Constraining Objects as Logic Theories

Andrea Omicini

LIA - DEIS - Università di Bologna
Viale Risorgimento, 2 – 40136, Bologna (Italy)
mailto:aomicini@deis.unibo.it
http://www-lia.deis.unibo.it/Staff/AndreaOmicini/

Abstract. The aim of this work is to present a computational model based on logic programming where constrained computations can be performed over application domains described through an object-oriented data model. This model is essentially based on two kind of constraints: *provability constraints* and *hierarchical constraints*.

Provability constraints concern the truth value of a formula with respect to a given logic theory, which is bounded to satisfy the formula. Message passing is then re-interpreted in a declarative way as a relation constraining specific object properties.

Hierarchical constraints make it possible to define an object taxonomy in terms of a hierarchy of theories. Relationships such as class/instance and class/superclass can then be uniformly handled as constraints, which can be either statically defined in the text of a program, or dynamically inferred during a computation. This suggests a new notion of logic program, which is no longer statically defined once for all, but may instead grow dynamically in a declarative way, as a result of a constrained computation.

Examples of constrained object-oriented computations are presented by using two simple multi-theory logic languages. Their operational semantics is finally given in form of a transition system, by extending the typical CLP semantics characterisation.

Keywords: *Constraints, Object-Oriented Programming, Multi-Theory Logic Languages*

Contents

1 Introduction

Declarative programming makes it possible to express programs and computations in terms of the properties of the elements of the domain of discourse, and of the relationships between them. The Logic Programming ability to describe domains through uninterpreted structures enables inferential activities to be performed over any application domain which can be described through definite clauses. In turn, Constraint Programming makes it possible to express and impose relations as constraints globally affecting the evolution of a computation. However, typical constraint languages (such as those fitting the CLP($\mathcal{X}$) scheme, see [5]) are meant to be used to compute over pre-defined domains (such as real numbers, finite domains, etc.) where the notion of truth is given *a-priori* in the domain, and not described through appropriate program structures.

On the other hand, object-oriented programming provides an effective approach to modelling application domain abstractions both in terms of program structure, and in terms of computation organisation. It has already been shown how a multi-theory language can be exploited so as to capture object-oriented mechanisms and abstractions into a first-order logic language [7, 9, 3]. By expressing objects as composition of logic theories, logic programs can be structured and logic computations can be organised in an object-oriented fashion.

The purpose of this work is then to show how multi-theory logic programming can be exploited so as to enable *constrained computations* over user-defined domains, modelled in an object-oriented way. The idea this work is based on is that both message passing operations and *is-a* relationships can be conceived as constraints in order to provide the power needed to express intra- and inter-object relations as constraints over object properties. Correspondingly, the notions of *provability constraint* and *hierarchical constraint* are introduced.

Provability constraints formalise the intuition that *demo* operations [2], typical of multi-theory logic languages (such as module-based ones), involve a notion of directionality which clearly contrasts with declarativeness. As a result, such operations should be conceived as *meta-level constraints*, bounding a formula to logically follow from a logic theory. These constraints, then, may be used not only to properly set values in a logic formula so that it follows from a given theory, but, conversely, also to configure (in any applicable sense) a logic theory so as to make it satisfy a formula.

Hierarchical constraints, in turn, descend from the idea that there is no conceptual reasons why *is-a* relationships, such as class/superclass, should be statically set in the body of a program, and defined once for all for any computation over the program. Instead, by conceiving them as constraints between abstractions of the application domain, such relations can be dynamically inferred and eventually given as results for a computation, to be used, for instance, to deal with classification problems.

As a result, we show how object-oriented computations can be performed by constraining both object properties (through provability constraints) and object representation (through hierarchical constraints). Moreover, through hierarchical constraints, we introduce the idea of *dynamic logic program* as a set of (meta-level) relations which can be either statically defined in the program text, or inferred as a result of a constrained computation.

The work is then organised as follows. Section 2 introduces $\mathcal{L}_{ob}$, a simple object-based logic language exploiting provability constraints. Some examples of constrained computations over an object-based data model are shown and discussed.

Section 3 presents $\mathcal{L}_{oo}$, a simple multi-theory logic language extending $\mathcal{L}_{ob}$ with theory composition, where object taxonomies can be defined by means of hierarchical constraints. Some examples are presented, suggesting a new notion of logic program. In fact, when intended as the knowledge from which new information can be derived by deduction, an $\mathcal{L}_{oo}$ program is partially given in the static textual definition, and partially determined during a proof, and may then constitute part of the result of a computation.

Section 4 gives the semantics characterisation of $\mathcal{L}_{oo}$ (subsuming the $\mathcal{L}_{ob}$ one) in terms of a transition system modelled after the typical CLP one. Finally, Section 5 contains conclusions and further remarks, in particular concerning new perspectives opened by the notion of dynamic logic program.

2 An Object-Based Logic Language with Constraints

2.1 A Simple Object-Based Logic Language

Object-oriented languages organise programs (as well as computations) around the elements of the domain of discourse. Thus, the structure of object-oriented programs directly reflects the programmer's own model of the application domain.

So, the main step towards an object-oriented logic language is concerned with the *structure* of a logic program. The idea of the logic program as a monolithic entity, expressing universal truthes about the *world* (the application domain) taken as a whole, has to be abandoned. Instead, the partition of a logic program into a multiplicity of logic theories better captures the idea of representing a collection of different abstractions of the domain of discourse through *partial*, *incomplete* knowledge chunks.

The most natural choice for an object-oriented approach in logic programming is then a multi-theory logic language, where a program consists of a multiplicity of logic theories. Multi-theory logic languages extends Horn clause syntax through *extension formulae* [8, 4], enabling clause bodies to depend on different theories than the one a clause belongs to. Take for instance the case of the simple logic language

$\mathcal{L}_{ob}$, which will be used throughout this Section.[1] $\mathcal{L}_{ob}$ makes it possible to define different logic theories, called *units*, and to use them separately. Units are denoted through an identifier (the *unit name*), which is a ground term for the program, and can be referred to in extension formulae through the *unit switch operator* $\not\vdash$ (`/-` in the code). A left-to-right selection rule will be assumed for $\mathcal{L}_{ob}$, following the operational behaviour of most used logic languages.

Example 2.1 illustrates a simple case of $\mathcal{L}_{ob}$ program, and discusses a possible computation.

Example 2.1. A program $\mathcal{P} \in \mathcal{L}_{ob}$ consists of the following three units, called `dog`, `cat`, and `mouse`, respectively.

```
:- unit dog.     :- unit cat.     :- unit mouse.
eats(bone).      eats(mouse).     eats(cheese).
hates(cat).      hates(dog).      hates(X) :- X /- eats(mouse).
```

The informal interpretation of the clauses of the first two units is very intuitive, since they are facts describing properties of the domain abstraction corresponding to the unit. Then, dogs hate cats and eat bones, while cats hate dogs and eat mice. Instead, the third unit contains both a ground fact and a clause with an extension goal in its body. While the unitary clause simply states that mice eat cheese, the second clause bounds the `hates` property to the property `eats` of a different domain abstraction, unknown at the time the program is written. The following query, asking who is hated by both dogs and mice,

```
?- dog /- hates(Who), mouse /- hates(Who).
```

succeeds with computed answer

```
Who = cat ?
```

In fact, the meaning of the unit switch operator $\not\vdash$ leads to interpret the extension goal `dog /- hates(Who)` as the request to prove formula `hates(Who)` using the clauses of unit `dog`. Then, the first subgoal succeeds by constraining `Who = cat`, given that unit `dog` contains the unitary clause `hates(cat)`.
Suppose now that the second clause of `mouse` is renamed apart as `hates(X') :- X' /- eats(mouse)`, using the fresh variable `X'`. Then, the second subgoal bounds `Who` to `X'`, and is rewritten as `X' /- eats(mouse)`. Given that `X' = cat`, and that unit `cat` contains the fact `eats(mouse)`, the computation finally succeeds with computed answer (output constraint) `Who = cat`, thus ascertaining that cats are hated by both dogs and mice. □

$\mathcal{L}_{ob}$ adopts the simplest, non-trivial approach to object representation for a multi-theory language, by representing each element of the application domain as a sepa-

[1] Since $\mathcal{L}_{ob}$ is a subset of $\mathcal{L}_{oo}$, used in Section 3, the description of its semantics can be taken from Section 4, where the full operational characterisation of $\mathcal{L}_{oo}$ is given.

rate logic theory.[2] An $\mathcal{L}_{ob}$ program consists of a set of logic theories, the units, each one representing an object, whose true properties are defined by the corresponding unit clauses.[3]

Units are *open logic theories* [1], since the truth value of a formula with respect to a given unit may also depend on the clauses of other units. This makes it possible to define the properties of an object in terms of the properties of others. In order a formula to be proved with respect to a different unit, the *unit switch operator* $\not\vdash$ is provided, as a sort of *demo* meta-operator [2], having the formula to be proven and the identifier of the logic theory where the proof has to be performed as its arguments. Adopting an object-oriented perspective, this reminds to message passing. A formula $u \not\vdash a$ can then be read as a message a sent to the object u.

$\mathcal{L}_{ob}$ adopts ground terms as unit identifiers. This choice naturally follows from the notions of Herbrand Universe and pre-interpretation of a logic program. In fact, abstractions of the domain of discourse are already mapped onto a logic program by means of its pre-interpretation. In particular, a pre-interpretation of a program $\mathcal{P}$ maps the elements of the application domain onto the Herbrand Universe $\mathcal{H}_{\mathcal{P}}$ of the program, that is, the collection of all ground terms which can be built starting from the constants and function symbols of $\mathcal{P}$. Thus, ground terms in $\mathcal{L}_{ob}$ can be used as unit identifiers too, since each unit represents an element of the application domain by itself. As a result, each unit of our framework is naturally denoted by an element of $\mathcal{H}_{\mathcal{P}}$ (as in [7]), and not by some extra-logic symbol (as in [3]).

Furthermore, we may easily think of *all* the elements of $\mathcal{H}_{\mathcal{P}}$ as unit names, not only the ground terms actually used as such in the text of a program $\mathcal{P}$. Thus, an $\mathcal{L}_{ob}$ program may be thought as explicitly defining a (presumably) finite number of units (henceforth called *core* units) as finite collections of first-order clauses, while implicitly defining a (generally) non-finite number of units (henceforth *non-core* units) as empty logic theories.

Correspondingly, we define $\mathcal{H}^c_{\mathcal{P}}$ as the set of the *core names*, that is the identifiers of the core units of a program $\mathcal{P}$, and $\mathcal{H}^{nc}_{\mathcal{P}}$ as the set of the *non-core names*, that is the identifiers of the non-core units of $\mathcal{P}$. As a result, we may think of a program $\mathcal{P} \in \mathcal{L}_{ob}$ as partitioning $\mathcal{H}_{\mathcal{P}}$ into two set, $\mathcal{H}^c_{\mathcal{P}}$ and $\mathcal{H}^{nc}_{\mathcal{P}}$, such that $\mathcal{H}^c_{\mathcal{P}} \cap \mathcal{H}^{nc}_{\mathcal{P}} = \emptyset$, and $\mathcal{H}^c_{\mathcal{P}} \cup \mathcal{H}^{nc}_{\mathcal{P}} = \mathcal{H}_{\mathcal{P}}$

Example 2.2. Take again the program of Example 2.1. There,

$$\mathcal{H}_{\mathcal{P}} = \{\texttt{bone}, \texttt{cat}, \texttt{cheese}, \texttt{dog}, \texttt{mouse}\}$$

[2] For an exhaustive survey about modularity in logic languages, see [4].

[3] This approach is roughly the same adopted by module systems of most wide-spread commercial Prolog frameworks (such as SICStus [14] and Quintus [13]). However, while modules are typically read as being mere syntactic containers, what we want to stress here is the object-based interpretation of units, where any logic theory actually corresponds to a specific abstraction of the application domain, rather than to a simple collection of procedures and data.

where $\mathcal{H}^c_{\mathcal{P}} = \{\texttt{cat}, \texttt{dog}, \texttt{mouse}\}$ denote core units, and $\mathcal{H}^{nc}_{\mathcal{P}} = \{\texttt{bone}, \texttt{cheese}\}$ denote non-core units. Then, units `bone` and `cheese` are implicitly defined as empty, in that they have no relevant properties as far as this program is concerned. We might "send" any formula to them for a proof, but such a proof is doomed to failure, as in the following example:

```
?- bone /- hates(Who).
```
no

□

2.2 Constraining Object Properties

Since any term can be used in order to denote units in extension formulae, it is then possible to write unit switch formulae where the unit identifier is not ground in a clause body. As in Example 2.1, computations may be aimed at determining the ground term denoting the unit with respect to which the truth value of a formula has to be computed.

What happens, then, when the unit identifier in an extension formula is non-ground and the formula is selected for resolution?

Example 2.3. Take, for instance, the program of the Examples 2.1-2.2, and suppose the `X' /- eats(mouse)` subgoal is selected *before* `X'` has been bound to `cat`. This is the case, for instance, of the following query

```
?- mouse /- hates(Who), dog /- hates(Who).
```

In fact, the first subgoal is unified with the head of the second clause of unit `mouse`, where `X` is properly renamed apart with the fresh variable `X'`, and constrains `Who = X'`. The resulting subgoal `X' /- eats(mouse)` is then selected, but its truth value cannot be determined yet, since `X'` is not a ground term, and we cannot say at this point of the computation which clauses to use for the proof of `eats(mouse)`. □

In that case, industrial module-based systems (like SICStus [14]) typically adopt a failure semantics.[4] Thus, however, the operational behaviour departs from the declarative semantics, in that the truth value of a goal depends on the subgoals evaluation order.[5]

One solution is obviously to define a specialised, data-driven selection rule, based on the idea that only formulae whose truth value can be actually computed are to be selected.

[4] Precisely, SICStus adopts an *abort* semantics, with exception raising.

[5] Take an $\mathcal{L}_{ob}$ program $\mathcal{P}$, where $t \in \mathcal{H}^c_{\mathcal{P}}$ and $\mathcal{P} \vdash t \not\vdash a$, and assume a sequential, left-to-right selection rule. Then, the goal $X \not\vdash a, X = t$ fails, while $X = t, X \not\vdash a$ succeeds.

However, a data-driven approach is not enough from a conceptual viewpoint. In fact, it does not solve the problem of what should happen to formulae which never reach the complete configuration: should they either succeed or fail, both cases would lead to a loss of declarativeness.[6]

A more general solution is to adopt a constraint-based approach. As a first result, this would solve the problem of the selection of formulae whose truth value cannot be computed yet, since selection and evaluation no longer coincide as far as constraints are concerned (see Example 2.4 below).

Adopting a constraint-based solution, however, obviously requires the definition of the constraints which are to be handled. The main idea behind the proposed model is to re-interpret the relationship between a formula and its associated unit as a *provability constraint*, bounding a formula and a logic theory so that the former logically follows from the latter. From this point of view, a unit switch formula in the resolvent may be seen both as a formula to be proven with respect to a given unit, and as a *meta-level constraint*. How it is actually handled by the inference engine depends on whether the truth value of the formula can be computed or not.

Example 2.4. Take Example 2.3, where the query

```
?- mouse /- hates(Who), dog /- hates(Who).
```

leads to the selection of the formula `X' /- eats(mouse)` for resolution, with `Who = X'`.

Since the truth value of `X' /- eats(mouse)` cannot be computed yet, it has to be handled as a provability constraint, bounding unit `X'` (whose clauses are unknown at this point of the computation) to satisfy atomic formula `eats(mouse)`.

Then, the second subgoal is selected, and succeeds by constraining `Who = cat`, given that unit `dog` contains the unitary clause `hates(cat)`.

Since now `X' = cat`, the constraint corresponding to `X' /- eats(mouse)` can be treated in the same way as the subgoal `cat /- eats(mouse)`, thus leading computation to success with the computed answer

```
Who = cat ?
```

By the way, this is the same computed answer of the goal of Example 2.1

```
?- dog /- hates(Who), mouse /- hates(Who).
```

independently of the different subgoal ordering, as well as of the selection rule, as expected. □

[6]Take an $\mathcal{L}_{ob}$ program $\mathcal{P}$ such that $\mathcal{H}^c_{\mathcal{P}} = \{t\}$, and $\mathcal{P} \vdash t \not\vdash a$. The goal $X \not\vdash a$ should either fail or succeed, depending on X (either $X = t$ or $X \neq t$), which however is still not bound when the computation ends.

In addition, in a typical constraint-based systems, unresolved constraints which cannot be evaluated for the lack of enough information are given as results of a computation. A constraint-based approach would then make it possible to re-interpret those formulae whose truth value cannot be computed for the lack of sufficient information as constraints of some sort.

Example 2.5. Consider again the program of the Examples 2.1-2.4, and suppose that `X'` in the subgoal `X' /- eats(mouse)` is never bound to `cat`, nor to any other ground term. This is the case, for instance, of the following simple query

```
?- mouse /- hates(Who).
```

The only subgoal is then unified with the head of the second clause of unit `mouse`, properly renamed with the fresh variable `X'`, thus constraining `Who = X'`. The resulting subgoal `X' /- eats(mouse)` is then selected and treated as a provability constraint, bounding unit `X'` (yet unknown) to satisfy `eats(mouse)`. Then, since no other subgoals remain to be selected, the computation ends while the constraint is still alive. Following the typical behaviour of constraint languages, the computation succeeds with a conditional answer represented by the remaining constraint, which is added to the computed answer as its final output. The computation finally results in the following output

```
?- mouse /- hates(Who).
Who = X', X' /- eats(mouse)?
```

which may be easily read as the statement (not arguable, indeed) that mice hate mouse-eaters. □

2.3 Constrained Object-Based Computations

The resulting $\mathcal{L}_{ob}$ computational model may be given different interpretations.

First of all, it endorses an object-based model for programs and computations. Programs are structured as flat collections of software components (logic theories) each one representing an abstraction of the domain of discourse. Such components encapsulate both methods and attributes of the objects, which are denoted through a unique identifier.[7] Computations evolve through message passing between objects, where a message is actually the delegation of the proof of a logic formula from one theory to another.

On the other hand, a full declarative model is provided. Object properties and their relationships are represented relationally, and a declarative semantics can be easily defined.

[7] Information hiding is not a concern of this work. However, it would be easy to define a more complex binding strategy exploiting many levels of visibility for either predicates or clauses of a unit.

The main point, however, lays in the ability to perform some kind of constrained computations over a user-defined domain. Any domain which can be described through definite Horn clauses can be computed over with constrained computations, where the relationships between different objects, and between different properties of a given object can be expressed as meta-level constraints.

Take for instance Example 2.5. There, a simple constraint bounds object `Who`, whatever it is, to have a particular property (in particular, to be a mouse-eater). Any subsequent evolution of the state of the computation affecting `Who` (as in Example 2.4) must satisfy that very constraint, or the computation is doomed to fail.

A first, obvious limitation of $\mathcal{L}_{ob}$ from an object-oriented viewpoint is that it is not an object-oriented language at all. The flatness of the collection of the program components prevents the incremental definition of objects in an $\mathcal{L}_{ob}$ program, since $\mathcal{L}_{ob}$ does not make it possible either to build hierarchies of objects, or to exploit inheritance mechanisms. This problem will be addressed in the next Section by providing mechanisms for theory composition, and by re-interpreting inheritance relations as a new kind of constraint.

Moreover, $\mathcal{L}_{ob}$ does not exploit any meaningful mechanism of constraint propagation for provability constraints. For instance, since the clauses of all units are given in an $\mathcal{L}_{ob}$ program $\mathcal{P}$, the provability constraint `X' /- eats(mouse)` resulting from the computation of Example 2.5 may be used to propagate the equality constraint `X' = cat`, since `cat` is the only unit of the program making `eats(mouse)` true. However, we do not mean to discuss here all the desirable features of the sample languages presented in this work. It should be enough to understand that the notion of provability constraint (or that of hierarchical constraint, presented in the next Section) can be fully exploited by providing many different kind of propagation mechanisms, depending on the peculiar language characteristics, and according to the desired language features.

3 An Object-Oriented Logic Language with Constraints

3.1 A Simple Object-Oriented Logic Language

The Object-Oriented paradigm promotes the incremental definition of software components, by organising programs in terms of hierarchies of classes. Each object of the application domain can be defined in terms of a peculiar part, describing its unique features, and a common part, shared with other domain abstractions, which represents a class of elements. Thanks to inheritance mechanisms, objects can delegate computations concerning shared features to more general components (static inheritance), as well as classes can exploit more specialised knowledge belonging to more specific elements (dynamic inheritance).

The idea is to represent each abstraction of the application domain as a separate logic theory, containing only that knowledge which is peculiar to a domain abstraction, while relying over other theories for the knowledge shared with other abstractions.

A multi-theory logic language with theory composition is then an obvious choice for an object-oriented approach in logic programming. Take for instance the case of a simple logic language $\mathcal{L}_{oo}$,[8] subsuming $\mathcal{L}_{ob}$. Like in $\mathcal{L}_{ob}$, introduced in Section 2, it is possible to define different logic theories, called *units*, and denote them through ground terms. Like $\mathcal{L}_{ob}$, $\mathcal{L}_{oo}$ adopts a left-to-right selection rule. In addition, $\mathcal{L}_{oo}$ extends $\mathcal{L}_{ob}$ by enabling unit composition into *contexts*, through a sort of *is-a* relation in form of the *parent* relation between units. Henceforth, we will denote such a relation with the infix predicate symbol $\Vdash\!<$.

Example 3.1 illustrates a simple $\mathcal{L}_{oo}$ program, and shows a possible computation.

Example 3.1. Consider the following $\mathcal{L}_{oo}$ program.

```
:- unit kitty |< cat.    :- unit jojo |< cat.    :- unit odie |< dog.
colour(white).           colour(black).          colour(brown).
sex(female).             sex(male).              sex(male).

:- unit dog |< pet.      :- unit cat |< pet.
eats(bone).              eats(mouse).
hates(cat).              hates(dog).

:- unit pet.
likes(X) :- #sex(female), X /- sex(male).
likes(X) :- #sex(male), X /- sex(female).
```

By means of the `|<` operator in unit declaration, units `kitty` and `jojo` declare `cat` as their *parent unit*, which is in its turn a *child unit* of `pet`. Unit `pet` declares no parent unit, and is the parent unit for `dog`, too, which is in its turn the parent unit of `odie`.

Thus, when referring a unit like `kitty` in a *context switch formula* (like `kitty /- likes(jojo)`), the context $\langle$`kitty, cat, pet`$\rangle$, obtained as a linear composition of units `kitty`, `cat`, and `pet`, has to be taken into account for formula demonstration, since `kitty` $\Vdash\!<$ `cat` and `cat` $\Vdash\!<$ `pet`.

As a result, the following query succeeds:

```
?- kitty /- likes(jojo).
yes
```

In fact, since it does not contain any suitable clauses, unit `kitty` delegates the proof of `likes(jojo)` to its ancestor units `cat` and `pet`, so that the first clause of `pet` is chosen for the proof, and renamed apart with a fresh variable,

[8] $\mathcal{L}_{oo}$ behaviour will be formally defined in Section 4.

say `X'`. Thus, the resolvent becomes `#sex(female), X' /- sex(male)`, with `X' = jojo`.

The self operator $\sharp$ (`#` in the code) introduces a different kind of binding, since it enables `pet` to delegate the proof of `sex(female)` upwards to the self object, represented here by unit `kitty`, where the formula is trivially found to be true, given that `kitty::sex(female)` is a clause of the program.

The computation finally succeeds, since the last subgoal `X' /- sex(male)` forces the proof of `sex(male)` in the unit `jojo` of the context $\langle$`jojo`, `cat`, `pet`$\rangle$, given that `X' = jojo`, `jojo` $\Vert\!<$ `cat`, and `cat` $\Vert\!<$ `pet`. $\square$

Thus, a $\mathcal{L}_{oo}$ program contains a collection of logic theories, called units, in the same way as an $\mathcal{L}_{ob}$ program does. However, $\mathcal{L}_{oo}$ promotes the incremental definition of software components, in that each unit of an $\mathcal{L}_{oo}$ program contains only that knowledge which is peculiar to a domain abstraction, and relies over other units for the knowledge shared with other abstractions through the *parent* relation. Then, all units of an $\mathcal{L}_{oo}$ program define one (and only one) *parent unit*. Units representing abstractions which are not conceptually based on other ones do not explicitly declare any parent unit, but are by default considered as child units of the empty unit $\emptyset_{th}$, denoted by `root` in any $\mathcal{L}_{oo}$ program.[9]

Thus, an $\mathcal{L}_{oo}$ program will contain also information about *parent* relations between logic units, denoted with the infix operator $\Vert\!<$. If $t, t' \in \mathcal{H}_{\mathcal{P}}$, $t \Vert\!< t' \in \mathcal{P}$ tells that unit t is a child unit of t', or that t' is the parent unit of t.[10] Since each unit of $\mathcal{L}_{oo}$ has one and only one parent unit, which can be another unit of the program or the empty unit $\emptyset_{th}$, each unit is implicitly associated with a *context* (that is, a sequence of units), defining the corresponding abstraction as a whole. In fact, if $t_1, \ldots, t_n \in \mathcal{H}_{\mathcal{P}}$ such that $\forall i, 1 \leq i < n, t_{i+1} \Vert\!< t_i \in \mathcal{P}$, and $t_1 \Vert\!< \emptyset_{th} \in \mathcal{P}$, then the whole information concerning the object denoted by t_n is given by the context $\langle t_n, \ldots, t_1\rangle$, which is called the *associated context* of t_n. In such a context, t_n will

[9] Consequently, by default, $\emptyset_{th} \in \mathcal{H}_{\mathcal{P}}$ (equivalently, `root` $\in \mathcal{H}_{\mathcal{P}}$) for any $\mathcal{L}_{oo}$ program $\mathcal{P}$.

[10] Two other relations ($\overset{\star}{\Vert\!<}$ and $\overset{\star}{\Vert\!\leq}$) will be useful in the following, based on the $\Vert\!<$ relation. If $t, t', t'' \in \mathcal{H}_{\mathcal{P}}$, then the following relations hold:

$$t \Vert\!< t' \in \mathcal{P} \implies \mathcal{P} \models t \overset{\star}{\Vert\!<} t' \tag{1}$$

$$t \Vert\!< t' \in \mathcal{P}, \mathcal{P} \models t' \overset{\star}{\Vert\!<} t'' \implies \mathcal{P} \models t \overset{\star}{\Vert\!<} t'' \tag{2}$$

and

$$\mathcal{P} \models t \overset{\star}{\Vert\!\leq} t \tag{3}$$

$$\mathcal{P} \models t \overset{\star}{\Vert\!<} t' \implies \mathcal{P} \models t \overset{\star}{\Vert\!\leq} t' \tag{4}$$

If $\mathcal{P} \models t \overset{\star}{\Vert\!\leq} t'$, we will say that t is a *descendant* of t', or that t' is an *ancestor* of t. If $\mathcal{P} \models t \overset{\star}{\Vert\!<} t'$, we will say that t is a *proper descendant* of t', or that t' is a *proper ancestor* of t.

represent the peculiar information about the abstraction, while the *parent context* $\langle t_{n-1}, \ldots, t_1 \rangle$ will describe the information shared with other abstractions. Thus, the whole information about a domain abstraction can be obtained by *composition* of all the units involved in a chain of *parent* relations.

The set of the *parent* relations in an $\mathcal{L}_{oo}$ program can be seen as a way to define the typical inheritance tree of object-oriented languages, where units can be used as a uniform representation for instances, classes, superclasses.[11]

Example 3.2. Consider again the program of Example 3.1, and re-interpret the *parent* relations as class/instance and class/superclass relations. Then, unit `kitty` represents an instance of class `cat`, which is a derived class of its superclass `pet`. In turn, since `pet` declares no parent unit, `pet` $\Vert\!\!<$ `root` implicitly holds, so that we can read `pet` as a base class.
The associated context of `kitty` is represented by the context $\langle$`kitty`, `cat`, `pet`$\rangle$, which can be thought as the linear composition of units `kitty`, `cat`, and `pet`, actually representing all the knowledge associated with a branch (from the instance to the root) of the inheritance tree of the program. □

Thus, when a context switch formula like $u \not\vdash a$ is encountered during a computation, it has to be read as a request to prove formula a using the clauses of the context associated with u. Roughly, any time the clauses contained in unit u do not suffice for the proof of a formula, the other units upon which u is based (according to Definitions 1-2) will be used, then adopting an *early binding* policy. This makes it possible to exploit of the *super* inheritance mechanism typical of object-oriented languages, which is used by classes to delegate computations to superclasses.

Instead, a *late binding* policy is obtained through the *self operator* $\sharp$ (`#` in the code). When a formula g is prefixed by such an operator in the body of a clause of a unit u in a context c, the proof of g will not depend on the clauses of the context associated with u only, but on all the clauses of context c. This enables classes to delegate method resolution to derived classes, thus reproducing the typical *self* inheritance mechanism.

Example 3.3. Consider again the program of Example 3.1, and the same computation. There, the query

```
?- kitty /- likes(jojo).
```

forces the proof of `likes(jojo)` in the associated context of the unit `kitty`. The context switch operator, then, makes context $\langle$`kitty`, `cat`, `pet`$\rangle$ become the current context of the computation. Since a clause for `likes(jojo)` is found in unit `pet`, this becomes the current unit for the computation, but the current context does not change. Then, when `#sex(female)` is encountered, the proof

[11] From this point of view, it should be noted that $\mathcal{L}_{oo}$ does not provide for multiple inheritance. Only simple inheritance is allowed, given that every unit can define one parent unit only.

of `sex(female)` does not depend on the clauses of $\langle$`pet`$\rangle$ only,[12] but on all the clauses of the current context $\langle$`kitty, cat, pet`$\rangle$, according to the semantics of the self operator $\sharp$. □

3.2 Constraining Object Representation

The main difference between $\mathcal{L}_{ob}$ and $\mathcal{L}_{oo}$ lays in the capability of expressing hierarchies of logic theories.[13] Elements of the application domain are conceptually organised in an inheritance tree through the *parent* relation, which captures both class/instance and class/superclass relationships in a uniform way. Domain abstractions are then represented as structured logic theories, obtained by composing the units associated with a branch of the inheritance tree into a context.

Usually, information about inheritance relationships is given in the text of a program, and is exploited *implicitly* during a computation. However, it may sometimes be useful to compute *explicitly* over inheritance relations, for instance in order to reason about classification.

Example 3.4. Consider again the program of Example 3.1: if we ask Kitty whether it likes Odie or not, the answer is yes:

```
?- kitty /- likes(odie).
yes
```

However, since it is a cat, we know that Kitty hates dogs,

```
?- kitty /- hates(Who).
Who = dog?
```

so it should hate Odie too, which is a dog. What we would need, then, is a way to say that the `likes/1` clauses of class `pet` should apply to pets of the same kind, in particular it should apply to instances of the same class derived from `pet` (two `dogs`, or two `cats`). □

Then, we may think of introducing a small collection of predicates, providing access to inheritance relations from inside a computation. For instance, suppose we define the following two operators: the `|<` binary operator, unifying its second argument with the term denoting the parent unit of the unit denoted by the first argument, and the `#|<` unary operator, unifying its argument with the base unit of the unit representing the self object at the time of the computation when the operator is encountered (a sort of "self version" of the first operator, intuitively). More sophisticated computations about classes can now be performed.

Example 3.5. Consider the $\mathcal{L}_{oo}$ program obtained from that of Example 3.1, by rewriting unit `pet` as follows.

---

[12] Since `pet` $\Vert\!\!<$ `root`, $\langle$`pet`$\rangle$ is the associated context of `pet`.

[13] An $\mathcal{L}_{ob}$ program $\mathcal{P}$ can be read as an $\mathcal{L}_{oo}$ program by assuming that $\forall t \in \mathcal{H}_{\mathcal{P}}, t \Vert\!\!< \emptyset_{th}$.

```
:- unit pet.
likes(X) :- #|< K, X |< K, #sex(female), X /- sex(male).
likes(X) :- #|< K, X |< K, #sex(male), X /- sex(female).
```

A pet, then, likes someone (`X`) if it is of the same kind (`K`) and of different sex. Thus, Kitty dislikes poor Odie.

```
?- kitty /- likes(odie).
no
```

In fact, as in the previous Examples, the `likes` formula is solved in the `kitty` associated context $\langle$`kitty, cat, pet`$\rangle$. Thus, the first clause of `pet`, which is the matching clause for `likes(odie)`, is renamed apart with fresh variables, say `X'` and `K'`, so that `X' = odie`, and the new resolvent becomes `#|< K', X' |< K', #sex(male), X' /- sex(female)`.

The predicate in the first subgoal `#|< K` gets the parent unit `cat` of the current self unit `kitty`, given that the current context is $\langle$`kitty, cat, pet`$\rangle$, and `kitty` $\Vdash\!<$ `cat`. As a consequence, `K'` is bound to `cat`.

Since `X' = odie` and `odie` $\Vdash\!<$ `dog`, the subgoal `X' |< K'` tries to unify `K'` with `dog` and fails, given that `K' = cat`. The second matching clause of `pet` obviously leads to failure in the same way. □

However, such an approach seems too *ad hoc* to our purposes, and introduces new problems to be solved, basically given by its essentially operational nature.

Example 3.6. Consider the $\mathcal{L}_{oo}$ program of Example 3.5. What result should we expect from the following query?

```
?- kitty /- likes(Who).
```

After the initial goal has been rewritten as `#|< K', X' |< K', #sex(female), X' /- sex(male)`, with `X' = Who`, the subgoal `#|< K'` unifies `K'` with `cat`. However, the subgoal `X' |< K'` cannot produce now any meaningful result, since `X'` is unbound, the parent unit of unit `X'` is unknown, and its identifier cannot be unified with `K'`. Thus, neither success nor failure should be accepted as satisfactory results for this computation. □

Even though some suspension semantics could be associated with the new operators, a more general and satisfactory approach consists of re-interpreting inheritance relations (*parent* relations, here) as *hierarchical constraints*. Such constraints bound any domain abstraction to be a node of an inheritance tree, involving in principle all the elements of the Herbrand Universe of a program. Thus, given a program $\mathcal{P}$ of $\mathcal{L}_{oo}$, each element of $\mathcal{H}_{\mathcal{P}}$ is in principle a unit identifier, which is bound to another one through a *parent* relation. Such a relation can be either given statically in the text of the program, or dynamically inferred from a set of hierarchical constraints during a computation.

For instance, the two operators defined above could be re-interpreted as constraint predicates: the first one constraining its second argument to denote the parent unit of the one identified by the first argument, the second one bounding its argument to denote the parent unit of the self unit, that is the unit representing the self object.

The main advantage of such an approach lays in its declarativeness, which frees the model from all the operational details needed for the operator description. A simple structure can be given as the constraint domain, and the corresponding notions of consistency and entailment can be exploited to perform simple inferences over the inheritance tree, as shown in Section 4.[14] The following Example should help catching the intuition laying behind hierarchical constraints.

Example 3.7. The same query of Example 3.6 produces now to the following result:

```
?- kitty /- likes(Who).
Who |< cat, Who /- sex(male)?
```

meaning that Kitty likes male cats, as desirable. In fact, when subgoal `X' |< K'` is selected, it is simply taken as a constraint bounding unit `K'` to be the parent unit of unit `X'` (that is, to be unit `cat`, since `X'` $\Vdash\!\!<$ `cat` at this point of the computation). The next subgoal (`#sex(female)`) simply succeeds, while the last results in the provability constraint `X' /- sex(male)`. Finally, the computation successfully ends, by returning as conditional answer the two alive constraints, the hierarchical one and the provability one, applying to the query variable `Who`, given that `Who = X'`. $\square$

Moreover, this approach makes it possible to fully exploit the dynamicity of the inheritance relations. In fact, since *parent* relations correspond to hierarchical constraints, inheritance relations can now

- be given as input constraints
- inferred during a computation
- provided as output results

In particular, we are no longer forced to define the whole taxonomy of the application domain once for all, statically, in the text of a program. Instead, a unit could be defined in a $\mathcal{L}_{oo}$ program without declaring its parent unit, through the `|<?` unary postfix operator. Its parent unit could instead be determined as a result of a computation, thus carrying out a simple process of *dynamic classification*.

[14] At any moment of the computation, a set $\mathcal{T}$ of hierarchical constraints is consistent if there exists at least another hierarchical constraint set $\mathcal{UT}$ and a substitution ϑ such that $\mathcal{T}\vartheta \subseteq \mathcal{UT}$, and $\mathcal{UT}$ defines an inheritance tree where (i) each node is labelled by an element of $\mathcal{H}_{\mathcal{P}}$, (ii) each element of $\mathcal{H}_{\mathcal{P}}$ labels one and only one node, and (iii) the root is labelled by $\emptyset_{th}$ (`root`).

Example 3.8. Consider the following $\mathcal{L}_{oo}$ program text.

```
:- unit kitty |< cat.   :- unit jojo |<?.       :- unit odie |<?.
colour(white).          colour(black).          colour(brown).
sex(female).            sex(male).              sex(male).

:- unit dog |< pet.     :- unit cat |< pet.
eats(bone).             eats(mouse).
hates(X) :- X |< cat.   hates(X) :- X |< dog.

:- unit pet.
likes(X) :- #|< K, X |< K, #sex(female), X /- sex(male).
likes(X) :- #|< K, X |< K, #sex(male), X /- sex(female).
```

`jojo` and `odie` do not declare their parent unit, so that we cannot tell from the program which classes Jojo and Odie are instance of. The following computation

```
?- jojo /- hates(odie), kitty /- likes(jojo).
jojo |< cat, odie |< dog?
```

succeeds and classifies Jojo and Odie as a cat and a dog, respectively.
The first subgoal is treated as a provability constraint, even though the unit identifier is the name of a unit of the program. The associated context of `jojo`, in fact, is unknown, since its parent unit is unknown in turn.
The second subgoal, instead, forces `jojo` to have the same parent unit as `kitty` (that is, `cat`), through the first clause of `pet`. Once the associated context of `jojo` is known to be $\langle$`jojo`, `cat`, `pet`$\rangle$ (since `jojo` $\|\!<$ `cat`), it can be used for the proof of `hates(odie)`. Then, the third clause of `cat` constrains hated `odie` to be a `dog`. The computation finally ends with success, and the two hierarchical constraints `jojo |< cat` and `odie |< dog` can be given as output results. □

Example 3.8 shows a simple case of dynamic classification. The query does not contain any variable, and could be read as representing a couple of observations about the domain of discourse. Then, the proof exploits the program so as to infer from such observations which classes the two abstractions (Jojo and Odie) belong to.

Dynamic classification represents an example of *declarative program extension*. At the end of the computation, the $\mathcal{L}_{oo}$ program of Example 3.8 contains several $\|\!<$ relations, whose some ones stated by the program text (such as `kitty` $\|\!<$ `cat` and `cat` $\|\!<$ `pet`), and some others inferred from the computation induced by the query (such as `jojo` $\|\!<$ `cat` and `odie` $\|\!<$ `dog`). What we would like to stress here is that the program obtained through dynamic extension in Example 3.8 has substantially the same behaviour of the program which we could obtain by adding the two output relations to the program text, as shown in Example 3.9 below.

Example 3.9. By extending the program text of Example 3.8 with the two relations `jojo` $\Vert\!\!<$ `cat` and `odie` $\Vert\!\!<$ `dog`,

```
:- unit kitty |< cat.    :- unit jojo |< cat.    :- unit odie |< dog.
colour(white).           colour(black).          colour(brown).
sex(female).             sex(male).              sex(male).

...                      ...                     ...
```

the query

```
?- jojo /- eats(W).
W = mouse?
```

produces the same result (apart from output constraints) of the query

```
?- jojo /- hates(odie), kitty /- likes(jojo), jojo /- eats(W).
jojo |< cat, odie |< dog, W = mouse?
```

when performed with respect to the program of Example 3.8. □

The main difference between the programs of Example 3.8 and Example 3.9 is that the latter bounds all computations to treat Odie as a dog and Jojo as a cat through a static, non-modifiable declaration. Instead, the former may produce either the same kind of classification (as in Example 3.8) or a completely different one, such as in the following Example.

Example 3.10. With respect to the program text of of Example 3.8, the query

```
?- odie /- hates(jojo), kitty /- likes(odie).
odie |< cat, jojo |< dog?
```

produces a quite different answer than the query of Example 3.8. □

3.3 Constrained Object-Oriented Computations

As in the case of $\mathcal{L}_{ob}$, the computational model of $\mathcal{L}_{oo}$ may be given different interpretations. Like $\mathcal{L}_{ob}$, logic theories encapsulates both methods and attributes of the objects, which are represented in a uniform way by means of clauses, and message passing is re-interpreted as the delegation of the proof of a logic formula from one theory to another.

In addition, $\mathcal{L}_{oo}$ endorses an object-oriented model for programs and computations. Programs are structured as a hierarchy of software components (units), providing for a uniform representation of both classes and instances in form of structured logic theories, called contexts, obtained by unit composition. As a result, classes are first-class objects as instance are, so they can be used as recipients for messages. Moreover, typical inheritance mechanisms, with super and self binding, can be easily exploited in $\mathcal{L}_{oo}$. The (simple) inheritance tree of an $\mathcal{L}_{oo}$ program is not statically defined once for all in the text of the program, but can be inferred (either partially or totally) in a dynamic way during a computation. Thus, the programmer is not forced to classify all the domain abstractions *a priori*, but can leave the task of classifying some of them to the computation.

On the other hand, a full declarative model is provided. Like in the case of $\mathcal{L}_{ob}$, object properties and their relationships are represented relationally, and a declarative semantics can be easily defined.

The main point, however, lays again in the ability to perform constrained computations over a user-defined domain modelled through an object-oriented data model. Both object properties (through provability constraint) and object classification (through hierarchical constraints) are handled as constraints, so that they can be given as inputs, imposed by a computation, or returned as output results. Take for instance Example 3.7, where the initial goal `kitty /- likes(Who)` bounds object `Who` to be liked by `kitty`. Then, the computation transforms such an input constraint into two output constraints, a hierarchical constraint classifying `Who` as a `cat`, and a provability constraint bounding the property `sex` of `Who` to `male`. Any possible further computation (that is, any computation produced by a goal having `kitty /- likes(Who)` as its first subgoal) would take automatically into account that `Who` is a male cat, so that `Who` could be subsequently bound to `jojo`, but neither to `odie`, which is a male dog, nor to `kitty`, which is a female cat.

A first, obvious limitation of $\mathcal{L}_{oo}$ from an object-oriented viewpoint is that it does not support any meaningful notion of object state. Since $\mathcal{L}_{oo}$ instances are fixed and non-mutable, they can be neither created nor modified during a computation, so all possible instances have to be foreseen and given *a priori* by the programmer in the program text. The problem of creating and configuring instances from class templates has been addressed in [10] by providing mechanisms for declarative theory creation and configuration, based on an abductive framework.

4 The Operational Semantics

4.1 The General Framework

As a multi-theory logic language, $\mathcal{L}_{oo}$ can be described in terms of the general framework defined in [12]. Within this framework, the semantics of a multi-theory language is given *transformationally*, by mapping the language onto a more general one (called *scope language*) whose semantics is taken as known.

In short, such a framework is based on the notion of *scope*, as the way in which any multi-theory logic language captures the abstractions of the application domain. In particular, a *scope language* replaces the standard notion of atomic formula with that of σ-formula, which formalises the idea that the truth value of a formula in a multi-theory logic language is no longer absolute, but depends on the particular domain abstraction which the formula is referred to, and which is described through a collection of logic axioms. Thus, a σ-formula is a formula associated with a scope, where its truth value has to be determined.

Say that $\mathcal{L}$ is a multi-theory language, and $\mathcal{A}_{\mathcal{L}}$ is the set of all its atomic formulae. Then, the behaviour of $\mathcal{L}$ can be fully described in terms of the corresponding scope language $\mathcal{S}\left\langle \mathcal{I}^{\sigma}_{\mathcal{L}}, \mathcal{A}_{\mathcal{L}}\right\rangle$ by defining

- the set of the scope identifiers $\mathcal{I}^{\sigma}_{\mathcal{L}}$
- the σ-function $\sigma_{\mathcal{L}}$
- the default scope $\emptyset^{\sigma}_{\mathcal{L}}$
- the σ-representation function $\rho^{\sigma}_{\mathcal{L}}$

The σ-function maps any formula in the body of an $\mathcal{L}$ clause into the corresponding σ-formula. In particular, according to [12], if $a \in \mathcal{A}_{\mathcal{L}}$ is an atomic formula of $\mathcal{L}$, and $s \in \mathcal{I}^{\sigma}_{\mathcal{L}}$ is a scope identifier, then $|a|_s$ is an atomic σ-formula of $\mathcal{S}\left\langle \mathcal{I}^{\sigma}_{\mathcal{L}}, \mathcal{A}_{\mathcal{L}}\right\rangle$. The default scope is the scope where any query has to be evaluated, so that an initial goal G for a multi-theory language $\mathcal{L}$ implies that the initial goal for the corresponding scope language is $\sigma_{\mathcal{L}}(\emptyset^{\sigma}_{\mathcal{L}}, G)$. The σ-representation function maps $\mathcal{L}$ programs into programs of the scope language, which are sets of σ-clauses. In particular, given a scope identifier $s \in \mathcal{I}^{\sigma}_{\mathcal{L}}$ and a program $\mathcal{P}$ of $\mathcal{L}$, $\rho^{\sigma}_{\mathcal{L}}(\mathcal{P}, s)$ returns the σ-set of s, that is the set of the σ-clauses representing s (see [12] for the details).

As a consequence, in order to formally define $\mathcal{L}_{oo}$, next Subsections will map it onto the corresponding scope language $\mathcal{S}\left\langle \mathcal{I}^{\sigma}_{\mathcal{L}_{oo}}, \mathcal{A}_{\mathcal{L}_{oo}}\right\rangle$, which will be denoted henceforth with the shorter notation $\mathcal{S}_{\mathcal{L}_{oo}}$. Subsection 4.2 will give the syntactic structures of $\mathcal{S}_{\mathcal{L}_{oo}}$, by defining $\mathcal{I}^{\sigma}_{\mathcal{L}_{oo}}$, while Subsection 4.3 will map $\mathcal{L}_{oo}$ onto $\mathcal{S}_{\mathcal{L}_{oo}}$ by defining $\sigma_{\mathcal{L}_{oo}}$, $\emptyset^{\sigma}_{\mathcal{L}_{oo}}$, and $\rho^{\sigma}_{\mathcal{L}_{oo}}$. Finally, Subsection 4.4 will introduce an operational characterisation of $\mathcal{S}_{\mathcal{L}_{oo}}$ in form of a transition system, extending the one in [12] in order to better cope with the new notions of constraint introduced. This will complete the description of the $\mathcal{L}_{oo}$ operational semantics.

4.2 The Syntax

Like any logic language, $\mathcal{L}_{oo}$ alphabet contains variables, constraints, functions, and predicate symbols. Say $\Sigma_{\mathcal{L}_{oo}}$ is the set of the constant and function symbols of $\mathcal{L}_{oo}$, $\mathcal{V}_{\mathcal{L}_{oo}}$ the variable set, $\Pi_{\mathcal{L}_{oo}}$ the predicate set. Then, we will denote with $\tau(\Sigma_{\mathcal{L}_{oo}}, \mathcal{V}_{\mathcal{L}_{oo}})$, or with the shorter notation $\tau_{\mathcal{L}_{oo}}$, the set of the $\mathcal{L}_{oo}$ terms which can be built from $\Sigma_{\mathcal{L}_{oo}}$ and $\mathcal{V}_{\mathcal{L}_{oo}}$, and with $\mathcal{A}_{\mathcal{L}_{oo}}$ the set of the $\mathcal{L}_{oo}$ atomic formulae which can be built applying predicate symbols of $\Pi_{\mathcal{L}_{oo}}$ to terms of $\tau_{\mathcal{L}_{oo}}$.

A scope in $\mathcal{L}_{oo}$ is represented by a unit in a context. The context (called the *current context*) represents the self object at any point of the computation, in that it contains all the knowledge related to the object whose properties are currently under evaluation. The unit (called *current unit*) is a unit of the current context, and represents that part of the self object actually in charge of the computation.

Take for instance Example 3.1 on page 11, showing the computation induced by the proof of the goal `kitty /- likes(odie)`. After a clause of `pet` has been selected among the clauses of context associated with `kitty`, `pet` represents the current unit of the computation, while the current context is obviously $\langle \mathtt{kitty}, \mathtt{cat}, \mathtt{pet} \rangle$. We may easily interpret this by saying that the `likes` property of `kitty` depends on its being a `pet`.

The set of the *parent* relations of an $\mathcal{L}_{oo}$ program associates one context to each unit. On the other hand, the current context of a computation is always the associated context of a unit, called the *self unit*, representing the peculiar part of the self object. As a consequence, we only need two unit identifiers in order to denote a scope of $\mathcal{L}_{oo}$: the self unit and the current unit identifier.

Since units in $\mathcal{L}_{oo}$ are denoted through terms (ground terms in the unit definition in the text of the program), a pair of terms can suffice for scope denotation. As a result, the set $\mathcal{I}^{\sigma}_{\mathcal{L}_{oo}}$ of the scope identifiers for $\mathcal{L}_{oo}$ is defined as follows:

$$\mathcal{I}^{\sigma}_{\mathcal{L}_{oo}} ::= \{ t \diamond t' \,|t, t' \in \tau_{\mathcal{L}_{oo}} \} \tag{5}$$

so that $o_i \diamond o_j$ represents the scope having self unit o_i and current unit o_j.

This completes the definition of the syntactic structure of $\mathcal{S}_{\mathcal{L}_{oo}}$.

4.3 The Mapping

The σ-function $\sigma_{\mathcal{L}_{oo}}$ maps goal formulae of $\mathcal{L}_{oo}$ onto σ-formulae of $\mathcal{S}_{\mathcal{L}_{oo}}$. Precisely, it takes a scope denotation (an element of $\mathcal{I}^{\sigma}_{\mathcal{L}_{oo}}$) and a goal formula of $\mathcal{L}_{oo}$, and returns a σ-formula of $\mathcal{S}_{\mathcal{L}_{oo}}$ as its arguments. Be $o, o_i, o_j \in \tau_{\mathcal{L}_{oo}}$ terms, a an atomic formula and g, g' goal formulae of $\mathcal{L}_{oo}$. Then, we define $\sigma_{\mathcal{L}_{oo}}$ as follows.

- $\sigma_{\mathcal{L}_{oo}}(o_i \diamond o_j, a) ::= |a|_{o_i \diamond o_j}$ $(\sigma 1)$
- $\sigma_{\mathcal{L}_{oo}}(o_i \diamond o_j, (g, g')) ::= \sigma_{\mathcal{L}_{oo}}(o_i \diamond o_j, g), \sigma_{\mathcal{L}_{oo}}(o_i \diamond o_j, g')$ $(\sigma 2)$
- $\sigma_{\mathcal{L}_{oo}}(o_i \diamond o_j, o \not\vdash g) ::= \sigma_{\mathcal{L}_{oo}}(o \diamond o, g)$ $(\sigma 3)$
- $\sigma_{\mathcal{L}_{oo}}(o_i \diamond o_j, \sharp g) ::= \sigma_{\mathcal{L}_{oo}}(o_i \diamond o_i, g)$ $(\sigma 4)$

Definition $(\sigma 1)$ handles atomic formulae, and can be taken as a simple illustration of the meaning of the σ-function: when evaluated with respect to a scope s, an atomic formula a of $\mathcal{L}_{oo}$ directly matches the σ-formula $|a|_s$ through the σ-function. Definition $(\sigma 2)$ trivially deals with conjunctions of $\mathcal{L}_{oo}$ formulae. Definition $(\sigma 3)$ and Definition $(\sigma 4)$ handle *extension formulae* (see [8, 4]) of $\mathcal{L}_{oo}$, containing those meta-operators directly manipulating a formula scope. Definition $(\sigma 3)$ defines the semantics of a context switch formula $o \not\vdash g$: independently of the current scope (the first argument of the σ-function), g has to be proven in a scope whose current and self units are both set to o. According to Definition ($\sigma 4$), instead, the truth value of $\sharp g$ depends on the current scope $o_i \diamond o_j$: by making the self unit o_i become the current unit, all the clauses of the current context (o_i associated context) can be used for the proof of the formula g.

A formal definition of hierarchical constraints is now needed in order to complete $\sigma_{\mathcal{L}_{oo}}$ definition. $\mathcal{L}_{oo}$ allows three[15] constraint predicate symbols to be used in $\mathcal{L}_{oo}$ clause bodies: the unary predicates $\vert\!\!<$ and $\sharp\vert\!\!<$, bounding their argument to denote the parent unit of the current and self unit, respectively, and the binary predicate $\vert\!\!<$, bounding its second argument to be the identifier of the parent unit of the unit denoted by its first argument. Correspondingly, $\mathcal{S}_{\mathcal{L}_{oo}}$ adopts only one binary predicate symbol, $\vert\!\!<$, representing a constraint directly matching a *parent* relation.

Definition 4.1. (*hierarchical constraint*) Given $t, t' \in \tau(\Sigma_{\mathcal{L}_{oo}}, \mathcal{V}_{\mathcal{L}_{oo}})$, the syntax

$$t \vert\!\!< t'$$

represents a *hierarchical constraint* bounding term t to denote a unit whose parent unit is denoted by t'. □

The correspondence between $\Vert\!\!<$ relations and $\vert\!\!<$ constraints can then be stated as follows. Given an $\mathcal{L}_{oo}$ program $\mathcal{P}$, a set of equality and hierarchical constraint C resulting from a formula proof w.r.t. $\mathcal{P}$, and two ground terms $o_i, o_j \in \mathcal{H}_{\mathcal{P}}$, it obviously holds that

$$C \models o_i \vert\!\!< o_j \iff o_i \Vert\!\!< o_j \in \mathcal{P} \tag{6}$$

It is worth to be noticed that $\vert\!\!<$ constraints range over pairs of generic (even non-ground) terms, while a $\Vert\!\!<$ relation is meaningful to an $\mathcal{L}_{oo}$ program only when it bounds a pair of ground terms.

Two other relations, represented through the infix operators $\overset{\star}{\vert\!\!<}$ and $\overset{\star}{\vert\!\!\leq}$,[16] will be useful in the following, based on the $\vert\!\!<$ constraint. Given a set C of equality and hierarchical constraints, then

$$C \models t \vert\!\!< t' \implies C \models t \overset{\star}{\vert\!\!<} t' \tag{7}$$

$$C \models t \vert\!\!< t',\ t' \overset{\star}{\vert\!\!<} t'' \implies C \models t \overset{\star}{\vert\!\!<} t'' \tag{8}$$

and

$$C \models t \overset{\star}{\vert\!\!\leq} t \tag{9}$$

$$C \models t \overset{\star}{\vert\!\!<} t' \implies C \models t \overset{\star}{\vert\!\!\leq} t' \tag{10}$$

[15] Only the last two have actually been discussed in the Examples above.

[16] This Definitions obviously match the $\overset{\star}{\Vert\!\!<}$ and $\overset{\star}{\Vert\!\!\leq}$ relations, respectively, from Definitions (1-4) on page 12.

The correspondence between $\mathcal{L}_{oo}$ and $\mathcal{S}_{\mathcal{L}_{oo}}$ is obviously set through the following suitable definition of the σ-function $\sigma_{\mathcal{L}_{oo}}$,[17] where $o, o', o_i, o_j \in \tau_{\mathcal{L}_{oo}}$, a is an $\mathcal{L}_{oo}$ atomic formula, g, g' are $\mathcal{L}_{oo}$ formulae.

- $\sigma_{\mathcal{L}_{oo}}(o_i \diamond o_j, \,|\!\!< o) ::= o_j \,|\!\!< o$ $\quad(\sigma 5)$
- $\sigma_{\mathcal{L}_{oo}}(o_i \diamond o_j, \sharp|\!\!< o) ::= o_i \,|\!\!< o$ $\quad(\sigma 6)$
- $\sigma_{\mathcal{L}_{oo}}(o_i \diamond o_j, o \,|\!\!< o') ::= o \,|\!\!< o'$ $\quad(\sigma 7)$

The default scope for $\mathcal{L}_{oo}$ is obviously represented by the empty theory $\emptyset_{th}$, so that we define $\emptyset^{\sigma}_{\mathcal{L}_{oo}}$ as follows:

$$\emptyset^{\sigma}_{\mathcal{L}_{oo}} ::= \emptyset_{th} \diamond \emptyset_{th} \tag{13}$$

Finally, given an $\mathcal{L}_{oo}$ program $\mathcal{P}$, and the terms $o, o_i, o_j \in \mathcal{I}^{\sigma}_{\mathcal{L}_{oo}}$, the σ-representation function $\rho^{\sigma}_{\mathcal{L}_{oo}}$ can be defined recursively as follows.

- $\rho^{\sigma}_{\mathcal{L}_{oo}}(\mathcal{P}, o \diamond \emptyset_{th}) ::= \emptyset$ $\quad(\rho 1)$
- $$\rho^{\sigma}_{\mathcal{L}_{oo}}(\mathcal{P}, o_i \diamond o_j) ::= \begin{array}{c} \left\{ |h|_{o_i \diamond o_j} \leftarrow \sigma_{\mathcal{L}_{oo}}(o_i \diamond o_j, B) \,\middle|\, o_j :: h \leftarrow B \ \in \mathcal{P} \right\} \\ \bigcup \\ \left\{ |h|_{o_i \diamond o_j} \leftarrow B \,\middle|\, |h|_{o_i \diamond o_k} \leftarrow B \in \rho^{\sigma}_{\mathcal{L}_{oo}}(\mathcal{P}, o_i \diamond o_k), o_j \,\|\!\!< o_k \right\} \end{array} \tag{$\rho 2$}$$

Definition ($\rho 2$) show how the σ-set of a generic $\mathcal{L}_{oo}$ scope has to be built, as a union[18] of two sets of σ-clauses: the first obtained by transforming the clauses of the current unit into suitable σ-clauses, the second (representing the recursive part of the definition) obtained by "lifting up" the σ-clauses belonging to the rest of the associated context of the current unit (in other words, the associated context of the parent unit of the current unit).[19] Definition ($\rho 1$) actually represents the termination condition of this recursive definition, by stating that no formula is true in the empty theory $\emptyset_{th}$ (`root` in the code), whatever the current context is.

[17] Only Definition ($\sigma 5$) would be actually needed, in case the two $\mathcal{L}_{oo}$ constraints used in Examples 3.7-3.8 were defined through the following equivalences:

```
#|< X   ≡   # (|< X)      (11)
X |< Y  ≡   X /- (|< Y)   (12)
```

[18] This implements a pure extension policy for unit composition. More complex strategies can be obtained by using more sophisticated operators instead of the union operator (see [11] for an example).

[19] By definition, only clauses from the context associated with the current unit are taken into account by the σ-representation function, thus making static inheritance the default $\mathcal{L}_{oo}$ inheritance mechanism.

4.4 The Semantics

The semantics characterisation of scope languages presented in [12] is powerful enough to capture most of the multi-theory languages known in literature, extending standard LP syntax with new operators enabling scope manipulation. However, a proof-theoretical characterisation relying on an inference system in the sequent calculus seems now inadequate to gracefully cope with the constraint-based computations of $\mathcal{L}_{oo}$. A more satisfactory solution is to rewrite the inference engine in terms of a transition system, as usual for CLP languages [6], where states are characterised by different (multi-)sets of atomic formulae and constraints, and transitions between states are described in form of four standard Transition Rules.

In the following, we will first sketch briefly the standard CLP proof-theoretical semantics, then we will extend it to cope with $\mathcal{L}_{oo}$, taking into account both provability and hierarchical constraints.

4.4.1 Top-down execution model for a constraint logic language

It is known that standard LP can be taken as a particular instance of the general CLP($\mathcal{X}$) scheme [5]. Prolog, for instance, can be interpreted as a constraint logic language whose constraints are equality relations over finite trees (terms of the Herbrand Universe).

Following [6], the operational semantics of a CLP language can be given by expressing its top-down execution model in terms of a transition system. Transitions occur between states represented as triples of the form $\langle A, C, S\rangle$. A contains the resolvent, consisting of atomic formulae and constraints, which has to be evaluated. C and S altogether constitute the *constraint store*, collecting both active (in C) and passive (in S) constraints.

The transition system is based on four different Transition Rules: *resolution* ($\rightarrow_r$), *constraining* ($\rightarrow_c$), *inference* ($\rightarrow_i$), and *satisfiability* ($\rightarrow_s$). Transition Rule $\rightarrow_r$ represents the resolution step of the inference engine, which takes an atomic formula and looks for a clause head to be unified with the selected subgoal.

- $\langle A \cup a, C, S\rangle \rightarrow_r \langle A \cup B, C, S \cup (a = h)\rangle$ $(r1)$

  where $h \leftarrow B$ is a program clause, a and h have the same predicate symbol, and $(a = h)$ summarises all the equations resulting from unification of a with the clause head h.

- $\langle A \cup a, C, S\rangle \rightarrow_r \mathit{fail}$ $(r1')$

  where no clause $h \leftarrow B$ exists whose head h has the same predicate symbol of a.

Transition Rule $\rightarrow_c$ adds a new constraint to the store, by taking it from the resolvent.

- $\langle A \cup c, C, S\rangle \rightarrow_c \langle A, C, S \cup c\rangle$ $(r2)$

  where c is an atomic constraint.

The general role of the Transition Rule $\rightarrow_i$ is to infer new constraints from the constraint store through the *infer* function, which takes C and S and returns a (possibly) different state (C', S').

- $\langle A, C, S\rangle \rightarrow_i \langle A, C', S'\rangle$ $(r3)$

  where $(C', S') = \mathit{infer}(C, S)$.

The last Transition Rule $\rightarrow_s$ is in charge of checking the satisfiability of the constraint store, through the definition of the *consistent* predicate, which applies to the active constraints in C.

- $\langle A, C, S\rangle \rightarrow_s \langle A, C, S\rangle$ $(r4)$

  where $\mathit{consistent}(C)$ holds.

- $\langle A, C, S\rangle \rightarrow_s \mathit{fail}$ $(r4')$

  where $\mathit{consistent}(C)$ does not hold.

While Transition Rules ($r1$–$r2$) are common to all CLP systems, Transition Rules ($r3$–$r4'$) are usually exploited in order to specialise the system of choice, by means of suitably tailored definitions of the *infer* function and of the *consistent* predicate.

The *derivation* of an initial goal G is then represented by a sequence of transitions starting from the initial state $\langle G, \emptyset, \emptyset\rangle$. Such a derivation ends when no more transitions can be applied, and succeeds when A is the empty set in the final state (fails otherwise).

4.4.2 Extending the model

Several things have to be taken into account in order to extend the aforementioned transition system so as to make it applicable to $\mathcal{S}_{\mathcal{L}_{oo}}$, the scope language of $\mathcal{L}_{oo}$. First, the system has to deal with σ-formulae, instead of standard logic formulae. Moreover, it has to cope with both provability and hierarchical constraints. For these purposes, the resolvent A is to be thought as a multi-set of atomic σ-formulae and constraints (hierarchical and provability ones).

What needs be formalised is the distinction between provability constraints and σ-formulae, both syntactically and semantically. From a syntactic viewpoint, we will denote a provability constraint as follows.

Definition 4.2. (*provability constraint*) Be $a \in \mathcal{A}_{\mathcal{L}_{oo}}$, $o_i \diamond o_j \in \mathcal{I}^{\sigma}_{\mathcal{L}_{oo}}$. Then, the notation

$$o_i \diamond o_j \not\vdash a$$

denotes a *provability constraint* bounding the atomic formula a to logically follow from unit o_j in the context o_i (i.e., the associated context of unit o_i). □

Intuitively, a provability constraint is a σ-formula whose scope is not sufficiently specified, so that its σ-set (that is, the set of the σ-clauses representing its scope) cannot be properly computed through the σ-representation function. Then, we should somehow define a notion of applicability of the σ-representation function, by delimiting its domain: whenever enough information is not given to the function, it can not produce the result, that is the σ-set of a given scope. In other terms, a provability constraint $s \not\vdash a$ should be handled as the σ-formula $|a|_s$ iff the σ-set of its scope can be properly determined. The idea is then to define our transition system so that at any time of the computation the resolvent A consists of σ-computable formulae and constraints only.[20]

For this purpose, we re-formulate Definition (σ1) as follows:

$$\sigma_{\mathcal{L}_{oo}}(o_i \diamond o_j, a) ::= o_i \diamond o_j \not\vdash a \qquad (\sigma 8)$$

Moreover, to the end of formally defining the notion of *σ-computability* (that is, the computability of the set of the σ-clauses representing a scope), we introduce the σ-computability relation $\sigma\text{-}comp_{\mathcal{L}_{oo}}$, based on the $\models\!<$ one.

Definition 4.3. ($\sigma\text{-}comp_{\mathcal{L}_{oo}}$) Given a set C of equality and hierarchical constraints of $\mathcal{S}_{\mathcal{L}_{oo}}$, and a scope identifier $o_i \diamond o_j \in \mathcal{I}^{\sigma}_{\mathcal{L}_{oo}}$, we define the unary relation $\sigma\text{-}comp_{\mathcal{L}_{oo}}$ such that

$$C \models \sigma\text{-}comp_{\mathcal{L}_{oo}}(o_i \diamond o_j) \iff \begin{array}{l} C \models o_i \overset{\star}{\models\!<} \emptyset_{th}, o_i \overset{\star}{\models\!<} o_j \\ C \models o_i \overset{\star}{\models\!<} o \Rightarrow o \in \mathcal{H}_{\mathcal{P}} \end{array}$$

Then, we say that scope $s \in \mathcal{I}^{\sigma}_{\mathcal{L}_{oo}}$ is σ-computable (with respect to C) iff $C \models \sigma\text{-}comp_{\mathcal{L}_{oo}}(s)$. Correspondingly, a σ-formula $|a|_s$ is said to be σ-computable if and only if its scope s is σ-computable. □

Intuitively, the $\sigma\text{-}comp_{\mathcal{L}_{oo}}$ relation holds if and only if it is applied to a scope identifier which is ground and such that the hierarchy associated with it is completely given in the constraint set.

[20] It is worth to be noticed that the symbol $\not\vdash$ has different meanings in $\mathcal{L}_{oo}$ and $\mathcal{S}_{\mathcal{L}_{oo}}$. It represents the $\mathcal{L}_{oo}$ context switch operator, and it is used to denote $\mathcal{S}_{\mathcal{L}_{oo}}$ provability constraints, too. This is mainly meant to stress the strong connection between the two notions, formalising the interpretation of a context switch operator as a provability constraint.

More precisely, given a term o and an atom a, for any scope s it holds that

$$\sigma_{\mathcal{L}_{oo}}(s, o \not\vdash a) \overset{\sigma 3}{=} \sigma_{\mathcal{L}_{oo}}(o \diamond o, a) \overset{\sigma 8}{=} o \diamond o \not\vdash a$$

where $o \not\vdash a$ is a (context switch) formula of $\mathcal{L}_{oo}$, while $o \diamond o \not\vdash a$ is a (provability constraint) formula of $\mathcal{S}_{\mathcal{L}_{oo}}$.

The idea is that all $\mathcal{L}_{oo}$ formulae are mapped onto $\mathcal{S}_{\mathcal{L}_{oo}}$ constraints by the σ-function, according to Definitions (σ2-σ8), and that a provability constraint $s \not\vdash a$ is transformed into a σ-formula $|a|_s$ and shifted from the constraint store to the resolvent A when and only when its scope is σ-computable, i.e. when $\sigma\text{-}comp_{\mathcal{L}_{oo}}(s)$ is true with respect to the constraint store.

As a consequence, Transition Rule ($r3$) (inference) has to be substantially modified, taking into account that provability constraints may result in new formulae to be added to the resolvent. Thus, the extended version of Transition Rule ($r3$) can be formalised as follows.

- $\langle A, C, S\rangle \rightarrow_i \langle A \cup A', C', S'\rangle$ ($r5$)
  where $(A', C', S') = infer(C, S)$.

In particular, the *infer* function

- moves all equality and hierarchical constraints from passive to active constraints, and propagates them. Apart from the obvious propagations, the following relation is exploited by the *infer* function:

$$C \cup S \models t \ltimes t',\, t \ltimes t'' \quad \Longrightarrow \quad t' = t'' \in C' \tag{14}$$

  representing simple inheritance in the system.

- transforms provability constraints whose scope is σ-computable into the corresponding σ-formulae, and moves them from passive constraints to the resolvent through the set A', while leaving the others untouched as passive constraints. In particular,

$$A' ::= \{|a|_s \mid s \not\vdash a \in S,\, C' \models \sigma\text{-}comp_{\mathcal{L}_{oo}}(s)\} \tag{15}$$

As a result, only equality and hierarchical constraints may be found in the active constraint store. Moreover, provability constraints are not propagated at all, and the constraint store should be actually thought as a two-sorted set of equality and hierarchical constraints joint to a multi-set of provability constraints. Finally, σ-formulae are put in the resolvent only when their scope is σ-computable.

Transition Rules ($r6$-$r6'$) generalise Transition Rules ($r1$-$r1'$). According to Definitions (σ2-σ8) and to the *infer* function definition, they are always applicable to any σ-formula occurring in A, since only σ-computable formulae can be found there.

- $\langle A \cup |p(\tilde{t})|_s, C, S\rangle \rightarrow_r \langle A \cup B, C, S \cup \{h = p(\tilde{t})\}\rangle$ ($r6$)
  where $s \in \mathcal{I}^{\sigma}_{\mathcal{L}_{oo}}$, $|h|_s \leftarrow B \in \sigma\text{-}set_p(s)$, and $\{h = p(\tilde{t})\}$ contains all the equations resulting from $p(\tilde{t})$ unification with the σ-clause head h.

- $\langle A \cup |p(\tilde{t})|_s, C, S\rangle \rightarrow_r \mathit{fail}$ $(r6')$
  where $\sigma\text{-}\mathit{set}_p(s) = \emptyset$.

On the other hand, as intuitive, Transition Rule $(r2)$ (constraining) has not to be changed, since formulae of the forms $s \not\vdash a$ and $o \models\!\!< o'$ are recognised as constraints by the transition system, and simply shifted from the resolvent A to the (passive) constraint store S.

Analogously, Transition Rule $(r4)$ has not to be changed, but a specific *consistent* definition has to be provided. Apart from handling equality relations, the *consistent* predicate has simply to check that the set of the hierarchical constraints is consistently built. In particular, the active constraint store C is consistent with respect to the hierarchical constraints if

- $C \models t \models\!\!< t' \implies C \models t \neq \emptyset_{th}$ (root)
- $C \models t \overset{\star}{\models\!\!<} t' \implies C \models t \neq t'$ $(\mathit{acyclicity})$
- $C \models t \models\!\!< t', t \models\!\!< t'' \implies C \models t' = t''$ $(\mathit{simple\ inheritance})$

Finally, in order to complete the specification of the $\mathcal{S}_{\mathcal{L}_{oo}}$ transition system, what is still to be given is the triple defining the initial state. As one may expect, given an initial goal G of $\mathcal{L}_{oo}$, the initial resolvent for $\mathcal{S}_{\mathcal{L}_{oo}}$ is defined to be $\sigma_{\mathcal{L}_{oo}}(\emptyset_{th} \diamond \emptyset_{th}, G)$, while the set of the passive constraints is empty. Instead, in order to make the $\sigma\text{-}\mathit{comp}_{\mathcal{L}_{oo}}$ relation meaningful from start, the initial set of the active constraints is represented by the (generally non-empty) set C_0 of the hierarchical constraints corresponding to the set of the $\|\!\!<$ relations contained in $\mathcal{P}_0$, that is the set of all the program relations given in the program text. So, coherently with (6) on page 22,

$$C_0 ::= \{\, o \models\!\!< o' \mid o \|\!\!< o' \in \mathcal{P}_0 \} \tag{16}$$

As a result, the initial state of a sequence of transitions representing an $\mathcal{S}_{\mathcal{L}_{oo}}$ derivation is defined as

$$\langle \sigma_{\mathcal{L}_{oo}}(\emptyset_{th} \diamond \emptyset_{th}, G), C_0, \emptyset\rangle$$

As usual, such a derivation ends when no more transitions can be applied, and succeeds when the final state has the empty resolvent (fails otherwise).

4.4.3 Program extension

The definition of the $\rho^{\sigma}_{\mathcal{L}_{oo}}$ function on page 23 makes its result depend on the program $\mathcal{P}$ taken as its first argument. However, one of the main points here is that an $\mathcal{L}_{oo}$ program is no longer bound to be statically given once for all by its textual definition. Instead, since $\|\!\!<$ relations correspond to hierarchical constraints, they can be either given statically in the text of a program, or dynamically inferred during a proof.

As a result, an $\mathcal{L}_{oo}$ program is not given once for all before the computations start, but may grow during a proof, so that, for instance, the number of scopes which are σ-computable possibly grows along with the set of the hierarchical constraints.

We have to define the notion of program which the σ-representation function can be applied. According to Definition ($\rho 2$), we have to specify exactly what an $\mathcal{L}_{oo}$ program is at any time of the computation. Generally speaking, we use to see a logic program as the knowledge used by the inference engine in order to derive new information by deduction. A Standard LP program, for instance, is represented by a fixed collection of definite Horn clauses.

Actually, an $\mathcal{L}_{oo}$ program $\mathcal{P}$ can be thought as a two-sorted set of relations, concerning the axioms and the taxonomy, respectively. In particular, $\mathcal{P}$ contains

- relations of the form $o :: c$, where $o \in \mathcal{H}_{\mathcal{P}}$ and c is an $\mathcal{L}_{oo}$ clause, constituting the axiomatic representation of the domain abstractions
- relations of the form $o \,\|\!\!< o'$, where $o, o' \in \mathcal{H}_{\mathcal{P}}$, representing the domain taxonomy.

If we denote the two set as $\mathcal{P}^{cl}$ and $\mathcal{P}^{\|\!<}$, respectively, it obviously holds that $\mathcal{P} = \mathcal{P}^{cl} \cup \mathcal{P}^{\|\!<}$. Now, if we denote with $\mathcal{P}_0$ the initial $\mathcal{L}_{oo}$ program from the textual definition, so that $\mathcal{P}_0 = \mathcal{P}_0^{cl} \cup \mathcal{P}_0^{\|\!<}$ (with $\mathcal{P}_0^{cl}$ and $\mathcal{P}_0^{\|\!<}$ having an obvious meaning), $\mathcal{P}_0^{cl} = \mathcal{P}^{cl}$ holds, since the clauses of an $\mathcal{L}_{oo}$ program do not change during a proof. Instead, $\mathcal{P}_0^{\|\!<} \subseteq \mathcal{P}^{\|\!<}$, since new $|\!\!<$ relations may be inferred from the set of the hierarchical constraints, possibly growing during a proof. By defining $\mathcal{P}^{\|\!<}$ as the set of the ground $|\!\!<$ relations which can be inferred from the (active) constraint store, we have formally established the (actually, quite intuitive) relationship between hierarchical constraints ($|\!\!<$) and *parent* relations ($\|\!\!<$).

$$\mathcal{P}^{\|\!<} ::= \{\, o \,\|\!\!< o' \mid C \models o \,|\!\!< o' \text{ and } o, o' \in \mathcal{H}_{\mathcal{P}} \} \tag{17}$$

where C is the active constraint store. Since now the set of relations constituting an $\mathcal{L}_{oo}$ program is defined at any time of the computation, $\rho^{\sigma}_{\mathcal{L}_{oo}}$ can be always applied. Moreover, the way in which hierarchical constraints are handled by the system ensures that only $\|\!\!<$ relations which are consistent with the previously existing ones can be added to the program, thus enforcing the declarative reading of $\mathcal{L}_{oo}$ programs.

5 Conclusions and Further Remarks

This work has presented a simple example of a constraint object-oriented computational model based on logic programming. Objects are represented as (structured) logic theories, organised in a hierarchy according to class/instance, class/superclass relationships. Computations evolve through message passing between objects. Both message passing and hierarchical relations between logic theories are re-interpreted here as meta-level constraints: respectively, provability constraints bound a logic formula to logically follow from a theory, while hierarchical constraints make it possible to infer the parent theory from a computation.

Apart from enforcing the declarative reading of a multi-theory logic language, these two kinds of constraints enable new forms of computations to be performed, where both object properties (through provability constraints) and object representations (through hierarchical constraint) can be both the subjects and the results of a logic proof. Throughout this paper, some simple examples have been given and discussed, illustrating the usefulness of the new notions introduced, by exploiting two simple multi-theory logic languages, $\mathcal{L}_{ob}$ and $\mathcal{L}_{oo}$.

However, one of the main criticisms which can be addressed to the approach presented here concerns the problem of object state. The languages presented do not allow any meaningful notion of object state to be captured: instances cannot be created from classes, and they can be neither configured nor modified during a computation. Actually, classes and instance only exist in the programmer's interpretation.

From this viewpoint, a more satisfactory approach for a multi-theory logic framework seems to be the one adopted by McCabe in [7], where parametric logic theories (where a logic variable used in a logic theory identifier has the whole theory as its scope) are used to define classes and instances of an object-oriented logic language. However, this approach too suffers from unacceptable limitations, such as the coincidence between object state and object identifier, thus badly addressing the problem of object identity, or the inability to gracefully cope with insufficiently specified theory names, thus weakening the declarative reading of the language.

Moreover, as discussed in particular in Subsection 4.4.2, the model presented here suggests a new notion of (multi-theory) logic program, where a program is no longer statically given once for all by its textual definition, but may instead be built as a result of a logic proof. Even though the languages presented here only partially exploit the power of such an idea, [10] shows how a multi-theory language of the same class of $\mathcal{L}_{oo}$ can be extended towards a notion of object state configuration. By enabling also the set of the clauses to grow during a computation in a constraint framework based on abduction, declarative object (theory) creation and configuration can be dynamically performed.

Intuitively, a program of $\mathcal{L}_{oo}$ can be conceived as a point in the space defined by the sets of all possible relations of the form $clause(unit, head, body)$ and of the possible taxonomy trees defined by *parent* relations. A program text, then, defines the program only partially, by restricting the subspace upon which it can range. Provability and hierarchical constraints, then, possibly restrict further such a space during a proof. Furthermore, many different sorts of constraints could be imagined and defined, so that provability and hierarchical ones can be seen as the simplest cases of constraints over object properties and over object representation, respectively. More generally, one could think of program relations (such as $o :: a \leftarrow B$, or $o_i \parallel\!\!< o_j$) being possibly modelled through *axiom constraints*, that is constraints bounding the clausal representation of domain abstractions (such as hierarchical constraint), while goal formulae, to be proven with respect to a theory (such as $o \not\vdash a$), could be modelled through *theorem constraints*, affecting the expected logical consequences of a logic theory (such as provability constraints). As a result, deduction, abduction, and induction could be all re-interpreted as inference mechanisms linking (either forward or backward) axiom constraints and theorem constraints. Future works will be devoted to explore the implications of such a re-interpretation of the notion of logic program.

Acknowledgements

First of all, I would like to thank Antonio Natali. Without its help and support, this research would have not even begun.

Other people contributed to make this work much better than what I could have done without their suggestions, remarks, and criticisms: the time that Enrico Denti, Evelina Lamma, Paola Mello, Michela Milano, and Fabrizio Riguzzi devoted to read and comment early versions of this report was really precious to me.

References

[1] A. Bossi, M. Gabbrielli, G. Levi, and M.C. Meo. Contributions to the semantics of open logic programs. In H. Tanaka, editor, *Proc. FGCS'92 Int. Conference*, pages 570–580, 1992.

[2] K.A. Bowen and R.A. Kowalski. Amalgamating Language and Metalanguage in Logic Programming. In K.L. Clark and S.A. Tarnlund, editors, *Logic Programming*, pages 153–173. Academic Press, 1982.

[3] A. Brogi, E. Lamma, and P. Mello. Objects in a Logic Programming Framework. In A. Voronkov, editor, *Logic Programming, LNCS*, pages 102–113. Springer-Verlag, 1992.

[4] M. Bugliesi, E. Lamma, and P. Mello. Modularity in Logic Programming. *Journal of Logic Programming*, 19-20:443–502, 1994.

[5] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. 14th Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[6] J. Jaffar and M.J. Mahrer. Constraint Logic Programming: a survey. *Journal of Logic Programming*, 19-20:503–581, 1994.

[7] F.G. McCabe. *Logic and Objects*. Prentice Hall International, London, 1992.

[8] D. Miller. A logical analysis of modules in LogicProgramming. *Journal of Logic Programming*, 6:79–108, 1989.

[9] L. Monteiro and A. Porto. Contextual logic programming. In G. Levi and M. Martelli, editors, *Proc. 6th Int. Conference on Logic Programming*, pages 284–302. The MIT Press, 1989.

[10] A. Omicini. Abduction and object state configuration. Technical Report DEIS-LIA-003-95, University of Bologna (Italy), 1995. LIA Series no. 3.

[11] A. Omicini. *Programmazione Logica Orientata agli Oggetti*. PhD thesis, Università di Bologna, Italy, February 1995.

[12] A. Omicini. A general framework for multi-theory logic languages. Technical Report DEIS-LIA-009-96, University of Bologna (Italy), 1996. LIA Series no. 16.

[13] Quintus Computer Systems, Inc., Mountain View, California. *Quintus Prolog User's Guide*, 1986.

[14] Swedish Institute of Computer Science, Kista, Sweden. *SICStus Prolog User's Manual*, 1994.