Logic Program Schemas, Constraints and Semi-Unification

Eric Chasseur and 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. Program schemas are known to be useful in different applications such as program synthesis, transformation, analysis, debugging, teaching ... This paper tackles two complementary aspects of program schemas. We first propose a language for the description of program schemas. It is based on a subset of second-order logic, enhanced with constraints and specific features of program schemas. One of the basic operations on schemas is the semi-unification of a schema with a program. We then express the semi-unification process over schemas as rewriting and reduction rules, using CLP techniques, where constraints are used to guide the semi-unification process.

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 [3]. Program *transformations* can advantageously be performed on *schemas* rather than on their instances (i.e. programs). See Fuchs, Fromherz [6], Vasconcelos, Fuchs [22, 23], Flener, Deville [4], Richardson, Fuchs [20], Büyükyıldız, Flener [1]. In fact, the distinction between transformation and synthesis is not definitive, as said in Deville, Lau [2].

In this introduction, we first describe the different ways of representing program schemas. We justify our choice of second-order objects for schema representation. Related works are then presented. We finally make precise the objectives and contributions of this paper.

1.1 Representing Schemas

A schema is an object representing a set of programs. Various representations can be used for the description of a schema. We do not aim at presenting here an exhaustive survey.

First-Order Representation. First-order representations of schemas (for example [1]) use first-order place-holders. In the following example (schema S):

Example 1. Schema S:

r(X,Y) $\leftarrow$ *minimal*(X),
solve(X,Y).
r(X,Y) $\leftarrow$ *nonminimal*(X),
decompose(X,Head,Tail),
r(Tail,Transf),
compose(Head,Transf,Compl),
Y=Compl.

the first-order template predicates of the schema S (in *italic* in the schema) must be replaced by instance (non-place holders) predicates. For example, we obtain the program P from S by replacing the corresponding place-holders by "reverse", "emptylist", "declist" and "addatend" (and switching some of their parameters):

Example 2. Program P:

reverse(X,Y) $\leftarrow$ emptylist(X),
emptylist(Y).
reverse(X,Y) $\leftarrow$ nonempty(X),
declist(X,H,T),
reverse(T,T2),
addatend(T2,H,Y2),
Y=Y2.

The main characteristics of this approach are the following :

1. There is no clear distinction between place-holder and non place-holder predicates.
2. It is not easy to introduce place-holders for constants (e.g. empty list). For this purpose, extra predicates could be added.
3. Considering our semi-unification objective, there is no easy formalization of the matching process between schemas and programs.

Other first-order representations are possible, such as the representation advocated by Flener, Lau, Ornaghi [5] where the concept of *open program* is used. In this framework, a schema can be seen as a program where some of its predicates are *open* (their implementation is not known, but they fulfill some given axiomatic properties).

Second-Order Representation. Higher-order terms are normally difficult to deal with, since higher-order unification is undecidable [11] and there is no most general unifier. To handle these difficulties we can either use "standard" or "specific" second-order logic.

Using "Standard" Second-Order Logic. One can accept the previous difficulties and use, for instance, the pre-unification procedure of Huet [13]. This procedure performs a systematic search for determining the existence of unifiers.

Using "standard" second-order logic has a major drawback: the lack of expressiveness if one generalizes too much, for instance, by allowing second-order variables to represent entire clauses.

Using "Specific" Second-Order Logic. One can also restrict 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 [19]. 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 without changing the language (for instance second-order matching [14]) if the problem permits it.

Using "specific" second-order logic does not necessarily mean using a subset of second-order logic. One can also *extend* second-order logic by introducing new features.

Our choice. In this paper, schemas are formalized using a second-order logic. The basis of the language is a subset of classical second-order logic. We also extend the language by first introducing specific features of program schemas, and by introducing constraints in the language.

The resulting language, described in the next sections, offers a powerful formalization of program schemas. The introduction of constraints enhances the expressiveness of the language and guides the semi-unification process.

In this language, the schema S of example 1 would be written as schema S':

Example 3. Schema S':

```
R(_E,Y) ← Min(_E),
          Solve(Y).
R(_N,Y) ← NMin(_N),
          Decompose(_N,_Head,_Tail),
          R(_Tail,_Tail2),
          Compose(_Tail2,_Head,_Res),
          Y=_Res.
```

The substitution θ, solution of $S'\theta = P$, is:
{ R/reverse, Solve/emptylist, _E/X, _N/X, Min/emptylist, NMin/nonempty, Decompose/declist, _Head/H, _Tail/T, _Tail2/T2, Compose/concat, _Res/Y2 }.
See Sects. 2 and 3 for details.

We now give a more elaborated example illustrating various aspects of such schemas.

Example 4.

$P([], \&_1) \leftarrow G_1.$
$P([\_H|\_T], \&_2) \leftarrow \ll Q(\_H) \gg, P(\_T, \&_3), G_2.$

In this schema, $\&_1$, $\&_2$ and $\&_3$ denote three sequences of terms, G_1 and G_2 denote an atom and the annotation $\ll\gg$ means that $Q(\_H)$ is optional. Possible global constraints on this schema are:

1. length_eq$(\&_1, L) \wedge$ length_eq$(\&_2, L) \wedge$ length_eq$(\&_3, L)$, which means that the instances of $\&_1$, $\&_2$ and $\&_3$ must have the same length,
2. variable$(_H) \wedge$ variable$(_T)$: $\_H$ and $\_T$ instances are first-order variables.

1.2 Related Work

Miller, Nadathur [17] present λProlog which is a higher-order extension of Prolog manipulating objects such as function and predicate variables, formulas and programs. Unification in λProlog is based on second-order unification. The *pre-unification procedure* of Huet is used to handle the second-order unification undecidability.

There is a major difference between λProlog and our schema language. Their goals are different. λProlog is a logic programming language in the same way Prolog is. It is aimed at computing some results from some program. Our purpose is not to execute program schemas, but to provide a powerful and expressive representation language for program schemas.

Kraan et al. [16] 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 [12] present source-to-source program transformers as meta-programs that manipulate programs as objects. They show how simple transformers can be used to specify more sophisticated transformers. They use the *pre-unification algorithm* of Huet.

Gegg-Harrison [8] 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 [9], he defines logic program schemas with the help of λProlog to avoid using any meta-language. In [10], he extends these λProlog program schemas by applying standard programming techniques, introducing additional arguments and combining existing schemas.

Flener, Deville [4] 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 [3] defines a logic algorithm schema as: $\forall X_1 ... \forall X_n \; R(X_1 ... X_n) \Leftrightarrow F$. This is a second-order form of Kraan's specification [16]: $\forall \overline{args}.prog(\overline{args}) \leftrightarrow$

$spec(\overline{args})$. The author presents semantic constraints on instances of placeholders. 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.

Huet, Lang [15] describe a program transformation method based on rewriting rules composed of second-order schemas. Fuchs, Fromherz [6] and Vasconcelos, Fuchs [22, 23] present schema-based transformation formalisms and techniques. Implicit *second-order matching* is used in these papers [15, 6, 22, 23].

The formalism of schemas, as defined in Vasconcelos, Fuchs [22, 23], 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 [22], 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 [22, 23]. But here constraints are separated from first- and second-order objects.

1.3 Objectives

Due to the extensive use of program schemas in various domains, there is a need for a powerful language for the description of schemas, and for operations (such as semi-unification) manipulating such schemas. The objective of the paper is first to provide a description language for schemas, where constraints are integrated, then to propose a semi-unification process over schemas.

Coming from Vasconcelos, Fuchs' work, this paper obviously stands in the field of program transformation, where semi-unification is useful to match second-order schemas and programs. That is the reason why we focus ourselves onto the semi-unification problem.

Let S be a schema, and c be the initial constraint set associated to S. Let P be the program with which S has to be semi-unified.

The starting pair $\langle S = P\ ,\ c\rangle$ is transformed via successive rewriting rules to $\langle \emptyset\ ,\ c'\rangle$. During the whole process, the successive versions of the constraint set remain consistent. At the end, there is a substitution $\theta \in c'$ such that θ satisfies c in S and $S\theta = P$.

1.4 Contributions

The main contributions of the paper are the following:

1. Definition of a description language for program schemas, where constraints are explicitly integrated. The introduction of constraints increases the expressive power of the language. It allows schemas to contain more knowledge. It also offers a guided search in the semi-unification process.
2. Definition of an extensible first-order constraint language over schemas,

3. Expression of the semi-unification process over schemas as rewriting and reduction rules,
4. Presentation of two semantics: one based on program instances, and the other on rewriting rules.

1.5 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 contains 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 of the semi-unification process. We thus choose a subset of full second-order logic.

2.1 Basic components

Basic components of programs and of schemas are first-order 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 ..., variables by X, Y, Z ..., function symbols by f, g, h ...or particular symbols like $\bullet$ (list function symbol) and predicate symbols by p, q, r ...

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, length variable (Vlength) and position variable (Vpos), will be introduced and their meanings explained.

In the following, term variables are denoted by $\_X$, $\_Y$, $\_Z$ ...(note the *underscore*), atom variables by P_1, P_2, P_3 ..., function symbol variables by F, G, H ..., predicate symbol variables by P, Q, R ..., sequence of terms variables by $\&_1$, $\&_2$, $\&_3$ ...and sequence of atoms variables by G_1, G_2, G_3 ...Vlength are denoted by L, L_1, L_2, L_3 ...and Vpos by p, p_1, p_2, p_3 ...

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	::= SOrdCl | SOrdCl Schema
SOrdCl	::= SOrdP | SOrdP $\leftarrow$ SOrdBody
SOrdBody	::= SOrdP | SOrdP, SOrdBody
SOrdP	::= Vseqatom | Vatom | Vps | Vps(SOrdArg) | atom | ps(SOrdArg)
SOrdArg	::= SOrdT | SOrdT, SOrdArg
SOrdT	::= Vseqterm | Vterm | Vfs | Vfs(SOrdArg) | term | fs(SOrdArg)

All terminal symbols have been defined in Section 2.1. In the remaining of the paper, SOrdP is called *second-order predicate* and 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* ($\wedge$) 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 Program (resp. FOrdBody, FOrdCl, FOrdP and FOrdT) as being Schema (resp. SOrdBody, SOrdCl, SOrdP and SOrdT) without second-order variables.

Programs are thus classical Horn clause programs (without extra logical features such as cuts). Schemas and programs do not contain negations in the body of clauses. This syntactical restriction can easily be removed without affecting the complexity of our results.

Example 5. Complete example.
Let the schema S be:

$$\boxed{\begin{array}{l} P([], \&_1) \leftarrow G_1. \\ P([\_H|\_T], \&_2) \leftarrow G_2,\ P(\_T, \&_3),\ G_3. \end{array}}$$

where

1. P is a predicate symbol variable (Vps)
2. $\&_1$, $\&_2$ and $\&_3$ are variables of sequence of terms (Vseqterm)
3. G_1, G_2 and G_3 are variables of sequence of atoms (Vseqatom)
4. $\_H$ and $\_T$ are term variables (Vterm)

Let the program $Prog$ be:

$$\boxed{\begin{array}{l} \text{sum}([\,],0). \\ \text{sum}([X|X_s], S) \leftarrow \text{sum}(X_s, SX_s),\ S \text{ is } X + SX_s. \end{array}}$$

S is semi-unified with $Prog$ ($S.\theta = Prog$) if θ={ P/sum, $\&_1/0$, $\&_2/S$, $\&_3/SX_s$, $\_H/X$, $\_T/X_s$, $G_1/\emptyset$, $G_2/\emptyset$, G_3/S is $X + SX_s$ }

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.

1. Term positions among arguments of predicates and functions,
2. Representation of possibly empty sequences of atoms,
3. Representation of possibly empty sequences of terms,
4. Argument length constraints,
5. Instantiation form constraints (constant, ground, var ...),
6. Optional atoms and terms,
7. Interchangeability of predicate and function parameters.

Most of these features are already present in [22, 23].

Restrictions.

1. Although the above characteristics are syntactical, semantic constraints on place-holders, such as defined in [3], could also be considered. Such constraints are useful to instantiate schemas into valid programs. However we do not consider such constraints in this paper. These could easily be included in the framework.
2. Interchangeability of clauses and predicates in bodies of clauses is not considered here. Such a reordering can be performed at the program level, as in the Mercury language [21].

In order to express such constraints on program schema, a first-order constraint language is now defined. It is necessary and useful in the context of program schema and program synthesis/transformation. Constraints are defined on schema place-holders. Some are global to the whole schema and others are local to occurrences of place-holders. Constraints on a schema will restrict the possible instantiations of the schemas to instances satisfying the constraints. The following set of constraints are extensible.

3.1 Global Constraints

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

1. *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.
2. *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 include *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.
3. *Global Constraint Combinators.* Form and length constraints can be linked by constraint combinators: $\wedge$ (logic and), $\vee$ (logic or) and $\neg$ (logic not).

   Example 6. Term variable instance constrained to be either a constant or a variable: constant($\_X$) $\vee$ var($\_X$) ; two Vseqterm variables instances constrained to have equal lengths: length_eq($\&_1$, L) $\wedge$ length_eq($\&_2$, L).

We choose an untyped second-order representation. Notice that types could be introduced at the constraint level (global constraints).

3.2 Local Constraints

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

1. *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 7. In the partial schema "$P(\&_1, \_X\#p)$, $Q(\&_2, \_Y, \&_3, \_Z\#p, \&_4)$", the instances of $\_X$ and $\_Z$ must have the same positions among the parameters of P and Q predicate instances.

2. *Interchangeability of Parameters.* Interchangeability constraints are defined via unordered groups, denoted by $\odot(\ldots)$. 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 parameters have fixed positions.

   Example 8. According to the schema part "$P(\odot(\&_1, \_X, \&_2))$", any instance of $\odot(\&_1, \_X, \&_2)$ in P will be a permutation of instances of $\&_1$, $\_X$ and $\&_2$.

3. *Optional Objects.* Optional arguments and atoms are denoted by: $\ll X \gg$. Option constraints apply to second-order predicates and terms.

   Example 9. According to the schema part "$P(\&_1, \ll \_X \gg, \&_2)$", the instance of $\ll \_X \gg$ in P is either the instance of $\_X$ itself or $\emptyset$.

Example 10. Complete example.
Over the following schema:

$$\boxed{\begin{array}{l} P([], \&_1) \leftarrow G_1. \\ P([\_H|\_T], \&_2) \leftarrow \ll Q(\_H) \gg, P(\_T, \&_3), G_2. \end{array}}$$

a local constraint (optional object) is defined: $\ll Q(\_H) \gg$. Other global constraints could be defined:

1. length_eq$(\&_1, L) \wedge$ length_eq$(\&_2, L) \wedge$ length_eq$(\&_3, L)$ which means that all $\&_i$ $(1 \leq i \leq 3)$ instances must be of same length,
2. variable$(_H) \wedge$ variable$(_T)$: $\_H$ and $\_T$ instances are first-order variables,
3. length_eq$(G_1, 0)$ which means that G_1 instance must be of length equal to 0.

We can easily extend the set of constraints by adding new constraint predicates. For example, we could use the following predicate to mean that G_2 does not contain predicate P: not_in(G_2, P).

From the examples of this section, one can easily see the expressiveness of the proposed constraint language. It allows the description of schemas to contain knowledge. It also allows a single schema to represent what would require several schemas using other representation schemes. The extensibility of the constraint part of the language is an advantage. One can add new constraints depending of the specific use of the schemas (program transformation, program synthesis $\ldots$).

4 Meaning of Schemas

In this section, we introduce the semantics of schemas. The semantics of program schemas can be defined in different ways. The first semantics defines the meaning of a schema as the set of all its possible instances which respect the constraints. This is close to the idea that a schema "represents" a class of programs. With this representational semantics of schemas, one can then choose any semantics at the program level. A second, constructive, semantics will be presented in Section 5.5.

Definition 5. A *schema* is a pair $\langle S, c \rangle$, where S is a program schema annotated with local constraints, and c is a set of global constraints.

4.1 Substitutions

Definition 6. A *substitution pair* (SP) is a pair SObject/FObject of type: Vterm/term, Vfs/fs, Vseqterm/seqterm, Vatom/atom, Vps/ps, Vseqatom/ seqatom, Vlength/integer or Vpos/integer. Since sequence of terms and atom variables may instantiate to the empty sequence (denoted $\emptyset$), the pairs Vseqterm/$\emptyset$ and Vseqatom/$\emptyset$ are allowed.

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

Definition 7. 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 the property that $\forall i, j \leq n : s_i = s_j \Rightarrow i = j$.

Example 11.
Vterm/term: $\_X/sum(X, Y, Z)$,
Vseqterm/seqterm: $\&_1/f(X), g(Y)$,
Vfs/fs: F/sum,
Vseqatom/seqatom: $G_1/father(X, Y), husband(X, Z)$,
Vps/ps: $P/father$.

Definition 8. 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 all occurrences of s_i by p_i in S.

4.2 Satisfaction of Constraints

Let us first make precise the concept of a substitution θ satisfying a global constraint c. We consider the different form of the constraint c :

1. *constant*($\_X$) is true iff $\_X\theta$ is a constant,
2. *ground*($\_X$) is true iff $\_X\theta$ is ground,
3. *var*($\_X$) is true iff $\_X\theta$ is a variable,

4. *length_eq*(X, L) (*length_geq*(X, L), *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 combinators.

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 θ 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 ($\_X$ is instantiated to some term t, and p to the integer k).

The definitions of satisfaction for the other local constraints can be defined similarly.

Definition 9. Let $\langle S, c\rangle$ be a schema and θ a substitution. θ *satisfies the constraints of* $\langle S, c\rangle$ iff θ satisfies c, and θ satisfies the local constraints in S.

4.3 Schema Instances

We are now in position to define the first semantics. The semantics of the schema $\langle S, c\rangle$, denoted by $[\![S, c]\!]_1$, is defined by means of all its possible program instances.

Definition 10. P is an *instance of schema* $\langle S, c\rangle$, denoted by $P \in [\![S, c]\!]_1$, iff there exists a substitution θ such that $P \simeq S\theta$, and θ satisfies the constraints of $\langle S, c\rangle$.

$P \simeq S\theta$ means $P = S\theta$ after elimination in $S\theta$ of the syntactic constructs attached to schemas by the local constraints ($\odot(\ )$, $\#$ and $\ll \gg$).

Example 12. Schema S: $Q(\_X\#1, \ll \_Y \gg)$ with $c = \emptyset$; program P: $q(X, Y)$. Then the substitution θ is: θ=$\{$ Q/q, $\_X/X$, $\_Y/Y$ $\}$. We have $P = q(X, Y) \simeq q(X\#1, \ll Y \gg) = S\theta$.

The semi-unification process is not deterministic. If $P \in [\![S, c]\!]_1$, there could exist different substitutions θ_1 and θ_2 such that $P \simeq S\theta_1$ and $P \simeq S\theta_2$.

Example 13.

Schema S: $P(\_X) \leftarrow G_1, Q(\_X), G_2$ with $c = \emptyset$
Program P: $p(X) \leftarrow q_1(X), q_2(X)$
The two substitutions θ_1=$\{$ P/p, $\_X/X$, $G_1/q_1(X)$, Q/q_2, $G_2/\emptyset$ $\}$ and θ_2=$\{$ P/p, $\_X/X$, $G_1/\emptyset$, Q/q_1, $G_2/q_2(X)$ $\}$ are such that $P \simeq S\theta_1$ and $P \simeq S\theta_2$.

5 Rewriting Rules

We present here a more constructive semantics, based on rewriting rules. This semantics will allow an implementation of a semi-unification algorithm.

5.1 Form of Equations

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

1. α and β are respectively second-order (Schema, SOrdBody, SOrCl, SOrdP, SOrdT) and first-order (Program, FOrdBody, FOrCl, FOrdP, FOrdT) expressions: for example, $P([\_H|\_T]), \&_2) = islist([T|Q])$,
2. *or* α is a [sequence of atoms] variable (Vseqatom) and β is a sequence of second-order predicates without Vseqatom: for example, $G_1 = P_1, P_2, P_3$,
3. *or* α is a [sequence of terms] variable (Vseqterm) and β is a sequence of second-order terms without Vseqterm: for example, $\&_1 = \_T_1, \_T_2, \_T_3$.

5.2 Starting Point

Let $\langle S, c\rangle$ be a schema to semi-unify with a program P. The associated constraint 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 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, considered as global constraints, are also added to c.

Rewriting rules are of two different forms:

$$\frac{\text{Applicability condition}}{\langle Eq\ ,\ c\rangle \longmapsto \textbf{failure}}$$

$$\frac{\text{Applicability condition}}{\langle Eq\ ,\ c\rangle \longmapsto \langle Eq'\ ,\ c'\rangle}$$

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:

$$\frac{\text{unsatisfiable}(Eq,\ c)}{\langle Eq\ ,\ c\rangle \longmapsto \textbf{failure}}$$

In addition to the applicability condition of each rewriting rule, it is assumed that the following invariant holds:

$$\frac{\text{satisfiable}(Eq,\ c)}{\langle Eq\ ,\ c\rangle \longmapsto \langle Eq'\ ,\ c'\rangle}$$

Finally, in the following, "$[eq_1]$, $[eq_2]$ ... $[eq_i] \bullet Eq$" means "the set of equations composed of $Eq \cup \{\ eq_1,\ eq_2 \ldots\ eq_i\ \}$".

Appendix A presents some representative rewriting rules.

5.4 Final Point

The process can fail or succeed:

1. failure: $\langle S = P\ ,\ c\rangle \longmapsto^*$ **failure**
2. success: $\langle S = P\ ,\ c\rangle \longmapsto^* \langle \emptyset\ ,\ c'\rangle$

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

5.5 Rewriting Semantics

Now we define the semantics of a schema according to the rewriting rules. The semantics of a schema $\langle S, c\rangle$, denoted by $[\![S, c]\!]_2$, is defined as follows.

Definition 11. P is an *instance of schema* $\langle S, c\rangle$, denoted by $P \in [\![S, c]\!]_2$, iff there exists a constraint set c' such that $\langle S = P\ ,\ c\rangle \longmapsto^* \langle \emptyset\ ,\ c'\rangle$.

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

Definition 12. Existence of substitution:
if $\langle S = P\ ,\ c\rangle \longmapsto^* \langle \emptyset\ ,\ c'\rangle$
and θ is the set of all substitution pairs of c'
then θ is a substitution.
θ is called *the complete substitution* of c'.

Conjecture 13. *Soundness of* $[\![S,\ c]\!]_2$ *wrt.* $[\![S,\ c]\!]_1$*:*
if $\langle S = P\ ,\ c\rangle \longmapsto^* \langle \emptyset\ ,\ c'\rangle$
and θ is *the complete substitution of* c'
then $S\theta \simeq P$ *and* θ *satisfies* $\langle S, c\rangle$.

Conjecture 14. *Completeness of* $[\![S,\ c]\!]_2$ *wrt.* $[\![S,\ c]\!]_1$*:*
if $P \in [\![S,\ c]\!]_1$
then $P \in [\![S,\ c]\!]_2$.

As the semi-unification process is not deterministic, some of the rewriting rules are non-deterministic (see Appendix A). In general, the semi-unification process in second-order logic is known to be decidable, but NP-complete [7]. Although our language is a subset of second-order logic (with some extensions), the potential exponential complexity is still present. However, the active use of constraints in the semi-unification offers a more efficient search for a correct solution.

6 Conclusion

In this paper, we proposed a language for the description of program schemas. This language is based on a subset of second-order logic enhanced with constraints and specific features of program schemas. The constraint language is extensible and permits the possible introduction of domain knowledge for the schemas. We then expressed the semi-unification process over schemas as rewriting and reduction rules, using CLP techniques, where constraints are used to guide the semi-unification process. Appendix A shows some of these rewriting and reduction rules. Two semantics were also presented. The first semantics is based on program instances and the second on the rewriting rules.

Further Research Steps. A first step will be the theoretical and practical analysis of the complexity of the semi-unification algorithm. We are also interested in the use of schemas in program synthesis. Starting from a schema and a set of constraints on its schema place-holders, the objective is the synthesis of a program. The synthesis 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 will require the extension of syntactical constraints to semantic constraints.

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 constraint set. We construct the program P by means of a derivation:

$\langle S\ ,\ c\rangle \longmapsto \langle S_1\ ,\ c_1\rangle \longmapsto \langle S_2\ ,\ c_2\rangle \longmapsto \ldots \longmapsto \langle P\ ,\ c_n\rangle$.

At each step i of this derivation, new constraints can be added to the resulting set of constraints c_i $(0 < i < n)$ 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 the reviewers for helping us to improve this paper. Special thanks to Pierre Flener for our fruitful and constructive discussions.

References

1. H. Büyükyıldız and P. Flener, *Generalized logic program transformation schemas*, In: N.E. Fuchs (ed.), Proc. of LOPSTR'97 (this volume)
2. Y. Deville and K.-K. Lau, *Logic program synthesis: A survey*, Journal of Logic Programming, 19-20:321-350, May/July 1994
3. P. Flener, *Logic Program Synthesis From Incomplete Information*, Kluwer Academic Publishers, 1995

4. P. Flener and Y. Deville, *Logic program transformation through generalization schemata*, In: M. Proietti (ed.), Proc. of LOPSTR'95, Springer-Verlag, 1996
5. P. Flener, K.-K. Lau and M. Ornaghi, *On Correct Program Schemas*, In: N.E. Fuchs (ed.), Proc. of LOPSTR'97 (this volume)
6. N.E. Fuchs and M.P.J. Fromherz, *Schema-Based Transformations of Logic Programs*, In: T.P. Clement, K.-K. Lau (eds.), Proc. of LOPSTR'91, Springer-Verlag, 1992
7. M.R. Garey and D.S. Johnson, *Computers and Intractability. A Guide to the Theory of NP-completeness*, W.H. Freeman and Company, 1979
8. T.S. Gegg-Harrison, *Learning Prolog in a Schema-Based Environment*, Instructional Science, 20:173-192, 1991
9. T.S. Gegg-Harrison, *Representing Logic Program Schemata in λProlog*, In: L. Sterling (ed.), Proc. of the 12th International Conference on Logic Programming, Japan, pp. 467-481, The MIT Press, 1995
10. T.S. Gegg-Harrison, *Extensible Logic Program Schemata*, In: J. Gallagher (ed.), Proc. of the 6th International Workshop on Logic Program Synthesis and Transformation, Stockholm, Sweden, pp. 256-274, Springer-Verlag, 1996
11. W.D. Goldfarb, *The Undecidability of the second-order unification problem*, Theoretical Computer Science, 13:225-230, 1981
12. J. Hannan and D. Miller, *Uses Of Higher-Order Unification For Implementing Program Transformers*, In: A. Kowalski, K.A. Bowen (eds.), Proc. of ICLP'88, The MIT Press, 1988
13. G. Huet, *A unification algorithm for lambda calculus*, Theoretical Computer Science, 1:27-57, 1975
14. G. Huet, *Résolution d'Équations dans les langages d'ordre 1, 2... ω*, PhD thesis, Université Paris VII, 1976
15. G. Huet and B. Lang, *Proving and Applying Program Transformations Expressed with Second-Order Patterns*, Acta Informatica 11 (1978), 31-55
16. I. Kraan, D. Basin and A. Bundy, *Middle-Out Reasoning for Logic Program Synthesis*, In: D.S. Warren (ed.), Proc. of ICLP'93, The MIT Press, 1993
17. D. Miller and G. Nadathur, *A logic programming approach to manipulating formulas and programs*, Proc. of the IEEE Fourth Symposium on Logic Programming, IEEE Press, 1987
18. A. Martelli and U. Montanari, *An Efficient Unification Algorithm*, ACM Transactions on Programming Languages and Systems, Vol. 4, No. 2, April 1982, pp.258-282
19. T. Nipkow, *Higher-order critical pairs*, In: Proc. 6th IEEE Symp. Logic in Computer Science, pp. 342-349, 1991
20. J. Richardson and N.E. Fuchs, *Development of correct transformation schemata for Prolog programs*, In: N.E. Fuchs (e.), Proc. of LOPSTR'97 (this volume).
21. Z. Somogyi, F. Henderson and T. Conway, *The execution algorithm of Mercury: an efficient purely declarative logic programming language*, Journal of Logic Programming, 29(1-3):17-64, October-December 1996
22. W.W. Vasconcelos and N.E. Fuchs, *Enhanced Schema-Based Transformations for Logic Programs and their Opportunistic Usage in Program Analysis and Optimisation*, technical report, Institut für Informatik, Universität Zürich, 1995
23. W.W. Vasconcelos and N.E. Fuchs, *An Opportunistic Approach for Logic Program Analysis and Optimisation Using Enhanced Schema-Based Transformations*, In: M. Proietti (ed.), Proc. of LOPSTR'95, Springer-Verlag, 1996

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 [18].

A.1 Constraints

Interchangeability of Parameters Rule (Non-Deterministic).

$X_1 \dots X_n \in$ SOrdT, $\overline{X}$ is one of the $n!$ permutations of $X_1 \dots X_n$, $t \in$ seqterm, A and B are sequences of SOrdT

$$\langle [A, \odot(X_1 \dots X_n), B = t] \bullet Eq \ , \ \sigma \rangle \longmapsto \langle [A, \overline{X}, B = t] \bullet Eq \ , \ \sigma \rangle$$

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.

$G_k \in$ Vseqatom, $i \in IN$

$$\langle Eq \ , \ \sigma \cup \{length\_eq(G_k, i)\} \rangle \longmapsto \langle [G_k = P_1 \dots P_i] \bullet Eq\{G_k / P_1 \dots P_i\} \ , \ \sigma \rangle$$

where $P_1 \dots P_i$ are brand-new Vatom variables

$\&_k \in$ Vseqterm, $i \in IN$

$$\langle Eq \ , \ \sigma \cup \{length\_eq(\&_k, i)\} \rangle \longmapsto \langle [\&_k = \_T_1 \dots \_T_i] \bullet Eq\{\&_k / \_T_1 \dots \_T_i\} \ , \ \sigma \rangle$$

where $\_T_1 \dots \_T_i$ 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 similarly expressed. The rule is non-deterministic because j can be chosen between 0 and n (if $j = 0$, $G_k = \emptyset$).

$G_k \in$ Vseqatom, $p_1 \dots p_n \in$ atom, $L \in$ Vlength, $0 \leq j \leq n$, A and B are sequences of SOrdP

$$\langle [A, G_k, B = p_1 \dots p_n] \bullet Eq \ , \ \sigma \cup \{length\_eq(G_k, L)\} \rangle$$
$$\longmapsto \langle [G_k = P_1 \dots P_j], [A, P_1 \dots P_j, B = p_1 \dots p_n] \bullet Eq\{G_k / P_1 \dots P_j\} \ , \ \sigma\{L/j\} \rangle$$

where $P_1 \dots P_j$ are brand-new Vatom variables

Hard Position Constraint Rules.

X $\in$ SOrdP, $X_1 \dots X_j, Y_1 \dots Y_k \in$ SOrdP, $p_1 \dots p_n \in$ atom, $1 \leq i \leq n$

$$\langle [X_1 \dots X_j, X\#i, Y_1 \dots Y_k = p_1 \dots p_n] \bullet Eq \ , \ \sigma \rangle$$
$$\longmapsto \langle [X_1 \dots X_j = p_1 \dots p_{i-1}], [X = p_i], [Y_1 \dots Y_k = p_{i+1} \dots p_n] \bullet Eq \ , \ \sigma \rangle$$

$$\frac{X \in \text{SOrdP},\ X_1 \ldots X_j, Y_1 \ldots Y_k \in \text{SOrdP},\ p_1 \ldots p_n \in \text{atom},\ i \leq 0 \text{ or } i > n}{\langle [X_1 \ldots X_j, X\#i, Y_1 \ldots Y_k = p_1 \ldots p_n] \bullet Eq\ ,\ \sigma \rangle \longmapsto \textbf{failure}}$$

Similar rules are for SOrdT.

Optional Objects Rule (Non-Deterministic).

$$\frac{X \in \text{SOrdP (resp. SOrdT)},\ x \in \text{seqatom (resp. seqterm)},\ A \text{ and } B \text{ are sequences of SOrdP (resp. SOrdT)}}{\langle [A, \ll X \gg, B = x] \bullet Eq\ ,\ \sigma \rangle \longmapsto \langle [A, X, B = x] \bullet Eq\ ,\ \sigma \rangle}$$

$$\frac{X \in \text{SOrdP (resp. SOrdT)},\ x \in \text{seqatom (resp. seqterm)},\ A \text{ and } B \text{ are sequences of SOrdP (resp. SOrdT)}}{\langle [A, \ll X \gg, B = x] \bullet Eq\ ,\ \sigma \rangle \longmapsto \langle [A, B = x] \bullet Eq\ ,\ \sigma \rangle}$$

A.2 Others

Vseqatom Rewriting Rule (Non-Deterministic).
Here we show the rewriting rules for the case of unconstrained Vseqatom.

$$\frac{G_k \in \text{Vseqatom}, p_1 \ldots p_n \in \text{atom}, 0 \leq j \leq n,\ A \text{ and } B \text{ are sequences of SOrdP}}{\begin{array}{l}\langle [A, G_k, B = p_1 \ldots p_n] \bullet Eq\ ,\ \sigma \rangle \\ \longmapsto \langle [G_k = P_1 \ldots P_j], [A, P_1 \ldots P_j, B = p_1 \ldots p_n] \bullet Eq\{G_k / P_1 \ldots P_j\}\ ,\ \sigma \rangle \\ \text{where } P_1 \ldots P_j \text{ are brand-new Vatom variables}\end{array}}$$

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

$$\frac{T_1 \ldots T_n \in \text{SOrdT and } T_1 \text{ without option and position constraint},\ t_1 \ldots t_m \in \text{term},}{\begin{array}{l}\langle [T_1 \ldots T_n = t_1 \ldots t_m] \bullet Eq\ ,\ \sigma \rangle \\ \longmapsto \langle [T_1 = t_1], [T_2 \ldots T_n = t_2 \ldots t_m] \bullet Eq\ ,\ \sigma \rangle\end{array}}$$

$$\frac{T_1 \ldots T_n \in \text{SOrdT and } T_1 \text{ without option and position constraint}}{\langle [T_1 \ldots T_n = \emptyset] \bullet Eq\ ,\ \sigma \rangle \longmapsto \textbf{failure}}$$

$$\frac{t_1 \ldots t_m \in \text{term}}{\langle [\emptyset = t_1 \ldots t_m] \bullet Eq\ ,\ \sigma \rangle \longmapsto \textbf{failure}}$$

Second-Order Substitution Rule.

$$\frac{X \in \text{SObject},\ x \in \text{FObject} \cup \{\ \emptyset\ \}}{\langle [X = x] \bullet Eq\ ,\ \sigma \rangle \longmapsto \langle Eq\{X/x\}\ ,\ \sigma \cup \{X/x\} \rangle}$$

First-Order Checking Rules.

$$\frac{x \in \text{FObject} \cup \{\ \emptyset\ \}}{\langle [x = x] \bullet Eq\ ,\ \sigma\rangle \longmapsto \langle Eq\ ,\ \sigma\rangle}$$

$$\frac{x_1,\ x_2 \in \text{FObject} \cup \{\ \emptyset\ \},\ \ x_1 \neq x_2}{\langle [x_1 = x_2] \bullet Eq\ ,\ \sigma\rangle \longmapsto \textbf{failure}}$$

B Working Example

Let the following schema $\langle S, c\rangle$, with

$$\boxed{\begin{array}{l}\text{Schema } S\text{: } P([\,], \&_1) \leftarrow G_1.\\ \qquad\qquad\quad P([\_H|\_T], \&_2) \leftarrow G_2,\ P(\_T, \&_3),\ G_3.\end{array}}$$

and global constraints

$$\boxed{c\text{=}\{\ \text{length\_eq}(\&_1, L),\ \text{length\_eq}(\&_2, L),\ \text{length\_eq}(\&_3, L)\ \}}$$

be semi-unified with the program P:

$$\boxed{\begin{array}{l}\text{Program } P\text{: islist}([\,]).\\ \qquad\qquad\quad \text{islist}([\ T|Q\ ]) \leftarrow \text{islist}(Q).\end{array}}$$

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

$\langle\ P([\,], \&_1) = \text{islist}([\,]),$
$\quad G_1 = \emptyset,$
$\quad P([\_H|\_T], \&_2) = \text{islist}([\ T|Q\ ]),$
$\quad G_2,\ P(\_T, \&_3),\ G_3 = \text{islist}(Q),$
$\quad c$
$\rangle$

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

1. **First equation**: $P([\,], \&_1) = \text{islist}([\,])$

   $\rightarrow$ by an undescribed rewriting rule instantiating predicate symbol variable P to predicate symbol islist:
   $\langle\ [\,], \&_1 = [\,],$
   $\quad \{\ P/\text{islist}, \text{length\_eq}(\&_1,L), \text{length\_eq}(\&_2,L), \text{length\_eq}(\&_3,L)\ \}\ \rangle$
   $\rightarrow$ by the non-deterministic rule A.1 (soft length constraint rule):
   $\langle\ \&_1 = \emptyset,$
   $\quad [\,], \emptyset = [\,],$
   $\quad \{\ P/\text{islist}, \text{length\_eq}(\&_2,0), \text{length\_eq}(\&_3,0)\ \}\ \rangle$
   Remark: if another branch of this non-deterministic rule is followed, the process fails. For example: $\langle\ \&_1 = \_T_1\ ;\ [\,],\ \_T_1 = [\,]\ ;\ \{\ P/\text{islist}, \text{length\_eq}(\&_2,1), \text{length\_eq}(\&_3,1)\ \}\ \rangle \rightarrow$ by rule A.2 (decomposition rule): $\langle\ \&_1 = \_T_1,\ [\,] = [\,],\ \_T_1 = \emptyset,\ \{\ P/\text{islist}, \text{length\_eq}(\&_2,1), \text{length\_eq}(\&_3,1)\ \}\ \rangle \rightarrow$ by rule A.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 (second-order substitution):
$\langle$ [ ], $\emptyset$ = [ ],
  { P/islist, length_eq(&$_2$,0), length_eq(&$_3$,0), &$_1/\emptyset$ } $\rangle$
$\rightarrow$ by rule A.2 (decomposition rule):
$\langle$ [ ] = [ ],
  $\emptyset = \emptyset$,
  { P/islist, length_eq(&$_2$,0), length_eq(&$_3$,0), &$_1/\emptyset$ } $\rangle$
$\rightarrow$ by rule A.2 (first-order checking): **success**
$\langle \emptyset$,
  { P/islist, length_eq(&$_2$,0), length_eq(&$_3$,0), &$_1/\emptyset$ } $\rangle$

2. **Second equation**: $G_1 = \emptyset$

   $\rightarrow$ from the result of the first equation and by rule A.2 (second-order substitution): **success**
   $\langle \emptyset$,
     { P/islist, length_eq(&$_2$,0), length_eq(&$_3$,0), &$_1/\emptyset$, $G_1/\emptyset$ } $\rangle$

3. **Third equation**: $P([\_H|\_T], \&_2)$ = islist([$T|Q$])

   $\rightarrow$ from the result of the previous equations:
   $\langle$ islist([$\_H|\_T$], &$_2$) = islist([$T|Q$]),
     { P/islist, length_eq(&$_2$,0), length_eq(&$_3$,0), &$_1/\emptyset$, $G_1/\emptyset$ } $\rangle$
   $\rightarrow$ by an undescribed rewriting rule checking predicate symbols (islist):
   $\langle$ [$\_H|\_T$], &$_2$ = [$T|Q$],
     { P/islist, length_eq(&$_2$,0), length_eq(&$_3$,0), &$_1/\emptyset$, $G_1/\emptyset$ } $\rangle$
   $\rightarrow$ by the rule A.1 (hard length constraint rule):
   $\langle$ &$_2$=$\emptyset$,
     [$\_H|\_T$] = [$T|Q$],
     { P/islist, length_eq(&$_3$,0), &$_1/\emptyset$, $G_1/\emptyset$ } $\rangle$
   $\rightarrow$ by a variant of rule A.2 (second-order substitution): **success**
   $\langle \emptyset$,
     { P/islist, length_eq(&$_3$,0), &$_1/\emptyset$, $G_1/\emptyset$, &$_2/\emptyset$, $\_H/T$, $\_T/Q$ } $\rangle$

4. **Fourth equation**: G_2, $P(\_T, \&_3)$, G_3 = islist(Q)

   $\rightarrow$ from the result of the previous equations:
   $\langle$ G_2, islist(Q, &$_3$), G_3 = islist(Q),
     { P/islist, length_eq(&$_3$,0), &$_1/\emptyset$, $G_1/\emptyset$, &$_2/\emptyset$, $\_H/T$, $\_T/Q$ } $\rangle$
   $\rightarrow$ by the non-deterministic rule A.2 (Vseqatom rewriting rule):
   $\langle$ $G_2 = \emptyset$,
     $G_3 = \emptyset$,
     islist(Q, &$_3$) = islist(Q),
     { P/islist, length_eq(&$_3$,0), &$_1/\emptyset$, $G_1/\emptyset$, &$_2/\emptyset$, $\_H/T$, $\_T/Q$ } $\rangle$
   $\rightarrow$ by rule A.2 (second-order substitution):

$\langle$ islist($Q, \&_3$) = islist(Q),
  { P/islist, length_eq($\&_3$,0), $\&_1/\emptyset$, $G_1/\emptyset$, $\&_2/\emptyset$, $\_H/T$, $\_T/Q$,
  $G_2/\emptyset$, $G_3/\emptyset$ } $\rangle$
$\rightarrow$ by an undescribed rule checking predicate symbols:
$\langle$ Q, $\&_3 = Q$,
  { P/islist, length_eq($\&_3$,0), $\&_1/\emptyset$, $G_1/\emptyset$, $\&_2/\emptyset$, $\_H/T$, $\_T/Q$,
  $G_2/\emptyset$, $G_3/\emptyset$ } $\rangle$
$\rightarrow$ by same rewriting rules as before: **success**
$\langle$ $\emptyset$,
  { P/islist, $\&_1/\emptyset$, $G_1/\emptyset$, $\&_2/\emptyset$, $\&_3/\emptyset$, $\_H/T$, $\_T/Q$, $G_2/\emptyset$, $G_3/\emptyset$ } $\rangle$

At the end of the resolution we have: $\langle\emptyset;\sigma\rangle$, with σ={ P/islist, $\&_1/\emptyset$, $\&_2/\emptyset$, $\&_3/\emptyset$, $\_H/T$, $\_T/Q$, $G_1/\emptyset$, $G_2/\emptyset$, $G_3/\emptyset$ }. From this constraint set, we derive the complete substitution which is σ itself. We have that $S\ \sigma \simeq P$.

In this complete example, 14 rewriting rules have been applied to find the substitution. For failure branches of the non-deterministic rules to be explored until failure, 10 extra rewriting rules have also to be applied.

This article was processed using the LaTeX macro package with LLNCS style