A Formal Framework with Late Binding*

Davide Ancona, Maura Cerioli, and Elena Zucca

DISI - Università di Genova
Via Dodecaneso, 35, 16146 Genova (Italy)
fax: +39-10-3536699
email: {davide,cerioli,zucca}@disi.unige.it

Abstract. We define a specification formalism (formally, an institution) which provides a notion of *dynamic type* (the type which is associated to a term by a particular evaluation) and *late binding* (the fact that the function version to be invoked in a function application depends on the dynamic type of one or more arguments). Hence, it constitutes a natural formal framework for modeling object-oriented and other dynamically-typed languages and a basis for adding to them a specification level. In this respect, the main novelty is the capability of writing axioms related to a given type which are not required to hold for subtypes, hence can be "overridden" in further refinements, thus lifting at the specification level the possibility of reusing code which is offered by the object-oriented approach.

Introduction

After many years of research on the foundations of object-oriented programming, a point of view which has recently emerged [2] is to consider as its distinguishing feature the fact that in method calls the correct variant of the method to be invoked is determined at run-time (*late* or *dynamic binding*), in other words a policy of *dynamic resolution of overloading* is applied.

We clarify further this terminology, adapting more or less the presentation of [2]. A distinction extensively used in language theory for the last two decades is that between *parametric* (or *universal*) polymorphism and *ad hoc* polymorphism. Parametric polymorphism allows one to write a function whose code can work on arguments of different types, while by ad hoc polymorphism it is possible to write functions that execute different code for arguments of different types. The first kind of polymorphism has been widely investigated on and developed, while the second form, usually known as *overloading*, has had little theoretical attention. This is probably due to the fact that traditional programming languages offer a very limited form of overloading, where the correct variant of the function to be applied is always decided at compile time, i.e. the *overloading resolution* is static (*early* or *static binding*). Clearly this form of overloading can be reduced to a useful syntactic abbreviation which does not significantly affect the language.

The real gain of power of overloading occurs with languages where types are computed during the execution. Indeed, in this case, the correct variant of the function to be applied can be decided depending on the *dynamic type* of arguments, i.e. the type computed at run-time; hence there is *late binding* of the function name to the code to be executed, or, in other words, dynamic resolution of overloading. This happens typically in object-oriented languages. In most of them dynamic overloading resolution is adopted only for what concerns the "receiver" (i.e. the first, implicit, parameter) of a method call. Nevertheless, the same policy can be applied to all the arguments of a function, as it happens e.g. in *multimethods* [8, 4].

* Partially supported by Murst 40% - Modelli della computazione e dei linguaggi di programmazione and CNR - Formalismi per la specifica e la descrizione di sistemi ad oggetti.

In this paper we define a specification formalism (formally, an institution in the sense of [6]) which provides dynamic resolution of overloading. The basic idea is to handle overloading at the *model* (semantic) level and not at the *signature* (syntactic) level. More precisely, we model a function $op: \overline{s} \to s$ which[1] has many different variants, as e.g. a method in the object-oriented case, by a unique function symbol in the signature, whose interpretation in a model is a *multifunction*, i.e. a family of functions $op_{\overline{u}}$, one for each existing subtype $\overline{u}$ of $\overline{s}$. In other words, the existence of different variants is seen as *redefinition*, and distinguished from static overloading.

We see two main motivations and directions of application for the approach we propose.

First, we provide a formal framework for modeling object-oriented languages or, more in general, languages which provide some form of dynamic overloading resolution. In particular, the term language of our formalism is, syntactically, a variant of the standard term language in, say, order-sorted frameworks [7], but we are able to define what is the *dynamic type* of a term (the type of the element obtained by its evaluation). Furthermore, in the evaluation of a function application the variant to be used is (possibly) determined using late binding. Hence, our term language with its semantic interpretation provides a unifying metalanguage allowing to express by immediate translation the semantics of languages with dynamic overloading resolution. Applications of this kind rely on just the model part (signatures and models) of our formalism.

Considering now the logical part (sentences and satisfaction relation), the main novelty is that we are able to express two different kinds of requirements over elements of a given type s:

- requirements which must hold for elements of any possible subtype of s, written $\forall x : s_{\leqslant}.\varphi$;
- requirements which must hold for elements which have s as most specific type, but not for elements of proper subtypes, written $\forall x : s.\varphi$.

Note that requirements of the second kind are not expressible by sentences of "usual" formalisms (without a notion of dynamic type).

This possibility is very interesting since it allows, in a sense, to lift at the specification level the possibility of reusing code which is the main advantage offered by the object-oriented approach (and more in general by late binding). Indeed, assume that we have a specification SP which describes a type s and its related functions. Later, we want to define a specialization s' of s which behaves "more or less" like s, but for instance one of the functions must be changed in a way that it does no longer satisfy some axiom, say $\forall x : s.\varphi$. In usual frameworks it is not possible to obtain a specification SP' of this specialization by reusing SP as it stands, i.e. by enrichment. Indeed, φ is required to hold for each element of (a subtype of) s. Hence, we have to give up either to reuse specifications or to write axioms which we are not sure should hold in all possible present or future refinements.

In our framework, since the sentence $\forall x : s.\varphi$ is not required to hold for subtypes, this axiom can coexist with other axioms for the subtype s' which specify a different behavior, even in contradiction. In other words, it is possible to have reuse of specifications possibly "overriding" some axioms, in analogy to what happens with programming languages. In our opinion this represents a true novelty, and makes our formalism a good starting point for adding a specification level to languages with late binding.

The paper is structured as follows. In Sect. 1 we provide an informal introduction to the formalism, showing how some simple Java classes can be semantically interpreted and how to write axioms specifying their expected behavior. In Sect. 2 we formally define

[1] Here and later let $\overline{x}$ denote the tuple $x_1 \ldots x_n$.

our formalism as an institution in the sense of [6] and prove that it also satisfies the amalgamation property. In Sect. 3 we discuss the interference between the redefinition (dynamically solved overloading) modeled in our framework and usual (statically solved) overloading. Finally, in the Conclusion we summarize the results of the paper and provide some comparison with related work.

Proofs are omitted here for reasons of space and can be found in [1].

1 An Informal Presentation

In this section, we will first show how our framework provides a semantic foundation for programming languages with late binding, typically object-oriented languages. Then, we will illustrate how it could be taken as starting point for adding a specification level to such languages.

We will consider a standard example written in a toy object-oriented language. We adopt a Java-like syntax for convenience, but consider in the following for simplicity a purely functional interpretation, where objects are records and a method with side effects on some object's components is seen instead as a function returning a record consisting of the updated components. Indeed, handling imperative aspects is orthogonal to the problems we aim at solving in this paper and there are standard techniques for that (see e.g. [11]), which could be applied to our framework as well.

Let us conside a software module consisting of two class definitions, describing points in the Cartesian plane (2D-points) and in the space (3D-points). We assume that, for some reason, moving a 3D-point has the side effect of incrementing by one its third coordinate.

```
class 2DPoint {
  private int x,y;
  int X () { return x;}
  int Y () { return y;}
  void move (int dx, int dy) { x = x + dx; y + dy;}
}

class 3DPoint extends 2DPoint {
  private int z;
  int Z () { return z;}
  void move (int dx, int dy) { x = x + dx; y = y + dy; z = z + 1;}
}
```

The semantic counterpart of this module is given, in our framework, by a signature Σ_P (P for "points") modeling the syntactic interface of the module to users and a model M_P over Σ_P providing an interpretation for symbols in the interface.

The definition of Σ_P is the following:

sig Σ_P =
  sorts $int, 3D \leq 2D$
  opns ...standard functions on integers ...
    $X, Y \colon 2D \to int$
    $move \colon 2D, int, int \to 2D$
    $Z \colon 3D \to int$

Note that there is a unique function symbol corresponding to `move`; indeed, as we will see below, redefinition is handled at the semantic level.

The model M_P defines an interpretation for the sort and function symbols. Sort symbols are interpreted as sets, as usual.

$int^{M_P} = \mathbb{Z}, 2D^{M_P} = \mathbb{Z} \times \mathbb{Z}, 3D^{M_P} = \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$

However, the intended meaning is different from the standard; in a model M, the interpretation s^M of a sort s defines the set of the elements which have s as most specific type, called the *proper carrier* of sort s (for instance, 2D-points for $2D$). The *(extended) carrier* of sort s, denoted $s^M_{\leqslant}$, is then defined to be the disjoint union of all the proper carriers of subtypes of s. In the example:

$$2D^{M_P}_{\leqslant} = \{p : 2D \mid p \in 2D^{M_P}\} \cup \{p : 3D \mid p \in 3D^{M_P}\},$$

while $3D^{M_P}_{\leqslant} = 3D^{M_P}$ since $3D$ has no proper subtypes. Note that by definition $s^M_{\leqslant} \subseteq {s'}^M_{\leqslant}$ holds whenever $s \leq s'$.

We illustrate now the interpretation of function symbols. As said above, redefinition of a method in a heir class is modeled at the semantic level. Indeed, the interpretation of *move* consists of two different functions, called *variants*, as shown below.

$$\begin{array}{l}
move^{M_P}_{2D}: 2D^{M_P}_{\leqslant} \times int^{M_P} \times int^{M_P} \to 2D^{M_P}_{\leqslant} \\
\quad move^{M_P}_{2D}(p, dx, dy) = \\
\quad\quad \langle x + dx, y + dy\rangle : 2D \text{ if } p = \langle x, y\rangle : 2D, \\
\quad\quad \langle x + dx, y + dy, z\rangle : 3D \text{ if } p = \langle x, y, z\rangle : 3D \\
move^{M_P}_{3D}: 3D^{M_P} \times int^{M_P} \times int^{M_P} \to 2D^{M_P}_{\leqslant} \\
\quad move^{M_P}_{3D}(\langle x, y, z\rangle, dx, dy) = \langle x + dx, y + dy, z + 1\rangle
\end{array}$$

The latter variant, acting on 3D-points only, corresponds to the new definition of the method in `3DPoint`; the former corresponds to the old definition, which can be invoked not only on 2D-points, but also on 3D-points (for instance in Java using the `super` keyword in the class `3DPoint`).

In general, the interpretation in a model M of a function symbol, say $op: \overline{s} \to s$, is a *multifunction*, i.e. a family of functions $op^M_{\overline{u}}: \overline{u}^M_{\leqslant} \to s^M_{\leqslant}$, one for each subtype $\overline{u}$ of $\overline{s}$. Of course in practice a function could be redefined only for some subtypes, but we assume that a model provides a variant for each possible subtype for sake of simplicity (no redefinition being simply obtained by having two variants which have the same behavior on arguments of the more specific type). Domain and codomain of each variant $op^M_{\overline{u}}$ are the extended carriers of $\overline{u}$ and s, reflecting the intuition that this variant could be applied to argument tuples of any subtype of $\overline{u}$ and, analogously, the result could be an element of any subtype of s. Note moreover that multiple inheritance can be modeled as well.

Assume now that we want to use the module consisting of the two classes `2DPoint` and `3DPoint` as an implementation for the following restricted interface Σ_{2D}.

sig Σ_{2D} =
  sorts $int, 2D$
  opns ... standard functions on integers ...
    $X, Y: 2D \to int$
    $move: 2D, int, int \to 2D$

This is intuitively sensible since the module has a richer interface and is captured, in the algebraic formalisms, by the notion of *reduct*; that is, it should be possible to define, starting from M_P, a model $M_{P|\Sigma_{2D}}$ over Σ_{2D} which is the formal counterpart of the behavior of the module when accessed only through the restricted interface. Intuitively, $M_{P|\Sigma_{2D}}$ should be obtained from M_P "forgetting" the type $3D$ and the function Z. However, this cannot be achieved by simply throwing away 3D-points, since 3D-points and the corresponding variant of *move* are still available because of late binding. Assume, to see this more clearly, that Σ_{2D} also offers a constant $myPoint: \to 2D$ whose implementation in M_P returns a 3D-point (this could correspond for instance in Java to a static method in the class `2DPoint` with body `return new 3DPoint()`). Then, the evaluation of the term $move(myPoint, 1, 1)$ (`2DPoint.myPoint.move(1,1)` in Java syntax) in the model M_P involves the $move_{3D}$ variant even if the user has no knowledge of the existence of the type `3DPoint`.

The technical solution we adopt is that of decoupling *syntactic types* (types which appear in the signature, and are used for static typing of terms) and *semantic types*, which are used as indexes in function variants. Hence a model M over a signature Σ with sorts S defines a set s^M for each s in a set of semantic types S^M and an (order-preserving) map $\iota : S \to S^M$ which specifies how syntactic types are mapped into semantic types. For instance, in the model $M_{P|\Sigma_{2D}}$ this map is just an inclusion, corresponding to the intuition that the type $3D$ is not visible at the syntactic level, but can be the dynamic type of some term.

In order to illustrate the logical part of our formalism, i.e. sentences expressing requirements over models, we first briefly present *terms*. Assume that $p2$ and $p3$ are two variables of (static) type $2D$ and $3D$, respectively. Terms are of three kinds: variables, function applications and casted terms.

A first form of function application is exemplified by $move(p2, 1, 1)$, $move(p3, 1, 1)$, $Z(p3)$. In this case, the syntactic form of terms is standard, but not the evaluation. Let us consider for instance the evaluation of $move(p2, 1, 1)$ w.r.t., say, M_P; reflecting what actually happens in languages with subtyping, the variable $p2$ can denote either a 2D-point or a 3D-point (formally, valuations of variables have as codomain extended carriers), and the variant used for interpreting $move$ (either $move_{2D}^{M_P}$ or $move_{3D}^{M_P}$) depends on the *dynamic type* of $p2$, that is the (most specific) type of the element denoted by $p2$.

Another form of function application is that where we force static binding, like e.g. in $move(p2 : 2D, 1, 1)$. In this second case, even if the dynamic type of $p2$ is $3D$, the variant which is invoked is that with index $2D$.

The combination of the first two forms where terms are of the general form $op(t_1[: s_1], \ldots, t_n[: s_n])$, where square brackets denote optionality, allows much more flexibility than in object-oriented languages, where the binding is always dynamic for the first (implicit) argument (apart the limited possibility offered by the `super` mechanism) and always static for the remaining arguments.

Finally, we allow casted terms, like e.g. $(3D)p2$. This feature is instead completely analogous to what offered e.g. in Java by *casting down* (type conversion from a supertype to a subtype) and is typically useful for using in a context of the subtype (hence through a richer interface) a term which we expect to have this subtype as dynamic type. For instance, if we write $Z((3D)p2)$, then we are able to get the third dimension of the point denoted by $p2$, if we have good reasons to suppose that this point is actually a 3D-point. If, on the other side, our supposition is wrong and $p2$ denotes a 2D-point, then we get a run time error (formally, the evaluation of the term is undefined). Note however that, as in Java, casting does not influence the dynamic type, hence in $move((3D)p2, 1, 1)$ the version which is invoked is still $move_{3D}^{M_P}$ if $p2$ denotes a 3D-point. Finally, we recall that this casting down is conceptually analogous to the so-called *retracts* in order-sorted frameworks [7], even if the technical treatment is different.

We do not consider casting up (type conversion from a subtype to a supertype) since it is not significant except than for static overloading resolution (see Sect. 3 for more details).

We can now show some sentences. Let us consider for instance the following axioms expressing requirements for the type $2D$.

1. $\forall p : 2D_{\leqslant}.\forall dx : int.\forall dy : int.X(move(p, dx, dy)) = X(p) + dx$
2. $\forall p : 2D_{\leqslant}.\forall dx : int.\forall dy : int.Y(move(p, dx, dy)) = Y(p) + dy$
3. $\forall p : 2D.move(p, 0, 0) = p$

4. $\forall p : \mathit{2D}.\forall dx : int.\forall dy : int.move(move(p, dx, 0), 0, dy) = move(p, dx, dy)$

The first two axioms express, intuitively, requirements that we want to hold for 2D-points and to be also preserved in any possible specialized version of them. For instance, these axioms must be verified by 3D-points, too.

On the contrary, the last two axioms express perfectly reasonable requirements for the type *2D*, which could however not hold for all subtypes; for instance, these axioms do not hold for 3D-points, since the *move* function is assumed to have an additional effect on them.

The difference between the two kinds of axioms is expressed by the two different quantifications, suggesting exactly the interpretation explained above.

The possibility of writing in a specification axioms of the two kinds is very important: for instance, we can write a specification SP_{2D} for 2D-points including the axioms above and then write a specification SP_{3D} for 3D-points as an enrichment of SP_{2D} by adding, e.g., the axiom[2]

$$\forall p : \mathit{3D}.\forall dx : int.\forall dy : int.Z(move(p, dx, dy)) = Z(p) + 1.$$

In usual formalisms, this axiom and axiom (3) and (4) would be in contradiction, hence the specification SP_{3D} could not be obtained by reusing SP_{2D}, but should be rewritten from scratch.

From the methodological point of view, the designer should choose between the two forms of quantification depending on the intuition about the property expressed by the axiom (either a *conservative* property, expected to hold in all the possible future refinements of a type, or a *specific* property, required to hold for that type but not necessarily for subtypes). This allows to have in the specification any desired degree of control over inheritance.

2 An Institution for Late Binding

This section is devoted to the formal definition of our framework. The readers are encouraged to compare the notions presented here with the examples of application of the previous section.

2.1 Syntax

The subtyping relation is represented by a *preorder*; thus, our notion of signature is similar to that of order-sorted signatures [7]. But, for sake of simplicity, we do not allow static overloading: that is, we do not allow the same function symbol to have two different functionalities. This is somehow drastic, because many cases of overloading are harmless and helpful, but allows us to focus our attention on *redefinition* (or *dynamic overloading*). In Sect. 3 we will see how this requirement can be relaxed, following quite standard techniques, in order to get a more user-friendly language.

Definition 1. *A* preorder $(S, \leq)$ *consists of a set* S *and a reflexive and transitive binary relation* $\leq$ *on* S. *Given preorders* $(S, \leq)$ *and* $(S', \lesssim)$, *a* morphism of preorders *is a function* $\sigma^S : S \to S'$ *s.t.* $s \leq s'$ *implies* $\sigma^S(s) \lesssim \sigma^S(s')$ *for all* $s, s' \in S$.

Any preorder $\leq$ *can be componentwise extended to sequences, that is* $\overline{s} \leq \overline{u}$ *holds iff* $\overline{s}$ *and* $\overline{u}$ *have the same length* n *and* $s_i \leq u_i$ *for all* $i = 1 \ldots n$. *We denote by* ***PreOrd*** *the category of preorders.*

[2] We use the standard quantification for integers, since we consider them as a fixed predefined data type which cannot be furtherly specialized.

An order-sorted signature without overloading, *from now on simply* signature, $\Sigma = (S, \leq, O, \delta 0, \delta 1)$ *consists of a preorder* $(S, \leq)$ *of* sorts, *a set* O *of* function symbols *and two functions* $\delta 0 \colon O \to S^*$ *and* $\delta 1 \colon O \to S$ *returning the* arity *and the* result type *of each function symbol, respectively.*

For each $op \in O$*, if* $\delta 0(op) = \overline{s}$ *and* $\delta 1(op) = s$ *then we will write* $op \colon \overline{s} \to s$*.*

Definition 2. *Given signatures* $\Sigma = (S, \leq, O, \delta 0, \delta 1)$ *and* $\Sigma' = (S', \lesssim, O', \delta 0', \delta 1')$*, a morphism* $\sigma \colon \Sigma \to \Sigma'$ *consists of a morphism of preorders* $\sigma^S \colon S \to S'$ *and a renaming of functions symbols* $\sigma^F \colon O \to O'$ *that is consistent with the sort renaming, that is s.t. for all* $op \colon \overline{s} \to s$ *we have* $\sigma^F(op) \colon \sigma^S(\overline{s}) \to \sigma^S(s)$*, where* σ^S *is componentwise extended to sequences of sorts.*

The category of signatures, where identities and compositions are defined in the obvious way, will be denoted by ***OSign****.*

Notation 1 *From now on, let* Σ *denote the signature* $(S, \leq, O, \delta 0, \delta 1)$*,* Σ' *the signature* $(S', \lesssim, O', \delta 0', \delta 1')$ *and* σ *the morphism* $\sigma \colon \Sigma \to \Sigma'$ *with components* (σ^S, σ^F)*. In the following we will simply denote* σ^S *and* σ^F *by* σ*, provided that no ambiguity arises.*

2.2 Semantics

As shown in Sect. 1, the models of a signature Σ in our framework are quite different from usual order-sorted algebras. Indeed, first of all the elements of a model are classified by a set of *semantic types*. This captures the intuition that we are using modules with a possibly larger collection of types through a restricted interface, described by the signature. Then, the interpretation of each semantic type (the *proper carrier*) consists only of the elements having that sort as most specific type. Thus, we have no subsumption among the proper carriers, though this property is recovered at the level of *extended carriers*. Finally, each function symbol $op \colon \overline{s} \to s$ of the signature is interpreted in a model M as a *multifunction*, that is a family of functions $op^M_{\overline{u}} \colon \iota(\overline{u})^M_{\leqslant} \to \iota(s)^M_{\leqslant}$ for each possible specialization $\overline{u} \leq \overline{s}$.

Definition 3. *A* Σ-model M *consists of*

- *a preorder* $(S^M, \preceq)$ *and a morphism of preorders* $\iota \colon (S, \leq) \to (S^M, \preceq)$*; if* ι *is an inclusion (as in most concrete cases), we will omit it;*
- *for each* $s \in S^M$*, a set* s^M*, called the* proper carrier *of sort* s *in* M*. We will denote by* $s^M_{\leqslant}$ *the disjoint union of* ${s'}^M$ *for all* $s' \preceq s$*, i.e. for all* $s \in S^M$*:*

$$s^M_{\leqslant} = \{a : s' \mid a \in {s'}^M, s' \preceq s\}$$

  The set $s^M_{\leqslant}$ *is called the* (extended) carrier *of sort* s*. Moreover, for each sequence* $\overline{s}$ *of sorts in* S^M *with length* n*, we will denote by* $\overline{s}^M_{\leqslant}$ *the Cartesian product* ${s_1}^M_{\leqslant} \times \ldots \times {s_n}^M_{\leqslant}$*.*
- *for each* $op \colon \overline{s} \to s$ *and each* $\overline{u} \preceq \iota(\overline{s})$*, a partial function* $op^M_{\overline{u}} \colon \iota(\overline{u})^M_{\leqslant} \to \iota(s)^M_{\leqslant}$*.*

We will denote by ***OMod***(Σ) *the set of all* Σ*-models.*

We allow the interpretation of function symbols to be *partial* in order to be able to give an immediate semantics to programming languages, where partiality is inherent, due to non-termination.

Notation 2 *Here and later let* M *denote the* Σ*-model with components* $(S^M, \preceq)$*,* ι*,* s^M *and* $op^M_{\overline{u}}$ *and, analogously,* M' *denote the* Σ'*-model with components* $(S^{M'}, \precsim)$*,* ι'*,* $s^{M'}$ *and* $op^{M'}_{\overline{s}}$*.*

We are not interested here in features related with model morphisms, like initiality. We plan to study the nature of the *category* of models in an extended forthcoming version of the paper.

Lemma 1 (Subsumption). *Let M be a model of Σ. Then, $s \preceq s'$ implies $s^M_{\leqslant} \subseteq s'^M_{\leqslant}$.*

As usual in algebraic approaches, we have to say how models can be translated along signature morphisms, in order to support structured specifications.

Definition 4. *Let the* reduct *M of M' along σ, denoted $M'_{|\sigma}$, be defined by*

- *$(S^M, \preceq) = (S^{M'}, \precsim)$ and ι is the composition of ι' and σ.*
- *for each $s \in S^M$, $s^M = s^{M'}$;*
- *for each function symbol $op: \overline{s} \to s$ and each $\overline{u} \preceq \iota(\overline{s})$ we define $op^M_{\overline{u}} = \sigma(op)^{M'}_{\overline{u}}$.*

Moreover, let $\boldsymbol{OMod}(\sigma)$: $\boldsymbol{OMod}(\Sigma') \to \boldsymbol{OMod}(\Sigma)$ be the function defined by $\boldsymbol{OMod}(\sigma)(M') = M'_{|\sigma}$.

Proposition 1. *Using the notation of Def. 3 and 4, $\boldsymbol{OMod}$: $\boldsymbol{OSign} \to \boldsymbol{Set}^{op}$ is a functor.*

Here we allow resolution of dynamic overloading on all the parameters, like in *multimethods*, while in most object-oriented languages it is solved only w.r.t. the first (implicit) parameter, the receiver. Thus, if we want to use our formalism only to give semantics to such languages, then we can simplify the models, requiring that for each $op: \overline{s} \to s$ the interpretation of op consists of a partial function $op^M_{\overline{u}}: \iota(\overline{u})^M_{\leqslant} \to \iota(s)^M_{\leqslant}$, for each $\overline{u}$ s.t. $u_1 \preceq s_1$ and $u_i = s_i$ for all $i = 2 \dots n$. In this way, models are "smaller", because multifunctions have less variants. The theory developed for this less general case is, hence, a trivial simplification of the theory presented here.

2.3 Amalgamated Sum

Even though not crucial for any kind of institution, it is widely recognized that the amalgamation property [5] makes institutions particularly suitable for dealing in an elegant way with modular software systems and specifications. Indeed, as already stated in the Introduction, two main motivations for this work are the definition of an algebraic framework for modeling languages with late binding and the possibility to lift at the specification level the reuse of code typical of the object-oriented approach. This implies that we have to face the problem of modularization at the level both of models (indeed, object-oriented programs are usually structured) and specifications (since, for reusing specifications, we necessarily deal with some modularization mechanism). However, we need to prove the amalgamation property only for models, since the corresponding property for axiomatic presentations comes for free from the former (see [13]).

We first show that **OSign** is finitely cocomplete, so that signatures can be combined together; then, we state the amalgamation property in its most general formulation, that is, by considering any kind of pushouts. Proofs are given in the Appendix.

The proof of finite cocompleteness is based on the fact that both **Set** and **PreOrd** are (finitely) cocomplete and shows that every colimit in **OSign** can be defined by simply putting together the two colimits in **PreOrd** and **Set** obtained by forgetting function and sort symbols, respectively. More formally, let U^S: $\mathbf{OSign} \to \mathbf{PreOrd}$, U^F: $\mathbf{OSign} \to \mathbf{Set}$ be the two forgetful functors defined by $U^S(S, \leq, O, \delta 0, \delta 1) = (S, \leq)$, $U^S(\sigma^S, \sigma^F) = \sigma^S$ and $U^F(S, \leq, O, \delta 0, \delta 1) = O$, $U^F(\sigma^S, \sigma^F) = \sigma^F$. Then U^S and U^F turn out to be finitely cocontinuous. This property allows the proof for the amalgamation property to be simpler, since we are able to reason at the level of the underlying categories **Set** and **PreOrd** rather than **OSign**.

Proposition 2. *The category **OSign** is finitely cocomplete. Furthermore, the two forgetful functors U^S and U^F are finitely cocontinuous.*

Proposition 3 (Amalgamation). *For any pushout diagram in **OSign***

$$\begin{tikzcd}
\Sigma_1 = (S_1, \leq_1, O_1, \delta 0_1, \delta 1_1) \arrow[r, "\sigma_1'"] & \Sigma = (S, \leq, O, \delta 0, \delta 1) \\
\Sigma_0 = (S_0, \leq_0, O_0, \delta 0_0, \delta 1_0) \arrow[u, "\sigma_1"] \arrow[r, "\sigma_2"'] & \Sigma_2 = (S_2, \leq_2, O_2, \delta 0_2, \delta 1_2) \arrow[u, "\sigma_2'"']
\end{tikzcd}$$

and for any Σ_1-model M_1 and any Σ_2-model M_2, if $M_{1|\sigma_1} = M_{2|\sigma_2}$, there exists a unique Σ-model M s.t. $M_{|\sigma_i'} = M_i$, $i = 1, 2$.

2.4 Language

As shown in Sect. 1, terms are of three kinds: variables, function applications and casted terms. In function applications, we allow, but not require, arguments to be explicitly typed, to direct the choice of which variant of the multifunction has to be used (static binding). If no type is provided, then the dynamic type of the argument is used as a default (late binding).

A casted term is of the form $(s')t$, where t is a term of some supersort of s' and denotes the value of t seen as an element of type s', if possible (i.e. if the dynamic type of t is a subtype of s'); it is undefined otherwise.

Finally, note that we do not have an explicit subsumption rule. Hence any term has a unique (static) type, due to the absence of static overloading. But a term can be used as argument for a function application whenever its type is smaller than the expected type.

Definition 5. *Given a set S, an S-*indexed family of variables* X *is any family $\{X_s\}_{s \in S}$ of pairwise disjoint (sub)sets X_s (of some fixed universe). In the following we will denote by X the (necessarily disjoint) union of the X_s's, too.*

For each S-indexed family of variables X, the S-indexed family $T_\Sigma(X)$ of terms *over Σ and X is inductively defined by*

- $X_s \subseteq T_\Sigma(X)_s$
- $t \in T_\Sigma(X)_s$ *and* $s' \leq s$ *implies* $(s')t \in T_\Sigma(X)_{s'}$ *(*casting*)*
- *if* $t_i \in T_\Sigma(X)_{s''_i}$ *and* $s''_i \leq s'_i \leq s_i$ *for all* $i = 1 \ldots n$*, then* $op(\overline{\tau}) \in T_\Sigma(X)_s$ *for each function symbol* $op\colon \overline{s} \to s$ *and each* $\overline{\tau}$ *of length* n *s.t.* τ_i *is* t_i *or* $t_i : s'_i$*.*

The free variables *of a term are inductively defined by*

- $FV(x) = \{x\}$ *for all* $x \in X_s$*;*
- $FV((s')t) = FV(t)$*;*
- $FV(op(\overline{\tau})) = \cup_{i=1\ldots n} FV(t_i)$ *if* τ_i *is either* t_i *or* $t_i : s'_i$*.*

For each $t \in T_\Sigma(X)_s$ we say that s is the static type *of t.*

Notice that we are able to give semantics to the casting construct only because we have adopted *partial* models. Indeed, the evaluation of such construct cannot yield any value if applied to a value outside the carrier of the sort on which we are casting.

Definition 6. *Given an S-indexed family of variables $X = \{X_s\}_{s\in S}$, a* valuation *of X into M, denoted by $V: X \to M$, is a function from X into the disjoint union of all the proper carriers of M s.t. if $x \in X_s$, then $V(x) \in \iota(s)^M_\leqslant$. Moreover, we will denote $[V: X \to M]$ the set of all valuations of X in M.*

For each valuation $V: X \to M$, the evaluation *$t^{M,V}$ of a term t in M w.r.t. V is inductively defined by:*

- $x^{M,V} = V(x)$ *for all* $x \in X$
- *if* $t^{M,V} \in \iota(s')^M_\leqslant$, *then* $(s')t^{M,V} = t^{M,V}$, *else* $(s')t^{M,V}$ *is undefined*
- *if* $t_i^{M,V} = a_i : s''_i$ *and* $u_i = \begin{cases} \iota(s'_i) & \text{if } \tau_i = t_i : s'_i \\ s''_i & \text{if } \tau_i = t_i \end{cases}$ *for all* $i = 1 \dots n$, *then* $op(\overline{\tau})^{M,V} = op^M_{\overline{u}}(t_1^{M,V}, \dots, t_n^{M,V})$.

If $t^{M,V} = a : s$, then s is the dynamic type *of t.*

Note that the evaluation of a term may be undefined not only because the term contains a casting, but also because the interpretation of function symbols are *partial* functions. However, if the evaluation of a term $t \in T_\Sigma(X)_s$ yields a value, then that value belongs to $\iota(s)^M_\leqslant$.

Expressions of an object-oriented language where dynamic binding is applied only to the receiver have to be transformed in our formalism in terms where each function call has all the arguments, but the first, explicitly typed by the types expected by the function. So, for instance, if we have `R.f(A1...An)` for some method `f` declared with argument types $s_1 \dots s_n$, then we translate this call into $f(r, a : s_1, \dots, a : s_n)$ (where the translation is recursively applied to the subterms as well). In this way we choose the variant to be used in term evaluation independently from the dynamic type of the arguments.

An alternative approach is to use simplified models where the variants of multifunction are indexed only by the subtypes of the receiver type, as sketched at the end of Sect. 2.2. In this case explicitly typing arguments but the first has no effect on the semantics and hence it is not needed.

Finally note that the `super` construct should be transformed into the explicit typing of the receiver by the parent class of its static type.

Terms are translated along signature morphisms replacing sort and function symbols by their translation, while variables are unaffected.

Definition 7. *For each S-indexed family X of variables, let $\sigma(X)$ be the S'-indexed family of variables defined by $\sigma(X)_{s'} = \cup_{\sigma(s)=s'} X_s$ for all $s' \in S'$.*

Moreover, let the translation of Σ-terms along σ be inductively defined by

- $\sigma(x) = x$ *for all variables* x;
- $\sigma((s')t) = (\sigma(s'))\sigma(t)$
- $\sigma(op(\overline{\tau})) = \sigma(op)(\overline{\tau'})$ *where* $\overline{\tau}$ *and* $\overline{\tau'}$ *are of the same length* n *and if* $\tau_i = t_i$, *then* $\tau'_i = \sigma(t_i)$ *else* $\tau_i = t_i : s'_i$ *and* $\tau'_i = \sigma(t_i) : \sigma(s'_i)$.

Lemma 2. *Using the notation of Def. 7, if $t \in T_\Sigma(X)_s$, then $\sigma(t) \in T_{\Sigma'}(\sigma(X))_{\sigma(s)}$.*

As expected, term evaluation in the reduct of a model coincides with the evaluation of translated terms in the source model.

Lemma 3. *Given an S-indexed family X of variables, let us denote by M the reduct $M'_{|\sigma}$. Then:*

- *the function χ from $[V: X \to M]$ into $[U: \sigma(X) \to M']$ associating each valuation V with the valuation $\sigma(V)$, defined by $\sigma(V)(y) = V(y)$ for all variables y, is an isomorphism.*
- *for each term $t \in T_\Sigma(X)$ and each valuation V, $t^{M,V} = \sigma(t)^{M',\sigma(V)}$.*

2.5 Logic

The main novelty of our approach is the definition of two different kinds of quantification, where variables range over elements of the proper and extended carrier, respectively, of a given sort. In the latter case we state properties holding for all possible realizations of the type, while in the former we impose conditions that can be overridden in further refinements. We illustrate the two kinds of quantification in the case of the Horn-Clauses on equality, but the approach extends naturally to existential quantification and first-order logic as well, adding predicates to signatures and their interpretation to models, following the same intuition as for function symbols and allowing variants.

Definition 8. *Given an S-indexed family of variables X, the* atoms *over Σ and X consists of (here and in the following t, possibly decorated, is a term over Σ and X):*

- definedness assertions *of the form $D(t)$;*
- (strong) equalities *of the form $t = t'$, with t and t' of static types having a common supersort;*

Then, Horn Clauses *have the form $\epsilon_1 \wedge \ldots \wedge \epsilon_n \supset \epsilon_{n+1}$, where each ϵ_i is an atom for $i = 1 \ldots n+1$. The ϵ_i for $i = 1 \ldots n$ are called* premises *and ϵ_{n+1} is called* consequence. *The set of all Horn Clauses over Σ and X will be denoted by $HC(\Sigma, X)$. We will write a Horn Clause with an empty set of premises simply as the atom that is its consequence. The free variables of a Horn Clause φ are the union of the free variables of its subterms, that is*

- $FV(D(t)) = FV(t)$;
- $FV(t = t') = FV(t) \cup FV(t')$;
- $FV(\epsilon_1 \wedge \ldots \wedge \epsilon_n \supset \epsilon_{n+1}) = \cup_{i=1 \ldots n+1} FV(\epsilon_i)$;

Finally, conditional sentences *have the form*

$$\forall x_1 : s_{1\leqslant} . \ldots \forall x_k : s_{k\leqslant} . \forall y_1 : s'_1 . \ldots \forall y_n : s'_n . \varphi$$

for each $\varphi \in HC(\Sigma, X)$ with the quantified variables pairwise distinct, where $x_i \in X_{s_i}$ for all $i = 1 \ldots k$, $y_j \in X_{s'_j}$ for all $j = 1 \ldots n$ and $FV(\varphi) \subseteq \{x_1, \ldots, x_k, y_1, \ldots, y_n\}$.

The set of all conditional sentences over a signature Σ will be denoted by $OSen(\Sigma)$.

Satisfaction is defined, as usual, on the basis of validity w.r.t. *total* variables valuations, that is each variable denotes a value, following the intuition that the variables quantified as the x_i's above can be instantiated on any value in the carrier, while the variables quantified as the y_j's above have to be instantiated on values in the *proper* carrier of their sort.

Definition 9. *Given an S-indexed family of variables X, the validity of Horn Clauses over Σ and X w.r.t. a valuation $V : X \to M$ is defined as follows.*

- $M \models_V D(t)$ *iff $t^{M,V}$ is defined;*
- $M \models_V t = t'$, *iff either $t^{M,V} = \alpha = t'^{M,V}$ for some $\alpha \in s^M_{\leqslant}$ or both $t^{M,V}$ and $t'^{M,V}$ are undefined;*
- $M \models_V \epsilon_1 \wedge \ldots \wedge \epsilon_n \supset \epsilon_{n+1}$ *iff $M \not\models_V \epsilon_i$ for some $i = 1 \ldots n$ or $M \models_V \epsilon_{n+1}$.*

Then

$$M \models \forall x_1 : s_{1\leqslant} . \ldots \forall x_k : s_{k\leqslant} . \forall y_1 : s'_1 . \ldots \forall y_n : s'_n . \varphi$$

iff $M \models_V \varphi$ for all valuations $V : \{x_1, \ldots, x_k, y_1, \ldots, y_n\} \to M$, s.t. $V(y_j) = a_j : \iota(s'_j)$ for some $a_j \in \iota(s'_j)^M$.

Sentence translation immediately follows from term translation.

Definition 10. *The translation of Σ-formulae along σ is inductively defined by*

- $\sigma(D(t)) = D(\sigma(t))$;
- $\sigma(t = t') = \sigma(t) = \sigma(t')$;
- $\sigma(\epsilon_1 \wedge \ldots \wedge \epsilon_n \supset \epsilon_{n+1}) = \sigma(\epsilon_1) \wedge \ldots \wedge \sigma(\epsilon_n) \supset \sigma(\epsilon_{n+1})$;

and the translation $OSen(\sigma)$: $OSen(\Sigma) \to OSen(\Sigma')$ of Σ-sentences along σ is defined by

$$\begin{array}{l} OSen(\sigma)(\forall x_1 : s_{1_\leqslant} \ldots \forall x_k : s_{k_\leqslant}.\forall y_1 : s'_1 \ldots \forall y_n : s'_n.\varphi) \quad = \\ \forall x_1 : \sigma(s_1)_\leqslant \ldots \forall x_k : \sigma(s_k)_\leqslant.\forall y_1 : \sigma(s'_1) \ldots \forall y_n : \sigma(s'_n).\sigma(\varphi). \end{array}$$

It is immediate to see that $OSen(\sigma)$ is a well-defined function.

Proposition 4. *Using the notation of Def. 8 and 10, $OSen$: $\boldsymbol{OSign} \to \boldsymbol{Set}$ is a functor.*

Proposition 5. *Using the notation of Lemma 3, $M \models_V \phi$ iff $M' \models_{\sigma(V)} \sigma(\phi)$ for any Horn Clause ϕ.*

Proposition 6. *For each Σ-sentence ϕ and each Σ'-model M'*

$$M' \models OSen(\sigma)(\phi) \qquad \Longleftrightarrow \qquad \boldsymbol{OMod}(\sigma)(M') \models \phi$$

The technical results presented in this section can be summarized by saying that we have defined an institution (see e.g. [6]). Besides guaranteeing some degree of internal coherence, the fact that our framework is an institution makes directly available all the *institution independent* constructions, like structured specification languages, the notion of implementation, the capability of importing entailment systems through (suitable) coding in richer framework and so on.

Theorem 3. *The tuple $(\boldsymbol{OSign}, OSen, \boldsymbol{OMod}, \models)$ is an institution.*

3 Static Overloading

In concretely used languages, it may be convenient to allow the same function symbol to be used to declare functions of different types, all visible at the same level of nesting. The difference between this static overloading and the dynamic overloading modeled by our formalism is that in the latter case, the decision of which variant of the multifunction has to be invoked can be made only at run time and, hence, depends on the particular execution. On the contrary, the former kind of overloading can be solved, once and for all, at compile time. More precisely, a distinct internal name is associated with each declaration; then each function call is coded by substituting the user-defined name by the corresponding internal one, if it is possible to select, following some language-dependent rule, one definition among all those fitting that call, otherwise it is rejected as statically incorrect.

The same strategy can be adopted within algebraic frameworks (see e.g. [3]), distinguishing the level of the language for the end-users from the actual signature of the corresponding semantics.

Let us consider for instance how the static overloading allowed in the Java language could be solved in our framework.

In Java, a method name of a parent class can be redeclared in a heir class provided that either the number or the type of at least one argument is different (i.e., double declarations cannot be distinguished by the result type). For instance, let us modify the examples in Sect. 1, adding the following methods to the classes `2DPoint` and `3DPoint`:

```
class 2DPoint {
  ...
  bool equals (2DPoint P)             { ... }
  ...
}

class 3DPoint extends 2DPoint {
  ...
  bool equals (2DPoint P)             { ... }
  bool equals (3DPoint P)             { ... }
  bool inLine(2DPoint P1, 3DPoint P2) { ... }
  bool inLine(3DPoint P1, 2DPoint P2) { ... }
  ...
}
```

We can translate this interface into a signature where static overloading has been solved by renaming function names (a canonical way is using $f^{\overline{s},s}$ for each $f: \overline{s} \to s$, but for sake of simplicity in the example below we use an ad hoc, even though equivalent, renaming). In the case of our running example (omitting all the functions but `equals` and `inLine`)

sig $\Sigma_P = \ldots$
sorts $int, \mathit{3D} \leq \mathit{2D}$
opns
$equals^2 : \mathit{2D}, \mathit{2D} \to bool$
$equals^3 : \mathit{3D}, \mathit{3D} \to bool$
$inLine^{2,3} : \mathit{3D}, \mathit{2D}, \mathit{3D} \to bool$
$inLine^{3,2} : \mathit{3D}, \mathit{3D}, \mathit{2D} \to bool$

Note that in Java the first `equals` in class `3DPoint` is interpreted as a redefinition of `equals` in class `2DPoint`, since the argument type is the same. On the contrary, the second is interpreted as a *new* method with the same name but different argument type. Correspondingly, in the signature we have two function symbols for the two methods with the same name, while there is no different function symbol corresponding to the redefinition.

Now, every method invocation in Java can be translated into a function application (a term over Σ_P) where all the actual parameters but the first (i.e. the receiver, for which the binding is dynamic) are explicitly typed; the name of the function and the types for the explicit typing of the parameters are determined by the algorithm for resolving static overloading in Java. For instance, assuming that `P2` and `P3` have (static) type `2DPoint` and `3DPoint`, respectively, the expression `P2.equals(P3)` is translated into $equals^2(p2, p3 : \mathit{2D})$, whereas the expression `P3.equals(P3)` is translated into $equals^3(p3, p3 : \mathit{3D})$. Note that there is no translation for the expression `P3.inLine(P3,P3)`, since it is rejected by the Java compiler as statically incorrect.

It is interesting to note that there exists another possible solution for dealing with static overloading in our framework, besides the canonical one sketched above. This second solution, however, can be applied only when a set of overloaded methods $M(op)$ (i.e. all the methods having the same name op) has the following property[3]:

$$\exists \overline{s} \in S^*, s \in S \ s.t. \ \forall op' : \overline{u} \to s' \in M(op) \ \ \overline{u} \leq \overline{s}, s' = s$$

where S ranges over all types defined in the Java program. If this property holds, then we can simply associate to the program a signature with a unique function symbol op with functionality $\overline{s} \to s$.

[3] We could relax the condition by requiring $s' \leq s$ instead of $s' = s$, but in this way we should use casting for having a correct translation.

For instance, in the example above, this property is satisfied both for op = `equals` and for op = `inLine`, with functionalities $2D, 2D \rightarrow bool$ and $3D, 3D, 3D \rightarrow bool$, respectively. Considering e.g. `equals`, we will have in the signature a function symbol $equals\colon 2D, 2D \rightarrow bool$ which expands in any model M_P to the variants

$$
\begin{array}{l}
equals^{M_P}_{2D\,2D}\colon \rightarrow 2D^{M_P}_{\leqslant} \times 2D^{M_P}_{\leqslant}\, bool^{M_P}_{\leqslant} \\
equals^{M_P}_{3D\,2D}\colon \rightarrow 3D^{M_P}_{\leqslant} \times 2D^{M_P}_{\leqslant}\, bool^{M_P}_{\leqslant} \\
equals^{M_P}_{2D\,3D}\colon \rightarrow 2D^{M_P}_{\leqslant} \times 3D^{M_P}_{\leqslant}\, bool^{M_P}_{\leqslant} \\
equals^{M_P}_{3D\,3D}\colon \rightarrow 3D^{M_P}_{\leqslant} \times 3D^{M_P}_{\leqslant}\, bool^{M_P}_{\leqslant}
\end{array}
$$

which can be used for defining the semantics of the several versions of `equals`, either redefined or overloaded; in our example, $equals^{M_P}_{2D\,2D}$ represents the unique definition of `equals` in the class `2DPoint`, $equals^{M_P}_{3D\,2D}$ its redefinition in the class `3DPoint` and $equals^{M_P}_{3D\,3D}$ the overloaded version added in `3DPoint`. In this case, the version $equals^{M_P}_{2D\,3D}$ is of no use.

Of course, as happens for the first approach, also here we have to explicitly type all the arguments but the receiver; now the expression `P2.equals(P3)` is translated into $equals(p2, p3 : 2D)$, whereas the expression `P3.equals(P3)` is translated into $equals(p3, p3 : 3D)$.

Conclusion

We have presented a formal framework suitable to deal with functions with late binding, a crucial feature of the object-oriented approach. We have proved this formal framework to be an institution, so that our approach provides both a clean way for modeling object-oriented languages (including languages with multimethods like CLOS) and a logic appropriate for reasoning about object-oriented programs and for dealing with the problems that code redefinition via method overriding raises at the level of modular specifications. Furthermore, this institution verifies the amalgamation property, hence it is well-suited also for handling modularization, an important issue for object-oriented systems.

Since our emphasis is on dynamic overloading, in the model we have not taken into account static overloading; however, we have shown that, as happens in many other algebraic frameworks, static overloading can be reduced to a useful syntactic abbreviation, by means of an appropriate renaming and corresponding translation of terms. On the contrary, this simple solution cannot be adopted for dealing with dynamic overloading.

The source of inspiration of this work has been with no doubts [2]. We are in debt with this book for the central idea motivating this paper, i.e. recognizing dynamic overloading resolution as, on one side, the most important distinguishing feature of the object-oriented approach, on the other an extremely powerful mechanism of programming languages which hence deserves a deep theoretical investigation. However, the work presented in this paper faces the problem in a completely different formal framework (a specification formalism rather than a calculus) and with different technical solutions.

In the field of algebraic specification, the formalisms most closely related to ours are the many variants of *order-sorted algebras* [7, 3], since they too handle subtyping and overloading. We have already pointed out in the paper the technical differences. From a more substantial point of view, order-sorted algebras only allow overloading which is *static* (there is no notion of dynamic type in terms) and *conservative* (two function symbols with the same name and type related by the subtyping relation must behave in the same way on elements of the subtype; in few words, no redefinition). These two restrictions are too strong to model "real" inheritance in object-oriented languages.

The possibility of writing axioms which are not required to hold in subtypes presents some similarity with the use of *defaults* in specifications (see e.g. [10]). However, the approach presented there is based on temporal logic and non-monotonic reasoning, whereas we use classical first-order logic.

Finally, a research direction which has some contact points with our work is that studying the use of assertions in case of inheritance (see e.g. [9,12]). Anyway, a more precise comparison with the two last mentioned approaches is matter of further analysis.

References

1. D. Ancona, M. Cerioli, and E. Zucca. A formal framework with late binding. Technical report DISI-TR-98-16, Università di Genova, 1999.
2. G. Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkhäuser, 1997.
3. M. Cerioli, A. Haxthausen, B. Krieg-Brückner, and T. Mossakowski. Permissive subsorted partial logic in CASL. In *Recent Trends in Algebraic Development Techniques (12th Intl. Workshop, WADT'97 - Selected Papers)*, LNCS 1349, pages 91–107, 1997. Springer Verlag.
4. C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP '92*, LNCS 615, 1992. Springer Verlag.
5. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics*, volume 6 of *EATCS Monograph in Computer Science*. Springer Verlag, 1985.
6. J.A. Goguen and R.M. Burstall. Institutions: Abstract model theory for computer science. *Journ. ACM*, 39:95–146, 1992.
7. J.A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial equations. *Theoretical Computer Science*, (105):217–273, 1992.
8. S.C. Keene. *Object Oriented Programming in Common Lisp: A Programming Guide in CLOS*. Addison-Wesley, 1989.
9. K. Lano and H. Haughton. Reasoning and refinement in object-oriented specification languages. In O. Lehrmann Madsen, editor, *ECOOP '92*, LNCS 615, pages 78–97, 1992. Springer Verlag.
10. U.W. Lipeck and S. Brass. Object-oriented system specification using defaults. In K. v. Luck and H. Marburger, editors, *Management and Processing Complex Data Structures, Proc. 3rd Workshop on Information Systems and Artificial Intelligence*, LNCS 777, 1994. Springer Verlag.
11. D. Sannella and A. Tarlecki. Towards formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica*, 25:233–281, 1988.
12. R. Stata and J.V. Guttag. Modular reasoning in the presence of subclassing. In *OOPSLA '95*, pages 200–214. ACM Press, October 1995. SIGPLAN Notices, volume 30, number 10.
13. A. Tarlecki. *Institutions: An Abstract Framework for Formal Specification*. Chapman and Hall, 1999. To appear.