SLD-Resolution with Reflection

Jonas Barklund  Stefania Costantini
Pierangelo Dell’Acqua*  Gaetano A. Lanzarone
Uppsala University  Università degli Studi di Milano

Uppsala University
Computing Science Department
Box 311, S-751 05 Uppsala, Sweden
Phone: +46–18–18 25 00
Fax: +46–18–51 19 25

Abstract

We present a language containing names of ground expressions and a corresponding simple extension to SLD-resolution which allows meta-level computation and interlevel communication through reflection. The extended language allows significant freedom in the choice of names and as an example of a possible policy we discuss self-naming expressions. We go on to present a language in which the choice of naming relation has been partly determined by specifying that names of compound expressions are compositional. This is a sensible design decision and we present in detail a rewrite system for extended unification for the language, having certain similarities with a constraint solving system over names. Comparisons are made with related languages and systems.

This paper was presented at the 1994 International Symposium on Logic Programming, held in Ithaca, New York.

*P. Dell’Acqua has been supported by both Università degli Studi di Milano and Uppsala Univ.
1 Introduction

This paper presents an extension of SLD-resolution with naming of expressions and two-way reflection. The extension is actually more like a schema than a specific language, because it admits many choices of naming strategies and also a variety of techniques for rewriting equalities involving names. For example, we can choose a naming relation and a rewrite system for names to obtain a slightly modified variant of Reflective Prolog [9, 10].

The question of using ground or nonground representations in metalogic programming has been discussed intensively for a long time. In this proposal we sidestep the question, as the proposed inference system restricts reflection so only ground expressions are ever communicated between levels. Therefore only naming of ground expressions is involved, and no one has proposed to use anything but ground expressions to name ground expressions. A more interesting question is then whether it is sufficient to let ground expressions be their own names or not. The proposed inference system appears to be general enough to allow either approach.

The decision to only pass representations of ground expressions between levels is not so restrictive if we let naming be compositional. In this case, interlevel communication of a partially instantiated expression allows the representations of the instantiated parts to be communicated while postponing communication of the uninstantiated parts.

The real limitation with never communicating nonground expressions is that it is impossible to pass information about the instantiation status of an object level goal to the metalevel. This restriction might complicate using metaprogramming for control purposes. However, communicating such information could destroy the independence of the selection rule enjoyed by SLD-resolution.¹

The extension to the inference system is similar to a constraint solving system that avoids search by delaying the computation of which values satisfy some constraint until enough variables have been instantiated to make the computation efficient. Because of the decision to compute names of ground expressions only, the computation of a name of a nonground expression must necessarily be delayed until some other part of the computation grounds these variables. That is the responsibility of the extended unification rewrite system, which is to be given as a parameter of the inference system.

Metalevel Architectures

The advantages deriving from the ability of explicitly expressing and using metalevel knowledge have been widely recognized, especially in the AI literature [1, 2]. The need to formally represent knowledge and metaknowledge

¹This is the difference with respect to Reflective Prolog, as mentioned above.
has led to the study of metalevel architectures [4], where these two kinds of knowledge are explicitly represented. The basic features of a metalevel architecture can be summarized in:

1. the \textit{naming relation}, providing names at the metalevel for the expressions of the object level theory;

2. the metalevel formalization of the object level properties;

3. the \textit{linking rules}, that establish the connection between object level and metalevel in the inference process.

So far, the only investigation of formal properties of naming relations in metalogic programming is that of van Harmelen [21]. He argues that naming relations should be definable in order to be adapted to the particular requirements of a given metatheory. Similar motivations underlie our approach of parametrizing the naming relation described in the following sections.

The distinguished metalevel predicates, expressing properties of the object level, are related to the object level itself by linking rules, having the form of inference rules:

\[ \frac{T \vdash O \alpha}{Pr \vdash M \text{Demo}(T',\alpha')} \quad \text{and} \quad \frac{Pr \vdash M \text{Demo}(T',\alpha')}{T \vdash O \alpha} \]

where \( Pr \) is a metalevel description of provability at the object level. The first rule, \textit{object-to-meta reflection}\textsuperscript{2}, allows one to assert the provability of \( \text{Demo}(T',\alpha') \) in the metatheory when \( \alpha \) can be proved in the object theory \( T \). The second rule, \textit{meta-to-object reflection}, allows one to assert the provability of \( \alpha \) in the object level theory \( T \) whenever it is possible to prove \( \text{Demo}(T',\alpha') \) in the metatheory.

\textbf{Summary}

In Section 2 we first present a variant of SLD-resolution, based on equations rather than substitutions. We extend the usual definite clause language to a language \( K \) whose terms contains names of expressions, under fairly general assumptions. The usual Herbrand equality theory is extended to an equality theory that includes names. We define an inference system SLD*-resolution, parametrized with a rewrite system \( UN \) for unification extended with names. We discuss some possible choices of names, including self-naming.

In Section 3 we present \( L \), an instance of \( K \), where the naming relation is more specified, in order to guarantee compositionality of names. We present

\textsuperscript{2}Some authors call these linking rules upward and downward reflection, respectively; other authors, instead, use upward reflection (and downward reflection) to indicate the procedural shift from the object level to the metalevel (and from the metalevel to the object level, respectively). Therefore, we introduce new names to avoid such ambiguities.
a further extended equality theory CNET and a rewrite system UNL for unification extended to handle compositional names. We discuss properties of UNL and show how Reflective Prolog can be seen as an instance of L.

Section 4 contains comparisons with related work and some concluding remarks.

In the full version of this paper, we give declarative and procedural semantics of SLD*-resolution for K (and thus L) and establish theorems of soundness and completeness. More specifically we show that (i) every answer computed by SLD*-resolution is a correct answer with respect to the Reflective Model semantics [10] and (ii) if UN is canonical, then SLD*-resolution is complete with respect to this semantics.

2 SLD*-Resolution
Let us first describe informally a variant of SLD-resolution that might delay unification. This system will be our starting point for a formal definition of SLD*-resolution.

SLD-Resolution with Equations
To begin with, note that it is well known how to reformulate SLD-resolution over definite clause programs in terms of sets of equations rather than substitutions (see, e.g., Clark [8]). A computation state is then a pair \( \langle G, H \rangle \) where \( G \) is a set of atoms and \( H \) is a Herbrand assignment, i.e., a set of equations \( \{ X_1 = t_1, \ldots, X_k = t_k \} \), where the variables \( X_1, \ldots, X_k \) are all distinct and none of them depends on itself. Such a set may also contain the atomic proposition \( \text{false} \); it is then inconsistent. Unification can in this process be seen as a rewriting system that takes a set of equations to an equivalent Herbrand assignment (under the Herbrand equality theory). Unification algorithms that can be expressed as such rewriting systems have been defined, e.g., by Huet [13] and Martelli & Montanari [19].

The assumption that unification rewrites the whole set of equations to a Herbrand assignment can be relaxed. Let a state instead consist of a triple \( \langle G, H, E \rangle \), where \( G \) is a set of atoms, \( H \) is a Herbrand assignment, and \( E \) is a set of equations over terms. Such a state represents the definite goal \( \langle \leftarrow (\wedge G) \wedge (\wedge E) \rangle \hat{H} \), where \( \hat{H} \) is the substitution corresponding to the Herbrand assignment \( H \). We can see \( H \) and \( E \) as the solved and unsolved part of a single equation system. The unification rewrite system now takes a pair \( \langle H, E \rangle \) to a new pair \( \langle H', E' \rangle \) such that \( H \cup E \) and \( H' \cup E' \) are equivalent and \( H \subseteq H' \). Note that the resulting unsolved part \( E' \) need not be empty. Thus, unsolvability of the set of equations might not be detected until further rewriting can take place. (This system is similar to

\footnote{In Chapter 5 (Unification dans les langages de premier ordre) of Huet’s Ph.D. thesis there is a very elegant unification algorithm for first-order expressions, which always terminates.}
AGLD-resolution, defined by Clark [8], originating from Wolfram, Maher and Lassez [22].

Adding Names and Reflection

Consider the usual language of definite clause programs [18] extended so that for every ground term or atom there is exactly one ground term that names it. Moreover, a term is the name of one expression, at most.\(^4\) As the names are themselves ground terms, it follows that in the language there must also be names of the names of every term and atom, and so on. When \(\alpha\) is an expression for which there is a name, we will write \(\alpha^1\) for it, \(\alpha^2\) for the name of \(\alpha^1\), etc. For convenience we also define \(\alpha^0 = \alpha\). Similarly, if \(\beta\) is the name of some expression, then we will write \(\beta^{-1}\) for that expression.

We will assume that the language, which we will refer to as \(K\), contains a unary predicate symbol \(\mathit{solve}\) and a binary predicate symbol \(\mathit{name}\) (besides =, as usual). We say that they are distinguished. A predication \(\mathit{solve}(\beta)\) is intended to state that \(\beta^{-1}\) is a theorem (with respect to the implicit program). A predication \(\mathit{name}(\beta, \alpha)\) is intended to state that \(\beta\) is the name of the term \(\alpha\). No clause may have a \(\mathit{name}\) atom as its head.

Consider a program \(P\) and a goal \(G_0\). We shall describe a process for refuting \(P \cup \{G_0\}\) based on successive rewriting of a state. As for the modified SLD-resolution, a state is a triple \((G, H, E)\), where \(G\) is a set of atoms, \(H\) is a Herbrand assignment and \(E\) is a set of equations. However, we allow the equations in \(E\) to contain also expressions on the form \(\uparrow\alpha\) and \(\downarrow\beta\), where \(\alpha\) is some expression having a name and \(\beta\) is an expression that is a name. The intuition is that \(\uparrow\alpha\) means the name of \(\alpha\) and \(\downarrow\beta\) what \(\beta\) names.

The Herbrand equality theory [7] is extended with axioms on the forms

- if \(\uparrow\alpha = \uparrow\beta\), then \(\alpha = \beta\);
- if \(\downarrow\alpha = \downarrow\beta\), then \(\alpha = \beta\);
- \(\downarrow\uparrow\alpha = \alpha\);
- \(\uparrow\alpha = \alpha^1\) and \(\downarrow\beta = \beta^{-1}\) for every ground expression \(\alpha\) and name \(\beta\);
- \(\uparrow\alpha \neq \gamma\) for every ground expression \(\alpha\) and every ground expression \(\gamma\) distinct from \(\alpha^1\);
- \(\downarrow\beta \neq \gamma\) for every name \(\beta\) and every ground expression \(\gamma\) distinct from \(\beta^{-1}\);

\(^4\)The assumption that the naming relation is functional and injective simplifies our presentation. In order to relax this assumption, certain steps in what follows could be made nondeterministic, choosing one particular name of an expression or one particular named expression.
• \( X \neq \uparrow \alpha[X] \) and \( X \neq \downarrow \beta[X] \) for every expression \( \alpha[X] \) and name \( \beta[X] \) containing some variable \( X \).

We will refer to this extended equality theory as NET (Name Equality Theory).

The choice of a naming scheme above is one parameter of this inference scheme. Together with this naming scheme, we must be given a rewrite system UN for unification properly extended for the naming scheme. It should terminate and rewrite a pair \( \langle H, E \rangle \) to a unique pair \( \text{UN}(\langle H, E \rangle) = \langle H', E' \rangle \), under the same restrictions as for the modified SLD-resolution above but where \( H \cup E \) and \( H' \cup E' \) are instead equivalent under NET.

Let \( P \) be a \( K \)-program and \( G \) a \( K \)-goal. An initial state when refuting \( P \cup \{ G_0 \} \) is a triple \( \langle G_0, \{ \}, \{ \} \rangle \), a failure state is a triple \( \langle G, H \cup \{ \text{false} \}, E \rangle \) and a success state is a triple \( \{ \}, H, E \) where \text{false} is not in \( H \) and \( H \cup E \) is solvable.

Yet another possibility is a state \( \{ \}, H, E \) that is neither a failure state, nor a success state, but where UN cannot rewrite \( \langle H, E \rangle \) to another pair and we do not know if \( H \cup E \) is solvable. We say that such a state is floundering.

We will nondeterministically construct a tree of states, branching from the initial state. Let \( S_i = \langle G_i, H_i, E_i \rangle \) be some state that is neither a failure state, nor a success state. There is a nondeterministic choice between five cases. Let \( A_i \) be the selected atom of \( G_i \), and assume that the variables of the selected clause, when applicable, are disjoint from those of \( S_i \).

1. Select a clause \( B_i \leftarrow D_i \) from \( P \), where \( A_i \) and \( B_i \) have the same predicate symbol. The new state is \( S_{i+1} = \langle G_i \setminus \{ A_i \} \cup D_i, H_i, E_i \cup \{ A_i = B_i \} \rangle \), where \( A_i = B_i \) is short for the set of equations that equates the corresponding arguments of \( A_i \) and \( B_i \).

2. \( A_i \) is \text{solve}(Q_i). Select a clause \( B_i \leftarrow D_i \) from \( P \), where the predicate symbol of \( B_i \) is not \text{solve}. The new state is \( S_{i+1} = \langle G_i \setminus \{ A_i \} \cup D_i, H_i, E_i \cup \{ Q_i = \uparrow B_i \} \rangle \).

3. The predicate symbol of \( A_i \) is not \text{solve}. Select a clause \( \text{solve}(Q_i) \leftarrow D_i \) from \( P \). The new state is \( S_{i+1} = \langle G_i \setminus \{ A_i \} \cup D_i, H_i, E_i \cup \{ Q_i = \uparrow A_i \} \rangle \).

4. \( A_i \) is \text{name}(\beta, \alpha). The new state is \( S_{i+1} = \langle G_i \setminus \{ A_i \}, H_i, E_i \cup \{ \beta = \uparrow \alpha \} \rangle \).

5. Suppose that \( \text{UN}(\langle H_i, E_i \rangle) = \langle H_{i+1}, E_{i+1} \rangle \); the new state is then \( S_{i+1} = \langle G_i, H_{i+1}, E_{i+1} \rangle \).

The first and the last case together correspond to the operations of the modified SLD-resolution discussed above. The second case is an \textit{object-to-meta} reflection, where an object level clause is used to prove a metalevel atom.
The third case is a meta-to-object reflection, where a metalevel clause is used to prove an object level atom. The fourth case, finally, is the mechanism to compute names of ground terms, or vice versa, when requested from a program. All the three first cases are similar: adding a set of goals and a set of unsolved equations to the state. The fifth case “cleans up” the equations.

An SLD*-derivation is a (finite or infinite) path in the tree of states above. We say that a SLD*-derivation is floundering if its final state is floundering. A SLD*-refutation is a finite path in the tree, ending with a success state.

To achieve termination of SLD*-resolution when possible, it is necessary that the rewriting system UN makes progress in some useful sense. Some rewrites that make progress are elimination of obviously true equations and moving solved equations to the solved part. One could define some norm on \( (H, E) \) in the nonnegative integers that must be decremented by each rule of UN, but we will not go that far here. (Rewriting systems based on the unification algorithms mentioned earlier certainly make useful progress. Later we will propose specific rewriting systems for which we can argue why they make progress.)

Note that SLD*-resolution is a quite small extension of the alternate form of SLD-resolution that adds a naming scheme to the language and adds interlevel inference and access to the names of terms to the inference system. The three additional inference rules could probably also be added to other inference systems for definite clause programs that have provisions for delaying computations.

3 A Language with Compositional Names

In this section we introduce the basic features of a metalogic definite clause language. The main refinement with respect to the language introduced informally in the previous section is that the names of this language are compositional, another refinement is that the language is two-sorted (name terms and terms, where the former is a subset of the latter). The naming scheme for compound expressions is therefore predetermined, but the choice of naming of symbols is largely left open. The advantage of compositional names is that an extended unification rewriting system can construct partially the name of a nonground expression. As a result, computations can no longer flounder, because we can always determine if the current set of constraints is solvable.

3.1 Language

The abstract language we are about to introduce will be called \( L \) throughout this presentation. It is an instance of the language \( K \) of the previous section. The language is that of definite programs, as defined by Lloyd [18], except that terms are defined differently in order to include name terms that are intended to denote expressions of the language itself. In order to allow
the results presented below to be as general as possible we use an abstract syntax for the terms, which could be concretized in various ways. Various extensions of $L$ can also be accommodated, provided that they are based on limited modifications of the same semantics.

**Alphabet** The difference in the alphabet of $L$, as compared with the usual alphabet of definite clause programs, lies in a distinction between object variables and metavariables and in the presence of various other metasymbols.

We assume that we are given an injective mapping that associates with every constant, function and predicate symbol a metasymbol that names it. We will say that the result of applying this mapping to a symbol yields the symbol intended as its name.

We extend the notation employed for names of expressions in the preceding section, so that if $\alpha$ is a non-variable symbol in $L$, then $\alpha^1$ denotes the metasymbol in $L$ intended as the name of $\alpha$. Note that we do not put any restriction on the metasymbols introduced, so also in this case $\alpha$ and $\alpha^1$ may even coincide.

The distinguished predicate symbols $solve$ and $name$ play a special role in SLD*-resolution and we do not assume that there are symbols naming them.

**Definition 3.1** The language $L$ has the following object symbols:

- the object variables;
- the object constants, where we distinguish between the constant symbols, for each $n$ ($n > 0$) the $n$-ary function symbols and for each $n$ ($n \geq 0$) the $n$-ary predicate symbols;
- the usual logical connectives and punctuation symbols.

The language $L$ has the following metasymbols:

- the metavariables;
- the metaconstants, where we distinguish between the constant names, the function names and the predicate names.

The variables of $L$ is the union of the object variables and the metavariables of $L$. The constants of $L$ is the union of the object constants and the metaconstants of $L$. The symbols of $L$ is the union of the object symbols and the metasymbols of $L$. (We will also use punctuation symbols freely.)

Given a program $P$ we can identify the alphabet of $P$ as the set of symbols occurring in $P$ extended with the symbols $\alpha^k$ for every $k$, $k \geq 0$, whenever a symbol $\alpha^l$ occurs in $P$, for some $l \geq 0$. 

7
Terms The terms of $L$ are defined as usual except that we introduce the name terms. The name of a ground compound expression $\alpha_0(\alpha_1,\ldots,\alpha_n)$ ($n > 0$) in $L$ is the compound name term $[\alpha'_0,\alpha'_1,\ldots,\alpha'_n]$, where each $\alpha'_i$ is the name of $\alpha_i$, $0 \leq i \leq n$. Furthermore, the name of the name of that compound expression is $[\alpha''_0,\alpha''_1,\ldots,\alpha''_n]$, where $\alpha''_i$ is the name of $\alpha'_i$, $0 \leq i \leq n$, etc.

Definition 3.2 We define the name terms of $L$ by an inductive definition:

- A metavariable is a name term.
- A metaconstant is a name term.
- If $\alpha_0$ is a metavariable, a function name or a predicate name and $\alpha_1,\ldots,\alpha_n$ are name terms, then a compound name term $\alpha = [\alpha_0,\alpha_1,\ldots,\alpha_n]$ is a name term. If $\alpha_0$ is a predicate name, then we call $\alpha$ a predication name.

No other expressions are name terms.

Definition 3.3 We define the terms of $L$ by an inductive definition:

- An object variable is a term.
- An object constant is a term.
- A name term is a term.
- If $f$ is a $n$-ary function symbol and $\alpha_1,\ldots,\alpha_n$ are terms, then a compound term $f(\alpha_1,\ldots,\alpha_n)$ is a term.

No other expressions are terms.

Name terms allow the representation of ground expressions of the language in the language itself. Letting names of compound expressions be compositional allows us to use unification for constructing name terms and accessing parts of name terms. For example, the name of the term $f(a)$ is the compound name term $[f^1,a^1]$, and the name of the atom $p(f(a),b^1)$ is the predication name $[p^1,[f^1,a^1],b^2]$. In general the “$k$-th” name of the term $f(a)$ is the term $[f^k,a^k]$.

Definition 3.4 Given an $L$ program $P$, the language of $P$ is the subset of $L$ that can be generated from the alphabet of $P$. 

8
The definitions of substitutions and variable assignments, as well as the unification rewriting system, have to be properly modified to take into account name terms and metavariables. Only name terms may be substituted for metavariables. If a compound name term is to actually be the name of some compound expression, then all components must be names and the first component must be a function name, a predicate name or a name (at some level) of such a symbol. For example, the name term \([f^1, a, b^1]\) is not the name of any term, nor is \([a^1, b^1, c^1]\) if \(a^1\) is not the name of a binary function or predicate symbol. However, all components need not be “at the same level”, for example, the name term \([f^1, a^2, b^3]\) is the name of the term \(f(a^1, b^2)\).

With a more elaborate system of sorts, we could define terms and name terms in a more restrictive way so that every well-formed name term would be the name of some expression.

### 3.2 A Compositional Name Equality Theory

We shall extend the equality theory NET of the previous section with axioms that characterize equality for compound name terms. The extension is similar to Clark’s axioms for compound terms.

Compound name terms of the same length are equal if and only if their components are. For all positive \(k\),

\[
[X_0, X_1, \ldots, X_k] = [Y_0, Y_1, \ldots, Y_k] \leftrightarrow X_0 = Y_0 \land X_1 = Y_1 \land \cdots \land X_k = Y_k.
\]

Compound name terms of different length are not equal. For all positive \(k\) and \(l, k \neq l\),

\[
[X_0, X_1, \ldots, X_k] \neq [Y_0, Y_1, \ldots, Y_l].
\]

Name terms are not equal to any other kind of term. For all positive \(k\), every constant \(\alpha\) and every function symbol \(f\) (with arity \(l\)),

\[
[X_0, X_1, \ldots, X_k] \neq \alpha \quad \text{and} \quad [X_0, X_1, \ldots, X_k] \neq f(Y_1, \ldots, Y_l).
\]

Next, we characterize the interaction between the \(\uparrow\) and \(\downarrow\) operators and equality. For every constant \(\alpha\) and every metaconstant \(\beta\),

\[
\uparrow\alpha = \alpha^1 \quad \text{and} \quad \downarrow\beta = \beta^{-1}.
\]

For every function symbol \(f\) (with arity \(k\)) and every function name \(f^1\),

\[
\uparrow f(X_1, \ldots, X_k) = [f^1, \uparrow X_1, \ldots, \uparrow X_k]
\]

and

\[
\downarrow [f^1, X_1, \ldots, X_k] = f(\downarrow X_1, \ldots, \downarrow X_k).
\]

For every positive \(k\),

\[
\uparrow [X_0, \ldots, X_k] = [\uparrow X_0, \ldots, \uparrow X_k] \quad \text{and} \quad \downarrow [Y_0, \ldots, Y_k] = [\downarrow Y_0, \ldots, \downarrow Y_k],
\]

if \(Y_0\) is not a function name. We call this equality theory CNET (compositional name equality theory).
3.3 Extended Unification

Now we will consider a unification rewriting system UNL for $L$. As before, it takes a pair $(H, E)$ to a pair $(H', E')$ under certain restrictions, but now $H \cup E$ and $H' \cup E'$ must be equivalent under CNET.

In order to simplify the definition of UNL, we shall define it in two parts. First, we define a rewrite system A with the purpose to rewrite $\uparrow$ and $\downarrow$ expressions in $E$. Second, we define a rewrite system B, which is a quite straightforward extension of Herbrand unification to handle name terms. Finally, we will define UNL as the union of these systems.

In all rules below it is an implicit condition that $H$ does not contain $false$. Moreover, we will assume that the equality in the unsolved set is symmetric, so when we write $E \cup \{\alpha = \beta\}$, we really mean $E \cup \{\alpha = \beta\}$ or $E \cup \{\beta = \alpha\}$.

Rewrite rules for A [Evaluate up arrows.] If any $\uparrow$ expression has become instantiated, then replace it with another expression.

\[
\begin{align*}
\langle H, E[\alpha] \rangle & \rightarrow \langle H, E[\sigma] \rangle & \text{if } \alpha \hat{H} \text{ is a constant } \sigma \\
\langle H, E[\alpha] \rangle & \rightarrow \langle H, E[\uparrow f, \uparrow \alpha_1, \ldots, \uparrow \alpha_k] \rangle & \text{if } \alpha \hat{H} \text{ is } f(\alpha_1, \ldots, \alpha_k) \\
\langle H, E[\alpha] \rangle & \rightarrow \langle H, E[\uparrow \alpha_0, \uparrow \alpha_1, \ldots, \uparrow \alpha_k] \rangle & \text{if } \alpha \hat{H} \text{ is } [\alpha_0, \alpha_1, \ldots, \alpha_k]
\end{align*}
\]

[Evaluate down arrows.] If any $\downarrow$ expression has become instantiated, then replace it with another expression.

\[
\begin{align*}
\langle H, E[\beta] \rangle & \rightarrow \langle H, E[\mu^{-1}] \rangle & \text{if } \beta \hat{H} \text{ is a metaconstant } \mu \\
\langle H, E[\beta] \rangle & \rightarrow \langle H, E[\downarrow \beta_0, \downarrow \beta_1, \ldots, \downarrow \beta_k] \rangle & \text{if } \beta \hat{H} \text{ is } [\beta_0, \beta_1, \ldots, \beta_k] \\
\langle H, E[\beta] \rangle & \rightarrow \langle H \cup \{false\}, E[\beta] \rangle
\end{align*}
\]

if $\beta \hat{H}$ is an object constant or a compound term

[Make compound term.] Convert to correct syntax for compound terms.

\[
\langle H, E[\alpha_0, \alpha_1, \ldots, \alpha_k] \rangle \rightarrow \langle H, E[f(\alpha_1, \ldots, \alpha_k)] \rangle & \text{if } \alpha_0 \hat{H} \text{ is } f
\]

[Reverse arrow.] There may be equations where the arrow is on the “wrong” term so it cannot be rewritten. We exploit that each arrow is the other’s inverse.

\[
\begin{align*}
\langle H, E \cup \{\uparrow \alpha = \beta\} \rangle & \rightarrow \langle H, E \cup \{\alpha = \downarrow \beta\} \rangle & \text{if } k > 0, \alpha \hat{H} \text{ is a variable and } \beta \hat{H} \text{ is not a variable} \\
\langle H, E \cup \{\downarrow \beta = \alpha\} \rangle & \rightarrow \langle H, E \cup \{\beta = \uparrow \alpha\} \rangle & \text{if } k > 0, \beta \hat{H} \text{ is a variable and } \alpha \hat{H} \text{ is not a variable}
\end{align*}
\]

Theorem 3.5 The rewrite system A is canonical.
Rewrite rules for $B$  These rewrite rules form a quite straightforward extension of the usual Herbrand unification to handle metavariables, metaconstants and compound name terms.

[Identity.] A trivial equality is superfluous.

$$\langle H, E \cup \{ \alpha = \alpha \} \rangle \rightarrow_B \langle H, E \rangle$$

[Variable.] Typically, an equation between a variable and something else can be moved to the solved part, but some equations cause failure (occur check or sort conflict) and there must be no arrow operators.

$$\langle H, E \cup \{ \alpha = \beta \} \rangle \rightarrow_B \langle H \cup \{ \alpha \beta = \beta \beta \}, E \rangle$$

if $\alpha \beta$ is an object variable; $\beta \beta$ does not properly contain $\alpha \beta$; $\beta \beta$ does not contain $\uparrow$ or $\downarrow$.

$$\langle H, E \cup \{ \alpha = \beta \} \rangle \rightarrow_B \langle H \cup \{ \alpha \beta = \beta \beta \}, E \rangle$$

if $\alpha \beta$ is a metavariable; $\beta \beta$ does not properly contain $\alpha \beta$; $\beta \beta$ is not an object variable, an object constant or a compound term; $\beta \beta$ does not contain $\uparrow$ or $\downarrow$.

$$\langle H, E \cup \{ \alpha = \beta \} \rangle \rightarrow_B \langle H \cup \{ \text{false} \}, E \rangle$$

if $\alpha \beta$ is a variable and $\beta \beta$ properly contains $\alpha \beta$.

$$\langle H, E \cup \{ \alpha = \beta \} \rangle \rightarrow_B \langle H \cup \{ \text{false} \}, E \rangle$$

if $\alpha \beta$ is a metavariable and $\beta \beta$ is an object constant, an object variable or a compound term.

[Inhomogeneity.] Object constants, compound terms, metaconstants and compound name terms are incompatible with each other.

$$\langle H, E \cup \{ \alpha = \beta \} \rangle \rightarrow_B \langle H \cup \{ \text{false} \}, E \rangle$$

if each of $\alpha \beta$ and $\beta \beta$ is an object constant, a compound term, a metaconstant or a compound term name and they are not of the same kind.

[Clash.] Different constants, compound terms or compound name terms.

$$\langle H, E \cup \{ \alpha = \beta \} \rangle \rightarrow_B \langle H \cup \{ \text{false} \}, E \rangle$$

if $\alpha \beta$ and $\beta \beta$ are distinct constants, compound terms with distinct principal symbols or compound name terms of different length.
Equality between compound terms is propagated to the corresponding subexpressions.

\[ \langle H, E \cup \{\alpha = \beta\} \rangle \rightarrow B \langle H, E \cup \{\alpha_1 = \beta_1, \ldots, \alpha_k = \beta_k\} \rangle \]

if \( \alpha \bar{H} \) is \( f(\alpha_1, \ldots, \alpha_k) \) and \( \beta \bar{H} \) is \( f(\beta_1, \ldots, \beta_k) \), for some \( k \).

Equality between compound name terms is also propagated.

\[ \langle H, E \cup \{\alpha = \beta\} \rangle \rightarrow B \langle H, E \cup \{\alpha_0 = \beta_0, \ldots, \alpha_k = \beta_k\} \rangle \]

if \( \alpha \bar{H} \) is \( [\alpha_0, \ldots, \alpha_k] \) and \( \beta \bar{H} \) is \( [\beta_0, \ldots, \beta_k] \), for some \( k \).

**Theorem 3.6** The rewrite system \( B \) is canonical.

It is not necessarily the case that the union of terminating and confluent rewrite systems is terminating or confluent. However, it is the case for \( A \cup B \).

**Theorem 3.7** The rewrite system \( UNL = A \cup B \) is canonical.

When a rewrite system \( R \) is terminating and confluent, it follows that every term \( t \) has a unique normal form \( t' \), obtained as \( t \xrightarrow{R} t' \), and we write \( R(t) \) for \( t' \). Thus, \( UNL(\langle H, E \rangle) \) is well defined.

The correctness of \( UNL \) is stated in the following theorem.

**Theorem 3.8** If \( UNL(\langle H, E \rangle) = \langle H', E' \rangle \), then \( CNET \models H \cup E \equiv H' \cup E' \).

### 3.4 Example: Reflective Prolog

In Reflective Prolog we are given sets of symbols together with a naming relation on them. For instance, the symbol \( a \) is an example of object constant, while the symbols \( "a" \), \( ""a"" \) and \( "\"a\"" \) are examples of constant names. Moreover, it is specified that \( a_1 = "a", a_2 = "a"^3 = "\"a\"", \) etc.

Similarly, the symbol \( f \) is an example of a function symbol, the symbol \( \{f\} \) is an example of a function name and the symbols \( "\{f\}" \) and \( "\"\{f\}\"" \) are further examples of constant names. Moreover, it is specified that \( f_1 = \{f\}, f_2 = "\{f\}" \), etc.

Except for the elaboration of the system of sorts mentioned above, SLD*-resolution together with the rewrite system for \( L \) can be used as presented for running Reflective Prolog programs.
4 Related Work and Concluding Remarks

We are aware of a few other attempts to combine metalogic programming and constraint logic programming. Here we present a brief overview, together with some other relevant work, in chronological order.

Heintze et al. [11] propose an approach that integrates metaprogramming facilities and the constraint paradigm. They extend the language CLP(R) [15] with the aim of allowing manipulations of CLP(R) programs and introduce some facilities to access the coded form of the current set of constraints. The addition of these facilities enables one to develop new applications and simplification algorithms.

Lim and Stuckey [17] propose a metalevel structure of computation in order to give a logical meaning to metaprogams. In particular, they extend the theory of CLP [14] with a metalevel structure that incorporates a method for representing object level terms and interpreted relations (constraints) for manipulating object level terms. Thus, each metaprogram can be seen as a constraint logic program over this structure.

Sato [20] has presented a style of metaprogramming based on a truth predicate in three-valued logic. His emphasis is not on presenting a proof system, but rather a metainterpreter for the truth predicate.

Cervenat and Rossi [5] present a metalevel extension of a logic programming language, called 'Log. The extension consists of a naming scheme that associates two different metarepresentations with every syntactic object of the language, and of a destructuring operator allowing us to relate these metarepresentations. They use constraint satisfaction techniques to characterize the destructuring operator, rather than reflection.

Jiang [16] proposes an approach quite different from ours, where he takes the syntactic ambiguity of Prolog even further. His logic intentionally tries to remove the distinction between predicate and function symbols, etc. For ground expressions his naming scheme can be captured in our framework, by self-naming.

We are aware of a study by Christiansen [6], whose idea is also to use a delay mechanism when computing names of expressions. His approach is somewhat different as he takes as his formalism a Prolog system extended with a delaying primitive.

The Gödel programming language has names as an abstract datatype [12]. Names of expressions will thus always appear as variables in metaprogams. The metalevel part of a program can use the operations associated with the datatype to manipulate names. The naming relation of Gödel is compositional and it seems to fit in our framework.

The Alloy metalogic programming language [3] exploits an extended version of the scheme described in this paper, including names also of non-ground expressions. This complicates the rewriting system but allows a
metalevel program to express properties of more general object level formulas.

Some of these approaches are only partly related to the present one, because they are mainly aimed at syntactic metaprogramming: program manipulation and transformation via metapograms. Our approach instead, is mainly intended for multilevel knowledge representation and reasoning. In fact, it allows reflective steps (and thus multilevel reasoning) to be modeled by extending SLD-Resolution so as to dynamically relate language expressions and their names. We have seen that such an approach is easily applicable to many multilevel formalisms. In fact, it is usable for both explicit and implicit reflection: for explicit reflection, the extended resolution will be applied only on demand, while using usual SLD-Resolution for all the other steps. The conditions for the applicability of the approach are the following. The reflective inference mechanism must be formally defined. It must be parametrized with respect to unification (which can thus be extended to include the equation rewrite algorithm). The declarative semantics must not depend on the specific features of the naming relation.

Acknowledgements

Pierangelo Dell’Acqua has been financially supported through a scholarship from Univ. degli Studi di Milano and later by the Swedish National Board for Technical and Industrial Development (NUTEK) under contract No. 92–10452 (ESPRIT BRP 6810: Computational Logic 2).

References


