. Sci. Inst., Univ. of Nijmegen, 1999.
4. K.B. Bruce, L. Cardelli, and B.C. Pierce. Comparing object encodings. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software*, number 1281 in Lect. Notes Comp. Sci., pages 415–438. Springer, Berlin, 1997.
5. L. Cardelli. A semantics of multiple inheritance. *Inf. & Comp.*, 76(2/3):138–164, 1988.
6. W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *Inf. & Comp.*, 114(2):329–350, 1995.
7. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
8. M. Hofmann, W. Naraschewski, M. Steffen, and T. Stroup. Inheritance of proofs. *Theory & Practice of Object Systems*, 4(1):51–69, 1998.
9. M. Huisman. *Reasoning about Java Programs in Higher-Order Logic, using PVS and Isabelle/HOL.* PhD thesis, Univ. Nijmegen, 2000. Forthcoming.
10. M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Fundamental Approaches to Software Engineering*, number 1783 in Lect. Notes Comp. Sci. Springer, Berlin, 2000.
11. B. Jacobs. Inheritance and cofree constructions. In P. Cointe, editor, *European Conference on Object-Oriented Programming*, number 1098 in Lect. Notes Comp. Sci., pages 210–231. 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*, pages 83–103. Kluwer Acad. Publ., 1996.
13. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.
14. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object-Oriented Programming, Systems, Languages and Applications*, pages 329–340. ACM Press, 1998.
15. J.C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Principles of Programming Languages*, pages 109–124. ACM Press, 1990.
16. W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics*, number 1479 in Lect. Notes Comp. Sci., pages 349–366. Springer, Berlin, 1998.
17. S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.
18. L.C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and computer science*, pages 361–386. Academic Press, London, 1990. The APIC series, vol. 31.
19. Loop Project. `http://www.cs.kun.nl/~bart/LOOP/`.