A General Framework for Multi-Theory Logic Languages

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. Multi-theory logic languages are a wide class of programming languages which extend the standard logic programming paradigm by partitioning the logic program into a multiplicity of logic theories. Such an extension is generally meant to address typical software engineering issues, such as modularity, component reuse, incremental software design and development. Despite of the many syntactic and semantic differences, what multi-theory logic languages share is the partitioning of the program according to the abstractions of the domain of discourse, by associating different collections of axioms to different domain elements. The aim of this work is then to introduce a general framework for multi-theory logic languages, providing a uniform approach for their definition and description. An extended notion of logic formula is introduced, which makes the relationship between formulae and domain abstractions explicit. A class of extended logic languages is consequently defined, which is general enough that any multi-theory logic language can be mapped onto it through suitable translation functions. Such languages, called *scope languages*, can then be used as a mean to express the semantics of any multi-theory logic language, and as a tool for the comparison of different languages of that kind.

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

Contents

1 Introduction **3**

2 Multi-Theory Logic Languages **3**

3 Logic Programming with Scopes **5**

3.1 The Syntax 5

3.2 The Mapping 7

3.3 The Operational Semantics 10

4 Some Examples **11**

4.1 Logic Programming with Modules 11

4.2 Contextual Logic Programming 12

4.3 Logic Objects 14

5 Conclusions **15**

1 Introduction

Modularity is an essential feature for any programming language to be effectively used as a tool for building real software systems. Many extensions to standard logic programming towards modularity have been designed and implemented [4, 5, 2, 10, 7, 8, 3, 17], allowing a multiplicity of logic theories (as separate collections of definite clauses) to be defined and used together in the same program. Many mechanisms for theory composition have been introduced, allowing more refined policies for software design and development. In this work, we will focus on those logic languages based on the extension of Horn clause logic, where theory composition operators can be embedded directly in clauses bodies, following [9].[1]

What all these language share is the abandonment of any assumption of universal validity for the axioms. A logic clause does no longer define a truth about the whole world, but is intended as stating a true property of an abstraction of the application domain. Correspondingly, the truth value of any formula depends on the particular domain abstraction which it is referred to.

These languages differ, instead (apart from the syntactic sugar) in how they represent the different elements of the domain of discourse: basically, how these are identified, and how axioms are associated with identifiers. By properly characterising these two peculiarities through appropriate structures and functions, a general logic language can be defined, which captures the behaviour of many known multi-theory languages in a uniform framework.

The aim of this work is then to define such a language, showing how some meaningful examples of multi-theory logic language can be simply mapped onto it. As a result, a uniform operational characterisation for multi-theory logic languages can be obtained, making it possible to put into evidence both common aspects and peculiarities.

2 Multi-Theory Logic Languages

Modular logic languages organise programs (as well as computations) as collections of separate logic theories. The structure of the programs of any multi-theory logic language can be seen as reflecting the programmer's own model of the application domain, where different collections of definite clauses are used to describe and represent different elements of the domain of discourse.

Multi-theory logic languages actually abandon the idea of the logic program as a monolithic entity, expressing universal truthes about the *world* (the application domain) taken as a whole. Instead, the partition of a logic program into a multiplicity

[1] So, we will not consider here the approaches based on algebraic meta-operators, inspired by the work of O' Keefe [13]. See [6] for a comprehensive survey on the subject, where both approaches are extensively discussed.

of logic theories better captures the idea of representing a collection of different abstractions of the domain of discourse through *partial*, *incomplete* knowledge chunks. Each knowledge chunk is then in principle a collection of axioms of some kind, representing what is true with respect to a given element of the domain.

Correspondingly, the truth value of a formula is no longer an absolute notion. In fact, as well as a property may hold for one object while not holding for another, each formula may be either true or false with respect to different collections of axioms representing different elements of the application domain.

So, each formula in a multi-theory logic language is to be associated with a specific abstraction of the application domain in order to be meaningful. A multi-theory language is then characterised by the following issues:

- how elements of the domain are denoted
- how elements of the domain are represented
- how formulae are associated with elements of the domain
- how the truth value of a formula with respect to a given element is determined

This work is based on the idea that a general abstract logic language can be defined, such that most multi-theory logic languages can be mapped onto it, whose operational semantics can be given in terms of a few, simple inferential rules. Such a language should be abstract enough to properly encapsulate any peculiarity of the mapped languages, by stressing instead their fundamental aspects such as the denotation and representation of the domain abstractions.

In order to be expressive enough for our purposes, such a general language should be parametrised with respect to the symbols belonging to each multi-theory language. In particular, each language is characterised by

- a first-order logic alphabet
- a set of theory identifiers

so that the syntax of the corresponding abstract language will be based on these two syntactic structures.

A suitable translation function is then to be defined for each language, providing the mapping between the abstract general language and the constructs of each multi-theory language, so that programs and formulae in the concrete language can be reformulated as programs and formulae in the proper syntax for the abstract language.

3 Logic Programming with Scopes

This Section introduces the concepts and definitions needed to build a general abstract language able to capture the behaviour of a multi-theory logic language. The main idea is that any logic formula has not a truth value *per se*, but only with respect to a given element of the application domain. We call *scope* the representation of any domain abstraction in a multi-theory program, since it constitutes the scope where the validity of a formula can be determined. Consequently, we will call *scope language* the general language for the definition and description of multi-theory languages.

This Section is organised as follows: first, Subsection 3.1 defines the basic notions and the syntax for the scope language, then Subsection 3.2 introduces functions and structures making it possible to map a multi-theory logic language onto the scope language itself. Finally, Subsection 3.3 defines the operational behaviour of the scope language, in terms of a simple inferential engine which subsumes and generalises the Standard LP engine.

3.1 The Syntax

From a syntactic viewpoint, the scope language is characterised by

- the set $\mathcal{I}^\sigma$ of the *scope identifiers*
- the set $\mathcal{A}$ of all the atomic logic formulae

A scope language will be then denoted in the following as $\mathcal{S}\langle\mathcal{I}^\sigma,\mathcal{A}\rangle$. So, this notation actually denotes a class of languages, parametrised according to the two sets $\mathcal{I}^\sigma$ and $\mathcal{A}$. Each multi-theory language, then, has its corresponding scope language according to its syntactic structure and to the choice of the scope identifiers.

As usual for first-order logic languages, $\mathcal{A}$ is on its turn based on a set of predicate symbols Π, a set of function and constant symbols Σ, and a set of first-order variables $\mathcal{V}$, from which atomic formulae of $\mathcal{A}$ are built. No restrictions apply on $\mathcal{I}^\sigma$, which may be represented (as we will see in some example) by a set of extra-logic symbols [3], or even by the set $\tau(\Sigma,\mathcal{V})$ of the terms built from Σ and $\mathcal{V}$. For instance, module-based languages like SICStus Prolog [18] allow constants to be used as module names. There, then, a scope corresponds to a Prolog module, that is a collection of Prolog clauses, and $\mathcal{I}^\sigma$ is represented by the set of the Prolog constant.

The basic elements of the syntax of a scope language, based on $\mathcal{I}^\sigma$ and $\mathcal{A}$, can now be defined.

Definition 3.1. (*atomic σ-formula*) Given a scope language $\mathcal{S}\langle\mathcal{I}^\sigma,\mathcal{A}\rangle$, let p be an n-ary predicate symbol of Π, and $\tilde{t}$ be a n-ple of terms of $\tau(\Sigma,\mathcal{V})$, so that $p(\tilde{t})\in\mathcal{A}$ is an atomic formula of $\mathcal{S}\langle\mathcal{I}^\sigma,\mathcal{A}\rangle$. If $s\in\mathcal{I}^\sigma$ denotes a scope, then

$$|p(\tilde{t})|_s$$

denotes an *atomic σ-formula* of $\mathcal{S}\langle\mathcal{I}^\sigma, \mathcal{A}\rangle$ whose scope is s and whose predicate symbol is p. □

Even though very simple, the notion of *atomic σ-formula* is actually the very fundamental notion upon which this approach is based. In fact, it extends the standard notion of atomic formula by formalising the idea of the association between a logic formula and its scope, which is the main point of any multi-theory logic language.

The general notion of σ-formula follows intuitively, by combining atomic σ-formulae with connectives, quantifiers, punctuation, so it will be not expressed here formally, but will be taken as understood. Instead, since they will be exploited throughout all this paper, the particular notions of *definite σ-clause* and of *σ-goal* are introduced below.

Definition 3.2. (*σ-clause*) Let $|h|_s, |b_1|_{s_1}, \ldots, |b_n|_{s_n}, n \geq 0$, be atomic σ-formulae of the scope language $\mathcal{S}\langle\mathcal{I}^\sigma, \mathcal{A}\rangle$. Then, a *definite σ-clause* (henceforth, simply *σ-clause*) of $\mathcal{S}\langle\mathcal{I}^\sigma, \mathcal{A}\rangle$ is a formula of the form

$$|h|_s \leftarrow |b_1|_{s_1}, \ldots, |b_n|_{s_n}$$

where $|h|_s$ is the *head* of the σ-clause, whose scope is s, while the conjunction of atomic σ-formulae $|b_1|_{s_1}, \ldots, |b_n|_{s_n}$ constitutes the *body* of the σ-clause. If $h \equiv p(\tilde{t})$, we say that p is the predicate symbol of the σ-clause $|h|_s \leftarrow |b_1|_{s_1}, \ldots, |b_n|_{s_n}$. □

Definition 3.3. (*σ-goal*) Let $|b_1|_{s_1}, \ldots, |b_n|_{s_n}, n \geq 0$, be atomic σ-formulae of the scope language $\mathcal{S}\langle\mathcal{I}^\sigma, \mathcal{A}\rangle$. Then, a *σ-goal* of $\mathcal{S}\langle\mathcal{I}^\sigma, \mathcal{A}\rangle$ is a formula of the form

$$\leftarrow |b_1|_{s_1}, \ldots, |b_n|_{s_n}$$

□

The notion of σ-theory follows accordingly.

Definition 3.4. (*σ-theory*) A *σ-theory* of $\mathcal{S}\langle\mathcal{I}^\sigma, \mathcal{A}\rangle$ is a set of σ-clauses of $\mathcal{S}\langle\mathcal{I}^\sigma, \mathcal{A}\rangle$. □

A *program* for a scope language is then a σ-theory. Any σ-clause of a program refers to a particular scope, by describing a property of the corresponding abstraction of the application domain. In particular, given a program $\mathcal{P}^\sigma$ of $\mathcal{S}\langle\mathcal{I}^\sigma, \mathcal{A}\rangle$, a scope $s \in \mathcal{I}^\sigma$ is represented by all the σ-clauses whose scope is s, according to Definition 3.2. More precisely, we define scope representation in terms of σ-sets as follows.

Definition 3.5. (*σ-set*) Given a program $\mathcal{P}^\sigma$ of a scope language $\mathcal{S}\langle\mathcal{I}^\sigma, \mathcal{A}\rangle$, and a scope identifier $s \in \mathcal{I}^\sigma$, we define $\sigma\text{-}set(s)$ as the *σ-set of* s, such that

$$\sigma\text{-}set(s) ::= \{|h|_s \leftarrow |b_1|_{s_1}, \ldots, |b_n|_{s_n} \in \mathcal{P}^\sigma\}$$

Moreover, we define $\sigma\text{-}set_p(s)$ as the *σ-set of s with respect to predicate p*, such that

$$\sigma\text{-}set_p(s) ::= \{|p(\tilde{t})|_s \leftarrow |b_1|_{s_1}, \ldots, |b_n|_{s_n} \in \mathcal{P}^\sigma\}$$

□

It is easy to see that $\sigma\text{-}set(s) \subseteq \mathcal{P}^\sigma$ is the σ-theory constituting the representation of scope s in the program $\mathcal{P}^\sigma$, while $\sigma\text{-}set_p(s) \subseteq \sigma\text{-}set(s)$ represents the projection of such a program subset with respect to predicate p.

3.2 The Mapping

In order to map a multi-theory logic language $\mathcal{L}$ onto a scope language, first we have to choose the set $\mathcal{I}^\sigma_\mathcal{L}$ of the scope identifiers. Since $\mathcal{A}_\mathcal{L}$ depends on $\mathcal{L}$, this choice automatically defines the corresponding scope language $\mathcal{S}\langle\mathcal{I}^\sigma_\mathcal{L}, \mathcal{A}_\mathcal{L}\rangle$.

Example 3.1. *Standard LP ($\mathcal{I}^\sigma$)* As a simple example, consider a Standard Logic Programming language, which we will henceforth denote as $\mathcal{L}_0$. Actually, we may re-interpret $\mathcal{L}_0$ as a trivial case of a multi-theory language where the only logic theory constituting the program captures the domain of discourse as a whole, such as a sort of global "world" abstraction.

Since the theory constituting the program represents the only "scope" that any formula can be associated with, only one symbol (say $I_\mathcal{P}$) is needed for $\mathcal{I}^\sigma_{\mathcal{L}_0}$ so that

$$\mathcal{I}^\sigma_{\mathcal{L}_0} ::= \{I_\mathcal{P}\} \tag{1}$$

Thus, if A_0 is the set of all the atomic formulae of $\mathcal{L}_0$, $\mathcal{A}_{\mathcal{L}_0} \equiv A_0$, and $\mathcal{S}\langle\{I_\mathcal{P}\}, A_0\rangle$ is the scope language for $\mathcal{L}_0$. □

Moreover, in any multi-theory language formulae are to be transposed in terms of σ-formulae of the corresponding scope language. Then, a suitable translation function has to be given, so that any formula of $\mathcal{L}$ can be mapped onto a suitably defined σ-formula of the corresponding scope language.

To this end, for each multi-theory language $\mathcal{L}$, we define the *σ-function* $\sigma_\mathcal{L}$ which is in charge of the semantics of the operators and constructs of the language, and gives meaning to any atomic formula, by associating to it the proper scope.

In particular, any multi-theory language presumably defines (meta-)operators making it possible to determine the truth value of a formula occurring in the body of a clause of a given theory with respect to another theory. In other terms, any multi-theory language presumably defines operators allowing the properties of a given abstraction of the application domain depend on another abstraction. Then, the syntax of such languages provides for new kinds of *goal formulae* (formulae in clause bodies), such as *extension goals* [9, 10], *module-prefixed* goals [18], context switch goals [17]. So, the σ-function is in charge of expressing the semantics of such operators, by defining how any goal formula of a multi-theory language $\mathcal{L}$ should be translated into the corresponding σ-formula of the scope language $\mathcal{S}\left\langle \mathcal{I}^\sigma_\mathcal{L}, \mathcal{A}_\mathcal{L} \right\rangle$.

The σ-function of a language $\mathcal{L}$, then, represents the bridge between a multi-theory language and the syntax of the corresponding scope language. $\sigma_\mathcal{L}$ takes a goal formula of $\mathcal{L}$ and a scope identifier as its arguments, and returns the σ-formula of $\mathcal{S}\left\langle \mathcal{I}^\sigma, \mathcal{A} \right\rangle$ corresponding to the given formula when taken in the given scope. For instance, the following definition for a conjunction of formulae trivially holds for the σ-function $\sigma_\mathcal{L}$ of any multi-theory language $\mathcal{L}$:

$$\sigma_\mathcal{L}(s, (a_1, \ldots, a_n)) ::= \sigma_\mathcal{L}(s, a_1), \ldots, \sigma_\mathcal{L}(s, a_n) \qquad (\sigma 1)$$

where $a_1, \ldots, a_n$ is a conjunction of atomic formulae, and $s \in \mathcal{I}^\sigma_\mathcal{L}$ is a scope identifier.

Example 3.2. *Standard LP (σ-function)* Take again the case of a Standard LP language $\mathcal{L}_0$. Here (see Example 3.1), $I_\mathcal{P}$ represents the only scope identifier, denoting then the only scope that any formula of $\mathcal{L}_0$ is implicitly associated with.

Trivially, no explicit scope denotation is needed in $\mathcal{L}_0$, nor is any connective necessary to associate scopes to formulae in the concrete syntax. Instead, any atomic formula $p(\tilde{t})$ in the body of a $I_\mathcal{P}$ clause is to be interpreted as the σ-formula $|p(\tilde{t})|_{I_\mathcal{P}}$.

This can be trivially expressed by the following definition of the σ-function $\sigma_{\mathcal{L}_0}$ for an atomic formula a.

$$\sigma_{\mathcal{L}_0}(I_\mathcal{P}, a) ::= |a|_{I_\mathcal{P}} \qquad (\sigma 2)$$

□

In addition, one should define the scope with which the formulae occurring in the starting goal have to be associated by default, so that the initial goal for the multi-theory language can be translated in suitable σ-formulae. Thus, given an initial goal G for a multi-theory language $\mathcal{L}$, the corresponding goal for the scope language $\mathcal{S}\left\langle \mathcal{I}^\sigma, \mathcal{A} \right\rangle$ is given by $\sigma_\mathcal{L}(\emptyset^\sigma_\mathcal{L}, G)$, where $\emptyset^\sigma_\mathcal{L}$ is the symbol denoting the *default scope* of the language $\mathcal{L}$.

Example 3.3. *Standard LP (Default scope)* In the case of the Standard LP language $\mathcal{L}_0$, the only scope is obviously also the default scope, that is

$$\emptyset^\sigma_{\mathcal{L}_0} ::= I_\mathcal{P} \qquad (2)$$

As a result, given an initial goal G of $\mathcal{L}_0$, the corresponding goal for $\mathcal{S}\left\langle \mathcal{I}^\sigma_{\mathcal{L}_0}, \mathcal{A}_{\mathcal{L}_0} \right\rangle$ is represented by $\sigma_{\mathcal{L}_0}(I_\mathcal{P}, G)$. Conceptually, this corresponds to saying that the truth value of any formula of $\mathcal{L}_0$ has to be set with respect to the program denoted by $I_\mathcal{P}$. □

Finally, programs of a multi-theory language have to be mapped onto programs of the corresponding scope language, by properly mapping clauses onto σ-clauses. What is needed, then, is way to compute the representation of any scope in terms of σ-clauses (its σ-set) given a program of a multi-theory language and a scope identifier.

Thus, for each language $\mathcal{L}$ we define a *σ-representation function* $\rho^\sigma_\mathcal{L}$, which takes a scope identifier $s \in \mathcal{I}^\sigma_\mathcal{L}$ and a program $\mathcal{P}$ of $\mathcal{L}$ and returning the σ-set of s. As a result,

$$\rho^\sigma_\mathcal{L}(\mathcal{P}, s) = \sigma\text{-}set(s) \tag{3}$$

holds by definition.

Then, given a program $\mathcal{P}$ of a multi-theory logic language $\mathcal{L}$, the corresponding $\mathcal{S}\left\langle \mathcal{I}^\sigma_\mathcal{L}, \mathcal{A}_\mathcal{L} \right\rangle$ program, which we denote as $\mathcal{P}^\sigma$ with an abuse of notation, can be given as the union of all the σ-set of the scopes of the program. In other terms,

$$\mathcal{P}^\sigma ::= \bigcup_{s \in \mathcal{I}^\sigma_\mathcal{L}} \rho^\sigma_\mathcal{L}(\mathcal{P}, s) \tag{4}$$

Example 3.4. *Standard LP (σ-representation function)* Let $\mathcal{P}$ be a program of the Standard LP language $\mathcal{L}_0$, denoted by the symbol $I_\mathcal{P} \in \mathcal{I}^\sigma_{\mathcal{L}_0}$. Then, we define $\rho^\sigma_{\mathcal{L}_0}$ such that

$$\rho^\sigma_{\mathcal{L}_0}(\mathcal{P}, I_\mathcal{P}) ::= \left\{ |h|_{I_\mathcal{P}} \leftarrow |b_1|_{I_\mathcal{P}}, \ldots, |b_n|_{I_\mathcal{P}} \;\middle|\; h \leftarrow b_1, \ldots, b_n \in \mathcal{P} \right\} \tag{$\rho 1$}$$

It is worth noting that, for any clause $h \leftarrow b_1, \ldots, b_n$ of $\mathcal{P}$, it holds that

$$|h|_{I_\mathcal{P}} \leftarrow |b_1|_{I_\mathcal{P}}, \ldots, |b_n|_{I_\mathcal{P}} = |h|_{I_\mathcal{P}} \leftarrow \sigma_{\mathcal{L}_0}(I_\mathcal{P}, b_1, \ldots, b_n) \tag{5}$$

showing the first example of the strong relationship between the σ-function and the σ-representation function of a multi-theory language. In particular, the σ-function is used by the σ-representation definition in order to map goal formulae (that is, formulae which can occur in a clause body) onto the corresponding σ-formulae, thus making explicit the scope with respect to which the truth value of any formula in a clause body has to be determined. More remarkable examples of this relationship can be found in Section 4.

Moreover, given that program $\mathcal{P}$ has the only scope $I_\mathcal{P}$, it obviously holds that

$$\mathcal{P}^\sigma = \sigma\text{-}set(I_\mathcal{P}) \tag{6}$$

□

In conclusion, a multi-theory language $\mathcal{L}$ can be easily mapped onto a scope language by specifying

- 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 $\rho^{\sigma}_{\mathcal{L}}$

3.3 The Operational Semantics

The operational behaviour of a scope language $\mathcal{S}\langle\mathcal{I}^{\sigma},\mathcal{A}\rangle$ can be given in terms of an inferential engine dealing with σ-formulae, σ-clauses, σ-theories and scope identifiers. In the following, we will exploit the typical operational characterisation used for LP languages, based on *sequents* and *Inference Rules* [9].

As one may expect, the resolvent of such an inferential engine simply consists of σ-formulae of the form $|a|_S$ instead of standard logic formulae. Thus, the following Inference Rules are (substantially) unchanged with respect to standard LP. There, $\mathcal{P}^{\sigma}$ denotes a program of $\mathcal{S}\langle\mathcal{I}^{\sigma},\mathcal{A}\rangle$, $|g_i|_{S_i}$ is an atomic σ-formula $\forall i, 1 \leq i \leq n$, $s \in \mathcal{I}^{\sigma}$ denotes a scope, θ and ϑ denote sets of equalities, while ε denotes the empty equality set.

- $$\frac{}{\mathcal{P}^{\sigma} \vdash_{\varepsilon} |true|_S} \qquad (i1)$$
- $$\frac{\mathcal{P}^{\sigma} \vdash_{\theta} |g_i|_{S_i} \qquad \mathcal{P}^{\sigma} \vdash_{\vartheta} |g_1|_{S_1},\ldots,|g_{i-1}|_{S_{i-1}},|g_{i+1}|_{S_{i+1}},\ldots,|g_n|_{S_n}}{\mathcal{P}^{\sigma} \vdash_{\theta\vartheta} |g_1|_{S_1},\ldots,|g_{i-1}|_{S_{i-1}},|g_i|_{S_i},|g_{i+1}|_{S_{i+1}},\ldots,|g_n|_{S_n}} \qquad (i2)$$

What has to be generalised in order to define the operational behaviour of a scope language $\mathcal{S}\langle\mathcal{I}^{\sigma},\mathcal{A}\rangle$ is the Inference Rule describing how an atomic σ-formula is handled when it is selected for resolution. Roughly, the intended behaviour can be described as follows: when a σ-formula $|p(\tilde{t})|_S$ is selected, it is bound to one (if any) of the σ-clauses of $\sigma\text{-}set_p(s)$, and new equalities are added to the computation state as the result of the unification of the atom of the σ-clause head with $p(\tilde{t})$. Finally, as intuitive, the σ-formulae contained in the body of the chosen σ-clause have to be added to the resolvent.

Correspondingly, the Inference Rule for the resolution of an atomic σ-formula $|p(\tilde{t})|_S$ can be expressed as follows, where all the symbols have the obvious expected meaning.

- $$\frac{\mathcal{P}^{\sigma} \vdash_{\vartheta} B\theta}{\mathcal{P}^{\sigma} \vdash_{\theta\vartheta} |p(\tilde{t})|_S} \qquad (|h|_S \leftarrow B \in \sigma\text{-}set_p(s)) \wedge (\theta = (h = p(\tilde{t}))) \qquad (i3)$$

where B is a conjunction $b_1,\ldots,b_n$ of atomic σ-formulae, while $h = p(\tilde{t})$ denotes the set of all the equalities resulting from the unification of the atomic formula $p(\tilde{t})$ with the head h of the matching clause.

The operational semantics for a scope language $\mathcal{S}\langle\mathcal{I}^{\sigma},\mathcal{A}\rangle$ can now be given by formalising the notions of proof and derivability of a σ-formula.

Definition 3.6. (*proof*) Given a program $\mathcal{P}^\sigma$ of $\mathcal{S}\langle\mathcal{I}^\sigma,\mathcal{A}\rangle$, let $s\in\mathcal{I}^\sigma$ be a scope identifier, G a σ-formula, and ϑ a set of equalities. Then, a *proof* for $\mathcal{P}^\sigma \vdash_{\vartheta} G$ is a tree whose nodes are labelled by sequents, and arcs are labelled by Inference Rules. In particular,

- the root is labelled by $\mathcal{P}^\sigma \vdash_{\vartheta} G$
- each internal node is derived from its parent according to one of the Inference Rules $i1$-$i3$.
- all the leaves are labelled by the empty sequent (that is, derived from the parent node using the Inference Rule ($i1$)).

□

Definition 3.7. (*derivability*) Given a program $\mathcal{P}^\sigma$ of $\mathcal{S}\langle\mathcal{I}^\sigma,\mathcal{A}\rangle$, let $s\in\mathcal{I}^\sigma$ be a scope identifier, G a σ-formula, and ϑ a set of equalities. Then, the formula G is *derivable* from $\mathcal{P}^\sigma$ with computed answer ϑ if and only if there exists a proof for $\mathcal{P}^\sigma \vdash_{\vartheta} G$. □

4 Some Examples

This Section presents some examples of how a multi-theory logic language can be described in terms of the corresponding scope language. Three cases will be discussed, taken from the literature: a logic language with modules and clause import mechanisms, Contextual Logic Programming [10], and an object-oriented logic language called Logic Objects [3]. Obviously, there will be no space here neither for a full explanation of the languages, nor for some enlightening examples, which would be quite useful for a good understanding. The original papers where the case studies have been taken from offer obviously the best introduction to the languages presented here, then there is no pretence to give a full account of them in this work.

4.1 Logic Programming with Modules

Let $\mathcal{L}_1$ denote a standard logic language extended with a module construct enabling to define a clause c as belonging to module m in the program $\mathcal{P}$, and let us denote such a relation with the notation $m :: c \in \mathcal{P}$. Moreover, suppose that some mechanisms for import/export of clauses are provided, as in widely used commercial systems like SICStus Prolog [18], so that a module m can import a clause c from another module n (m imports $n :: c \in \mathcal{P}$).

Like in such commercial systems, $\mathcal{L}_1$ allows constants of $\Sigma_{\mathcal{L}_1}$ (which we denote with $\mathcal{C}_{\mathcal{L}_1}$, with $\mathcal{C}_{\mathcal{L}_1} \subseteq \Sigma_{\mathcal{L}_1}$) to be used as module names, which, however, can be explicitly referred to in goal formulae also through variables of $\mathcal{V}_{\mathcal{L}_1}$. As a result,

$$\mathcal{I}^\sigma_{\mathcal{L}_1} ::= \mathcal{V}_{\mathcal{L}_1} \cup \mathcal{C}_{\mathcal{L}_1} \tag{7}$$

is the scope identifier set for the scope language $\mathcal{S}\left\langle \mathcal{I}^{\sigma}_{\mathcal{L}_1}, \mathcal{A}_{\mathcal{L}_1} \right\rangle$.

Non-standard formulae in such a language are limited to module-prefixed atomic formulae. A proper meta-operator (:) can be used in the body of a clause of a module, making the truth of an atomic formula of $\mathcal{L}_1$ depend on the clauses of another module. Instead, any formula which is not module-prefixed, should be intended as referred by default to the module where the formula occurs.

Correspondingly, the definition of the σ-function for $\mathcal{L}_1$ can be given as follows for atomic formulae

$$\sigma_{\mathcal{L}_1}(m, a) ::= |a|_m \tag{$\sigma3$}$$

and for module-prefixed formulae

$$\sigma_{\mathcal{L}_1}(m, n : g) ::= \sigma_{\mathcal{L}_1}(n, g) \tag{$\sigma4$}$$

where $a \in \mathcal{A}_{\mathcal{L}_1}$ is an atomic formula and g a goal formula of $\mathcal{L}_1$, and $m, n \in \mathcal{I}^{\sigma}_{\mathcal{L}_1}$ are module names.

Goals have to be proven with respect to a given module. A default module can be provided, where all formulae which are not explicitly associated with a module are to be proved. Following SICStus Prolog, $\mathcal{L}_1$ calls *user* this module, which plays therefore the role of the default scope of the language:

$$\emptyset^{\sigma}_{\mathcal{L}_1} ::= \mathit{user} \tag{8}$$

Thus, given an initial goal G, the corresponding $\mathcal{S}\left\langle \mathcal{I}^{\sigma}_{\mathcal{L}_1}, \mathcal{A}_{\mathcal{L}_1} \right\rangle$ goal is given by $\sigma_{\mathcal{L}_1}(\mathit{user}, G)$.

Given a program $\mathcal{P}$ of $\mathcal{L}_1$, the representation $\sigma\text{-}\mathit{set}(m)$ of the scope corresponding to module m is given by the σ-representation function as $\rho^{\sigma}_{\mathcal{L}_1}(\mathcal{P}, m)$. The definition of such a function takes into account the scope rules for the clauses of a module, as well as the rules for imported clauses. In particular, importing a clause c from a module n to a module m means that formulae matching the head of c can be proved in m depending on the truth value of the body goals in n. This is formally expressed through the following definition of the σ-representation function:

$$\rho^{\sigma}_{\mathcal{L}_1}(\mathcal{P}, m) ::= \begin{array}{c} \{\, |h|_m \leftarrow \sigma_{\mathcal{L}_1}(m, B) \mid m :: h \leftarrow B \ \in \mathcal{P}\} \\ \bigcup \\ \{\, |h|_m \leftarrow \sigma_{\mathcal{L}_1}(n, B) \mid m \text{ imports } n :: h \leftarrow B \ \in \mathcal{P}\} \end{array} \tag{$\rho2$}$$

where $\mathcal{P}$ is an $\mathcal{L}_1$ program, and $m \in \mathcal{I}^{\sigma}_{\mathcal{L}_1}$ is a module name.

4.2 Contextual Logic Programming

The $\mathcal{L}_2$ language discussed below is based on the Contextual Logic Programming paradigm (henceforth, CtxLP) proposed by [10], from which operators and concepts are taken. The underlying idea of CtxLP is that the knowledge to be used for the proof of a formula can be extended *ad hoc* with a new set of clauses.

A contextual program $\mathcal{P}$ is composed of *units*, collections of definite clauses denoted by a unique identifier belonging to the set $\mathcal{U}$ of the unit names. However, units are not used directly as scopes for formula demonstration. Instead, units are composed into *contexts*, which are sequences of units, through *extension formulae* of the form $u \gg a$, where $u \in \mathcal{U}$ and $a \in \mathcal{A}_{\mathcal{L}_2}$. The proof of an extension formula $u \gg a$ in the context c corresponds to the proof of the formula a in the context $u.c$ (where the operator . denotes unit composition), obtained by composing unit u with context c. Thus, contexts represent scopes in $\mathcal{L}_2$. As a result, while the language $\mathcal{L}_2$ deals with units and unit identifiers, the corresponding scope language uses contexts as scopes, and context identifiers for scope denotation.

The default scope is represented by the *empty context*, containing no clauses, denoted as λ, following [10].

$$\emptyset^{\sigma}_{\mathcal{L}_2} ::= \lambda \tag{9}$$

while the set of the scope identifiers is represented by the name of the empty context and by all the context names which can be obtained by composing unit names through the . operator:

$$\mathcal{I}^{\sigma}_{\mathcal{L}_2} ::= \lambda \cup \{u.c \mid u \in \mathcal{U} \text{ and } c \in \mathcal{I}^{\sigma}_{\mathcal{L}_2}\} \tag{10}$$

The definition of the σ-function deals again with atomic formulae

$$\sigma_{\mathcal{L}_2}(c, a) ::= |a|_c \tag{σ5}$$

and extension formulae

$$\sigma_{\mathcal{L}_2}(c, u \gg g) ::= \sigma_{\mathcal{L}_2}(u.c, g) \tag{σ6}$$

where $a \in \mathcal{A}_{\mathcal{L}_2}$ is an atomic formula and g a goal formula of $\mathcal{L}_2$, $u \in \mathcal{U}$ is a unit name, and $c \in \mathcal{I}^{\sigma}_{\mathcal{L}_2}$ is a context name.

Given a program $\mathcal{P}$, its translation $\mathcal{P}^{\sigma}$ in terms of the corresponding scope language can be obtained as the union of all the context representations through the σ-representation function $\rho^{\sigma}_{\mathcal{L}_2}$, according to Definition 4 on page 9. In order to give a formal definition of the σ-representation function, we need an additional concept, that is the notion of *subcontext* of a given context. Such a notion is defined recursively as a relation $\prec_{ctx}$ over the set of the context identifiers, where $c' \prec_{ctx} c$ is to be read as "c' is a subcontext of c", or, equivalently, as a boolean function

$$\prec_{ctx} : \mathcal{I}^{\sigma}_{\mathcal{L}_2} \times \mathcal{I}^{\sigma}_{\mathcal{L}_2} \rightarrow \mathit{boolean}$$

The subcontext relation can be defined recursively as follows:

$$c \prec_{ctx} u.c \tag{11}$$

$$c' \prec_{ctx} u.c \text{ if } c' \prec_{ctx} c \tag{12}$$

where $u \in \mathcal{U}$, and $c, c' \in \mathcal{I}^{\sigma}_{\mathcal{L}_2}$.

If we define the $\preceq_{ctx}$ relation such that, $\forall c, c' \in \mathcal{I}^{\sigma}_{\mathcal{L}_2}$, the following relations hold by definition

$$c \preceq_{ctx} c \tag{13}$$

$$c' \preceq_{ctx} c \text{ if } c' \prec_{ctx} c \tag{14}$$

the σ-representation function can be defined as follows:

$$\rho^{\sigma}_{\mathcal{L}_2}(\mathcal{P}, c) ::= \left\{ |h|_c \leftarrow \sigma_{\mathcal{L}_2}(u.c', B) \;\middle|\; c' \in \mathcal{I}^{\sigma}_{\mathcal{L}_2},\; u.c' \preceq_{ctx} c,\; u :: h \leftarrow B \;\in \mathcal{P} \right\} \tag{$\rho 3$}$$

for any context $c \in \mathcal{I}^{\sigma}_{\mathcal{L}_2}$.

This definition results in a very simple composition policy (pure extension) as well as in the *early* [1] binding policy, exploiting *static inheritance*.

A different binding policy, *late* binding, could be obtained through a different definition of the σ-representation function:

$$\rho^{\sigma}_{\mathcal{L}_2}(\mathcal{P}, c) ::= \left\{ |h|_c \leftarrow \sigma_{\mathcal{L}_2}(c, B) \;\middle|\; c' \in \mathcal{I}^{\sigma}_{\mathcal{L}_2},\; u.c' \preceq_{ctx} c,\; u :: h \leftarrow B \;\in \mathcal{P} \right\} \tag{$\rho 4$}$$

The two binding policies, however, could be used together only by adopting a more complex notion of scope, as well as an equivalent more sophisticated definition of scope identifier. Obviously, new binding operators would be needed too, to be handled appropriately by the σ-function. An example of a multi-theory language implementing both binding policies is presented in the following Subsection (see [15] for another example). Moreover, instead of pure extension, more sophisticated composition policies could be represented by re-formulating the σ-representation definition recursively (like in Subsection 4.3), in terms of union of σ-clauses, then by replacing the union operator with a more sophisticated composition operator (such as in [16]). There is no interest here, however, in further discussing these specific topics.

4.3 Logic Objects

Language $\mathcal{L}_3$ is based on the object-oriented logic language presented in [3], called Logic Objects. $\mathcal{L}_3$ are collections of *objects*, which are logic theories denoted by an extra-logic symbol. Inheritance mechanisms are represented through meta-level axioms of the form o isa o', defining an object hierarchy. A message passing formula of the form $o : f$ is re-interpreted as a request to prove a formula (the message sent, f) with respect to the clauses representing an object (the receiver, o). The default binding policy is early binding, implementing static inheritance, but dynamic inheritance can be achieved through the use of the *self* operator in message passing formulae (*self* : f).

If we denote with $O_{\mathcal{P}}$ the set of the extra-logic symbols used to denote objects, a scope identifier is a pair of object identifiers, one denoting the current object, another denoting the self object. As a result, we can define the set $\mathcal{I}^{\sigma}_{\mathcal{L}_3}$ of the scope identifiers of $\mathcal{L}_3$ as follows:

$$\mathcal{I}^{\sigma}_{\mathcal{L}_3} ::= \{ o_i \diamond o_j \mid o_i, o_j \in O_{\mathcal{P}}, o_i \preceq_{obj} o_j \} \tag{15}$$

where the relation $o \preceq_{obj} o'$ between two objects $o, o' \in O_{\mathcal{P}}$ is defined as follows:

$$o \preceq_{obj} o \tag{16}$$

$$o \preceq_{obj} o' \text{ if } o \text{ isa } o' \tag{17}$$

$$o \preceq_{obj} o'' \text{ if } o \text{ isa } o' \text{ and } o' \preceq_{obj} o'' \tag{18}$$

The σ-function of $\mathcal{L}_3$ should then take into account atomic formulae, message passing and self references. If a is an atomic formula and g a goal formula of $\mathcal{L}_3$, and $o, o_i, o_j \in O_{\mathcal{P}}, o_j \preceq_{obj} o_i$, then

- $\sigma_{\mathcal{L}_3}(o_i \diamond o_j, a) ::= |a|_{o_i \diamond o_j}$ $(\sigma 7)$
- $\sigma_{\mathcal{L}_3}(o_i \diamond o_j, o : g) ::= \sigma_{\mathcal{L}_3}(o \diamond o, g)$ $(\sigma 8)$
- $\sigma_{\mathcal{L}_3}(o_i \diamond o_j, self : g) ::= \sigma_{\mathcal{L}_3}(o_i \diamond o_i, g)$ $(\sigma 9)$

Moreover, the root of any $\mathcal{L}_3$ object hierarchy is represented by object *root*, whose identifier belongs by definition to the set $O_{\mathcal{P}}$ of any $\mathcal{L}_3$ program. Then, the default scope is obviously represented by *root*:

$$\emptyset^{\sigma}_{\mathcal{L}_3} ::= root \diamond root \tag{19}$$

Finally, given an $\mathcal{L}_3$ program $\mathcal{P}$, and the object identifiers $o_i, o_j, o_k \in O_{\mathcal{P}}$, such that o_j isa o_k, the σ-representation function can be defined recursively as follows.

- $\rho^{\sigma}_{\mathcal{L}_3}(\mathcal{P}, o_i \diamond root) ::= \emptyset$ $(\rho 5)$
- $\rho^{\sigma}_{\mathcal{L}_3}(\mathcal{P}, o_i \diamond o_j) ::= \begin{array}{c} \left\{ |h|_{o_i \diamond o_j} \leftarrow B \;\middle|\; |h|_{o_i \diamond o_k} \leftarrow B \;\in\; \rho^{\sigma}_{\mathcal{L}_3}(\mathcal{P}, o_i \diamond o_k) \right\} \\ \bigcup \\ \left\{ |h|_{o_i \diamond o_j} \leftarrow \sigma_{\mathcal{L}_3}(o_i \diamond o_j, B) \;\middle|\; o_j :: h \leftarrow B \;\in \mathcal{P} \right\} \end{array}$ $(\rho 6)$

5 Conclusions

This work introduces a new approach to the semantic characterisation of multi-theory logic languages based on the extension of Horn clauses. By adopting a transformational approach, any multi-theory language can be mapped through appropriate structures and functions onto a more general language, whose semantics is taken as known. Unlike other transformational approaches [11, 12], however, such a reference language does not rely over standard Horn clauses, since this would ultimately be inadequate, as shown in [6]. Instead, a new class of languages has been defined, embedding the idea that formulae in a multi-theory language always refer to a specific domain abstraction through the notion of σ-formula. Scope languages, then, can be used as a mean to express the semantics of multi-theory logic languages, and as a tool for the comparison of different languages of such a class. Some meaningful cases have been shown here, in order to give the general idea of how to perform the mapping. Different examples can be found in other papers [15, 14], together with examples of computations which this paper lacks.

However, several problems have not been faced here. First, this work is based on the assumption of the staticity of the logic program. Otherwise, the results of the σ-representation function, depending on the program as well as on the scope, would give different answers for the same scope in different moments of the same computation. In [15, 14], for instance, the representation of a given scope may not be known at a certain point of the computation, that is, given an element of the set of the scope identifiers, its σ-set is not known yet. This problem has been faced in [15], by introducing the notion of σ-computable formula, thus actually adding to the σ-representation function a boolean function bounding its domain of application.

Moreover, the problem of the effect of a given selection rule (such as the Prolog one) has not been discussed here. In particular, it has not been shown what happens, or what should happen, when a σ-formula is selected for resolution, but its scope is not completely defined, for instance because its identifier is an unbound variable. Typical results of failure or exception raise are unsatisfactory, since they undermine the completeness of the language. A more refined scheme, based on constraints, should be used to extend the definition of the scope machine, and has actually been exploited in [15, 14] for constraint-based object-oriented logic languages.

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, and Michela Milano devoted to read and comment early versions of this report was really precious to me.

References

[1] A. Brogi, E. Lamma, and P. Mello. A general framework for structuring logic

programs. Technical report, 4/1, CNR Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, 1990.

[2] A. Brogi, E. Lamma, and P. Mello. Compositional Model-theoretic Semantics for Logic Programs. *New Generation Computing*, 11(1):1–21, 1992.

[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] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular Logic Programming. *ACM TOPLAS*, 16(4):1361–1398, 1994.

[5] M. Bugliesi. A declarative view of inheritance in logic programming. In K. Apt, editor, *Proc. Joint Int. Conference and Symposium on Logic Programming*, pages 113–130. The MIT Press, 1992.

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

[7] L. Giordano, A. Martelli, and G.F. Rossi. Extending Horn Clause Logic with Modules Constructs. *Theoretical Computer Science*, 95:43–74, 1992.

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

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

[10] 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.

[11] L. Monteiro and A. Porto. A transformational view of inheritance in Logic Programming. In D.H.D. Warren and P. Szeredi, editors, *Proc. 7th Int. Conference on Logic Programming*, pages 481–494. The MIT Press, 1990.

[12] Y. Moscowitz and E. Shapiro. Lexical logic programs. In K. Furukawa, editor, *Proc. 8th Int. Conference on Logic Programming*, pages 349–363. The MIT Press, 1991.

[13] R. O'Keefe. Towards an algebra for constructing logic programs. In J. Cohen and J. Conery, editors, *Proceedings of IEEE Symposium on Logic Programming*, pages 152–160. IEEE Computer Society Press, 1985.

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

[15] A. Omicini. Constraining objects as logic theories. Technical Report DEIS-LIA-002-95, University of Bologna (Italy), 1995. LIA Series no. 2.

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

[17] A. Omicini and A. Natali. Object-oriented computations in logic programming. In M. Tokoro and R. Pareschi, editors, *Object-Oriented Programming*, volume 821 of *LNCS*, pages 194–212. Springer-Verlag, 1994. 8th European Conference on Object-Oriented Programming, ECOOP'94, Bologna, Italy, July 4–8, 1994.

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