

Logical Foundations for Declarative Object-oriented Programming*

Christoph Kreitz¹, Kung-Kiu Lau¹ and Mario Ornaghi²

¹ Fachgebiet Intellektik, Fachbereich Informatik, Technische Hochschule Darmstadt

Alexanderstr. 10, D-64283 Darmstadt, Germany

{kreitz,lau}@intellektik.informatik.th-darmstadt.de

² Dipartimento di Scienze dell'Informazione

Universita' degli studi di Milano, Via Comelico 39/41, Milano, Italy

ornaghi@hermes.mc.dsi.unimi.it

Abstract. We present a formalism for reasoning about declarative object-oriented programs. Classes are represented as first-order theories that contain logic programs as methods. Inheritance, genericity and related concepts are represented by operations on such theories which preserve the consistency of these theories as well as the correctness of the programs specified by their axioms. Our approach provides a logical basis for the construction of correct and reusable programming systems.

Issues: Our main motivation is *formal program development*, i.e. developing programs that are formally correct wrt their formal specifications. Our goal in this paper is to provide the necessary logical foundations for formal program development in an object-oriented paradigm which has a suitable (declarative) semantics for this purpose.

Object-oriented programming is widely used for software development in industry because it is seen to meet the key requirements of *modularity*, *reusability*, and *reliability*. However, conventional object-oriented programming (e.g. [12]) is based on the *imperative* programming paradigm, and does not have a declarative semantics. This means that formal reasoning about programs is not at all straightforward, since modularity and reusability are characterised at code level. Instead, invariants, pre and postconditions have to be inserted into the code and checked if some level of reliability is to be guaranteed.

Declarative object-oriented programming languages have been proposed, by both the *functional* and *logic* programming communities. Examples include TOOPL [3], L&O [11], and Prolog++ [15]. These languages, though declarative in their *methods*, usually lack a suitable semantics for reasoning about formal object-oriented program development. That is, in these languages, classes are theories (with *initial* semantics) which are assumed to be correct, i.e. they are used as executable specifications. Thus in these languages it is not meaningful to talk about correctness wrt general (non-executable) specifications.

Main Results: Our main contribution is to define classes and methods declaratively, such that classes are full first-order theories (with *isoinitial* semantics), methods are logic programs that are synthesised from their specifications (in classes), and under our chosen (isoinitial) semantics, we can reason about the *correctness*, and *correct reuse* of both classes and their programs. Both our classes and programs may be *open*, i.e. they may have parameters.

* The first author is the contact author (phone: +49 6151 16 2863 fax: +49 6151 16 5326). The second author is on leave from the University of Manchester, UK, and is supported by the European Union HCM project on Logic Program Synthesis and Transformation, contract no. 93/414.

Correctness of a class is defined as its *adequacy*, and class reuse corresponds to class composition that preserves adequacy. Correctness of a program is defined wrt a specification: a program is correct if its (initial) model coincides with the interpretation of the specification in the (isoinitial) model of the class. We show that these notions of correctness enable us to characterise *correct reuse* of both classes and programs via composition. Thus altogether our results provide a basis for developing object-oriented software that is not just reliable or reusable, but in fact *formally correct* and *correctly reusable*.

However, in our formalisation we do not yet have the notion of *state*, or a mechanism for *message passing*. Although these are not directly relevant to our main concern of formal program development, they are important from a practical point of view. We intend to consider them in future work, and hopefully also define them logically. Some relevant current work (e.g. [14, 6]) should provide a few useful pointers.

Significance and Relevance: Our results are significant because they provide the logical foundations of declarative object-oriented programming which can be used for formal program development. Our characterisation of class, and class composition, can be regarded as a logical definition of class and inheritance. Our characterisation of correctness and correct reuse of both classes and programs provides a semantic, logical definition of correct reuse (as opposed to code reuse in imperative object-oriented programming) of both classes and programs. The relevance to LICS is the use of logic as the underlying formalism for a declarative view of object-oriented programming that is not only significant to computer science in general, but crucial to software engineering (of verifiably correct modular software).

1 Introduction

According to the object-oriented paradigm (e.g. [12]) the development of software systems proceeds in two essential stages. First one has to design the general structure of the system in the form of *class* definitions, relations between classes (i.e. import/export, genericity/instantiation, inheritance), class invariants, and specifications of class features (i.e. objects, functions, and routines). Then the individual *programs* of a class have to be constructed (possibly according to their specifications). This methodology supports the development of *reliable* and *flexible* software systems which can easily be *modified* and *reused* within another application. However, for developing programs that are *formally correct* wrt their specifications, current modular and object-oriented languages lack a suitable formal semantics. This means that reasoning about classes, programs, and reuse has to be done without a notion of (formal) correctness.

In this paper, we introduce a formalisation of classes which can be used as a logical foundation for reasoning about object-oriented programs and is suitable for a formal construction of correct and reusable software systems. We define classes as first-order theories (called *frameworks*) that contain logic programs.¹ A framework represents axiomatisations of the problem domain as well as the specifications of programs contained in a class. Since we are interested

¹ We shall use the logic programming paradigm for simplicity, but our approach is very general, and should be applicable to any programming paradigm with suitable logical semantics.

in formal program development we focus on frameworks which are consistent (i.e. which correspond to specifications that can be implemented) and therefore *adequate* for constructing software systems, and on programs which are correct wrt their specification within a given framework.

Some of the components of a framework or a program may depend on *parameters* which are not specified within the class. Relations between classes such as inheritance (*extension* and *renaming*) and instantiation of generic classes and operations for constructing frameworks will be expressed in terms of these parameters.

The main operation we shall investigate in this paper is *framework composition*. It can be interpreted as generalization of multiple inheritance (without additional extensions within the new class) and of instantiation of generic classes. We shall prove that composition preserves both the adequacy of the composed frameworks (theorem 5) and the correctness and termination of (inherited) programs (theorems 8 and 11). Thus composition allows us to *reuse* already existing classes in a correct and flexible way. Our results provide a theoretical basis for formal reasoning about classes, programs, and operations on classes and are therefore a promising step towards a formal construction of correct and reusable programming systems.

Our intended method of developing software systems by composing adequate frameworks that contain correct programs is similar in spirit to existing work on object-oriented program construction, but different in substance. Basically we have raised modularity and reusability to a *semantic* level. Thus our view of object orientation is different from that used in object oriented-logic programming (see e.g. [11, 15]) where the conventional object-oriented notions are imported virtually wholesale at the syntactic level. Our characterisation of modularity and reusability also differs from modular logic programming (see e.g. [2, 4]) where composition is not performed in the context of a framework and a notion of correct reusability is missing.

The paper is organised as follows. In Section 2, we introduce the background theory together with our notation and terminology. In Section 3 we investigate classes and inheritance using frameworks, relations between frameworks, and operations on frameworks that preserve consistency. Section 4 focuses on methods, i.e. programs specified in a framework, correctness, and their behaviour under the fundamental operations on frameworks. Then in Section 5 we hint at a calculus for classes and methods, to do formal reasoning about frameworks and programs.

2 Background

We will denote a many-sorted *signature* by $\Sigma = \langle S, F, R \rangle$, where S is a set of *sort* symbols, F is a set of *function* declarations and R is the set of *relation* declarations. Each function declaration has the form $f : a \rightarrow s$, where f is the declared function symbol, $a = (s_{i_1}, \dots, s_{i_n})$ its arity, and s its sort. Each relation declaration has the form $r : a$, where r is the declared relation symbol and $a = (s_{k_1}, \dots, s_{k_m})$ is its arity. Constants will be functions with empty arity, i.e. $c : \rightarrow s$. We allow *overloaded* relation symbols; overloaded identity $= : (s, s)$ for every sort s will be understood. A signature $\Sigma = \langle S, F, R \rangle$ is a *subsignature* of $\Delta = \langle S', F', R' \rangle$, written $\Sigma \preceq \Delta$, iff $S \subseteq S'$, $F \subseteq F'$, and $R \subseteq R'$.

A Σ -*interpretation* \mathcal{I} interprets the symbols of the signature in the usual way. By $\sigma^{\mathcal{I}}$ we indicate the interpretation of a symbol σ of the signature. The first-order language L_{Σ} generated by a signature Σ (and by some set of sorted variables) is defined in the usual way, as are terms and formulas. We will write $\tau : s$ to indicate that the sort of τ is s . Formulas of L_{Σ} will be called Σ -formulas.

An *assignment* \mathbf{a} over an interpretation \mathcal{I} is a map that associates with every variable $x : s$ an element $\mathbf{a}(x)$ of $s^{\mathcal{I}}$. Let \mathcal{I} be an interpretation and \mathbf{a} be an assignment. Then every term $\tau : s$ evaluates (in \mathcal{I} , \mathbf{a}) to a value $val_{\mathcal{I}}(\tau, \mathbf{a}) \in s^{\mathcal{I}}$ and every formula F evaluates (in \mathcal{I} , \mathbf{a}) to *true* or *false*. We will write $\mathcal{I} \models_{\mathbf{a}} F$ to indicate that F evaluates to *true* (in \mathcal{I}, \mathbf{a}). If F is closed, we will simply write $\mathcal{I} \models F$, and similarly the value of a ground term τ will be indicated by $val_{\mathcal{I}}(\tau)$.

Many-sorted homomorphisms, isomorphisms and isomorphic embeddings are defined in the usual way. Let Σ be a signature and ρ be a function that maps the symbols of Σ into another set of symbols. Then ρ *generates* the triple $\rho(\Sigma) = \langle \rho(S), \rho(F), \rho(R) \rangle$, where $\rho(S)$ is the image of S , $\rho(F)$ contains the declarations $\rho(f) : \rho(a) \rightarrow \rho(s)$ such that $f : a \rightarrow s \in F$, and $\rho(R)$ the declarations $\rho(r) : \rho(a)$ such that $r : a \in R$. If ρ is a map from the symbols of Σ to those of another signature Δ , and if $\rho(\Sigma) \preceq \Delta$, then we say that ρ is a *signature morphism* from Σ to Δ . If it is an injective map, then ρ is called a *signature extension*. If bijective, it is called a *renaming*.

Let $\rho : \Sigma \rightarrow \Delta$ be a signature morphism. The interpretations $int(\Delta)$ of Δ are related to the interpretations $int(\Sigma)$ of Σ by the *reduct* operation $|\rho : int(\Delta) \rightarrow int(\Sigma)$ defined thus:

$$\begin{aligned} s^{\mathcal{I}}|\rho &= \rho(s)^{\mathcal{I}} \\ f^{\mathcal{I}}|\rho : a \rightarrow s &= \rho(f)^{\mathcal{I}} : \rho(a) \rightarrow \rho(s) \\ r^{\mathcal{I}}|\rho : a &= \rho(r)^{\mathcal{I}} : \rho(a) \end{aligned}$$

$\mathcal{I}|\rho$ will be called the ρ -*reduct* of \mathcal{I} . If $\rho(\sigma) = \sigma$ for every symbol σ , i.e. $\Sigma \preceq \Delta$, we get the usual notion of reduct to a subsignature. In this case, the reduct will also be denoted by $\mathcal{I}|\Sigma$.

It is immediate to extend a morphism $\rho : \Sigma \rightarrow \Delta$ to a map $\rho : L_{\Sigma} \rightarrow L_{\Delta}$ and the following satisfaction condition (see [7]) can be easily proved:

Theorem 1. *For every closed Σ -formula F and every Δ -interpretation \mathcal{I} :*

$$\mathcal{I} \models \rho(F) \Leftrightarrow \mathcal{I}|\rho \models F \tag{1}$$

Finally, given a signature morphism $\rho : \Sigma \rightarrow \Delta$ and a theory $\mathcal{T} \subseteq L_{\Sigma}$, $\rho(\mathcal{T})$ denotes the ρ -*image* of \mathcal{T} .

3 Classes and Inheritance

Our notion of a class is that it consists of a first-order theory \mathcal{F} together with logic programs $prog_1, prog_2, \dots$, that can be specified and derived in \mathcal{F} . The theory \mathcal{F} is called a *framework* and contains axiomatisations of abstract data types (ADTs), *any* relevant axioms for reasoning about these ADTs (e.g. induction axioms), as well as other axioms for reasoning about the problem domain.

Formally, a framework is a 4-tuple $\mathcal{F} = \langle \Sigma, \mathcal{T}, \mathcal{TH}, \Pi \rangle$, where Σ is a many-sorted *signature*, \mathcal{T} is Σ -theory, namely a set of Σ -axioms, \mathcal{TH} is a set of theorems that have been proved² and Π is a set of sort and relation symbols of Σ , that we call the *parameters* of the framework. We use the notation $\mathcal{F} : \Delta \Leftarrow \Pi$ to express the fact that the function and relation symbols Δ of \mathcal{F} depend on the parameters Π and call $\Delta \Leftarrow \Pi$ the *dependency type* of \mathcal{F} .

In this section we shall investigate the general properties of frameworks. We firstly discuss *closed* frameworks, namely frameworks that axiomatise an intended model according to the isoinitial semantics explained below. Then we study parametric (or *open*) frameworks that axiomatise a class of intended models, depending on the interpretation of the parameters.

Let $\mathcal{C} = \langle \Sigma, \mathcal{T}, \mathcal{TH}, \emptyset \rangle$ be a (closed) framework and \mathcal{I} be a model of \mathcal{T} . We say that \mathcal{I} is *reachable* iff, for every sort s and $\alpha \in s^{\mathcal{I}}$, there is a ground term τ of sort s such that $\alpha = \text{val}_{\mathcal{I}}(\tau)$. \mathcal{I} is an *isoinitial model* of \mathcal{T} iff, for every other model \mathcal{M} of \mathcal{T} , there is a unique isomorphic embedding $\mu : \mathcal{I} \rightarrow \mathcal{M}$. For theories with reachable models, the following very useful theorem holds:

Theorem 2. *Let \mathcal{T} be a theory that has a reachable model \mathcal{I} . Then \mathcal{I} is an isoinitial model of \mathcal{T} iff, for every closed atomic formula A :*

$$\mathcal{T} \vdash A \quad \text{or} \quad \mathcal{T} \vdash \neg A \quad (2)$$

For the sake of conciseness, we will say that a theory \mathcal{T} is *atomically complete* if it satisfies (2).³

As we mentioned in the introduction, correctness of a class described by a framework \mathcal{F} refers to the consistency of \mathcal{F} . Moreover, we also have to preserve the isoinitial model of \mathcal{F} in order to keep correctness meaningful for the programs. To achieve both of these targets, we introduce the notion of *adequate frameworks*.

Definition 1. *A closed framework is adequate iff it has a reachable isoinitial model.*

In the sequel, we will simply say ‘closed framework’ instead of ‘adequate closed framework’.

Example 1. We associate with a definite program P the theory $\mathcal{T}(P) = \text{free}(P) \cup \text{Ocomp}(P)$, where:

- $\text{Ocomp}(P)$ (the *open completion* of P) is the set of completed definitions of the predicates defined by P (i.e. of symbols which occur in the head of at least one clause of P).
- $\text{free}(P)$ is the set of freeness axioms for the constant and function symbols of P , or more in general for a set of constant and function symbols containing the ones of P .

The theory $\mathcal{T}(P)$ induces a framework which for simplicity will also be called $\mathcal{T}(P)$. We say that P is closed if it defines all its predicates. If P is a definite closed logic program, and it terminates for every ground atomic goal $\leftarrow A$, then the theory $\mathcal{T}(P)$ is an adequate closed framework (see [10, Theorem 14]).

An adequate *open* framework, in contrast, axiomatises a class of intended models. Roughly speaking, an open framework $\mathcal{F}(\Pi)$ is adequate if, for every interpretation \mathcal{I}_{Π} of the parameters, the set of the \mathcal{I}_{Π} -models of $\mathcal{F}(\Pi)$ contains a unique Π -isoinitial model (the intended \mathcal{I} -model), where \mathcal{I}_{Π} -models are identical to \mathcal{I}_{Π} when restricted to the subsignature containing Π , and Π -isomorphic embeddings behave in a suitable way.

² For each theorem the framework contains its proof.

³ A proof of Theorem 2 can be found in [1]. Isoinitial semantics is closely related to initial semantics used in algebraic ADTs [16]. A discussion of these semantics can also be found in [13].

In this paper we are not interested in a model-theoretic characterisation. Rather we want to devise some effective methods to deal with frameworks. For this purpose we introduce operations on frameworks, where each framework has a finite or recursive presentation. The main operation on which all other operations can be based is *framework composition* which essentially means that the components and axioms of two frameworks will be united in a correctness preserving way.

Let $\mathcal{F}(\Pi)$ be a parametric framework (recall that Π is a set of sort symbols or relation declarations). We say that a sort symbol is used in Π if it occurs in Π or in some declaration of Π , while a relation declaration is used in Π iff it occurs in it. Let $\Sigma_1 = \langle S_1, F_1, R_1 \rangle$ and $\Sigma_2 = \langle S_2, F_2, R_2 \rangle$ be two signatures and Π be a set of parameters. The Π -*amalgamation operation*

$$\Sigma_1 +_{\Pi} \Sigma_2 = \langle S_1 \cup S_2, F_1 \cup F_2, R_1 \cup R_2 \rangle$$

is defined only if Σ_1 and Σ_2 coincide for the symbols and declarations used in Π , whilst they are disjoint for the other ones.

Definition 2. Let $\mathcal{F}_1 = \langle \Sigma_1, \mathcal{T}_1, \mathcal{TH}_1, \Pi_1 \rangle$ and $\mathcal{F}_2 = \langle \Sigma_2, \mathcal{T}_2, \mathcal{TH}_2, \Pi_2 \rangle$ be two frameworks. The composition operation $\mathcal{F}_1[\mathcal{F}_2]$ is defined if $\Sigma_1 +_{\Pi_1} \Sigma_2$ is defined and the following proof-obligation is satisfied:

$$\mathcal{T}_1|_{\Pi_1} \subseteq \mathcal{T}_2 \cup \mathcal{TH}_2 \quad (3)$$

and the resulting framework is:

$$\mathcal{F}_1[\mathcal{F}_2] = \langle \Sigma_1 +_{\Pi_1} \Sigma_2, \mathcal{T}_2 \cup (\mathcal{T}_1 - \mathcal{TH}_2), \mathcal{TH}_2 \cup (\mathcal{TH}_1 - \mathcal{T}_2), \Pi_2 \rangle$$

Since a framework contains also the proofs, some of the proofs of $\mathcal{TH}_1 - \mathcal{T}_2$ are modified⁴ in order to take into account the new organisation of axioms and theorems. Notice that, even if we maintain theorems and axioms disjointly, we do not pretend to have a set of independent axioms.

A remarkable and useful property of composition is its associativity, namely:

Theorem 3. If $\mathcal{F}_0[\mathcal{F}_1][\mathcal{F}_2]$ ⁵ is defined, then $\mathcal{F}_0[\mathcal{F}_1[\mathcal{F}_2]]$ is defined and

$$\mathcal{F}_0[\mathcal{F}_1][\mathcal{F}_2] = \mathcal{F}_0[\mathcal{F}_1[\mathcal{F}_2]]$$

Composition is a way of extending frameworks, according to the following definition and theorem.

Definition 3. Let $\mathcal{F}(\Pi) = \langle \Sigma, \mathcal{T}, \mathcal{TH}, \Pi \rangle$ and $\mathcal{G}(\Pi') = \langle \Sigma', \mathcal{T}', \mathcal{TH}', \Pi' \rangle$ be two frameworks. We say that $\mathcal{G}(\Pi')$ is an extension of \mathcal{F} , written $\mathcal{F}(\Pi) \preceq \mathcal{G}(\Pi')$ iff $\Sigma \preceq \Sigma'$, $\mathcal{T} \subseteq \mathcal{T}' \cup \mathcal{TH}'$, and the symbols σ , that are not used in Π , do not occur in Π' .

Theorem 4. If $\mathcal{F}[\mathcal{F}_1]$ is defined, then $\mathcal{F}(\Pi) \preceq \mathcal{F}[\mathcal{F}_1](\Pi_1)$ and $\mathcal{F}_1(\Pi_1) \preceq \mathcal{F}[\mathcal{F}_1](\Pi_1)$.

Thus $\mathcal{F}[\mathcal{F}_1]$ is an extension of both \mathcal{F} and \mathcal{F}_1 . The relation \preceq will be the basis for introducing the notion of adequate extension (Definition 7), that guarantees inheritance.

For every \mathcal{F} , $\mathcal{F} \preceq \mathcal{F}$, whilst \preceq is not transitive. Indeed, consider $\mathcal{F}_1(\Pi_1) \preceq \mathcal{F}_2(\Pi_2)$ and $\mathcal{F}_2(\Pi_2) \preceq \mathcal{F}_3(\Pi_3)$. Some symbols of Π_3 not occurring in Σ_2 could occur in Σ_1 , but not in Π_1 . However, we can rename such symbols.

⁴ This can be done in a short and effective way.

⁵ We consider composition to be left-associative, i.e. $\mathcal{F}_0[\mathcal{F}_1][\mathcal{F}_2]$ means $(\mathcal{F}_0[\mathcal{F}_1])[\mathcal{F}_2]$.

Definition 4. Let $\mathcal{F} = \langle \Sigma, \mathcal{T}, \mathcal{TH}, \Pi \rangle$ be a framework. Let ρ be a bijective map from the symbols of Σ to another set of symbols. We define

$$\rho(\mathcal{F}) = \langle \rho(\Sigma), \rho(\mathcal{T}), \rho(\mathcal{TH}), \rho(\Pi) \rangle$$

$\rho(\mathcal{F})$ will be called a renaming of \mathcal{F} .

Proposition 1. If $\mathcal{F}_1(\Pi_1) \preceq \mathcal{F}_2(\Pi_2)$ and $\mathcal{F}_2(\Pi_2) \preceq \mathcal{F}_3(\Pi_3)$, then there is a renamings ρ such that $\rho(\mathcal{F}_1(\Pi_1)) \preceq \mathcal{F}_3(\Pi_3)$.

Now we can introduce our notion of adequate framework. This notion is the basis of our treatment, since it corresponds to inheritance.

Definition 5. \mathcal{G} is an adequate closed extension of a closed framework \mathcal{C} , written $\mathcal{C} \sqsubseteq \mathcal{G}$, iff $\mathcal{C} \preceq \mathcal{G}$, \mathcal{G} has a reachable isoinitial model \mathcal{I} ,⁶ i.e. it is closed, and the reduct $\mathcal{I}|_{\Sigma_{\mathcal{C}}}$ is an isoinitial model of \mathcal{C} . If $\rho\mathcal{F} \sqsubseteq \mathcal{G}$, we will write also $\mathcal{F} \sqsubseteq_{\rho} \mathcal{G}$.

In object oriented terminology, $\mathcal{C} \sqsubseteq \mathcal{G}$ means that \mathcal{G} is a subclass of \mathcal{C} . Inheritance means that the isoinitial model, namely the intended interpretation of the old symbols, is inherited, and the union of the axioms and theorems are inherited. As we will see, also the correct programs are inherited.

Definition 6. An open framework $\mathcal{F}(\Pi)$ is adequate iff, for every closed framework \mathcal{C} , if $\mathcal{F}[\mathcal{C}]$ is defined, then $\mathcal{C} \sqsubseteq \mathcal{F}[\mathcal{C}]$.

Inheritance for open frameworks can be defined in the following way.

Definition 7. $\mathcal{G}(\Pi')$ is an adequate extension of an adequate open framework $\mathcal{F}(\Pi)$, written $\mathcal{F}(\Pi) \sqsubseteq \mathcal{G}(\Pi')$, iff $\mathcal{F}(\Pi) \preceq \mathcal{G}(\Pi')$, $\mathcal{G}(\Pi')$ is adequate and, for every closed framework \mathcal{C} , if $\mathcal{G}[\mathcal{C}]$ is defined, then $\mathcal{F}[\mathcal{C}]$ is defined and $\mathcal{F}[\mathcal{C}] \sqsubseteq \mathcal{G}[\mathcal{C}]$.

If $\mathcal{F}(\Pi) \sqsubseteq \mathcal{G}(\Pi')$, then $\mathcal{G}(\Pi')$ is a subclass in the following sense: for every \mathcal{C} , if $\mathcal{G}[\mathcal{C}]$ is defined, then $\mathcal{F}[\mathcal{C}]$ is defined, but $\mathcal{F}(\Pi)$ may have a larger class of instances. Inheritance means that $\mathcal{F}(\Pi) \preceq \mathcal{G}(\Pi')$ (the union of theorems, axioms, and programs is inherited) and every instance $\mathcal{G}[\mathcal{C}]$ inherits from $\mathcal{F}[\mathcal{C}]$.

Composition behaves correctly with respect to framework extension, as stated by the following theorem, that is based on the associativity of framework composition (see Theorem 3).

Theorem 5. If $\mathcal{F}(\Pi)$ and $\mathcal{G}(\Pi')$ are adequate, and $\mathcal{F}[\mathcal{G}]$ is defined, then $\mathcal{F}[\mathcal{G}]$ is adequate and $\mathcal{G}(\Pi') \sqsubseteq \mathcal{F}[\rho, \mathcal{G}](\Pi')$.

The above results require that composition is well-defined. In some cases (as for the transitivity of \preceq) renaming may be necessary to achieve this. This is, however, not a problem, since one can show that renaming does not change any feature of an adequate framework.

Framework composition is the main operation. Other operations, like a more general composition (based on composition steps of the kind considered here), can be derived with minimal effort, but we shall not dwell on them here. Instead, we will now focus on the other key issue of derivating correct (and correctly reusable) programs in closed and open frameworks.

⁶ Note that if $\mathcal{C} \sqsubseteq \mathcal{G}$, then \mathcal{G} is consistent by definition, since it has a model.

4 Methods

Apart from a framework \mathcal{F} , a class also contains methods which are logic programs that correctly implement a relation specified by the axioms of \mathcal{F} . These programs, however, are not just ordinary programs. They can be *open* programs that must be correct in all possible instances of \mathcal{F} . We call this kind of correctness *steadfastness*, and a model-theoretic formalisation can be found in [9]. We use the notation $P : \delta \Leftarrow \pi$ to express that the defined relations (δ) of P depend on the relation parameters π and call $\delta \Leftarrow \pi$ the *dependency type* of P .

A specification of a relation r to be computed is an explicit definition $\forall(r(x) \leftrightarrow R(x))$.⁷ A logic program P for computing r can iteratively be derived in the framework through a *program synthesis* process. In the process new relation symbols may be introduced through explicit definitions. Let D_P be the set of the explicit definitions associated to P . By $\mathcal{C} + D_P$ we indicate the framework with the signature containing the symbols of \mathcal{C} and the new relation symbols defined by D_P , and with D_P as new axioms. The synthesis process is performed in such a way that:

- $\mathcal{C} + D_P \vdash \mathcal{T}(P)$ and
- P existentially terminates, that is for every ground atom A , the goal $\leftarrow A$ with program P has a refutation or finitely fails.

Moreover the framework \mathcal{C} contains the freeness axioms $free(K)$ of all the constant and function symbols that are allowed in the derived programs, but that may be not used in P . Under the above conditions one can prove the following

Theorem 6. *Let P be a program derived within a closed framework \mathcal{C} . Then $\mathcal{C} \sqsubseteq \mathcal{C} + D_P$ and $\mathcal{T}(P) + free(K) \sqsubseteq \mathcal{C} + D_P$.*

Proof. Since P existentially terminates, then $\mathcal{T}(P) + free(K)$ is a closed framework by example 1. Since no new function or constant symbols are added and the axioms of $\mathcal{T}(P) + free(K)$ are theorems of $\mathcal{C} + D_P$, then the latter is atomically complete. Moreover, the isoinitial model $\mathcal{I}_{\mathcal{C}}$ of \mathcal{C} can be expanded to a reachable model of $\mathcal{C} + D_P$, since D_P are definition axioms. Therefore $\mathcal{C} \sqsubseteq \mathcal{C} + D_P$. Moreover, since the sorts used in programs are reachable by the signature of $free(K)$, the reduct of $\mathcal{I}_{\mathcal{C}}$ to the signature of $\mathcal{T}(P) + free(K)$ is a reachable model of it, i.e. $\mathcal{T}(P) + free(K) \sqsubseteq \mathcal{C} + D_P$.

Theorem 7. *Let P be a program derived within a closed framework \mathcal{C} . Then for every goal G for P , if $\mathcal{C} + D_P \vdash \exists(G)$, then $\mathcal{T}(P) \vdash \exists(G)$. Moreover, if P terminates for G , then $\mathcal{C} + D_P \vdash \neg\exists(G)$ entails $\mathcal{T}(P) \vdash \neg\exists(G)$.*

Proof. Let us assume that $\mathcal{C} + D_P \vdash \exists(G)$ and (by absurdity) that $\mathcal{T}(P) \not\vdash \exists(G)$. This means that, for every instance σG , $\mathcal{T}(P) \not\vdash \sigma G$. Then, by termination, $\mathcal{T}(P) \vdash \neg\sigma G$. Therefore $\mathcal{T}(P) + free(K) \vdash \neg\sigma G$, and hence $\mathcal{C} + D_P \vdash \neg\sigma G$. But, since $\mathcal{C} + D_P$ has an isoinitial model, there is some instance σG such that $\mathcal{C} + D_P \vdash \sigma G$, a contradiction ($\mathcal{C} + D_P$ being consistent). Now, let us assume that $\mathcal{C} + D_P \vdash \neg\exists(G)$. Then, by termination and the consistency hypothesis, P must finitely fail and we get our thesis.

The second theorem says that $\mathcal{T}(P)$ is a complete subframework with respect to the set of the goals of P , that is, even if $\mathcal{T}(P)$ is notably weaker than $\mathcal{C} \cup D_P$, it is equivalent to it

⁷ As it is well known, explicit definitions give rise to conservative extensions.

when we consider only the goals G for P . Notice that, in general, $\mathcal{C} \sqsubseteq \mathcal{C} + D_P$ does not hold, because the latter may not have an isoinitial model. Thus program synthesis is also a means to guarantee that some explicit definitions can be added while preserving the existence of an intended model.

Now, let us consider open programs. The derivation of an open program $P : \delta \Leftarrow \pi$ can be seen as an intermediate step of a derivation of a complete program, and we get that $\mathcal{C} \vdash \mathcal{T}(P)$. However, our aim is to use open programs as methods that can be derived separately and then composed in a safe way. $\mathcal{T}(P)$ is compositional, i.e. for any two programs $P_1 : \delta_1 \Leftarrow \pi_1$, $P_2 : \delta_2 \Leftarrow \pi_2$, if $\delta_1 \cap \delta_2 = \emptyset$ then $\mathcal{T}(P_1 \cup P_2) = \mathcal{T}(P_1) \cup \mathcal{T}(P_2)$. Therefore, to guarantee composability we need methods to deal with termination.

Let $P : \delta \Leftarrow \pi$ be an open program and \mathcal{SS} a set of π -ground goals and \mathcal{FF} a set of formulas of the form $\neg\exists G$.

- $\Leftarrow G_0$ is successful wrt \mathcal{SS} iff there is a P -derivation $G_0 \dots G_n$ such that some instance σG_n belongs to \mathcal{SS} .
- A derivation $G_0 \dots G_n$ fails wrt \mathcal{FF} iff the selected atom A in G_n has relation symbol from δ and A does not unify with the head of any clause of P , or A has relation symbol from Π and $\neg\exists(A) \in \mathcal{FF}$. G_0 finitely fails wrt \mathcal{FF} iff, for some constant k , all fair derivations fail wrt \mathcal{FF} with a length not greater than k .
- The success and failure sets of P wrt $\mathcal{SS} \cup \mathcal{FF}$, indicated by $\mathbf{SS}(P \cup \mathcal{SS} \cup \mathcal{FF})$ and $\mathbf{FF}(P \cup \mathcal{SS} \cup \mathcal{FF})$, are defined as follows: $\mathbf{SS}(P \cup \mathcal{SS} \cup \mathcal{FF})$ is the set of the ground G such that $\Leftarrow G$ is succesful wrt \mathcal{SS} , and $\mathbf{FF}(P \cup \mathcal{SS} \cup \mathcal{FF})$ is the set of the formulas $\neg\exists(G)$ such that $\Leftarrow G$ finitely fails wrt \mathcal{FF} .

In the following let $P : \delta \Leftarrow \pi$ be a program, $\mathcal{SS}_{\mathcal{C}}$ be the set of ground goals with relation symbols from π such that $\mathcal{C} + D_P \vdash G$, and $\mathcal{FF}_{\mathcal{C}}$ be the set of formulas $\neg\exists(G')$ such that G' is a (possibly open) goal with relations from π and $\mathcal{C} + D_P \vdash \neg\exists G'$.

Definition 8. We say that P is correct in \mathcal{C} iff, for every closed goal G for P , $\mathcal{C} + D_P \vdash G$ iff $G \in \mathbf{SS}(P \cup \mathcal{SS}_{\mathcal{C}} \cup \mathcal{FF}_{\mathcal{C}})$.

Definition 9. P terminates within \mathcal{C} iff, for every ground $\Leftarrow G$, $\Leftarrow G$ is successful wrt $\mathcal{SS}_{\mathcal{C}}$ or finitely fails wrt $\mathcal{FF}_{\mathcal{C}}$.

Theorem 8. If P terminates within \mathcal{C} and $\mathcal{C} \vdash \mathcal{T}(P)$, then P is correct in \mathcal{C} .

Proof. Let G be a ground goal such that $\mathcal{C} + D_P \vdash G$, and assume that $G \notin \mathbf{SS}(P \cup \mathcal{SS}_{\mathcal{C}} \cup \mathcal{FF}_{\mathcal{C}})$. Since P terminates within \mathcal{C} , then $G \in \mathbf{FF}(P \cup \mathcal{SS}_{\mathcal{C}} \cup \mathcal{FF}_{\mathcal{C}})$. One can show that $\mathcal{T}(P) + \mathcal{SS}_{\mathcal{C}} + \mathcal{FF}_{\mathcal{C}} \vdash \neg G$. Since $\mathcal{T}(P) + \mathcal{SS}_{\mathcal{C}} + \mathcal{FF}_{\mathcal{C}}$ is contained in $\mathcal{C} + D_P$, we get $\mathcal{C} + D_P \vdash \neg G$, a contradiction ($\mathcal{C} + D_P$ being consistent).

To obtain program composability, we introduce the following definition of strong termination.

Definition 10. Let $P : \delta \Leftarrow \pi$ and $\mathcal{C} + D_P$ as before, and let $\mathcal{FF}_{\mathcal{C}}^{\text{ground}}$ be the restriction of $\mathcal{FF}_{\mathcal{C}}$ to the ground goals. We say that P strongly terminates within $\mathcal{C} + D_P$ iff for every ground $\Leftarrow G$, $\Leftarrow G$ is successful wrt $\mathcal{SS}_{\mathcal{C}}$ or finitely fails wrt $\mathcal{FF}_{\mathcal{C}}^{\text{ground}}$.

Theorem 9. If $P_1 : \delta_1 \Leftarrow \pi_1$ and $P_2 : \delta_2 \Leftarrow \pi_2$ strongly terminate within \mathcal{C} , and no relation symbol of δ_1 occurs in P_2 , then $P_1 \cup P_2 : \delta_1 \cup \delta_2 \Leftarrow \pi_2 \cup (\pi_1 - \delta_2)$ strongly terminates within \mathcal{C} .

Proof. Let $\leftarrow G_0$ be a ground goal. If G_0 does not contain symbols of δ_1 , then termination is the one of P_2 . If some symbol of δ_1 occurs in $\leftarrow G_0$, then we first select only atoms with relation symbol from δ_1 , by a fair computation rule for P_1 . By the termination of P_1 we get success wrt $\mathcal{SS}_C|\pi_1$ (namely \mathcal{SS}_C restricted to the formulas with symbols from π_1) or finite failure wrt $\mathcal{FF}_C^{ground}|\pi_1$.

In the case of success, there is a derivation G_0, \dots, G_n with program P_1 such that $G_n \in \mathcal{SS}_C|\pi_1$. The only (possible) clauses that can be applied to G_n are those of P_2 . Since P_2 terminates, it is correct and there is a derivation G_n, \dots, G_k such that $G_k \in \mathcal{SS}_C|(\pi_2 \cup (\pi_1 - \delta_2))$. Therefore G_0 has a derivation $G_0, \dots, G_n, \dots, G_k$, that is successful wrt $\mathcal{SS}_C|(\pi_2 \cup (\pi_1 - \delta_2))$. In the case of finite failure, every fair derivation with program P_1 fails, ending with a goal G_h . If G_h fails with selected atom A with relation symbol $r \notin \delta_2$, then the derivation is failed wrt $\mathcal{FF}_C^{ground}|(\pi_2 \cup (\pi_1 - \delta_2))$. Otherwise, since P_2 terminates, it is correct and $\leftarrow A$ finitely fails wrt $\mathcal{FF}_C^{ground}|\pi_2$. Therefore, G_0 finitely fails with program $P_1 \cup P_2$ wrt $\mathcal{FF}_C^{ground}|(\pi_2 \cup (\pi_1 - \delta_2))$.

The above theorem shows the role of a framework in modular synthesis. It allows us to state termination properties without having to know the programs that will be used in composition. One can see (in the proof) that composability follows from the fact that the success and finite failure sets of P_2 wrt \mathcal{FF}_C and \mathcal{FF}_C^{ground} coincide with the ground goals for P_2 that are provable in \mathcal{C} . This is because P_2 terminates for ground goals. If one has stronger termination properties of P_2 , then one can substitute \mathcal{FF}_C^{ground} by a larger finite failure set. This corresponds to weaker termination requirements for P_1 .

Now we consider programs in adequate open frameworks. We extend the definition of termination in the following way.

Definition 11. $P : \delta \Leftarrow \pi$ strongly terminates in $\mathcal{F}(II)$ iff, for every closed framework \mathcal{C} , if $\mathcal{F}[\mathcal{C}]$ is defined, then $P : \delta \Leftarrow \pi$ strongly terminates in $\mathcal{F}[\mathcal{C}]$.

We get the following theorems.

Theorem 10. If $P_1 : \delta_1 \Leftarrow \pi_1$ and $P_2 : \delta_2 \Leftarrow \pi_2$, if $\delta_1 \cap \delta_2 = \emptyset$ strongly terminate within $\mathcal{F}(II)$, then $P_1 \cup P_2$ strongly terminates within $\mathcal{F}(II)$.

Theorem 11. Let $\mathcal{F}[\mathcal{G}]$ be the composition of two adequate frameworks. Let P be any open program of \mathcal{F} .

If P strongly terminates in \mathcal{F} , then it strongly terminates in $\mathcal{F}[\mathcal{G}]$.

If Q strongly terminates in \mathcal{G} , then it strongly terminates in $\mathcal{F}[\mathcal{G}]$.

That is, programs (i.e. methods) are inherited by framework composition. Moreover, by Theorem 11, after framework composition the methods coming from the two frameworks can be composed. In particular, let $P : \delta \Leftarrow \pi$ be a method of $\mathcal{F}(II)$ with parameters $\pi \subseteq II$; it can be composed with a method from \mathcal{G} , that computes (some of) the parameters π .

Theorem 12. If P strongly terminates in an adequate framework $\mathcal{F}(II)$ and $\mathcal{F}(II) \sqsubseteq \mathcal{G}(Pi')$, then P strongly terminates in $\mathcal{G}(II')$.

This theorem shows that a subclass \mathcal{G} inherits methods from the parent class.

The above theorems show that we have a general theory where we can model classes and inheritance, and that we can import these features of object-oriented into logic programming. However, to do so we need method for reasoning about adequacy of open frameworks and termination of programs in a framework. We want also rules to work on more general kinds of termination properties. We shortly discuss this issue in the next section.

5 A Calculus for Classes and Methods

Roughly speaking, the adequacy property of an open framework $\mathcal{F}(II)$ can be expressed as follows: if we have a framework \mathcal{C} that decides the parameters II , then $\mathcal{F}[\mathcal{C}]$ decides the defined symbols Δ of \mathcal{F} . We can express this by the dependency type $\mathbf{dec}(\Delta) \Leftarrow \mathbf{dec}(II)$.

$\mathbf{dec}(r)$ is the formula $\forall x (r(x) \vee \neg r(x))$ and an evaluation of it by a framework \mathcal{C} is a collection of \mathcal{C} -provable instances, called an evaluation collection, such that for every τ either $r(\tau)$ or $\neg r(\tau)$ belongs to the collection. More general kinds of formulas and collections evaluating them can be given, and a more general notion of dependency type $\Delta \Leftarrow II$ can be defined.

Some rules to work on adequacy of frameworks have been introduced in [10], based on the notion of dependency type. In the full paper we will give a short account of such rules, and introduce their use for dealing with termination properties of programs that are more general than strong termination considered in the previous section.

Conclusion

We have presented a formalisation of classes, programs and operations on classes that preserve correctness of classes and programs. Our formalisation allows us to embed object oriented features into logic programming in a way that reflects the semantics of these features and provides a basis for formal reasoning about object oriented programs, correctness, and reusability. However, our work is only in its initial stage and much more (for instance a detailed investigation of a calculus based on our theoretical results) remains to be done. Nevertheless we are convinced that it is a promising step towards the formal construction of correct and reusable software.

References

1. A. Bertoni, G. Mauri and P. Miglioli. On the power of model theory in specifying abstract data types and in capturing their recursiveness. *Fundamenta Informaticae* VI(2):127–170, 1983.
2. A. Brogi, P. Mancarella, D. Pedreschi and F. Turini. Modular logic programming. *ACM TOPLAS* 16(4):1361–1398, 1994.
3. K.M. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *J. Functional Programming* 4(2):127–206, 1994.
4. M. Bugliesi, E. Lamma and P. Mello. Modularity in logic programming. *J. Logic Programming* 19,20:443–502, 1994.
5. C.C. Chang and H.J. Keisler. *Model Theory*. North-Holland, 1973.
6. G. Delzано and M. Martelli. Objects in Forum. In J. Lloyd, editor, *Proc. 1995 Int. Logic Programming Symp.*, pages 115–129, MIT Press 1995.
7. J.A. Goguen and R.M. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM* 39(1):95–146, 1992.
8. K.K. Lau and M. Ornaghi. On specification frameworks and deductive synthesis of logic programs. In L. Fribourg and F. Turini, editors, *LNCS 883*, pages 104–121, Springer-Verlag, 1994.
9. K.K. Lau, M. Ornaghi and S.-Å. Tärnlund. Steadfast logic programs. Submitted to *J. Logic Programming*.
10. C. Kreitz, K.K. Lau and M. Ornaghi. Formal reasoning about modules, reuse, and their correctness. Submitted to *FAPR'96*.
11. F.G. McCabe. *L&O: Logic and Objects*. Prentice-Hall, 1992.
12. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.
13. P. Miglioli, U. Moscato and M. Ornaghi. Abstract parametric classes and abstract data types defined by classical and constructive logical methods. *J. Symb. Comp.* 18:41–81, 1994.
14. D. Miller. A multi-conclusion meta-logic. In *Proc. LICS 1994*, pages 272–281, IEEE Computer Society Press, 1994.
15. C.D.S. Moss. *Prolog++: The Power of Object-Oriented and Logic Programming*. Addison Wesley, 1994.
16. M. Wirsing. Algebraic specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. Elsevier, 1990.