Object-Oriented Computations in Logic Programming

Andrea Omicini and Antonio Natali

DEIS, Università di Bologna
Viale Risorgimento, 2
40136 – Bologna (ITALY)
{aomicini, anatali}@deis.unibo.it

Abstract. When interpreted as a model for structuring programs and organizing computations, the object-oriented paradigm can be thought as a set of abstractions independent of the host language. By generalizing the notion of object state configuration with respect to that embedded in languages based on assignment, this paper explores a relational approach to object-oriented programming. An object-oriented model based on the notion of object as structured logic theory, is introduced, allowing instance creation and configuration as well as computations with partially configured objects. The model is founded on an abductive framework rooted in the basic class/instance model of O-OP, which reconciles dynamic object creation with the declarative reading of LP. Meta-level constraints provide the computational support for the abstract model based on abduction. A simple first-order logic language implementing this model is presented, along with some examples of object-oriented logic computations dealing with intra- and inter-object constraints and with partially specified instances.

Keywords: Integration of object-oriented and logic paradigms, state configuration, partially configured instances, abduction, meta-level constraints.

1 Introduction

The usual perception of what computing with objects means, involves notions like object identity, message passing, class, inheritance, encapsulation, and so on. Many different definitions have been proposed, trying to capture the essentials of the object-oriented paradigm [1,2]. However, these proposals sometimes fail to properly distinguish between what should be considered characteristic of the object-oriented approach and what comes from the linguistic paradigm of choice. As an example, object-oriented and logic paradigms should not be considered as two opponents in the programming arena: instead, object-oriented imperative languages could actually be compared to object-oriented logic languages, so as to point out the advantages and the disadvantages of the two approaches.

Since the features of both object-oriented and logic programming have great impact from a software engineering perspective, the integration of the two paradigms is a subject of widespread interest. From the object-oriented side, logic languages offer several, well-known advantages with respect to traditional languages: the main one lies in their dual semantics (declarative and procedural), making logic languages particularly effective in bridging the gap between specification and coding. From a logic viewpoint, the object-oriented metaphors seem particularly suited to overcome the main drawback of logic languages in the design and development of large applications, namely the lack of mechanisms for organizing logic programs.

Thus, in the field of logic programming several attempts have been made to integrate the two models [3, 4, 5, 6, 7, and many others]. Many of the differences between the various proposals amount to a different perception of what “object-oriented” means.

In this paper, the object-oriented paradigm is interpreted as a model for structuring programs and organising computations. An object-oriented program structures the application domain by statically classifying its elements in a taxonomy. An object-oriented computation consists of a collection of independent elements of the application domain, each belonging to a class, which can be explicitly referred to by means of a name, and which communicate by message passing. An object may evolve in response to the messages it receives and its state encapsulates part of the overall state of the computation. Consequently, the result of an object-oriented computation can be defined as the final configuration of the state of the involved objects. Instance state configuration is then the intended result of any object-oriented computation.

Object state is also the most critical point when integrating objects in logic. While object statics (program structuring) can easily be captured in a logic framework, object dynamics seems to forfeit the declarative reading of logic languages. In fact, the proposals capturing the notion of mutable object state in a logic framework generally miss the fundamental connection between declarative and procedural semantics [5, 6, 8].

On the other hand, object-oriented languages based on assignment force a misleading interpretation of the very notion of object state. First of all, the state of an instance has to be completely specified for the instance to be used in a computation. As a result, state configuration in a traditional object-oriented language consists basically of an *initialization phase*, where an object is assigned a *null* state which has actually to precede any elaboration involving the object. During a computation, objects reach their final configurations by means of subsequent steps of *modification* by assignment, each one also introducing problems of state consistency. Then, the notion of state configuration is often confused with that of state modification, since any form of state configuration has first to rely on variable assignments, then on state modification.

Actually, the issue of state modification is strictly related to time-dependent application domains. An object state changes when it is intended to represent a mutable element of the application domain at different moments. Thus, the notion of object state should be kept distinct from that of mutable state. In particular, the concept of state configuration is essential to understand the distinction: configuring an object state leads to a given state specification. Whenever the object is intended to be a mutable object, a re-configuration of its state may be needed.

The traditional approach to the state issue is then somehow unsatisfactory. In particular, what is lacking is a notion of state consistency. State configuration by assignments is not necessarily an atomic operation, and suitable programming techniques have to be used in order to guarantee the preservation of state consistency. As a result, proposals such as that of [9, 10] were meant to introduce constraints in object-oriented programming by allowing programmers to specify different kinds of relations that hold in the application domain.

In fact, when state configuration issue is concerned, programming languages based on relations rather than functions, and on variable unification rather than variable assignment, offer a radically different perspective. Unification allows directionality in variable assignment to be dropped, and introduces the notion of write-once variable. An approach based on relations and unification permits in principle working with partially specified objects, where object properties can be naturally expressed in the form of (intra- and inter-object) constraints. Object state configuration may then take place incrementally, via the introduction of new relations (new constraints) concerning the object itself. The introduction of a new relation represents an atomic step of state configuration. State consistency can then be guaranteed by checking whether a newly introduced relation is consistent with respect to the previous ones.

The main goal of this paper is then to show how an object-oriented logic language dealing with instance state configuration could be built, starting from Prolog or, more generally, from any CLP($\mathcal{X}$) language. A multiple-theory logic framework with (static and/or dynamic) composition of theories is the most natural, as well as the most satisfactory approach when dealing with object representation in a logic language [11, 7, 12]. In particular, Contextual Logic Programming (CtxLP in the following) is our paradigm of choice for structuring logic programs. First of all, CtxLP has a clear model-theoretic semantics, which can be used to clarify the notion of object as logic theory [11]. Moreover, as shown in [13], multiple interpretations available for contexts make CtxLP suitable to express a wide spectrum of different concepts, ranging from the notion of binding environment to that of current line of reasoning. In addition, our CtxLP model is well settled in a stable, efficient implementation [14], which represents an ideal basis for further developments.

Needless to say, only a rather limited notion of object can be reproduced in the CtxLP model, since it cannot deal with instance creation and configuration. In this paper, then, we aim to show how a multiple-theory logic language can be extended so as to allow fully object-oriented computations by means of two key ideas: *abduction* and *meta-level constraints*. Abduction is exploited so as to perform program extension in a declarative way. Meta-level constraints allow objects with partial configured state to be dealt with. As a result, the proposed model provides instance creation and state configuration, and deals with computations involving partially configured objects. At the same time, the full declarative reading of logic programming is maintained.

This paper is organized as follows. Section 2 discusses the notion of object and object-oriented computation in a logic programming framework. The abstract model based on abduction (Section 3) and the computational model based on meta-level constraints (Section 4) are then discussed with reference to a simplified framework (multiple labelled theories with no theory composition) in order to avoid complicating the exposition with unneeded technical details. Section 5 presents a simple language based on the extended CtxLP model, along with some examples of computations dealing with (partially configured) logic objects. Finally, Section 6 is devoted to conclusions and comparisons with related works.

2 Objects in logic programming

2.1 Objects as finite trees

Objects of the application domain are usually represented in a logic program by means of terms, that is, by mapping them into finite trees. Roughly speaking, logic programs are designed to be used more as “term constructors” than as theorem provers, since they are built around the notion of a constructive proof rather than on the success/failure dynamics of a logic computation.

A logic computation can be seen as a sequence of progressive refinements of the structures representing semantic objects. A top goal can be read as a set of relations among the objects of the domain of discourse. The corresponding proof procedure is then a process which exploits a logic program to transform such relations into a consistent collection of constraints over the variables occurring in the structures representing objects. Finally, each involved structure is constrained to a particular configuration representing the state of the corresponding object.

The concept of object configuration is then quite relevant in logic programming. On the one hand, the configuration of terms is a typical intended result of a logic computation. On the other hand, by keeping track of the evolution of a computation, the structures representing objects somehow encapsulate the notion of state of an object.

A fundamental feature of the logic programming approach lies in the possibility to deal with partially configured objects. At any step of a logic computation, each object structure may either be completely specified or contain some free variable. In particular, the result of a logic computation may be read as a collection of objects whose states may be either partially or completely configured. In addition, a global notion of consistency is provided: each object state is consistent both in itself and with respect to the other objects.

Example 1. Consider for instance the following CLP($\mathcal{R}$) program describing simple electric circuits made of resistors.

```
ohm(resistor(V,R,I)) :- V = R*I.

voltage(resistor(V,_,_),V).

value(resistor(_,R,_),R).

current(resistor(_,_,I),I).

equivalent(R,series(R1,R2))  :-
    ohm(R), ohm(R1), ohm(R2),
    current(R,I), current(R1,I), current(R2,I),
    voltage(R,V), voltage(R1,V1), voltage(R2,V2),
    V = V1 + V2.
```

Goal **(g1)** below collects a set of relations over object **Res**, resulting in the complete configuration of **Res** like a resistor object with a fully specified state.

```
?-  current(Res,2), value(Res,5), ohm(Res).        (g1)
Res = resistor(10,5,2)  ?
```

The computation induced by the proof of **(g1)** can be seen as an example of a logic computation configuring an object by progressive refinement steps. The first subgoal builds **Res** as a resistor object, and constrains its current value to be equal to **2**.

```
?-  current(Res,2).                                (g1')
Res = resistor(_,_,2)  ?
```

Note that goal **(g1')** also shows how partially specified objects can be the intended result of a logic computation. The second subgoal further specifies the **Res** state by constraining its resistance value to **5**.

```
?-  current(Res,2), value(Res,5).                  (g1")
Res = resistor(_,5,2)  ?
```

Finally, the last subgoal set the final value for **Res** voltage by constraining the **Res** state to following Ohm's Law.

Goal **(g2)** shows how intra-object and inter-object relations can be exploited so as to compute over collections of objects, and have their states configured consistently.

```
?-  current(Res2,2), voltage(Res1,10),              (g2)
    equivalent(Res2,series(Res1,Res1)).
Res1 = resistor(10,5,2),
Res2 = resistor(20,10,2)  ?
```

■

The main concern in representing semantic objects by means of terms is that from a programming language perspective they are not "objects" at all. This approach, in fact, provides neither encapsulation nor data abstraction, and no way to express the notions of class, hierarchy and inheritance.[1] In particular, the object identity issue is not addressed, since there is no distinction between the object and its identifying term. In addition, as a final, more practical consideration, it is worth pointing out that mapping complex objects into terms is likely to be a really painful task for a programmer.

2.2 Objects as logic theories

The difficulty of representing and dealing with complex objects by means of terms suggests shifting from term representation to clause representation. Then, clauses of a program must be somehow "labelled" so that they can be mapped into objects of the domain of discourse. Each label, then, denotes a set of clauses, that is, a logic theory, describing a semantic object. As a result, programs consist of a collection of labelled logic theories. Labelled theories may be represented through collections of meta-facts of the form $O :: H{:-}B$, where O is a theory label and $H{:-}B$ is a first-order clause. Then, programs in a logic framework with multiple labelled theories can actually be seen as meta-programs.

The fundamental idea is that an object in a logic framework is what we know to be true about it [7]. This approach calls for a different notion of truth of a formula with respect to traditional logic languages. In the same way that axioms no longer refer to

[1] LOGIN [118] is a proposal aimed to incorporate inheritance directly in the term representation of objects. Term arguments represent attributes of the objects, which are organised into hierarchies by means of labels. Unification has to be extended so as to take object taxonomies into account.

universal truths, logic formulae can be stated to be true only with respect to a particular abstraction of the domain of discourse, that is, a given object's set of clauses. Thus, since each theory is denoted through a label, only *labelled formulae* of the form $O:G$ can be given a truth value. In particular, $O:G$ is true if G logically follows from the logic theory denoted by O.

We do not to introduce here any special set of symbols to label logic theories, by allowing any symbol of the Herbrand Universe[2] of a program (henceforth denoted by $\mathcal{H}$) to work as a theory identifier. The declarative semantics of a logic program with multiple labelled theories might be expressed as a slight variation on the scheme presented in [11].

Example 2. Assume extending a CLP($\mathcal{R}$) framework with multiple labelled theories. There are many different ways to rewrite Example 1 in this framework. Consider for instance the following program:

```
resistor :: isa(Res,V,R,I) :- Res:state(V,I),
                               Res:resistance(R),
                               V = R*I.
resistor :: equivalent(Res,series(Res1,Res2)) :-
                      isa(Res,V,_,I),
                      isa(Res1,V1,_,I),
                      isa(Res2,V2,_,I),
                      V = V1 + V2.
resistor :: voltage(Res,V) :- Res:state(V,_).
resistor :: current(Res,I) :- Res:state(_,I).
resistor :: value(Res,R) :- Res:resistance(R).

res1 :: state(10,2).
res1 :: resistance(5).

res2 :: state(20,2).
res2 :: resistance(10).
```

Here, methods of the theory `resistor` can be used as state selectors (such as goal **(g3)**) or consistency checkers (such as goal **(g4)**).

```
?-  resistor:(current(res1,I),value(res1,R)).      (g3)
I = 2, R = 5 ?

?-  resistor:equivalent(res2,series(res1,res1)).   (g4)
yes
```

However, they cannot be used either for instance creation or for state configuration, since theories have to be fully specified once-for-all in the program text.

■

From an object-oriented viewpoint, a labelled-theory framework offers several features. First of all, logic theories encapsulate all procedures and data of the corresponding object. Labels address the object identity issue. The notion of proof of labelled formulae seems well suited for a message passing interpretation: we may read

[2] Given a set of well-formed formulae P, its Herbrand Universe $\mathcal{H}(P)$ is defined as the collection of the ground terms (i.e., not containing variables) which can be built from the constants and the function symbols occurring in P. When no need exists to refer P, we will use the symbol $\mathcal{H}$ instead of $\mathcal{H}(P)$.

$O : G$ as the message G sent to object O for an answer. By introducing mechanisms for theory composition, notions like class, super class, and instance can be mimicked, while suitable binding mechanisms can be used to reproduce static and dynamic inheritance (see for instance [13] and [7]). However, since these mechanisms concern essentially object statics, and the present work concentrates rather on object dynamics, we will ignore them in the core of this paper. In fact, the notions introduced in Section 3 and 4 are nearly orthogonal with respect to theory composition, and working in a simplified framework makes it easier to point out the essentials of our proposal.

The main drawback of representing objects by means of logic theories is that we lack a way to create and configure new logic theories. Thus, new objects cannot be generated during a logic computation, and have rather to be statically defined in the text of the program. Moreover, no notion of object state (neither state configuration nor state modification) can be reproduced. Needless to say, approaches based on clause assertion cannot be considered satisfactory, since they forfeit the declarative model of logic computations.

3 The conceptual framework

3.1 Program extension through abduction

In the previously defined labelled-theory framework, the text of a program is assumed to include all the logic theories which can be used during a computation, henceforth called the *core theories* of the program. In other words, the program is supposed to completely describe all the semantic objects through a finite number of logic theories, in turn denoted by a corresponding number of identifiers. This amounts to putting an a-priori constraint on the set of the *core names*, that is, those ground terms used in order to label core theories. Since the range of the elements of the Herbrand Universe $\mathcal{H}$ which can be used in order to denote an object is limited, the formula $O : G$ has to be considered false whenever O does not represent a core theory.

Dropping this assumption is the first step towards the form of (implicit or explicit) theory creation which we are looking for. From a syntactic viewpoint, this amounts to abandoning the restriction on theory identifiers: thus, any formula $O : G$, with O ranging over the domain $\mathcal{H}$, should be admissible in principle. Now, the problem is how to deal with those identifiers which are not known to represent any of the core theories of the program. The basic idea is to consider them as potential identifiers for new theories to be created. As a result, the problem of program extension is then reduced to finding out the collection of clauses which has to be bound to the new identifier, that is, to configure theories which extend the program (*extension theories*).

Computations dealing with *extension* (non-core) *names* should then provide enough information to allow extension theories to be configured. The fundamental intuition is that clauses of an extension unit have to be obtained as a result of an *abduction* rather than an assertion so as to preserve a declarative model for computations. Abduction, in fact, is the only kind of computation that we can perform over logic theories which we know nothing about. For example, if the proof of a given goal G in a theory T involves the proof of a goal P in the extension theory O, there would be no way to deduce the truth value of P, if nothing were known about O. Then, G might be proved by assuming $O : P$ as a possible explanation for $T : G$, that is, by assuming by default P true with respect to theory O. Needless to say, since we want to exploit abduction for the creation and configuration of instance in an object-oriented framework, we need to put some restrictions on what is abducible and what is not, so as to make abduction fit into our object-oriented scheme.

More specifically, the Abductive Logic Programming approach [15] can be easily adapted to our case. In particular, once stated that clauses are labelled clauses, that only labelled formulae can be given a truth value, and that explanations have to be labelled formulae, too, we can directly apply to our model some of the ALP ideas. First of all, the set of hypotheses which can be adopted during a computation is restricted to the atoms built from a given set of predicates, called *abducibles*. In our labelled theory framework, hypotheses are again labelled atoms, since we can assume a formula to be true only with respect to a given theory. As a consequence, also abducible predicates have to be associated to a theory. In particular, it seems coherent with our approach to associate abducible only to extension theories, by considering core theories as complete, closed theories. On the other hand, each extension theory must be associated to a set of abducible predicates if we want to perform abductions so as to configure it. Henceforth, we will use the term *extension predicate* as a synonym for abducible. In particular, since our final goal is to represent instances through extension theories, the notion of abducible predicate for an extension theory immediately matches the notion of attribute for an instance. Thus, the most natural approach is to define a class as a set of extension predicates, so that whenever an extension theory is declared to be an instance of a given class, its set of attributes is known.[3]

Moreover, we restrict the set of atoms that can be taken as hypotheses to ground atoms. Since extension predicates actually correspond to instance attributes, this amounts to giving an extensional representation of objects state, which is quite natural. In particular, a stronger restriction on extension predicates is required in order to specify the state of an object as usual in object oriented languages, that is, as a collection of attributes which may assume only a single value at a time. Thus, some extension predicates may be constrained to being *single-extension* predicates: that is, if p is a single-extension predicate for extension theory O, then only one formula $O : p(t)$ can be adopted as a hypothesis. This amounts to saying that only one ground atom with predicate symbol p is to be contained in the full configuration of O.

3.2 Meta-predicates

In our labelled theory framework, abduction is used as a conceptual tool to extend logic programs in a declarative way. From this perspective, a program does not coincide here with its textual description. Instead, it results from the union of two sets of meta-relations, that is those expressed in the text, and those introduced by the computational process. Programs can then be seen as consisting of a fixed part (henceforth called the *core* part) expressed in the text of the program, and of an *extended* part which is built during a computation. For instance, meta-relations of the form $O :: C$ (where O denotes a theory and C a program clause) may concur in

[3] In this simplified framework, class hierarchies cannot be built by theory composition as in [5] or in CtxLP. Thus, special symbols for classes would have to be introduced into a language implementing the model with no theory composition in order to denote the set of extension attributes which have to be associated to each instance theory. Section 5 shows instead how no symbols, other than the theory names are needed to build class hierarchies when mechanisms for theory composition are available.

forming either the core or the extended part of a program, according to O (that is, whether it denotes a core theory or not). Of course, some other meta-predicates have to be introduced so as to completely define the abductive scheme. For each meta-predicate we should specify whether the information it represents concerns program extension. In fact, when extending a program with new meta-relations, a global notion of program consistency has to be given. Generally speaking, new information P' can be added to program P whenever $P \cup P'$ is consistent.

Since we have to express class specifications and class/instance relationships, we introduce three new meta-predicates: *isa*, connecting instance theories to a given class, *extension* and *single_extension*, defining the abducible predicates for each class. Following the typical approach of O-O languages, we introduce two further hypotheses. First, class-specific information, which consists here of the instance templates only (that is knowledge concerning abducibles) has to be included in the text of a program: thus, *extension* and *single_extension* meta-relations concern the core part of a program. Secondly, it must be possible for class/instance relationships to be introduced dynamically through *isa* meta-relations, which may consequently concern the extended part of a program.

In order to formally express the notion of consistent program extension, it suffices to state that:

- $P \cup \{O\ isa\ C\}$ is consistent if O is an extension name, C denotes a class, and $O\ isa\ C' \in P \Rightarrow C = C'$ holds.
- $P \cup \{O :: p(\tilde{t})\}$ is consistent when $O\ isa\ C \in P$, and it is either $extension(C, p) \in P$, or $(single\_extension(C, p) \in P) \wedge (O :: p(\tilde{t}') \in P \Rightarrow \tilde{t} = \tilde{t}')$

Since any program extension in this framework can be expressed as a sequence of the two basic extensions above, our abstract model based on abduction is now sufficiently specified. What is left is to define a computational model supporting program extension through abduction.

4 The operational framework

4.1 Meta-level constraints

Mapping elements of the domain of discourse into logic theories amounts to describing them through the Herbrand model of their corresponding theory. Each semantic object has a meta-level representation and an object-level denotation, since each theory is denoted by a ground term. From this perspective, the elements of $\mathcal{H}$ are object-level symbols which may be assigned a meta-level value (a theory). In particular, some of them (the core names) are bound statically, while others (the extension names) have to be bound dynamically during a computation.

How to build an extension theory and bind it to an extension name is actually the central point of the discussion. Since we have restricted abduction to (labelled) ground atoms, the notion of logical consequence for an extension theory comes down to set inclusion. As a result, instance configuration can be achieved by building the Herbrand model for the corresponding extension theory. If $\mathcal{B}$ is the Herbrand Base[4] of a program, extension names actually work as meta-level variables whose domain is the powerset $\mathcal{P}(\mathcal{B})$. Then, relations over the domain of discourse have to be expressed with respect to $\mathcal{P}(\mathcal{B})$, representing the space of the Herbrand intepretations. Adding new relations between objects and computing with these relations involves computing over such a meta-domain.

Thus, an object-oriented computation in the logic framework based on the abductive scheme presented in the previous section can be seen as a process which progressively narrows the domains of a set of symbols of $\mathcal{H}$ from the whole space $\mathcal{P}(\mathcal{B})$ to either a subset (partial configuration) or a single element (complete configuration) of that space, by exploiting relations over the symbols and the properties of the domain. From this perspective, meta-relations in our framework act as constraints. Since they actually concern program structure, we speak of *meta-level constraints*.

Apart from the intrinsic difficulties (symbols ranging over the meta-level domain are not variables of the language, but ground terms; the computation domain is expressed in the language itself), the formal introduction of complete CLP($\mathcal{Meta}$) scheme is out of the scope of this work. However, in order to define the operational framework supporting the abductive model sketched above, we can take as a basis the general framework presented in [16] for top-down execution operational semantics of CLP languages, and concentrate on the peculiarities of our approach.

4.2 The operational semantics

Generally speaking, any language fitting our scheme will eventually provide a syntax for expressing meta-relations corresponding to the meta-predicates seen so far: *isa*, $::$, *extension*, *single_extension*. Moreover, in a first-order logic language, labelled formulae can be read as being built from the $:$ (*demo*) meta-predicate. However, according to the scheme presented in the previous section, classes as instance templates are defined once-for-all in the program text, so that information over abducible is not subject to change during a computation. Thus, *extension* and *single_extension* meta-relations belongs to the core program. Moreover, core theory clauses are defined statically, and extension theory facts cannot be asserted. Thus, dynamic handling of $O :: C$ meta-relations is never related to explicit language constructs (e.g. assert).

As a result, defining the computational behaviour of our model simply amounts to specifying how to treat *demo* and *isa* meta-predicates. Since the top-down execution scheme adopted in [16] has been used to cover most of the major CLP systems, we will present the operational semantics of our model by reproducing and (partially) instantiating this scheme. Such a semantics is a transition systems on states, which are represented as tuples $\langle A,C,S \rangle$, where A is a collection of (labelled) atoms, and C and S are collections of constraints (respectively, active and passive constraints) called *constraint store* when taken as a whole. The transitions are $\rightarrow_r$ (for resolution), $\rightarrow_c$ (introducing constraints), $\rightarrow_s$ (for consistency), and $\rightarrow_i$ (for inference). The model is parameterised by a computation rule (which selects a type of transition, and an

[4] Given a set of well-formed formulae P, its Herbrand Base $\mathcal{B}(P)$ is defined as the collection of the ground atoms which can be built applying the predicate symbols occurring in P to the elements of the Herbrand Universe $\mathcal{H}(P)$. When no need exists to refer P, we will use the symbol $\mathcal{B}$ instead of $\mathcal{B}(P)$.

element of A if necessary), a *consistent* predicate (verifying the consistency of *C*) and a *infer* function (transforming the constraint store). In particular, the *infer* function is used in $\rightarrow_i$ transitions of the form

$$\langle A, C, S\rangle \rightarrow_i \langle A, C', S'\rangle$$

where $(C', S') = infer(C, S)$.

When an *isa* meta-predicate is encountered, what is to be ensured in order to preserve consistency is that no contrasting *isa* information already exists. Thus, we could simply write[5]

$$infer\big(C, S \cup isa(O,T)\big) = \big(C \cup isa(O,T), S\big)$$

and

$$infer\big(C \cup isa(O,T) \cup isa(O,T'), S\big) = \big(C \cup isa(O,T) \cup (T{=}T'), S\big)$$

so that meta-level consistency is simply mapped onto object-level consistency (*T*=*T'*).

As far as the *demo* predicate is concerned, the transition rule has simply to be extended so as to treat labelled atoms. The transition

$$\langle A \cup T{:}G, C, S\rangle \rightarrow_r \big\langle A \cup B, C, S \cup (G{=}H)\big\rangle$$

can be applied if *T*:*G* is the selected labelled atom, *T* is a core theory, $H{:}{-}B$ is a clause of *T*, and *G* and *H* have the same predicate symbol. In particular, $(G{=}H)$ is supposed to summarise all the equalities between the arguments of *G* and *H*. Correspondingly, we define

$$infer\big(C, S \cup (t{=}t')\big) = \big(C \cup (t{=}t'), S\big)$$

where *t* and *t'* are generic terms.

However, when the label does not denote a core theory, the labelled atom has to be treated as a constraint through a $\rightarrow_c$ transition. Thus,

$$\langle A \cup O{:}G, C, S\rangle \rightarrow_c \langle A, C, S \cup O{:}G\rangle$$

is selected when *O* denotes an extension theory, and *G* predicate symbol is an extension (or single extension) predicate for *O*. New *infer* definitions have to be introduced to treat these meta-level constraints. Apart from the obvious

$$infer\big(C, S \cup O{:}G\big) = \big(C \cup O{:}G, S\big)$$

the notion of single extension predicate is handled by an $\rightarrow_i$ transition according to the following definition:

$$infer\big(C \cup O{:}p(\tilde{t}) \cup O{:}p(\tilde{t}'), S\big) = \big(C \cup O{:}p(\tilde{t}) \cup (\tilde{t}{=}\tilde{t}'), S\big)$$

when *p* is a single extension predicate for extension theory *O*.

Since these definitions are sufficient to capture the conceptual framework defined in Section 3, there is no need to further specify the semantic model by giving a formal definition for the *consistent* predicate (which could be actually taken to be the same of Prolog, given our *infer* definitions), or for the computation rule (many different choices could be made, without affecting the substance of this approach). In fact, the

[5] Of course, this applies only to extension theories. Formally, this could be achieved for instance by supposing that a meta-relation *isa(T,core)* has been implicitly defined for each core theory *T*, where core is a special class name which no abducible are defined for. This would formalise the idea that core theories are fixed theories.

aim of this work is to present a computational model where constraints can be used in order to capture object-oriented abstractions in a logic framework, rather than to completely define a CLP system.

5 A simple object-oriented logic language

This section shows a simple Prolog-like language called *Class&Instance* (henceforth, *C*&*I*) implementing the computational model presented in the previous sections. *C*&*I* is based on a CtxLP framework built as an extension of SICStus Prolog [17], called CSM, which is described in [14]. After a summary of contextual programming concepts and mechanisms, a simplified version of the *C*&*I* syntax is presented, so that some examples of object-oriented computations in a logic language can be finally given.

5.1 Contexts

The contextual logic model (originally defined in [12], and subsequently further developed in [13] and [18]) is a multiple-theory framework with theory composition which can be fruitfully taken as a basis for object-oriented logic programming. In such a framework, labelled logic theories are called *units*, and theory labels are ground terms. Structured logic theories, obtained by unit composition, are called *contexts*. A context can be seen as an sequence of units, and list notation will be used for context representation. In particular, a context consisting of units T_n, T_{n-1}, …, T_2, T_1 is represented by the list $[T_n,T_{n-1},…,T_2,T_1]$.

Each unit is implicitly associated to a context. The *super* relation, defined by means of a static declaration, involves two units in that each unit may declare another unit as being its *super unit*. The context associated to a unit results from the transitive closure of the super relation. If the super unit of T_k is T_{k-1} for any k, $n \geq k > 1$, and T_1 is a *root unit* (that is, it has no super unit), then $[T_n,T_{n-1},…,T_2,T_1]$ is the context associated with T_n. As a consequence, the associated context of a unit can be directly referred to through the unit name: whenever a context is expected in a formula, a unit identifier denotes the context associated to the unit. As an example, consider the message-passing goal **O<-G**, which is defined to be true whenever **G** can be logically derived from the context denoted by **O**. If **O** is a unit identifier, then **G** should be proved in the context associated to **O** unit.

Binding of logic procedures is performed by applying the Closed World Assumption to each unit, except when *open goals* are concerned. That is, any predicate call performed in a given unit is solved locally by using procedures defined in the unit itself, except when it is a call of an open goal. Different kinds of open goals exist, allowing different binding policies to be performed. *Out-open* goals of the form **O<–G** explicitly delegates the proof of goal **G** to object **O**. *Up-open* and *down-open* goals, henceforth denoted with prefixes **self** and **super**, respectively, exploit an implicit form of delegation. A goal **self G** (respectively, **super G**) called in unit T_i of the context $[T_n,…,T_i,T_{i-1},…,T_1]$ is solved by using the whole context $[T_n,…,T_i,T_{i-1},…,T_1]$ (respectively, the subcontext $[T_{i-1},…,T_1]$) as the binding context.

Contextual logic programming provides some basic tools for capturing object-oriented abstractions in logic programming. Objects of the intended application domain can be mapped into contexts. An out-open goal `O<-G` may be interpreted as sending a message `G` to the object represented by context `O`. Suppose `O` is a unit, and `[O|C]` its associated context. Then, object `O` may be seen as an instance of the class `C`, where unit `O` represent the *self unit* of the object, possibly modelling object specific attributes, and context `C` is the *class context* of `O`, possibly modelling the behavioural knowledge of the object. In its turn, class context `C` may consist of a hierarchy of (*class*) units, statically defined by means of the super relation. Then, super relation may be interpreted as an *is-a* relationship between classes, so that class taxonomies can be built. Each unit may inherit from its super class(es) *methods*, and may refer to instance *attributes* stored in the self unit by means of open goals. In particular, up- and down-open goals allow self and super binding policies of traditional object-oriented programming languages to be reproduced. In the end, notions like object, class, hierarchy of classes, inheritance, message passing can be mimicked in this extended logic framework, as discussed in [13] and [18].

5.2 Extensions

In order to represent instance creation and configuration, the notion of *extension unit* (corresponding to the extending theories of previous sections) have to be introduced. *Core units* are again statically-defined labelled theories, and *core contexts* are contexts built of core units only. Since in the extension of a contextual program units are intended to be used solely for representing instances of a class, they are constrained to having a super unit actually being a core unit. By consequence, *extension contexts* representing instance objects consist of an extension unit as their self (top) unit, and of a core context as their class context. Extension contexts can be built by allowing *super* relations to be introduced dynamically. To this end, meta-predicate `isa` allows extension units to be declared as instances of a given class. If `O` is an extension unit, `T` a core unit, and `C` its associated context, `O isa T` declares `O` as an instance of class `C`, and builds `[O|C]` as the context associated to `O`.

As required by the object-oriented paradigm, classes must provide a template for their instances. A suitable declaration can be introduced so as to allow each unit to define the set of its *attribute predicates*. The union of the attribute predicates of the units of a core context taken as a class provides by definition the set of the extension predicates of the instances of that class.

As a result, once associated to a super core unit, the predicate set of an extension unit is known. By consequence, any information about the predicate set of an extension context can be derived from the program, so that contextual binding rules can be applied to extension contexts too. For instance, suppose that `O` is an instance of class `C`, that is, `O` is an extension unit and `C` is its class context, so that `[O|C]` is the context representing instance `O`. Then, a goal $\mathtt{O<-p}(\tilde{t})$, where $\tilde{t}$ is a tuple of terms and `p` a predicate symbol, can be further elaborated before to be solved. If `p` is defined in a unit `T` of the class context (that is, it is a method of class `C`), then $\mathtt{O<-p}(\tilde{t})$ can be transformed in the labelled atom $\mathtt{T:p}(\tilde{t})$ and possibly solved. Instead, in case `p` is an extension predicate of `O`, $\mathtt{O<-p}(\tilde{t})$ can be reduced to the meta-constraint $\mathtt{O:p}(\tilde{t})$ concerning the extension unit `O`.

5.3 *C&I* syntax

The skeleton syntax of the language is given with the minimum of details needed. A full Prolog-like syntax with "`!`", "`;`", precedence rules with parentheses, and usual meta-predicates may be obviously assumed by default, as well as basic constraint solving capability[6]. The implementation scheme for this language (not presented here) consists in a quite simple extension of the contextual logic programming environment CSM [14] based on the popular SICStus Prolog [17].

Unit (class) declaration. A *C&I* core unit has the following structure:

$$\langle \mathit{Unit} \rangle ::= \langle \mathit{UnitDecl} \rangle \{\langle \mathit{AttributeDecl} \rangle\} \{\langle \mathit{Clause} \rangle\}$$

A unit declaration has the following form:

$$\langle \mathit{UnitDecl} \rangle ::= \texttt{:-}\ \texttt{unit}\ \langle \mathit{UnitName} \rangle \left[\texttt{isa}\ \langle \mathit{SuperUnitName} \rangle\right]\texttt{.}$$

where $\langle \mathit{UnitName} \rangle$ and $\langle \mathit{SuperUnitName} \rangle$ are ground terms. The absence of **`isa`** in a unit declaration characterise root units, while its presence provides information on the *super unit* too, allowing static hierarchies to be built. The name of a unit can be used in order to denote its associated context.

Instance attributes. Each core units can define its set of single-extension predicate symbols by means of an attribute declaration.

$$\langle \mathit{AttributeDecl} \rangle ::= \texttt{:-}\ \texttt{attribute}\ \langle P \rangle \{\texttt{,} \langle P \rangle\}\texttt{.}$$

where $\langle P \rangle$ is a predicate symbol of the form *name/arity*. The predicate set of an extension unit is represented by the attribute predicates of its class context, built as the union of the attribute predicates of its composing units.

Clauses. A clause of *C&I* has the usual form

$$\langle \mathit{Clause} \rangle ::= \langle \mathit{Head} \rangle \left[\texttt{:-}\ \langle \mathit{Body} \rangle\right]\texttt{.}$$

$\langle \mathit{Head} \rangle$ and $\langle \mathit{Body} \rangle$ are defined as follows

$$\langle \mathit{Head} \rangle ::= \langle A \rangle$$

$$\langle \mathit{Body} \rangle ::= \langle G \rangle \{\texttt{,} \langle G \rangle\}$$

$$\langle G \rangle ::= \langle A \rangle \mid \texttt{(}\langle G \rangle\{\texttt{,}\langle G \rangle\}\texttt{)} \mid \texttt{self}\ \langle G \rangle \mid \texttt{super}\ \langle G \rangle \mid \langle T \rangle \texttt{<-} \langle G \rangle \mid$$
$$\langle T \rangle\ \texttt{isa}\ \langle T \rangle$$

where $\langle A \rangle$ is an atom, and $\langle T \rangle$ is a term.

When used in class bodies, the **`isa`** operator performs instance creation. A goal **`O isa T`** constrains unit **`O`** to having core unit **`T`** as its super unit, if this is consistent with previous information. In particular, if **`O`** is an extension unit, its

6 Our model has been thought to be built upon a constraint system. Then, `=` in the example has to be read as an equality constraint, rather than as a unification operator. For our purposes, it does not matter if this is implemented either via simple suspension mechanisms or by means of a fully-featured constraint solver.

predicate set is forced to be the collection of the attribute predicates defined by the context associated with **T**, representing **O** class context.

5.4 An example

In the following, we will show some examples of *C*&*I* programs and computations. By reproducing the behaviours described in Section 2, we aimed to point out how *C*&*I* fully preserve the declarative model of logic computations while capturing the O-O essentials. Consider for instance the following example of a *C*&*I* program:

```
:- unit resistor.

:- attribute state/2, resistance/1.

ohm(V, R, I) :-
        self state(V,I), self resistance(R),
        V = R * I.

voltage(V) :- self state(V,_).

value(R) :- self resistance(R).

current(I) :- self state(_,I).

equivalent(series(Res1,Res2)) :-
        ohm(V,_,I), V = V1 + V2,
        Res1 <- ohm(V1,_,I),  Res2 <- ohm(V2,_,I).
```

From an object-oriented point of view, unit **resistor** represents a class of objects having three attributes (representing voltage, resistance value, and current) and responding to several class methods. A fully configured instance of class **resistor** can be built through the goal **(g5)**, roughly corresponding to goal **(g1)** in Example 1:

```
?-  r1 isa resistor, r1 <- current(2),            (g5)
    r1 <- value(5), r1 <- ohm(V,_,_).
V = 10,
r1 isa resistor,
r1 :: state(10,2), r1 :: resistance(5)  ?
```

It is worth to point out that, like **(g1)** in Example 1, the computation induced by the proof of **(g5)** can be seen as an example of a logic computation configuring an object by progressive refinement steps. Even though objects are represented by means of logic theories, computations still evolve in the typical style of logic programming. The first subgoal creates object **r1** as an instance of class **resistor**. In other words, constant **r1** is taken as an identifier for an extension unit constituted by two ground facts built from the extension predicates. **[r1,resistor]** is then the context associated to unit **r1**, which represents the resistor object **r1**.

```
?-  r1 isa resistor.                             (g5')
r1 isa resistor  ?
```

The second subgoal results in applying class method **current/1**, which in turn induces a self call **state(_,2)** in the current context **[r1,resistor]**. The current value of **r1** is then constrained to being **2** by means of a meta-level constraint **r1:state(_,2)**, which is given as an output of the computation.

```
?-  r1 isa resistor, r1 <- current(2).           (g5″)
r1 isa resistor, r1 : state(_,2)  ?
```

After the third subgoal has been executed, a new meta-level constraint **`r1:resistance(5)`** is added. However, since it is completely ground, the atom **`resistance(5)`** can be adopted as an axiom of theory **`r1`**, actually exploiting abduction. This may be pointed out in the answer through the **`r1::resistance(5)`** relation.

```
?-  r1 isa resistor, r1 <- current(2),          (g5″′)
    r1 <- value(5).
r1 isa resistor,
r1 : state(_,2),
r1 :: resistance(5) ?
```

In the end, last subgoal forces **`r1`** attributes to follow Ohm's Law, so that instance **`r1`** comes to be completely configured in conclusion of the abduction process.

In order to fully accomplish the declarative model of logic computations, *C*&*I* computations adopt a data-driven approach by means of a selection rule which chooses the leftmost executable subgoal. Leftmost goals insufficiently specified are suspended, to be later resumed as soon as enough information is available. By consequence, goal **`(g5″″)`** will give basically the same results as **`(g5)`**.

```
?-  R1 <- current(2), R1 isa resistor,          (g5″″)
    r1 <- value(5),  r1 <- ohm(V,_,_),  R1  =  r1.
R1 = r1, V = 10,
r1 isa resistor,
r1 :: state(10,2), r1 :: resistance(5)  ?
```

The step-by-step evolution of the computation induced by the proof of goal **`(g5)`** shows how the model deals with partially specified objects. In particular, partially specified instances can be actually given as results of a goal evaluation.

Goal **`(g6)`** may provide one of the basic intuitions behind the notion of declarative theory creation: a completely configured extension unit behaves the same as a core unit.

```
?-  r1 isa resistor, r2  isa  resistor,              (g6)
    r2  <-  equivalent(series(r1,r1)),
    r1 <- ohm(V,5,I), r2 <- current(2).
I = 2, V = 10,
r1 isa resistor, r2 isa resistor,
r1 :: state(10,2), r1 :: resistance(5),
r2 :: state(20,2), r2 :: resistance(10),  ?
```

The proof of **`(g6)`** would lead to the same result (the creation of unit **`r2`**) even though unit **`r1`** were a core unit, statically declared in the text of the program (accordingly the above result, of course). Even more, if both **`r1`** and **`r2`** were core units, goal **`(g6)`** would succeed as well, trivially resulting in binding **`V`** to **`10`** and **`I`** to **`2`**. Thus, the same clauses which are used to configure an extension unit can be used as well to perform logic proofs in a fully configured unit, defined either statically or dynamically. As a result, programs can be written having in mind the usual model of logic programming of computations as deductions, and instance configuration is achieved automatically by the system, extracting information from logic computations via abduction.

Finally, the computation induced by the evaluation of goal **`(g6)`** seems particularly suited to be read as the evolution of a small, two-object network of independent computing elements. After defining both **`r1`** and **`r2`** as **`resistor`** instances, the third subgoal bounds each other the states of the two objects, according to the notion of resistor equivalence. From now on, any subsequent evolution of one of the two objects will affect the other. In particular, note that the fourth and the fifth subgoals would respectively configure **`r1`** and **`r2`** only partially, if taken isolated. Instead, since they come to work in a globally constrained system (**`r1`** and **`r2`** states are correlated by the third subgoal), they result in fully configuring both instance units. Furthermore, consistency is no longer a single object issue, but rather a matter of the software system as a whole. For instance, trying to setting **`r1`** current to **`3`**, even though not in contrast with the constraints directly referred to that object, would result in a failure, because of the global constraints on the whole computation.

6 Related works and conclusions

Among the many different approaches to the integration between object-oriented and logic programming which can be found in the literature, the proposals more strictly related to ours are obviously those based on the notion of object as logic theory. In particular, [11] describes the basic notions of object, message-passing and inheritance in a contextual-like logic framework, by providing both a declarative and an operational semantics. A clean discussion of the inheritance issue in a declarative framework is provided in [19]. [20] deals with logic classes and instances, and provides primitives for creating instances from sets of ground terms. By exploiting the distinction between deductive and active logic computations in a contextual framework, [18] presents an approach to state modification in logic. [7] constitutes the most extensive approach to the integration of object-oriented and logic models (as well as functional one), providing a satisfactory solution for the problem of class/instance relationship in a labelled-theory framework. However, it eventually fails in capturing some fundamental concepts of object-oriented languages, such as object identity and information hiding.

This is not the case of the approach presented in this work, which exploits abduction as as a link between the declarative computational model of logic computations and key-concepts of O-O programming such as instance creation and object identity. The declarative approach to state configuration intrinsically promoted by logic programming is achieved here through meta-level constraints. Any language based on the proposed model should then provide the benefits of both O-O and logic paradigms, and considerably extend the application area of logic programming. In particular, O-O declarative programs are intrinsically built as networks of protected and even partially configured objects whose evolution is controlled by a collection of intra- and inter-object constraints.

Further issues, such as model theoretic semantics, object persistency, information hiding, mutable instances, and implementation techniques, not discussed here, are partially covered by [21] and will be subjects for further work.

Acknowledgements

This paper has greatly benefited from fruitful discussions we had with António Porto and Luís Monteiro. This work has been partially supported by “Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo” of C.N.R. under grant n. 93.01627.PF69.

Bibliography

1. P. Wegner: Dimensions of Object-Based Language Design. Proceedings of OOPSLA ’87. ACM, 1987.
2. P. Wegner: Dimensions of Object-Oriented Modeling. IEEE Computer, October 1992, pp. 12-20.
3. C. Zaniolo: Object Oriented Programming in Prolog. In: Proceedings of the International Symposium on Logic Programming, Atlantic City, 1984.
4. H. Aït-Kaci, R. Nasr: LOGIN: A Logic Programming Language with Built-in Inheritance. Journal of Logic programming, 3(3), 1986, pp. 185-215.
5. J. Conery: Logical objects. In: Proceedings of the Fifth International Conference and Symposium on Logic Programming. Seattle, 1988.
6. A. Andreoli, R. Pareschi: Lo and Behold! Concurrent Structured Processes. In: Proceedings of OOPLSA’91, 1991.
7. F.G. McCabe: Logic and Objects. London: Prentice Hall International 1992.
8. E. Shapiro, A. Takeuchi: Object Oriented Programming in Concurrent Prolog. New Generation Computing, 1(1), 1983.
9. A. Borning, R. Duisburg, B. Freeman-Benson, A. Kramer, M. Wolf: Constraint Hierarchies. In: Proceedings of OOPLSA’87, 1987, pp. 48-60.
10. B. Freeman-Benson: Kaleidoscope: Mixing Objects, Constraints and Imperative Programming. In: Proceedings of ECOOP/OOPLSA’90, 1990, pp. 77-87.
11. A. Brogi, E. Lamma, P. Mello: Objects in a Logic Programming Framework. In: A. Voronkov (ed.): Logic Programming. Lecture Notes in Artificial Intelligence 592. Berlin: Springer-Verlag 1992, pp. 102-113.
12. L. Monteiro, A. Porto: Contextual Logic Programming. In: G. Levi, M. Martelli (eds.): Proceedings of the 6th International Conference on Logic Programming. Cambridge: The MIT Press 1989.
13. A. Brogi, E. Lamma, P. Mello: A General Framework for Structuring Logic Programs. C.N.R. Technical Report “Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo” 4/1, May 1990.
14. E. Denti, A. Natali, A. Omicini: Moving Prolog Toward Objects. In: E. Tick, G. Succi (eds.): Implementations of Logic Programming Systems. To be published by Kluwer, 1994. pp. 92-104.
15. A. Kakas, R. Kowalski, F. Toni: Abductive Logic Programming. Journal of Logic and Computation, Vol. 2, 1992. pp. 719-770.
16. J. Jaffar, M.J. Mahrer: Constraint Logic Programming: A Survey. In: Ten Years of Logic Programming. Special Issue of the Journal of Logic Programming. New York: Elsevier. To appear.
17. Swedish Institute of Computer Science: SICStus Prolog User’s Manual. Kista (Sweden) 1993.

18. A. Natali, A. Omicini: Objects with State in Contextual Logic Programming. In: M. Bruynooghe, J. Penjam (eds.): Programming Language Implementation and Logic Programming. Lecture Notes in Computer Science 714. Berlin: Springer-Verlag 1993, pp. 220-234.
19. M. Bugliesi: A declarative view of inheritance in logic programming. In: K. Apt (ed.): Proceedings of the Joint International Conference and Symposium on Logic Programming. The MIT Press 1992, pp. 113-130.
20. C. Ruggieri, M. Bugliesi. A Prolog Object-Oriented System: an Exercise in Contextual Logic Programming. Proceedings of the 6th Italian Conference on Logic Programming GULP 91, June 12-14,1991, Pisa, Italy.
21. A. Omicini: Integration of Object-Oriented and Logic Programming. Ph.D. Thesis, University of Bologna, Italy.