Logic Program Schemas, Semi-Unification and Constraints

Eric Chasseur          Yves Deville

Université catholique de Louvain
Department of Computing Science and Engineering
Place Sainte-Barbe
1348 Louvain-la-Neuve
Belgium
{ec,yde}@info.ucl.ac.be

Abstract

The use of schemas is a classical way of synthesizing, transforming and analyzing logic programs. Operations on schemas are needed, in particular, the semi-unification of schemas with programs. Since the schemas used in the paper are second-order objects, the related semi-unification is the second-order semi-unification, which is decidable but NP-complete. The non-determinism implied by the NP-completeness slows down the search for a substitution.

The present paper expresses the semi-unification process over schemas as rewriting and reduction rules. Global and local constraints are associated to the schema to extend the expressivity of schema description and to speed up the search for a second-order substitution between programs and schemas. CLP techniques and notations are used.

1 Introduction

In logic programming, the use of program schemas is a very promising technique. In program synthesis, program schemas can formalize particular resolution methods (divide-and-conquer, generate-and-test approaches...), as investigated by Flener [2]. Program transformations can advantageously be performed on schemas rather than on their instances (i.e., programs). See Fuchs, Fromherz [4], Vasconcelos, Fuchs [18, 19], Flener [2]. In fact, the distinction between transformation and synthesis is not definitive, as said in Deville, Lau [1].

The schemas used in the paper are defined with second-order variables. Higher-order terms are normally difficult to deal with, since unification is undecidable [9] and there is no most general unifier. When using higher-order terms, one either accepts this and uses, for instance, the pre-unification procedure of Huet [11] which performs a systematic search for determining the existence of unifiers, or one restricts oneself to a subset of higher-order terms which is tractable. Higher-order patterns form such a subset of higher-order terms which was investigated among others by Nipkow [17]. Higher-order patterns unification is decidable and there exists a most general unifier of unifiable terms. Another alternative is to use decidable subcases of higher-order unification (for instance second-order matching [12]) if the problem permits it.

Properties of schemas, such as recursive or non-recursive parts, parameter form, length or position are added via constraints on schema place-holder instances. See Flener [2] again for examples.
**Related work**  Miller, Nadathur [15] present \texttt{\lambda Prolog} which is a higher-order extension of Prolog manipulating objects such as function and predicate variables, formulas and programs. \texttt{\lambda Prolog} is a logic programming language in the same way Prolog is. It is based on second-order unification. The *pre-unification procedure* of Huet is used to handle with second-order unification undecidability.

Kraan et al. [14] synthesize logic programs as a by-product of the planning of their verification proofs. This is achieved by using higher-order meta-variables at the proof planning level, which become instantiated in the course of planning. These higher-order variables can represent functions and predicates applied to bound variables. The formulas containing them are *higher-order patterns*.

Hannan, Miller [10] present source-to-source program transformers as meta-programs that manipulate programs as objects. They expose how simple transformers can be used to specify more sophisticated transformers. They use the *pre-unification algorithm* of Huet.

Gegg-Harrison [6] proposes a hierarchy of fourteen logic program schemas which are second-order logic expressions and generalize classes of programs in the most specific generalization (msg) sense. In [7], he defines logic program schemas with the help of \texttt{\lambda Prolog} to avoid using any meta-language. In [8], he extends these \texttt{\lambda Prolog} program schemas by applying standard programming techniques, introducing additional arguments and combining existing schemas.

Flener, Deville [3] show that some logic program generalization techniques can be pre-compiled at the program schema level so that the corresponding transformation can be fully automated. They also use *second-order matching* implicitly.

Flener [2] defines a logic algorithm schema as: \( \forall X_1 \ldots \forall X_n \; R(X_1 \ldots X_n) \Rightarrow F \). This is a second-order form of Kraan’s specification [14]: \( \forall \text{args}. \text{prog} \Rightarrow \text{spec} \). The author presents integrity constraints on instances of place-holders. In particular, he details constraints on the divide-and-conquer schema. For the instantiation of the latter to result in valid divide-and-conquer logic algorithms, constraints are expressed on the induction parameter, for example. [2] also is about schema-based transformation.

Huet, Lang [13] describe a program transformation method based on rewriting rules composed of second-order schemas. Fuchs, Frommer [4] and Vasconcelos, Fuchs [18, 19] present schema-based transformation formalisms and techniques. Implicit *second-order matching* is used in these papers [13, 4, 18, 19].

The formalism of schemas, as defined in Vasconcelos, Fuchs [18, 19], allows one schema to describe a class of logic programs in a suitable Horn-clause notation. Vasconcelos, Fuchs introduce features adding expressiveness to schemas: predicate and function symbol variables, possibly empty sequences of goals or terms. They also introduce constraints over schemas: argument positions, argument optionality, recursive or non-recursive predicates. In [18], constraints are part of schemas and take part of the expressiveness augmentation.

In the paper, the formalism describing schemas is a variant of that of Vasconcelos, Fuchs [18, 19]. But here constraints are separated from first- and second-order objects.

**Objectives**  Due to the extensive use of program schemas, efficient manipulation of schemas is needed, among which the semi-unification of schemas with programs. The objective of the paper is to provide a framework in which schemas, associated to constraints, are semi-unified with programs by means of CLP techniques.

The work is based on second-order semi-unification (or matching) of schemas with programs. Second-order semi-unification is decidable [12] but NP-complete [5].

Let \( S \) be a schema, and \( c \) be the initial constraints set associated to \( S \). Let \( P \) be the program with which \( S \) has to be semi-unified.
The starting pair \( S = P, c \) is transformed via successive rewriting rules to \( \emptyset, c' \). During the whole process, the successive versions of the constraints set remain consistent. At the end, there is a substitution \( \theta \in c' \) such that \( \theta \) satisfies \( c \) in \( S \) and \( S\theta = P \).

**Contributions** The main contributions of the paper are as follows:
- explicit integration of constraints in the program schema and in the semi-unification process,
- definition of an extensible first-order constraint language over schemas,
- expression of the semi-unification process over schemas as rewriting and reduction rules,
- two semantics are presented: one based on program instances, and the other on rewriting rules.

**Structure of the paper** Section 2 gives the syntax of schemas. It defines first- and second-order objects. Section 3 presents needed features of schemas and the first-order language of constraints. We make the distinction between global and local constraints. The fourth section gives the meaning of schemas relating to their instances (semantics 1). It also defines substitution pairs. Section 5 presents the general form and spirit of the rewriting rules. It also presents the rewriting semantics of schemas (semantics 2). Finally, we conclude in section 6 and give further research steps. Appendix A presents a subset of rewriting rules. Appendix B gives an example.

# 2 Syntax of schemas

A schema can be represented as a second-order object. Those used in this paper are second-order objects. They contain second-order and first-order objects. In order to simplify the presentation, the number of clauses in a schema will be fixed, although some interesting schemas have a variable number of clauses. The technical results can easily be extended to remove this restriction.

In our framework, no variable can represent an entire clause, but only atoms and sequences of atoms in a clause. This is a compromise between expressiveness and efficiency. We thus choose a subset of full second-order logic.

## 2.1 Basic components

Basic components of programs and of our schemas are first- and second-order objects. **First-order objects** are present in schemas and in programs. **Second-order objects** only appear in schemas.

**Definition 1** A first-order object (FObject) is either a term (term), in particular constant or variable, an atom (atom), a function symbol (fs), a predicate symbol (ps), a sequence of terms (seqterm) or a sequence of atoms (seqatom).

Constants are denoted by \( a, b, c, \ldots \), variables by \( X, Y, Z, \ldots \), function symbols by \( f, g, h, \ldots \) or particular symbols like \( \bullet \) (list function symbol) and predicate symbols by \( p, q, r, \ldots \).

**Definition 2** A second-order variable (SObject), also called place-holder, is either a term variable (VTerm), an atom variable (Vatom), a function symbol variable (Vfs), a predicate symbol variable (Vps), a sequence of terms variable (Vseqterm) or a sequence of atoms variable (Vseqatom).
In next sections, two other place-holders, \textbf{Vlength} (length variable) and \textbf{Vpos} (position variable), will be introduced and their meanings explained.

In the following, term variables are denoted by $X, Y, Z, \ldots$ (note the underscore), atom variables by $P_1, P_2, P_3, \ldots$, function symbol variables by $F, G, H, \ldots$, predicate symbol variables by $P, Q, R, \ldots$, sequence of terms variables by $\&, \&, \&\&$, and sequence of atoms variables by $G_1, G_2, G_3, \ldots$. \textbf{Vlength} are denoted by $L, L_1, L_2, L_3, \ldots$ and \textbf{Vpos} by $p, p_1, p_2, p_3, \ldots$

### 2.2 Grammar of schema

All place-holders are implicitly universally quantified. It means that all second-order variables are global to the schema. Thus there is a difference between schemas and programs in which first-order variables are local to clauses.

**Definition 3** A second-order schema is defined by the grammar:

- **Schema** ::= \textbf{SOrdCl} | \textbf{SOrdCl Schema}
- \textbf{SOrdCl} ::= \textbf{SOrdP} | \textbf{SO} \textbf{r}dP \textbf{—} \textbf{SOrdBody}
- \textbf{SOrdBody} ::= \textbf{SOrdP} | \textbf{SO} \textbf{r}dP, \textbf{SOrdBody}
- \textbf{SOrdP} ::= \textbf{Vseqatom} | \textbf{Vatom} | \textbf{Vps} | \textbf{Vps} (\textbf{SOrdArg}) | \textbf{atom} | \textbf{ps} (\textbf{SOrdArg})
- \textbf{SOrdArg} ::= \textbf{SOrdT} | \textbf{SOrdT}, \textbf{SOrdArg}
- \textbf{SOrdT} ::= \textbf{Vseqterm} | \textbf{Vterm} | \textbf{Vfs} | \textbf{Vfs} (\textbf{SOrdArg}) | \textbf{term} | \textbf{fs} (\textbf{SOrdArg})

All terminal symbols have been defined in Section 2.1. In the remaining of the paper, \textbf{SOrdP} is called second-order predicate and \textbf{SOrdT} second-order term. Characteristics of the syntax are described next.

In a program schema (and in a logic program), a comma separating two atoms has the same meaning as the logic and ($\&$) but constrains the order of atoms. A comma separating two terms is an argument separator which also constrains the order of the parameters.

In a schema, first-order objects may co-exist with second-order objects. This is viewed as a partially instantiated schema. For instance: in the one-clause schema $p(\_X) \leftarrow Q(Y)$, $p$ is a predicate symbol, $Q$ a predicate symbol variable, $Y$ a first-order term (a first-order variable) and $\_X$ a second-order term variable.

**Definition 4** A program is a schema without second-order variables. More precisely, we define \textbf{Program} (resp. \textbf{FOrdBody}, \textbf{FOrdCl}, \textbf{FOrdP} and \textbf{FOrdT}) as being \textbf{Schema} (resp. \textbf{SOrdBody}, \textbf{SOrdCl}, \textbf{SOrdP} and \textbf{SOrdT}) without second-order variables.

### 3 Constraints language

In our framework, a schema is not a classical second-order object. It needs incorporating features essential to the objectives of program representation and manipulation. The characteristics below should be present:

- Term positions among arguments of predicates and functions,
- Representation of possibly empty sequences of atoms,
- Representation of possibly empty sequences of terms,
- Argument length constraints,
- Instantiation form constraints (constant, ground, var, \ldots),
- Optional atoms and terms,
- Commutativity of predicate and function parameters.
Restrictions

- Most features are already present in [18, 19]. The constraints presented above are syntactical constraints. Flener [2] also defines integrity constraints on instances of place-holders. Such constraints seem to be necessary for schemas to instantiate into valid programs. However we do not consider such constraints in this paper.
- Commutativity of clauses and predicates in bodies of clauses is not considered here, since in most cases modifying their orders can lead to totally different computations.

Next, a first-order constraint language is defined. It is necessary and useful in the context of program schema and program synthesis/translation. Constraints are defined on schema place-holders. Some are global to the whole schema and others are local to occurrences of place-holders.

3.1 Global constraints

Global constraints handle forms and lengths of instances of second-order variables. In our framework, instantiation of second-order variables is also a global constraint.

- **Form constraints**
  Term variable instances can be constrained to be constant, variable or ground, and atom variable instances to be ground.

  Possible form constraint predicates are: constant($X$), ground($X$) and var($X$) for terms and ground($P_k$) for atoms.

- **Length constraints**
  Global constraints can also apply on the length of the instances of sequence of terms (resp. atoms) variables, i.e. on the number of terms (resp. atoms) in Vseqterm (resp. Vseqatom) instances. Length constraint predicates are: length_eq($X$, L), length_geq($X$, L) and length_leq($X$, L). Length constraints regroup hard constraints (length is compared to an integer) and soft constraints (length is compared to a variable). Variables constraining the lengths of sequence of terms and atoms variable instances are called Vlength variables and will have to be instantiated to integers.

- **Global constraint combinators**
  Form and length constraints can be linked by constraint combinators: $\land$ (logic and), $\lor$ (logic or) and $\neg$ (logic not).

  **Example 5** Term variable instance constrained to be either a constant or a variable: constant($X$) $\lor$ var($X$); two Vseqterm variables instances constrained to have equal lengths: length_eq($\&_1$, L) $\land$ length_eq($\&_2$, L).

3.2 Local constraints

Local constraints relate to positions of parameters, commutativity of groups of parameters, and locally empty place-holder instances.

- **Position constraint**
  The position constraint applies to second-order predicates and terms, except sequence variables (Vseqatom and Vseqterm). A position constraint is denoted by $\#$ followed by an integer (hard constraint) or a position variable (soft constraint) which will have to be instantiated to integers.
Example 6 According to the schema part \( P(\&_1, X \# p), Q(\&_2, Y \# p, \&_3) \), the instances of \( X \) and \( Y \) must have the same positions among the parameters of \( P \) and \( Q \) predicate instances.

- **Commutativity of parameters**
  Commutativity constraints are defined via unordered groups, denoted by \( \bigcirc(\ldots) \). This introduces the commutativity of predicate and function parameters in schemas. Inside unordered groups, only second-order terms may appear. In such groups, second-order terms order is not fixed. It is only at the instance level that the instances have fixed positions. Imbrications of unordered groups at the same level are not permitted. **Example:** \( P(\bigcirc( X, \bigcirc( X, Y )) ) \) is not permitted but \( Q(\bigcirc( X, F(\bigcirc( Y, Z )) )) ) \) is.

Example 7 According to the schema part \( P(\bigcirc( \&_1, X, \&_2 )) \), any instance of \( \bigcirc(\&_1, X, \&_2 \&_1 \&_2 \&_2) \) in \( P \) will be a permutation of instances of \( \&_1, X \) and \( \&_2 \).

- **Optional objects**
  Optional arguments and atoms are denoted by \( : \& X \). Option constraints apply to second-order predicates and terms.

**Example 8** According to the schema part \( P(\&_1, X \& X, \&_2) \), the instance of \( X \& X \) in \( P \) is either the instance of \( X \) itself or \( \emptyset \).

4 **Meaning of schemas**

4.1 **Substitutions**

In our approach, substitution components themselves are viewed as constraints on schema place-holders.

**Definition 9** A substitution pair (SP) is a pair \( S\text{Object}/ F\text{Object} \) of type: \( V\text{Term} / \text{Term} \), \( V\text{fs}/ fs \), \( V\text{seqterm}/ \text{SeqTerm} \), \( V\text{atom}/ \text{Atom} \), \( V\text{ps}/ ps \), \( V\text{seqatom}/ \text{SeqAtom} \), \( V\text{length}/ \text{Integer} \) or \( V\text{pos}/ \text{Integer} \). Since sequence of terms and atoms variables may instantiate to the empty sequence (denoted \( \emptyset \)), the pairs \( V\text{seqterm}/\emptyset, V\text{seqatom}/\emptyset \) are added.

**Definition 10** A substitution is a finite set of substitution pairs \( s_i/p_i \), i.e. \( \sigma = \{ s_i/p_i \mid 1 \leq i \leq n \} \) with \( \forall i,j \leq n \; s_i = s_j \Rightarrow i = j \).

**Example 11** \( V\text{Term} / \text{Term} : X/ \text{add}(X, Y, Z), V\text{seqterm}/ \text{SeqTerm} : \&_1/ f(X), g(Y), V\text{fs}/ fs : F/ \text{add}, V\text{seqatom}/ \text{SeqAtom} : G_1/ \text{Father}(X,Y), \text{husband}(X,Z), V\text{ps}/ ps : P/ \text{Father} \).

**Definition 12** The application of a substitution \( \sigma = \{ s_i/p_i \mid 1 \leq i \leq n \} \) to a schema \( S \), denoted \( S\sigma \), is obtained by the simultaneous replacement of occurrences of \( s_i \) by \( p_i \) in \( S \).

In order to simplify the presentation of the semantics, substitution pairs will also be considered as global constraints.
4.2 Satisfaction of constraints

Let us first precise when a substitution $\theta$ satisfies a global constraint. \texttt{constant}(X) is true iff $X\theta$ is a constant. \texttt{ground}(X) is true iff $X\theta$ is ground. \texttt{var}(X) is true iff $X\theta$ is a variable. \texttt{length\_eq}(X,L) (\texttt{length\_geq}(X,L), \texttt{length\_leq}(X,L)) is true iff $X\theta$ has length equal (greater than or equal, less than or equal) to $L$. This extends easily to constraint combiners.

The satisfaction of the local position constraint is now defined. Let $X\#p$ be a position constraint occurring in a subformula $F(\ldots, X\#p, \ldots)$ of a schema $S$. A substitution $\theta$ satisfies this position constraint in $S$ iff, in $S\theta$, the above subformula is instantiated to $f(e_1, \ldots, e_{k-1}, t\#k, e_{k+1}, \ldots, e_n)$ for some terms $e_1, \ldots, e_n$, and some predicate or function symbol $f$. The definitions of satisfaction for the other local constraints can be defined similarly.

Definition 13 Let $S$ be a schema and $\theta$ a substitution. $\theta$ satisfies the constraints of $S$ iff $\theta$ satisfies the global and the local constraints of $S$.

4.3 Schema instances

We are now in position to define the semantics of a schema whose associated constraints set is $c$. The semantics of the schema $S$, denoted $[S, c]_1$, is defined by means of all its possible program instances. We shall define a second semantics $[S, c]_2$ in Section 5.5.

Definition 14 $P$ is an instance of schema $S$, i.e. $P \in [S, c]_1$ iff \exists $\theta$ substitution $P \cong S, \theta$ 
$\land \theta$ satisfies $c$ in $S$.

$P \cong S, \theta$ means $P=S, \theta$ after syntactic constructs attached to schemas by the local constraints ($\otimes$( ), $\#$ and $\ll\gg$) are eliminated from $S, \theta$.

Example 15 schema $S : Q(\_X\#1, \ll Y \gg) ;$ program $P : q(X, Y)$. Then the substitution $\theta$ is : $\theta=$ \{ $Q/q, \_X\!/X, \_Y\!/Y$ \}. We have $P=q(X, Y) \simeq q( X\#1, \ll Y \gg)\cong S, \theta$.

5 Rewriting rules

5.1 Form of rewriting rules

Rewriting rules handle the semi-unification process as well as constraint satisfaction. During the process, global constraints can be deleted from and added to the constraint set $c$. Substitutions are also added to $c$.

Rewriting rules are of two different forms:

\[
\begin{align*}
\text{Condition of applicability} & \quad \langle Eq, c \rangle \longrightarrow \text{failure} \\
\text{Condition of applicability} & \quad \langle Eq, c \rangle \longrightarrow \langle Eq', c' \rangle
\end{align*}
\]

We also keep the invariant that, in a pair $\langle Eq, c \rangle$, $c$ is satisfiable in $Eq$. Otherwise, this leads to failure. We thus have the rewriting rule:

\[
\begin{align*}
\text{unsatisfiable}(Eq, \sigma) & \quad \langle Eq, \sigma \rangle \longrightarrow \text{failure}
\end{align*}
\]
The condition of applicability of each rewriting rule is implicitly completed by:

\[
\text{satisfiable}(Eq, \sigma) \quad \frac{}{\langle Eq, \sigma \rangle \longrightarrow \langle Eq', \sigma' \rangle}
\]

Finally, in the following, \([e_{q_1}, [e_{q_2}], \ldots, [e_{q_k}] \bullet Eq]\) means "the set of equations composed of \(Eq \cup \{e_{q_1}, e_{q_2}, \ldots, e_{q_k}\}\)."

Appendix A presents some rewriting rules.

### 5.2 Starting point

Let \(S\) be a schema to semi-unify with a program \(P\). The associated constraints set is \(c\). It is composed of global constraints and substitutions considered as global constraints in this framework.

The starting point is: \(\langle Eq, c \rangle\) with equation \(Eq \equiv S = P\).

### 5.3 Final point

The process can fail or succeed:

- failure: \(\langle S = P, c \rangle \longmapsto^* \text{failure}\)
- success: \(\langle S = P, c \rangle \longmapsto^* \langle \emptyset, c' \rangle\)

where \(\longmapsto^*\) is the transitive closure of \(\longmapsto\), the rewriting symbol.

### 5.4 Form of equations

An equation \(Eq\) appearing in a rewriting rule is an equation (or a set of equations) of type \(\alpha = \beta\) where:

- \(\alpha\) is a second-order expression (Schema, SOrdBody, SOrCl, SOrdP or SOrdT) and \(\beta\) is a first-order expression (Program, FOrdBody, FOrCl, FOrdP, FOrdT)
- or \(\alpha\) is a sequence of atoms variable (Vseqatom) and \(\beta\) is a sequence of second-order predicate without Vseqatom
- or \(\alpha\) is a sequence of terms variable (Vseqterm) and \(\beta\) is a sequence of second-order term without Vseqterm.

### 5.5 Rewriting semantics

Now we define the semantics of a schema according to the rewriting rules. The semantics of a schema \(S\), denoted \([S, c]_2\), is defined as follows.

**Definition 16** \(P\) is an instance of schema \(S\) whose associated constraints set is \(c\), i.e. \(P \in [S, c]_2\)

\[
\text{iff } \exists \text{ constraint set } c' : \\
\langle S = P, c \rangle \longmapsto^* \langle \emptyset, c' \rangle.
\]

Obviously, the semantics \([S, c]_1\) and \([S, c]_2\) should be equivalent. The formal proof will not be developed in this paper.
Conjecture 1 Let $S$ be a schema, $P$ a program.

if $\langle S = P , c \rangle \rightarrow^{*} \langle \emptyset , c' \rangle$
then let $\theta$ be the set of all $SP \in c'$ ($SP =$ substitution pair)

- $\theta$ is a substitution.

$\theta$ is called the complete substitution of $c'$.

Conjecture 2 Soundness of $[S, c]_2$ wrt. $[S, c]_1$:

if $\langle S = P , c \rangle \rightarrow^{*} \langle \emptyset , c' \rangle$
then let $\theta$ be the complete substitution of $c'$

- $S\theta \simeq P$,
- $\theta$ satisfies $c$ in $S$.

Conjecture 3 Completeness of $[S, c]_2$ wrt. $[S, c]_1$:

if $P \in [S, c]_1$
then $P \in [S, c]_2$.

6 Conclusion

The semi-unification process is known to be decidable but NP-complete. The non-determinism implied by NP-completeness slows down the search for substitutions. We have investigated the issue of semi-unifying second-order schemas and programs by means of CLP techniques. Global and local constraints have been defined on schema place-holders. The set of constraints is extensible. Constraints give expressiveness to schema descriptions. In the paper, we have distinguished constraints from schemas. Constraints are associated to schemas in the semi-unification process. In this framework, substitutions are also viewed as constraints. The semi-unification process has been expressed as rewriting and reduction rules.

Further research steps A next step of research is synthesis. Starting from a schema and a set of constraints on its schema place-holders, the objective is the construction of a program. The construction is guided by successive additions of constraints. The initial schema becomes more and more instantiated until the program level is reached. The constraints used to instantiate the successive schema versions come from heuristics, specifications and user demands.

This problem does not handle equations as defined in the paper, but only partially instantiated schemas (left-side of current equations). Let $S$ be the initial schema, $c$ the associated constraints set. We construct the program $P$ by means of a derivation:

$\langle S , c \rangle \longleftarrow \langle S_1 , c_1 \rangle \longleftarrow \langle S_2 , c_2 \rangle \longleftarrow \ldots \longleftarrow \langle P , c_i \rangle$.

At each step of this derivation, new constraints can be added to the resulting set of constraints $c_j$ ($0 < j < i$) to guide the synthesis process.

Acknowledgements

This research is supported by the subvention *Actions de recherche concertées* of the Direction générale de la Recherche Scientifique - Communauté Française de Belgique. We also acknowledge Pierre Flener for his comments on a very early version of the text. Finally, we acknowledge the reviewers for helping us to improve this paper.
References


A Rewriting rules examples

We only present here a subset of the rewriting rules. We focus on the most significant ones. The set of rewriting rules contains the classical rules for first-order unification [16].

A.1 Constraints

A.1.1 Commutativity of parameters rule (non-deterministic)

\[ X_1 \ldots X_n \in \text{SOrdT}, \quad X \text{ is one of the } n! \text{ permutations of } X_1 \ldots X_n, \quad t \in \text{seqterm} \]
A and B are sequences of SOrdT

\[ \langle [A, \bigodot(X_1 \ldots X_n), B = t] \cdot E q, \sigma \rangle \longrightarrow \langle [A, X, B = t] \cdot E q, \sigma \rangle \]

A.1.2 Length constraints rules

Hard length constraints rule Hard length constraint rewriting rules are presented for the length_eq predicate constraint. The first rule is about constraint on Vseqatom and the second, on Vseqterm.

\[ G_k \in \text{Vseqatom}, \quad i \in \mathbb{N} \]
\[ \langle E q, \sigma \cup \{\text{length}\_eq(G_k, i)\} \rangle \longrightarrow \langle [G_k = P_1, \ldots, P_i] \cdot E q\{G_k/P_1, \ldots, P_i\}, \sigma \rangle \]
where \( P_1, \ldots, P_i \) are brand-new Vatom variables

\[ \kappa_k \in \text{Vseqterm}, \quad i \in \mathbb{N} \]
\[ \langle E q, \sigma \cup \{\text{length}\_eq(\kappa_k, i)\} \rangle \longrightarrow \langle [\kappa_k = \mathcal{T}_1, \ldots, \mathcal{T}_i] \cdot E q\{\kappa_k/\mathcal{T}_1, \ldots, \mathcal{T}_i\}, \sigma \rangle \]
where \( \mathcal{T}_1, \ldots, \mathcal{T}_i \) are brand-new Vterm variables

Soft length constraints rule (non-deterministic) Soft length constraint rewriting rules are presented for the length_eq predicate constraint on Vseqatom. The Vseqterm case is identically expressed. The rule is non-deterministic because \( j \) can be chosen between 0 and \( n \) (if \( j = 0, G_k = \emptyset \)).

\[ G_k \in \text{Vseqatom}, \quad p_1 \ldots p_n \in \text{atom}, \quad L \in \text{Vlength}, \quad 0 \leq j \leq n \]
A and B are sequences of SOrdP

\[ \langle [A, G_k, B = p_1, \ldots, p_n] \cdot E q, \sigma \cup \{\text{length}\_eq(G_k, L)\} \rangle \]
\[ \longrightarrow \langle [G_k = P_1, \ldots, P_j], [A, P_1, \ldots, P_j, B = p_1, \ldots, p_n] \cdot E q\{G_k/P_1, \ldots, P_j\}, \sigma \{L/j\} \rangle \]
where \( P_1, \ldots, P_j \) are brand-new Vatom variables

A.1.3 Hard position constraint rules

\[ X \in \text{SOrdP}, \quad X_1, \ldots, X_j, Y_1, \ldots, Y_k \in \text{SOrdP}, \quad p_1 \ldots p_n \in \text{atom}, \quad 1 \leq i \leq n \]
\[ \langle [X_1, \ldots, X_j, X \neq i, Y_1, \ldots, Y_k = p_1, \ldots, p_n] \cdot E q, \sigma \rangle \]
\[ \longrightarrow \langle [X_1, \ldots, X_j = p_i, \ldots, p_{i-1}], [X = p_i], [Y_1, \ldots, Y_k = p_{i+1}, \ldots, p_n] \cdot E q, \sigma \rangle \]
X ∈ SOrdP, X1, ..., Xj, Y1, ..., Yk ∈ SOrdP, p1 ... pn ∈ atom, i > n or i ≤ 0

\( \langle [X_1, ..., X_j, X \# i, Y_1, ..., Y_k = p_1, ..., p_n] \bullet E q, \sigma \rangle \leftarrow \text{failure} \)

Similar rules are for SOrdT.

A.1.4 Optional objects rule (non-deterministic)

\( X \in \text{SOrdP (resp. SOrdT)}, \ x \in \text{seqatom (resp. seqterm)} \)
\( A \text{ and } B \text{ are sequences of SOrdP (resp. SOrdT)} \)

\[ \langle [A, \ll X \gg, B = x] \bullet E q, \sigma \rangle \leftarrow \langle [A, X, B = x] \bullet E q, \sigma \rangle \]

A.2 Others

A.2.1 Vseqatom rewriting rule (non-deterministic)

Here we expose the rewriting rules for the case of unconstrained Vseqatom.

\( G_k \in \text{Vseqatom}, p_1 ... p_n \in \text{atom}, 0 \leq j \leq n \)
\( A \text{ and } B \text{ are sequences of SOrdP} \)

\[ \langle [A, G_k, B = p_1, ..., p_n] \bullet E q, \sigma \rangle \]
\[ \leftarrow \langle [G_k = P_1, ..., P_j], [A, P_1, ..., P_j, B = p_1, ..., p_n] \bullet E q\{G_k / P_1, ..., P_j\}, \sigma \rangle \]

where \( P_1, ..., P_j \) are brand-new Vatom variables

A.2.2 A decomposition rule

Here is an example of a decomposition rule applying on terms. Another identical rule applies on atoms. Failure rules are also needed if the numbers of the left-side and right-side arguments are not the same.

\( T_1 \ldots T_n \in \text{SOrdT and } T_1 \text{ without option and position constraint} \)
\( t_1 \ldots t_m \in \text{term} \)

\[ \langle [T_1 \ldots T_n = t_1 \ldots t_m] \bullet E q, \sigma \rangle \]
\[ \leftarrow \langle [T_1 = t_1], [T_2 \ldots T_n = t_2 \ldots t_m] \bullet E q, \sigma \rangle \]

\( T_1 \ldots T_n \in \text{SOrdT and } T_1 \text{ without option and position constraint} \)

\[ \langle [T_1 \ldots T_n = \emptyset] \bullet E q, \sigma \rangle \leftarrow \text{failure} \]

\( t_1 \ldots t_m \in \text{term} \)

\[ \langle [\emptyset = t_1 \ldots t_m] \bullet E q, \sigma \rangle \leftarrow \text{failure} \]

A.2.3 Second-order substitution rule

\( X \in \text{SObject}, x \in \text{FObject} \cup \{ \emptyset \} \)

\[ \langle [X = x] \bullet E q, \sigma \rangle \leftarrow \langle E q\{X/x\}, \sigma \cup \{X/x\} \rangle \]

A.2.4 First-order checking rules

\( x \in \text{FObject} \cup \{ \emptyset \} \)

\[ \langle [x = x] \bullet E q, \sigma \rangle \leftarrow \langle E q, \sigma \rangle \]
\[
x_1, x_2 \in \text{Object} \cup \{ \emptyset \}, x_1 \neq x_2
\]
\[
\langle x_1 = x_2 \rangle \bullet Eq, \sigma \longrightarrow \text{failure}
\]

The semantics chosen in Sections 4.3 and 5.5 implies that all second-order variables are global to schemas. As a consequence, first-order variables, local to clauses of programs, must have the same name if their related second-order variables, global to the schema, have the same name. This can be handled by considering program variants.

**B Example**

Let the following schema:

Schema S:

\[
P([\_H | \_T], \_1) \Rightarrow G_2, P(\_T, \_3), G_3.
\]

whose associated constraints set is \(\sigma = \{ \text{length}_{eq}(\_1, L), \text{length}_{eq}(\_2, L), \text{length}_{eq}(\_3, L) \} \)

be semi-unified with the program:

Program P:

\[
islist([\_]).
\]

\[
islist([T | Q]) \Rightarrow islist(Q).
\]

As a first step, starting from \(\langle S = P, \sigma \rangle\), a previously undescribed rewriting rule derives the following set of equations:

\[
\langle \begin{align*}
P([\_], \_1) &= \text{islist}(\_1), \\
G_1 &= \emptyset, \\
P([\_H | \_T], \_2) &= \text{islist}([T | Q]), \\
G_2, P(\_T, \_3), G_3 &= \text{islist}(Q), \\
\sigma
\end{align*} \rangle
\]

For clarity, we will handle the four equations separately now:

- **First equation**: \(P([\_], \_1) = \text{islist}(\_1)\)

  \(\rightarrow\) by an undescribed rewriting rule instantiating predicate symbol variable \(P\) to predicate symbol \(islist\):

  \[
  \langle \begin{align*}
  [\_], \_1 &= [], \\
P/\text{islist}, \text{length}_{eq}(\_1, L), \text{length}_{eq}(\_2, L), \text{length}_{eq}(\_3, L) \rangle
  \end{align*} \rangle
  \]

  \(\rightarrow\) by the non-deterministic rule A.1.2 (soft length constraint rule):

  \[
  \langle \begin{align*}
  &\_1 = \emptyset, \\
  [], \emptyset &= [], \\
P/\text{islist}, \text{length}_{eq}(\_2, 0), \text{length}_{eq}(\_3, 0) \rangle
  \end{align*} \rangle
  \]

**Remark**: if another branch of this non-deterministic rule is followed, the process fails. For example:

\[
\langle \begin{align*}
&\_1 = T_1, T_1 = [], \\
P/\text{islist}, \text{length}_{eq}(\_2, 1), \text{length}_{eq}(\_3, 1)\rangle
\end{align*} \rangle
\]

\(\rightarrow\) by rule A.2.2 (decomposition rule):

\[
\langle \begin{align*}
&\_1 = T_1, T_1 = [], \\
P/\text{islist}, \text{length}_{eq}(\_2, 1), \text{length}_{eq}(\_3, 1)\rangle
\end{align*} \rangle
\]

\(\rightarrow\) by rule A.2.2 again: **failure** due to \(T_1 = \emptyset\). In the remaining of the example, we shall follow success branches only:

\(\rightarrow\) by rule A.2.3 (second-order substitution):

\[
\langle \begin{align*}
&\_1 = [], \\
P/\text{islist}, \text{length}_{eq}(\_2, 0), \text{length}_{eq}(\_3, 0), \_1/\emptyset \rangle
\end{align*} \rangle
\]

\(\rightarrow\) by rule A.2.2 (decomposition rule):

\[
\langle \begin{align*}
&\_1 = [], \\
\emptyset &= \emptyset, \\
P/\text{islist}, \text{length}_{eq}(\_2, 0), \text{length}_{eq}(\_3, 0), \_1/\emptyset \rangle
\end{align*} \rangle
\]

\(\rightarrow\) by rule A.2.4 (first-order checking): **success**

\[
\langle \emptyset, \{ P/\text{islist}, \text{length}_{eq}(\_2, 0), \text{length}_{eq}(\_3, 0), \_1/\emptyset \} \rangle
\]
- **Second equation**: $G_1 = \emptyset$
  
  → from the result of the first equation and by rule A.2.3 (second-order substitution): success
  
  \[
  \begin{cases}
  \emptyset, \\
  \{ \text{P/\text{islist, length}_{\text{eq}}(\&_2,0), length}_{\text{eq}}(\&_3,0), \&_1/\emptyset, G_1/\emptyset } \}
  \end{cases}
  \]

- **Third equation**: $P([\text{H} | \text{T }], \&_2) = \text{islist}([\text{T } | \text{Q }])$
  
  → from the result of the previous equations:
  
  \[
  \begin{cases}
  \text{islist([H | T ], \&_2 ) = islist([T | Q ]),} \\
  \{ \text{P/\text{islist, length}_{\text{eq}}(\&_2,0), length}_{\text{eq}}(\&_3,0), \&_1/\emptyset, G_1/\emptyset } \}
  \end{cases}
  \]
  
  → by an undescribed rewriting rule checking predicate symbols (\text{islist}):
  
  \[
  \begin{cases}
  \text{islist([H | T ], \&_2 ) = islist([T | Q ]),} \\
  \{ \text{P/\text{islist, length}_{\text{eq}}(\&_2,0), length}_{\text{eq}}(\&_3,0), \&_1/\emptyset, G_1/\emptyset } \}
  \end{cases}
  \]
  
  → by the rule A.1.2 (hard length constraint rule):
  
  \[
  \begin{cases}
  \&_2 = \emptyset, \\
  \text{islist([H | T ] = [T | Q ],} \\
  \{ \text{P/\text{islist, length}_{\text{eq}}(\&_3,0), \&_1/\emptyset, G_1/\emptyset } \}
  \end{cases}
  \]
  
  → by a variant of rule A.2.3 (second-order substitution): success
  
  \[
  \begin{cases}
  \emptyset, \{ \text{P/\text{islist, length}_{\text{eq}}(\&_3,0), \&_1/\emptyset, G_1/\emptyset, \&_2/\emptyset, \text{H}/\text{T }, \text{T }/\text{Q } } \}
  \end{cases}
  \]

- **Fourth equation**: $G_2, P(\text{H }, \&_3), G_3 = \text{islist}(\text{Q })$
  
  → from the result of the previous equations:
  
  \[
  \begin{cases}
  G_2 = \emptyset, \\
  G_3 = \emptyset, \\
  \text{islist}(\text{Q }, \&_3 ) = \text{islist}(\text{Q }), \\
  \{ \text{P/\text{islist, length}_{\text{eq}}(\&_3,0), \&_1/\emptyset, G_1/\emptyset, \&_2/\emptyset, \text{H}/\text{T }, \text{T }/\text{Q } } \}
  \end{cases}
  \]
  
  → by the non-deterministic rule A.2.1 (Vseqatom rewriting rule):
  
  \[
  \begin{cases}
  G_2 = \emptyset, \\
  G_3 = \emptyset, \\
  \text{islist}(\text{Q }, \&_3 ) = \text{islist}(\text{Q }), \\
  \{ \text{P/\text{islist, length}_{\text{eq}}(\&_3,0), \&_1/\emptyset, G_1/\emptyset, \&_2/\emptyset, \text{H}/\text{T }, \text{T }/\text{Q } } \}
  \end{cases}
  \]
  
  → by rule A.2.3 (second-order substitution):
  
  \[
  \begin{cases}
  \text{islist}(\text{Q }, \&_3 ) = \text{islist}(\text{Q }), \\
  \{ \text{P/\text{islist, length}_{\text{eq}}(\&_3,0), \&_1/\emptyset, G_1/\emptyset, \&_2/\emptyset, \text{H}/\text{T }, \text{T }/\text{Q }, G_2/\emptyset, G_3/\emptyset } \}
  \end{cases}
  \]
  
  → by an undescribed rule checking predicate symbols :
  
  \[
  \begin{cases}
  \text{Q }, \&_3 = \text{Q }, \\
  \{ \text{P/\text{islist, length}_{\text{eq}}(\&_3,0), \&_1/\emptyset, G_1/\emptyset, \&_2/\emptyset, \text{H}/\text{T }, \text{T }/\text{Q }, G_2/\emptyset, G_3/\emptyset } \}
  \end{cases}
  \]
  
  → by same rewriting rules as before: success
  
  \[
  \begin{cases}
  \emptyset, \{ \text{P/\text{islist, \&_1/\emptyset, \&_2/\emptyset, \&_3/\emptyset, \text{H}/\text{T }, \text{T }/\text{Q }, G_1/\emptyset, G_2/\emptyset, G_3/\emptyset } \}
  \end{cases}
  \]

At the end of the resolution we have: $\sigma = \{ \text{P/\text{islist, \&_1/\emptyset, \&_2/\emptyset, \&_3/\emptyset, \text{H}/\text{T }, \text{T }/\text{Q }, G_1/\emptyset, G_2/\emptyset, G_3/\emptyset } \}$. From this constraints set, we derive the complete substitution which is $\sigma$ itself. We have that $S \sigma \simeq P$.  

14