Introducing Weak Unification into the WAM *

Pascual Julián-Iranzo and Clemente Rubio-Manzano

Department of Computer Science, University of Castilla-La Mancha.
{Pascual.Julian},{Clemente.Rubio}@uclm.es

Abstract. In [21] Maria I. Sessa extended the SLD resolution principle with the ability of performing approximate reasoning and flexible query answering. The operational mechanism of similarity-based SLD resolution can be used as the basis for a new logic language that allows to manage uncertain and imprecise information in a declarative framework, hence its importance. Similarity-based SLD resolution can be seen as an extension of the classical SLD resolution procedure where the syntactic unification algorithm has been replaced by a fuzzy unification algorithm. In this paper we address the problem of adapting the implementation of a WAM to incorporate fuzzy unification. As a result, we obtain a Prolog implementation based on similarity relations that we call S-Prolog. To the best of our knowledge this is the first WAM implementation that supports similarity-based SLD resolution.

Keywords: Fuzzy Logic, Fuzzy Prolog, Unification by Similarity, Warren Abstract Machine.

1 Introduction

Fuzzy Logic relays on the concept of fuzzy set [25, 26] and can be seen as an extension of classical predicate logic able to model uncertainty and/or vagueness. During the last decades fuzzy Logic has proved its usefulness in a number of practical applications (such as control systems, database or expert systems) where the management of imprecise information is mandatory. A comprehensive introduction to the subject of fuzzy Logic can be found in [19].

Fuzzy Logic Programming integrates fuzzy logic and pure logic programming [14], in order to provide these languages with the ability of dealing with uncertainty and approximated reasoning. One of the main advantages of this combination is the construction of programming languages that allow us to deal with imprecise information by using declarative techniques. It is important to say that there is no common method for introducing fuzzy concepts into logic programming. We have found two major approaches:

* This work has been partially supported by the EU, under FEDER, and the Spanish Science and Education Ministry (MEC) under grant TIN 2004-07943-C04-03.

– The first approach, replaces the syntactic unification mechanism of classical SLD–resolution by a weak unification algorithm, based on similarity relations. This algorithm provides a weak most general unifier as well as a numerical value, called the *unification degree*. Intuitively, the unification degree represents the truth degree associated with the (query) computed instance. Programs written in this kind of languages consist, in essence, in a set of ordinary (Prolog) clauses jointly with a set of "similarity equations" which play an important role during the unification process. Examples of this kind of languages are described in [4–7] and [21].

– In the second approach, programs are considered fuzzy subsets of (clausal) formulas, where the *truth degree* of each clause is explicitly annotated. The work of computing and propagating truth degrees relies on an extension of the resolution principle, whereas the (syntactic) unification mechanism remains untouched. Examples of this kind of languages are described in [8–10, 12, 13, 3, 16–18, 23] and [22].

In this paper we are interested in the implementation of the first class of fuzzy logic languages.

We follow the conceptual approach introduced in [21] where the notion of "approximation" is managed at a syntactic level by means of similarity relations. A similarity relation is an extension of the crisp notion of equivalence relation and it can be useful in any context where the concept of equality must be weakened. In [21] a new modified version of the SLD resolution procedure, named *similarity-based* SLD, is defined. Roughly speaking the similarity-based SLD resolution principle works as it is shown by the following example (adapted from [21]).

Example 1. Assume a database storing information on books, including readers preferences and some subjective information concerning the similarity between some syntactic entities. Then it is possible to perform an inference reasoning step where the antecedent of a conditional formula is allowed to match with some premise only approximately.

$$\frac{\begin{array}{l}\text{if } x \text{ is a } \texttt{mystery} \text{ book then } x \text{ is a } \texttt{good} \text{ one;}\\ \texttt{dracula} \text{ is a } \texttt{horror} \text{ book;}\\ \texttt{horror} \text{ is similar to } \texttt{mystery} \text{ with degree } 0.9\end{array}}{\texttt{dracula} \text{ is a } \texttt{good} \text{ book with degree } 0.9}$$

Since `horror` is similar to `mystery` with a certainty/truth degree of 0.9, also the conclusion will be affected by the similarity degree assigned to the relation between `horror` and `mystery`.

In this paper we are interested in the implementation of a fuzzy logic language that follows this inference scheme. More precisely, our goal is to incorporate the Sessa's similarity-based SLD resolution principle into the core of a Warren Abstract Machine (WAM) [24]. As a result, we obtain a Prolog implementation based on similarity relations that we call S-Prolog. The WAM is a virtual computer that aids in the compilation and implementation of the Prolog programming language and offers techniques for compiling symbolic languages that can be generalized beyond Prolog. A tutorial reconstruction for the WAM can be found in [1].

The structure of the paper is as follows. Basic definitions are given in Section 2. Section 3 recalls the definition of the similarity-based SLD resolution principle. In Section 4, the syntax and semantics of the S-Prolog language is described. Section 5 presents the main features of the Similarity WAM machine: we described the structure and behavior of the compiler and virtual machine as well as the implementation details related with the introduction of weak unification. To illustrate the compilation process a concrete example is shown in Subsection 5.4. Also, in Section 6, we give a formalization of the Similarity WAM Machine operational semantics and we demonstrate its equivalence with Sessa's Weak SLD resolution rule. Finally, in Section 7 we give our conclusions and some lines of future research.

In the following, we assume familiarity with the theory and practice of logic programming [2].

2 Similarity Relations and Unification by Similarity

In this section we present the weak unification algorithm of [21], where the syntactical identity between symbols is replaced by a test on a similarity relation. Before giving a formal definition of the weak unification procedure we need to remember some preliminary concepts.

2.1 Similarity relations

Given a set U, an ordinary subset A of U can be defined in terms of its *characteristic function* $\chi_A(x)$ (that returns 1 if $x \in A$ or 0 otherwise). On the other hand, a *fuzzy subset* A of U is a function $A : U \to [0,1]$. The function A is called the *membership function*, and the value $A(x)$ represents the *degree of membership* of x in the fuzzy subset A, being a generalization of the notion of characteristic function. Similarly, an ordinary binary relation on U is a subset of $U \times U$ and it can be identified by its characteristic function $U \times U \to \{0,1\}$. Therefore, the easy extension of this concept to the fuzzy case is to agree that, a *fuzzy binary relation* is a fuzzy subset on $U \times U$ (that is, a mapping $U \times U \to [0,1]$).

Definition 1. *A* similarity relation *on a set U is a fuzzy binary relation on $U \times U$, $\mathcal{R} : U \times U \to [0,1]$, holding the following properties:*

1. **(Reflexive)** $\mathcal{R}(x,x) = 1$ *for any* $x \in U$*;*
2. **(Symmetric)** $\mathcal{R}(x,y) = \mathcal{R}(y,x)$ *for any* $x, y \in U$*;*
3. **(Transitive)** $\mathcal{R}(x,z) \geq \mathcal{R}(x,y) \triangle \mathcal{R}(y,x)$ *for any* $x, y, z \in U$*;*

where the operator '$\triangle$' is an arbitrary t-norm[1]*.*

[1] A t-norm $\triangle : [0,1] \times [0,1] \to [0,1]$ is a binary operator which is commutative, associative, monotone in both arguments and $x \triangle 1 = x$ (hence, it subsumes the classical two-valued conjunction operator) [20].

In [19], when the operator $\triangle = \wedge$ (that is, it is the minimum of two elements), similarity relations are called fuzzy equivalence relations. Certainly, in this case, there exits a close relation between similarity relations and equivalence relations.

Definition 2. *Let U be a set and $\mathcal{R} : U \times U \to [0, 1]$ be a similarity relation. The binary relation $\equiv_{\mathcal{R},\lambda}$ on U defined as $\equiv_{\mathcal{R},\lambda}= \{(x, y) \mid \mathcal{R}(x, y) \geq \lambda\}$ is called the* λ-cut *of $\mathcal{R}$.*

Proposition 1. *[19] Let U be a set and $\mathcal{R} : U \times U \to [0, 1]$ be a fuzzy equivalence relation. For any λ, the λ-cut of $\mathcal{R}$, $\equiv_{\mathcal{R},\lambda}$, is an equivalence relation.*

Since an equivalence relation $\equiv_{\mathcal{R},\lambda}$ can be considered as a generalization of the identity relation, intuitively, a fuzzy equivalence on a set specifies when two elements may be considered equal with regard to a property that is not sharply defined.

Following [21], in the sequel, we restrict ourself to similarity relations that are fuzzy equivalence relations. Moreover we are interested in fuzzy equivalence relations at a syntactic level.

2.2 Similarity relations on syntactic domains

In classical Logic Programming different syntactic symbols represent distinct information. This restriction can be relaxed by introducing a similarity relation $\mathcal{R}$ on the alphabet of a first order language, allowing $\mathcal{R}$ to provide a possible non-zero value for function/predicate symbols with the same arity, whereas it is the identity relation for variables.

The similarity relation $\mathcal{R}$ on the alphabet of a first order language can be extended to terms and atomic formulas by structural induction in the usual way:

1. Let f and g be two n-ary function symbols and let $t_1, \dots, t_n, s_1, \dots, s_n$ be terms. $\mathcal{R}(f(t_1, \dots, t_n), g(s_1, \dots, s_n)) = \mathcal{R}(f, g) \wedge (\bigwedge_{i=1}^{n} \mathcal{R}(t_i, s_i))$;
2. Let p and q be two n-ary predicate symbols and let $t_1, \dots, t_n, s_1, \dots, s_n$ be terms. $\mathcal{R}(p(t_1, \dots, t_n), q(s_1, \dots, s_n)) = \mathcal{R}(p, q) \wedge (\bigwedge_{i=1}^{n} \mathcal{R}(t_i, s_i))$.

2.3 Unification by similarity

In presence of similarity relations on syntactic domains, it is possible to define an extended notion of a unifier and a more general unifier of two expressions[2].

Definition 3. *Let $\mathcal{R}$ be a similarity relation and $\mathcal{E}_1$ and $\mathcal{E}_2$ be two expressions. The substitution θ is a* weak unifier *of $\mathcal{E}_1$ and $\mathcal{E}_2$ w.r.t $\mathcal{R}$ if its* unification degree, *$\nu_{\mathcal{R}}(\theta(\mathcal{E}_1), \theta(\mathcal{E}_1))$, defined as $\nu_{\mathcal{R}}(\theta(\mathcal{E}_1), \theta(\mathcal{E}_1)) = \mathcal{R}(\theta(\mathcal{E}_1), \theta(\mathcal{E}_1))$, is greater than zero. When the unification degree $\nu_{\mathcal{R}}(\theta(\mathcal{E}_1), \theta(\mathcal{E}_1)) = \lambda > 0$ we also say that θ is a λ-unifier of $\mathcal{E}_1$ and $\mathcal{E}_2$.*

[2] We mean by "expression" a first order term or an atomic formula.

Definition 4. *Let $\mathcal{R}$ be a similarity relation. The substitution θ is* more general *than the substitution σ* with level λ, *denoted by $\theta \leq_{\mathcal{R},\lambda} \sigma$, if there exist a substitution δ such that, for any variable x in the domain of θ or σ, $\mathcal{R}(\sigma(x), \delta \circ \sigma(x)) \geq \lambda$ (that is, $\sigma(x) \equiv_{\mathcal{R},\lambda} \delta \circ \sigma(x)$).*

Definition 5. *Let $\mathcal{R}$ be a similarity relation and $\mathcal{E}_1$ and $\mathcal{E}_2$ two expressions. The substitution θ is a* weak more general unifier *(w.m.g.u.) of $\mathcal{E}_1$ and $\mathcal{E}_2$ w.r.t $\mathcal{R}$, denoted by $wmgu(\mathcal{E}_1, \mathcal{E}_2)$, if:*

1. *θ is a λ-unifier of $\mathcal{E}_1$ and $\mathcal{E}_2$; and*
2. *$\theta \leq_{\mathcal{R},\lambda} \sigma$, for any λ-unifier σ of $\mathcal{E}_1$ and $\mathcal{E}_2$.*

The weak unification algorithm we are going to present is a reformulation of the one appeared in [21], with the advantage of a clearer operational reading. It is an extension of Martelli and Montanari's unification algorithm for syntactic unification [2, 11, 15] and it is based in the following observation: The task of obtaining a w.m.g.u of two expressions $\mathcal{E}_1 = f(t_1, \ldots, t_n)$ and $\mathcal{E}_2 = g(s_1, \ldots, s_n)$ with $\mathcal{R}(f, g) = \alpha > 0$, where $\mathcal{R}$ is a similarity relation, is not a failure but it is equivalent to solve the (initial) set of equations $G = \{t_1 \sim s_1, \ldots, t_n \sim s_n\}$ coupled with the similarity degree α. Here, the symbol "$\sim$" represents the possibility that the arguments in $\mathcal{E}_1$ and $\mathcal{E}_2$ be equals by similarity.

The weak unification algorithm is formalized as a transition system based on a similarity-based unification relation "$\Rightarrow$". The unification of the expressions $\mathcal{E}_1$ and $\mathcal{E}_2$ is obtained by a state transformation sequence starting from an initial state $\langle G, id, \alpha \rangle$, where id is the identity substitution:

$$\langle G, id, \alpha \rangle \Rightarrow \langle G1, \theta_1, \alpha_1 \rangle \Rightarrow \langle G2, \theta_2, \alpha_2 \rangle \Rightarrow \ldots \Rightarrow \langle G_n, \theta_n, \alpha_n \rangle.$$

When the final state $\langle G_n, \theta_n, \alpha_n \rangle$, with $G_n = \emptyset$, is reached (i.e., the equations in the initial state have been solved), the expressions $\mathcal{E}_1$ and $\mathcal{E}_2$ are unifiable by similarity with w.m.g.u. θ_n and unification degree α_n. Therefore, the final state $\langle \emptyset, \theta_n, \alpha_n \rangle$ signals out the unification success. On the other hand, when expressions $\mathcal{E}_1$ and $\mathcal{E}_2$ are not unifiable, the state transformation sequence ends with failure (i.e., $G_n = Fail$).

The next definition provides the set of rules defining the similarity-based unification relation "$\Rightarrow$".

Definition 6 (similarity-based unification relation). *Let $\mathcal{R}$ be a similarity relation. The* similarity-based unification relation*, "$\Rightarrow$", is the smallest relation defined by the following set of transition rules:*

1. `Term decomposition`*:*

$$\frac{\langle \{f(t_1, \ldots, t_n) \sim f(s_1, \ldots, s_n)\} \cup E, \theta, \alpha \rangle}{\langle \{t_1 \sim s_1, \ldots, t_n \sim s_n\} \cup E, \theta, \alpha \rangle}$$

2. `Term decomposition by similarity`*:*

$$\frac{\langle \{f(t_1, \ldots, t_n) \sim g(s_1, \ldots, s_n)\} \cup E, \theta, \alpha \rangle \ , \ R(f, g) = \beta > 0}{\langle \{t_1 \sim s_1, \ldots, t_n \sim s_n\} \cup E, \theta, (\alpha \wedge \beta) \rangle}$$

3. `Removal of trivial equations`*:*

$$\frac{\langle \{X \sim X\} \cup E, \theta, \alpha\rangle}{\langle E, \theta, \alpha\rangle}$$

4. `Swap`*:*

$$\frac{\langle \{t \sim X\} \cup E, \theta, \alpha\rangle}{\langle \{X \sim t\} \cup E, \theta, \alpha\rangle}, \qquad \textit{if } t \textit{ is not a variable.}$$

5. `Variable elimination`*:*

$$\frac{\langle \{X \sim t\} \cup E, \theta, \alpha\rangle}{\langle \{X/t\}(E), \{X/t\} \circ \theta, \alpha\rangle}, \quad \textit{if the variable } X \textit{ does not occur in } t.$$

6. `Failure rule`*:*

$$\frac{\langle \{f(t_1, \ldots, t_n) \sim g(s_1, \ldots, s_n)\} \cup E, \theta, \alpha\rangle \ , \ R(f, g) = 0}{\langle Fail, \theta, \alpha\rangle}$$

7. `Occur check`*:*

$$\frac{\langle \{X = t\} \cup E, \theta, \alpha\rangle}{\langle Fail, \theta, \alpha\rangle}, \qquad \textit{if the variable } X \textit{ does occur in } t.$$

In the rules above, E denotes a set of (remaining) equational goals in the preceding state.

Note that, when the similarity relation $\mathcal{R}$ is the diagonal relation[3], the new algorithm conforms with the classical unification algorithm. Note also that, case 2 subsumes case 1, since by definition of similarity relation $\mathcal{R}(a, a) = 1$, for any symbol in the alphabet, however we let case 1 by efficiency reasons.

In general, the weak unification algorithm allows us to check if a set of expressions $S = \{\mathcal{E}_1 \sim \mathcal{E}'_1, \ldots, \mathcal{E}_n \sim \mathcal{E}'_n\}$ is weak unifiable. The w.m.g.u. of the set S is denoted by $wmgu(S)$. The following example illustrates the behavior of the weak unification algorithm.

Example 2. Let $\mathcal{R}$ be a similarity relation such that $\mathcal{R}(p, q) = 0.6$, $\mathcal{R}(b, d) = 0.3$, $\mathcal{R}(e, c) = 0.4$ and $\mathcal{R}(r, s) = 0.5$. In order to determine whether the set $S = \{p(X, Y, b) \sim q(a, b, d), r(Z, e) \sim s(T, c)\}$ is unifiable by similarity, we proceed as follows:

1. We build the initial state configuration:

$$\langle \{p(X, Y, b) \sim q(a, b, d), r(Z, e) \sim s(T, c)\}, id, 1\rangle$$

[3] That is, $\mathcal{R}(a, a) = 1$ and $\mathcal{R}(a, b) = 0$ (being a and b distinct symbols in the alphabet).

2. Then, we apply the unification rules in Definition 6, until a success or fail state is reached:

$$\begin{array}{ll}
\langle \{p(X,Y,b) \sim q(a,b,d), r(Z,e) \sim s(T,c)\}, id, 1\rangle & \Rightarrow_2 \\
\langle \{X \sim a, Y \sim b, b \sim d, r(Z,e) \sim s(T,c)\}, id, 0.6\rangle & \Rightarrow_5 \\
\langle \{Y \sim b, b \sim d, r(Z,e) \sim s(T,c)\}, \{X/a\}, 0.6\rangle & \Rightarrow_5 \\
\langle \{b \sim d, r(Z,e) \sim s(T,c)\}, \{X/a, Y/b\}, 0.6\rangle & \Rightarrow_2 \\
\langle \{r(Z,e) \sim s(T,c)\}, \{X/a, Y/b\}, 0.3\rangle & \Rightarrow_2 \\
\langle \{Z \sim T, e \sim c\}, \{X/a, Y/b\}, 0.3\rangle & \Rightarrow_5 \\
\langle \{e \sim c\}, \{X/a, Y/b, Z/T\}, 0.3\rangle & \Rightarrow_2 \\
\langle \{\}, \{X/a, Y/b, Z/T\}, 0.3\rangle &
\end{array}$$

Since the final state is a successful configuration, the initial set of equational goals S is unifiable with w.m.g.u. $\{X/a, Y/b, Z/T\}$ and unification degree 0.3.

Given a similarity relation $\mathcal{R}$ on a first order alphabet, it is possible to prove that if $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ is a w.m.g.u. of two expressions $\mathcal{E}_1$ and $\mathcal{E}_2$, with unification degree λ, then whatever substitution $\theta' = \{x_1/s_1, \ldots, x_n/s_n\}$, holding that $s_i \equiv_{\mathcal{R},\lambda} t_i$, for any $1 \leq i \leq n$, is also a w.m.g.u. of $\mathcal{E}_1$ and $\mathcal{E}_2$ with unification degree λ. Therefore, the weak unification algorithm computes a representative of a w.m.g.u. class.

3 Similarity-Based SLD Resolution

Let Π be a set of Horn clauses and $\mathcal{R}$ a similarity relation on the first order alphabet induced by Π. We define *Weak SLD* (WSLD) *resolution* as a transition system $\langle E, \Longrightarrow_{WSLD}\rangle$ where E is a set of triples $\langle \mathcal{G}, \theta, \alpha\rangle$ (goal, substitution, approximation degree), that we call the *state* of a computation, and whose transition relation $\Longrightarrow_{WSLD} \subseteq (E \times E)$ is the smallest relation which satisfies:

$$\frac{\mathcal{C} = (\mathcal{A} \leftarrow \mathcal{Q}) \ll \Pi, \sigma = wmgu(\mathcal{A},\mathcal{A}') \neq fail,\ \lambda = \nu_{\mathcal{R}}(\sigma(\mathcal{A}),\sigma(\mathcal{A}'))}{\langle (\leftarrow \mathcal{A}',\mathcal{Q}'), \theta, \alpha\rangle \ \Longrightarrow_{WSLD}\ \langle \leftarrow \sigma(\mathcal{Q}, \mathcal{Q}'), \sigma \circ \theta, \lambda \wedge \alpha\rangle}$$

where $\mathcal{Q}$, $\mathcal{Q}'$ are conjunctions of atoms and the notation "$\mathcal{C} \ll \Pi$" is representing that $\mathcal{C}$ is a standardized apart clause in Π.

A WSLD derivation for $\Pi \cup \{\mathcal{G}_0\}$ is a sequence of steps

$$\langle \mathcal{G}_0, id, 1\rangle \Longrightarrow_{WSLD} \langle \mathcal{G}_1, \theta_1, \lambda_1\rangle \Longrightarrow_{WSLD} \ldots \Longrightarrow_{WSLD} \langle \mathcal{G}_n, \theta_n, \lambda_n\rangle.$$

And a WSLD refutation is a WSLD derivation $\langle \mathcal{G}_0, id, 1\rangle \Longrightarrow_{WSLD}{}^{*} \langle \Box, \sigma, \lambda\rangle$, where σ is a computed answer and λ is its *approximation degree*. Certainly, a WSLD refutation computes a family of answers, in the sense that, if $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ then whatever substitution $\theta' = \{x_1/s_1, \ldots, x_n/s_n\}$, holding that $s_i \equiv_{\mathcal{R},\lambda} t_i$, for any $1 \leq i \leq n$, is also a computed answer with approximation degree λ.

4 S-Prolog: Syntax and Semantics

The language we call S-Prolog is an extension of the pure Prolog language with a similarity relation defined on a syntactic domain. Therefore, the syntax of the extended language is easy. It is just the Prolog syntax but enriched with a built-in symbol "$\sim$" used for describing similarity relations by means of *similarity equations* of the form:

```
    <alphabet symbol> ~ <alphabet symbol> = <similarity degree>
```

meaning that two constants, n-ary function symbols or n-ary predicate symbols are similar with a certain degree. More precisely, we use the built-in symbol "$\sim$" as a compressed notation for the symmetric closure of an arbitrary fuzzy binary relation $\mathcal{R}$ (that is, a similarity equation $a \sim b = \alpha$ can be understood in both directions: a is similar to b and b is similar to a with degree α). The user can supply an initial subset of similarity equations and then, the system generates a reflexive and transitive closure to obtain a similarity relation. Hence, a S-Prolog program is a sequence of Prolog facts and rules followed by a sequence of similarity equations.

Example 3. This S-Prolog program fragment specify features and preferences on books stored in a data base. The preferences are specified by means of similarity equations[4]:

```
% FACTS
adventures(treasure_island).
adventures(the_call_of_the_wild).
mystery(the_murders_in_the_rue_morgue).
horror(dracula).
science_fiction(the_city_and_the_stars).
science_fiction(the_martian_chronicles).

% RULES
good(X) :- interesting(X).

% SIMILARITY EQUATIONS
% Direct connections
adventures ~ mystery = 0.5
adventures ~ science_fiction = 0.8
adventures ~ interesting = 0.9
mystery ~ horror = 0.9
mystery ~ science_fiction = 0.5
science_fiction ~ horror = 0.5

% Transitive connections
```

[4] In order to facilitate later discussions, we explicitly give the similarity equations that complete the transitive closure of the initial fuzzy binary relation.

```
adventures ~ horror = 0.5
mystery ~ interesting = 0.5
interesting ~ horror = 0.5
science_fiction ~ interesting = 0.8
```

The operational semantics of S-Prolog conforms with the similarity-based SLD principle [21] as it is defined in Section 3. Therefore, S-Prolog computes answers as well as approximation degrees.

5 The Similarity WAM Machine

In this section we present the main features of the Similarity WAM machine (SWAM), a virtual machine for executing S-Prolog programs. Roughly speaking, it is an adaptation of the classical WAM, as described in [1], able to execute a Prolog program in the context of a similarity relation defined on the first order alphabet induced by that program. As we shall show, the SWAM uses an operational mechanism that conforms the weak SLD principle.

5.1 Structure and behavior of the machine

The structure of the S-Prolog compiler is depicted in Figure 1, being the SWAM machine the basis for the compiler implementation. The *Analyzer* performs a syntactical analysis and, at the same time, it translates the source program into an internal representation. The *Adapter* takes that internal representation and it obtains some auxiliary representations that facilitate the code generation task. The *Code Generator* produces the machine code associated to the source program. All these phases, except the one related with the Adapter, have been implemented following standard techniques described in [1].

Once the machine code is generated, it is stored in the Code Area, an addressable array of memory words. One or more memory words may contain a possibly labeled instruction consisting of an operation code followed by operands. Labels are symbolic entry points into the Code Area which are used by *control instructions* to alter the standard sequential execution order of machine instructions. Additionally, multi-labels are also used with other purposes, such as to guide some stages of the weak unification process (see below in the next section and Section 5.4). On the other hand, the similarity relation is stored into the *Similarity Matrix* memory area and its information is used: i) at compilation time, by the Adapter (see Section 5.3). ii) at execution time, when it is necessary during the unification process. An array of memory cells, which is called the *Heap*, is used to represent atoms and terms internally. Explicit *tags* are used to distinguish variables from other sort of structures. The allocation of structures in the Heap is started by the execution of the machine code instructions obtained when the query is compiled. The *Stack* stores control information. It contains procedure activation frames, called *environments*, as well as choice point frames. Since the size of these frames is variable, the stack is organized as a linked list. The

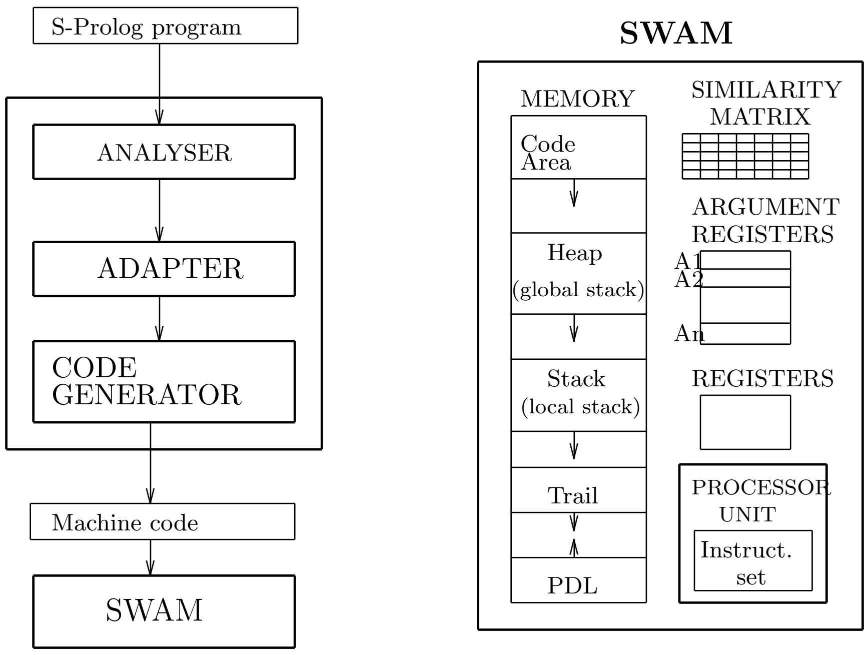

Fig. 1. S-Prolog compiler structure and SWAM structure

Trail maintains a record of those variables that need to be reset to "unbound" upon backtracking. It is represented by an array which is managed as a stack. Registers are pointers to the arguments of expressions (stored in the Heap) to be unified or pointers to the Code Area, the Heap, the Stack or the Trail, used for control purposes. Also, we declare a new global register to accumulate the computed approximation degree during the weak unification process.

In the sequel, we shall comment more deeply the main points where the SWAM design diverges from the standard WAM, but before doing that it is necessary to introduce a note about how syntactical unification of expressions is performed.

5.2 Standard versus weak unification

It is noteworthy that the syntactical unification algorithm is implemented into the WAM as a distributed procedure which includes two phases:

Phase 1 Unification of the predicate symbol rooting a (sub)goal and the heads of the clauses defining that predicate. This unification stage is immediate and produces a choice point. From the code generation point of view, it mainly produces the following set of machine instructions: `try_me_else`, `retry_me_else` and `trust_me` (when the program clauses are compiled).

Phase 2 Unification of the corresponding arguments of the (sub)goal and the clause heads being unified. From the code generation point of view, the visible effect is a set of machine instructions: `get_structure`, `unify_variable` and `unify_value` (when the program clauses are compiled). However, in this phase, the argument unification is not tested immediately, but at execution time by specific code portions inside the `get_structure` instruction and the procedure `unify` which is called by the `unify_value` instruction.

Therefore, if we want to introduce weak unification into a WAM context, it is necessary to modify both phases of the distributed unification procedure above described.

Phase 1 This phase controls the "flexible" matching of predicate symbols during the unification process when programs are augmented with similarity relations. This is a critical phase, since it is not obvious at a first glance how to proceed. We see that it requires the introduction of a program transformation step, which transforms the original program into a set of "clauses" whose bodies contain information about the similarity degree between predicate symbols. The Adapter carry out the transformation and manages it in order to facilitate the code generation. Section 5.3 describes the main features of the transformation.

Phase 2 The adaptation of this phase is easy. It only requires the modification of some portions of the machine instruction `get_structure` and the procedure `unify`, in order to perform a "flexible" matching of function and constant symbols which is guided by the similarity equations.

Finally, note that, when unifying expressions in presence of similarity relations it is necessary to store, as a part of the computation state, the current computed approximation degree. To cover this task, we use a specific global register in the SWAM which works as an accumulator register. We call this, the `AD` register.

5.3 The Adapter and the first phase of the weak unification procedure

In a logic program, a predicate p is defined by the set of clauses whose head is rooted by p. However, in a logic program extended with a similarity relation, a clause defining the predicate p can also be considered as defining each predicate q which is similar to p. On the other hand, as it was commented, the structure of the WAM is designed to test a "crisp" matching of predicate symbols. Therefore, if we want a "flexible" matching of predicate symbols without forcing the structure of the WAM, given a clause defining a predicate p, we need to introduce a new clause for each predicate q which is similar to p. We do it in order to simulate a "flexible" matching with a "crisp" technique.

The following definition formalizes the program transformation performed by the Adapter. We need to introduce an extended language obtained by adding to the object language alphabet the elements of the lattice $[0, 1]$ (of similarity degrees). Clauses in this extended language contain bodies with literals which are similarity degrees. We call these clauses 'e-clauses'. Also e-clauses with an empty head are called 'e-goals'.

Definition 7. *Let Π be a logic program and $\mathcal{R}$ a similarity relation on the syntactic domain generated by Π. Let $p(t_1, \dots, t_n) \leftarrow \mathcal{Q}$ be a clause in Π defining the n-ary predicate p. Then, for each $\mathcal{R}(p, q) = \alpha > 0$ add to the transformed program Π' the new e-clause $q(t_1, \dots, t_n) \leftarrow \alpha, \mathcal{Q}$. Hence, the transformed program $\Pi' = \{q(t_1, \dots, t_n) \leftarrow \alpha, \mathcal{Q} \mid (p(t_1, \dots, t_n) \leftarrow \mathcal{Q}) \in \Pi$ and $\mathcal{R}(p, q) = \alpha > 0\}$.*

Observe that, since $\mathcal{R}(p, p) = 1$ for any symbol p, if $p(t_1, \dots, t_n) \leftarrow \mathcal{Q}$ is in the original program, the e-clause $p(t_1, \dots, t_n) \leftarrow 1, \mathcal{Q}$ will be in the transformed program. Thus we give an uniform treatment for all clauses in the transformed program.

We illustrate the behavior of the transformation by means of the following two examples.

Example 4. The following is a fragment of the transformed program obtained when Definition 7 is applied to the clause `adventures(treasure_island)` in the original program of Example 3:

```
adventures(treasure_island):-1.0.
mystery(treasure_island):-0.5.
science_fiction(treasure_island):-0.8.
interesting(treasure_island):-0.9.
horror(treasure_island):-0.5.
```

Example 5. A fragment of the transformed program showing the clauses defining the predicate `adventures`:

```
adventures(treasure_island):-1.0.
adventures(the_call_of_the_wild):-1.0.
adventures(the_murders_in_the_rue_morge):-0.5.
adventures(dracula):-0.5.
adventures(the_city_and_the_stars):-0.8.
adventures(the_martian_chronicles):-0.8.
```

5.4 Compilation of S-Prolog Programs

The compilation of the transformed program to machine code is done using standard techniques. In essence, clauses of a transformed program are translated into the same machine instruction set a standard implementation would have produced. The only difference is that similarity degrees, in the body of transformed clauses, are "stored" in a multi-label field of the `try_me_else`, `retry_me_else` and `trust_me` machine instructions. The values in the multi-label field will be used during the computation of the unification degree in a WSDL resolution step.

Example 6. The following shows the compiled code for the program fragment of Example 5:

```
0  : adventures/1 [ 1.0 ] :try_me_else 3
1  : get_structure treasure_island 0,1
2  : proceed
3  : [ 1.0 ] :retry_me_else 6
4  : get_structure the_call_of_the_wild 0,1
5  : proceed
6  : [ 0.5 ] :retry_me_else 9
7  : get_structure the_murders_in_the_rue_morge 0,1
8  : proceed
9  : [ 0.5 ] :retry_me_else 12
10 : get_structure dracula/0,1
11 : proceed
12 : [ 0.8 ] :retry_me_else 15
13 : get_structure the_city_and_the_stars 0,1
14 : proceed
15 : [ 0.8 ] :trust_me
16 : get_structure the_martian_chronicles 0,1
17 : proceed
```

In general, given an adapted program, defining a predicate *p*:

```
p :- 1.0, Q_1
p :- alpha_j, Q_j
.
.
p :- alpha_m, Q_m
p :- alpha_n, Q_n
```

where the `alpha_i` are similarity degrees and the `Q_i` are conjunction of atoms, it is translated into the following set of machine instructions:

```
Li  : adventures/1 [ 1.0 ] :try_me_else Lj
     % code for the arguments in the head atom p
     % code for the body atoms in Q_1
     proceed
Lj  : [ alpha_j ] :retry_me_else Lk
     % code for the arguments in the head atom p
     % code for the body atoms in Q_j
     proceed
Lk  :  . . .
.
.
Lm : [ alpha_m ] :retry_me_else Ln
     % code for the arguments in the head atom p
     % code for the body atoms in Q_m
     proceed
Ln : [ alpha_n ] :trust_me
```

```
% code for the arguments in the head atom p
% code for the body atoms in Q_n
proceed
```

5.5 Specific machine instructions for approximation degree control

In this section we describe how the SWAM controls the computation of the approximation degree when a choice point is created. In order to accomplish this task properly, we need:

1. to introduce a global register, called the `AD` register, to store the approximation degree computed at each WDSL resolution step;
2. to modify the standard choice point frame structure by adding a new field to save the value stored in the `AD` register, prior to the creation of a choice point; this is because, when the computation backtracks and the next clause in an alternative is taken, we need to restart the computation (of the approximation degree) at the point it was left before the former clause was try.

A choice point frame is used to save the computation state, that is, all the information required to continue the computation upon backtracking. The figure shows the structure of a choice point frame, where the additional field to save the `AD` register value is placed at the end of the frame.

B	`n` (number of arguments)
B+1	`A_1` (argument register 1)
	⋮
B+n	`A_n` (argument register n)
B+n+1	`CE` (continuation environment)
B+n+2	`CP` (program continuation pointer)
B+n+3	`B` (previous choice point)
B+n+4	`BP` (next clause in the choice point)
B+n+5	`TR` (trail pointer)
B+n+6	`H` (heap pointer)
B+n+7	`AD` **(approximation degree register)**

A detailed explanation of the role played by each standard field and the reason for which we must to save them is given in [1].

As the choice point frame structure has been modified, the machine instructions that work in combination with it need also to be modified. In the following we briefly comment the functionality of these instructions, specially, regarding with the control of the approximation degree.

The `try_me_else` machine instruction builds a new choice point frame on top of the stack, setting its fields according to the current context. Certainly, it stores the current value of the `AD` register.

```
private void try_me_else(int L) {
int newB;
if( e > b ) newB = e + heap[e+2].direccion() + 3;
else newB = b + heap[b].direccion() + 8; //<===
heap[newB]=new Termino("num_of_args",num_of_args);
int N=heap[newB].direccion();
for(int i=1 ; i <= N ; i++ )
heap[newB+i]=new Termino(heap[i].etiqueta(),heap[+i].direccion());
heap[newB+N+1]=new Termino("e",e);
heap[newB+N+2]=new Termino("cp",cp);
heap[newB+N+3]=new Termino("b",b);
heap[newB+N+4]=new Termino("L",L);
heap[newB+N+5]=new Termino("tr",tr);
heap[newB+N+6]=new Termino("h",h);
heap[newB+N+7]=new Termino("ad",(double)ad); //<===
// stores the current value of the {\tt AD} register
b=newB;
hb=h;
p=p+1;
}
```

When the computation backtracks, the `retry_me_else` instruction resets all the necessary informations from the current choice point frame. Specifically, the value which the `AD` register had, at the time the choice point frame was created, is restored. Then it is set as the minimum of its value and the similarity degree “stored” at the multi-label field of the `retry_me_else` instruction.

```
private void retry_me_else(int L) {
int N=heap[b].direccion();
for(int i=1; i <= N ; i++ ) {
heap[i].setEtiqueta(heap[b+i].etiqueta());
heap[i].setDireccion(heap[b+i].direccion());
}
e=heap[b+N+1].direccion();
cp=heap[b+N+2].direccion();
heap[b+N+4]=new Termino("",L);
unwind_trail(heap[b+N+5].direccion(),tr);
tr=heap[b+N+5].direccion();
h=heap[b+N+6].direccion();
ad=heap[b+N+7].rdireccion(); //<===
ad=min(ad,Double.parseDouble(actual.etiqueta().ad()));  //<===
// the {\tt AD} register is restored and then set
hb=h;
p=p+1;
}
```

The `trust_me` instruction behaves in a similar way as the `retry_me_else` instruction does. The only difference is that the former discards the current choice point frame after the context (including the `AD` register) have been restored.

```
private void trust_me(int L) {

int N=heap[b].direccion();
for(int i=1; i <= N ; i++ ) {
heap[i].setEtiqueta(heap[b+i].etiqueta());
heap[i].setDireccion(heap[b+i].direccion());
}
e=heap[b+N+1].direccion();
cp=heap[b+N+2].direccion();
unwind_trail(heap[b+N+5].direccion(),tr);
tr=heap[b+N+5].direccion();
h=heap[b+N+6].direccion();
ad=heap[b+N+7].rdireccion();  //<===  restores the AD register
// content before destroying the current choice point frame
ad=min(ad,Double.parseDouble(actual.etiqueta().ad())); //<===
// the {\tt AD} register is set
B=heap[b+N+3].direccion();
hb=heap[b+N+6].direccion();
p=p+1;
}
```

Finally, note that, since we have altered the size of a choice point frame, by adding a new field to save the `AD` register, the machine instruction `allocate` must be slightly modified. This is because `allocate` builds a new environment frame on the top of the stack and the top of the stack is computed differently depending on whether an environment or choice point frame is the last pushed structure on the stack. On the other hand the machine instruction `deallocate` remains unchanged.

Summarizing, as for controlling the approximation degree computation, the SAM behaves as follows: the machine instruction `try_me_else` allocates a new choice point frame on the top of the stack, which contains an additional field to store the current value of the `AD` register. Afterwards, the `retry_me_else` and `trust_me` instructions restore the `AD` register value, when the computation backtracks, which is used to restart the computation at the point it was left.

5.6 Specific machine instructions for argument weak unification

Before ending this section, we comment the main features of machine instructions involved in the process of argument unification.

The machine instruction `get_structure`, presented below, tests the similarity of constant and function symbols of terms in predicate arguments. More precisely, a call `get_structure(f,n,A)` acts as follows (we explain the cases directly related with the weak unification process): if the heap cell referenced by the argument register `A` contains a structure (a STR tag) pointing to a function symbol `f` with arity `n` we are in the classical case. This signals out a successful unification step where the unification degree remains unchanged. However, when the heap cell referenced by the argument register `A` is pointing to a function symbol which is not syntactically equal to `f` but is similar to `f` with degree `alpha`, the unification degree is recomputed (as the minimum of its previous value and `alpha`) and stored into the `AD` register. Otherwise, the unification process fails and the procedure `backtrack` is called.

```
public void get_structure( String nombre , int aridad , int i ) {
   int addr = 0;
   int caso;
   boolean fail = false;
   addr = deref( i );
   if( heap[addr].etiqueta().equals("REF") ) caso=1;
   else {if( heap[addr].etiqueta().equals("STR") ) caso=2;
     else caso=3;}

   switch( caso ) {
        ...
     case 2:
     int a = heap[addr].direccion();
         if((heap[a].etiqueta()).equals(nombre) &&
                                heap[a].direccion()== aridad ) {
           s = a+1;
           modo = 0; //read mode
        }else { //<===  dealing with similarity relations
           Relacion r=similar(heap[a].etiqueta(),nombre);
           if( r!=null ) {
              ad=min(ad,r.ad());
              s = a+1;
              modo = 0;//read
            }else {
              fail=true;
              System.out.println("No\n");
            }
        }
     break;
        ...
   }
   if ( fail ) backtrack();
   else p=p+1;
   return;
}
```

Finally, the machine instruction `unify_value` calls the procedure `unify`, which carries out the other part of the argument unification process. The procedure `unify` implements the weak unification algorithm defined in Section 2.3.

```
private boolean unify( int a1 , int a2 ) {
   int aux1,aux2,d1,d2,v1,v2,n1,n2;
   String t1,t2,f1,f2;
   pdl.push(new Integer(a1));
   pdl.push(new Integer(a2));
   boolean fail=false;

   while( !(pdl.isEmpty() || fail) ) {
      aux1 = ((Integer)pdl.pop()).intValue();
      aux2 = ((Integer)pdl.pop()).intValue();
      d1 = deref( aux1 );
      d2 = deref( aux2 );
      if( d1 != d2 ) {
           t1 = heap[d1].etiqueta();
           v1 = heap[d1].direccion();
           t2 = heap[d2].etiqueta();
           v2 = heap[d2].direccion();

           if( t1.equals("REF") || t2.equals("REF") ) {
              bind(d1,d2);
           } else {
              f1 = heap[v1].etiqueta();
             n1 = heap[v1].direccion();
             f2 = heap[v2].etiqueta();
             n2 = heap[v2].direccion();
             if( (f1.equals(f2)) && ( n1 == n2 ) ) {
              for( int i=1 ; i < n1 ; i++ ) {
              pdl.push(new Integer(v1+i));
              pdl.push(new Integer(v2+i));
              }
             } else { //<=== dealing with similarity relations
                Relacion r=similar(f1,f2);
             if( r!=null ) {
                 ad=min(ad,r.ad());
             }else {
                 fail=true;
                }
             }
           }
        }
      }
      return fail;
}
```

6 SWAM Operational Semantics

This section formally describes the operational semantics of the SWAM, which is an adaptation of the WSLD resolution rule aiming to preserve the architecture of a standard WAM. The goal of this section is to establish the equivalence of the former operational mechanism and the WSLD resolution principle.

In the reminder of this section we shall work inside the framework of the extended language built by e-clauses and e-goals. Π' denotes a transformed program obtained by applying Definition 7 on a logic program Π equipped with a similarity relation $\mathcal{R}$.

Definition 8. *We define the SWAM operational semantics as a transition system $\langle E, \Longrightarrow_{SWAM}\rangle$ where E is a set of triples $\langle \mathcal{G}, \theta, \alpha\rangle$ (e-goal, substitution, approximation degree), and whose transition relation $\Longrightarrow_{SWAM} \subseteq (E \times E)$ is the smallest relation which satisfies:*

Rule 1:

$$\frac{\beta \in (0,1]}{\langle(\leftarrow \beta, \mathcal{Q}'), \theta, \alpha\rangle \Longrightarrow_{SWAM} \langle \leftarrow \sigma(\mathcal{Q}, \mathcal{Q}'), \theta, \beta \wedge \alpha\rangle}$$

Rule 2:

$$\frac{\begin{array}{c}(p(t_1, \ldots, t_n) \leftarrow \mathcal{Q}) \ll \Pi', \\ \sigma = wmgu(p(t_1, \ldots, t_n), p(s_1, \ldots, s_n)) \neq fail,\ \lambda_i = \nu_{\mathcal{R}}(\sigma(t_i), \sigma(s_i))\end{array}}{\langle(\leftarrow p(s_1, \ldots, s_n), \mathcal{Q}'), \theta, \alpha\rangle \Longrightarrow_{SWAM} \langle \leftarrow \sigma(\mathcal{Q}, \mathcal{Q}'), \sigma \circ \theta, (\bigwedge_{i=1}^{n} \lambda_i) \wedge \alpha\rangle}$$

where $\mathcal{Q}$, $\mathcal{Q}'$ are conjunctions of atoms.

In the sequel, we prove the semantic equivalence between the WSLD rule and the operational mechanism of Definition 8.

Lemma 1. *Given a logic program Π with a similarity relation $\mathcal{R}$, let Π' be the transformed program obtained by applying Definition 7. If there is a step*

$$\mathcal{S} = (\langle(\leftarrow p(s_1, \ldots, s_n), \mathcal{Q}'), \theta, \alpha\rangle \Longrightarrow_{WSLD} \langle \leftarrow \sigma(\mathcal{Q}, \mathcal{Q}'), \sigma \circ \theta, \lambda \wedge \alpha\rangle)$$

in Π, then there exists a derivation

$$\langle(\leftarrow p(s_1, \ldots, s_n), \mathcal{Q}'), \theta, \alpha\rangle \Longrightarrow_{SWAM}{}^{+} \langle \leftarrow \sigma(\mathcal{Q}, \mathcal{Q}'), \sigma \circ \theta, \lambda \wedge \alpha\rangle$$

in Π', which computes the same state.

Proof. If there is a step $\mathcal{S}$ in Π, is because there exists a clause $\mathcal{C} = (q(t_1, \ldots, t_n) \leftarrow \mathcal{Q})$ in Π such that

$$\sigma = wmgu(q(t_1, \ldots, t_n), p(s_1, \ldots, s_n)) = wmgu(\{t_1 \sim s_1, \ldots, t_n \sim s_n\}),$$

with approximation degree

$$\begin{aligned}\lambda &= \nu_{\mathcal{R}}(\sigma(q(t_1, \ldots, t_n)), \sigma(p(s_1, \ldots, s_n))) = \mathcal{R}(p, q) \wedge (\textstyle\bigwedge_{i=1}^{n} \mathcal{R}(\sigma(t_i), \sigma(s_i))) \\ &= \beta \wedge (\textstyle\bigwedge_{i=1}^{n} \lambda_i).\end{aligned}$$

On the other hand, by Definition 7, if $\mathcal{R}(p, q) = \beta > 0$, there is a clause $p(t_1, \ldots, t_n) \leftarrow \beta, \mathcal{Q}$ in Π'. Therefore, it is possible to construct the following derivation in Π':

$$\begin{aligned}\langle(\leftarrow p(s_1, \ldots, s_n), \mathcal{Q}'), \theta, \alpha\rangle &\Longrightarrow_{SWAM} \langle(\leftarrow \beta, \sigma(\mathcal{Q}, \mathcal{Q}')), \sigma \circ \theta, (\textstyle\bigwedge_{i=1}^{n} \lambda_i) \wedge \alpha\rangle \\ &\Longrightarrow_{SWAM} \langle\leftarrow \sigma(\mathcal{Q}, \mathcal{Q}'), \sigma \circ \theta, \lambda \wedge \alpha\rangle\end{aligned}$$

since, $wmgu(p(t_1, \ldots, t_n), p(s_1, \ldots, s_n)) = wmgu(\{t_1 \sim s_1, \ldots, t_n \sim s_n\}) = \sigma$, and $\nu_{\mathcal{R}}(\sigma(t_i),\sigma(s_i)) = \mathcal{R}(\sigma(t_i), \sigma(s_i)) = \lambda_i$.

The following proposition establishes a kind of completeness result where we prove that derivations in the original program using the WSLD resolution rule can be reproduced by the SWAM operational mechanism in the transformed program.

Proposition 2. *Given a logic program Π with a similarity relation $\mathcal{R}$, let Π' be the transformed program obtained by applying Definition 7. If there exists a derivation $\mathcal{D} = (\langle(\leftarrow \mathcal{Q}'_0), \theta_0, \alpha_0\rangle \Longrightarrow_{WSLD}{}^* \langle\leftarrow \mathcal{Q}'_n, \theta_n, \alpha_n\rangle)$ in Π, then there exists a derivation $\langle(\leftarrow \mathcal{Q}'_0), \theta_0, \alpha_0\rangle \Longrightarrow_{SWAM}{}^* \langle\leftarrow \mathcal{Q}'_n, \theta_n, \alpha_n\rangle$ in Π', which computes the same state.*

Proof. By induction on the length of the derivation $\mathcal{D}$ and Lemma 1.

Now we proceed by demonstrating the reverse of the last proposition, which constitute a kind of soundness result.

Lemma 2. *Given a logic program Π with a similarity relation $\mathcal{R}$, let Π' be the transformed program obtained by applying Definition 7. If there exists a derivation $\mathcal{D}'$:*

$$\begin{aligned}\langle(\leftarrow p(s_1, \ldots, s_n), \mathcal{Q}'), \theta, \alpha\rangle &\Longrightarrow_{SWAM} \langle(\leftarrow \beta, \sigma(\mathcal{Q}, \mathcal{Q}')), \sigma \circ \theta, (\textstyle\bigwedge_{i=1}^{n} \lambda_i) \wedge \alpha\rangle \\ &\Longrightarrow_{SWAM} \langle\leftarrow \sigma(\mathcal{Q}, \mathcal{Q}'), \sigma \circ \theta, \lambda \wedge \alpha\rangle\end{aligned}$$

in Π', with $\lambda_i = \nu_{\mathcal{R}}(\sigma(t_i),\sigma(s_i))$, then there is a step $\mathcal{S}$:

$$\langle(\leftarrow p(s_1, \ldots, s_n), \mathcal{Q}'), \theta, \alpha\rangle \Longrightarrow_{WSLD} \langle\leftarrow \sigma(\mathcal{Q}, \mathcal{Q}'), \sigma \circ \theta, \lambda \wedge \alpha\rangle$$

in Π', with $\lambda = \beta \wedge \bigwedge_{i=1}^{n} \lambda_i$.

Proof. If the first step of derivation $\mathcal{D}'$ is performed using the clause $\mathcal{C}' = (p(t_1, \ldots, t_n) \leftarrow \beta, \mathcal{Q})$ in Π' is because $\mathcal{R}(q, p) = \beta > 0$ and there exists a clause $\mathcal{C} = (q(t_1, \ldots, t_n) \leftarrow \mathcal{Q})$ in Π. Therefore, $wmgu(q(t_1, \ldots, t_n), p(s_1, \ldots, s_n)) = wmgu(\{t_1 \sim s_1, \ldots, t_n \sim s_n\}) = \sigma$, and $\nu_{\mathcal{R}}(\sigma(q(t_1, \ldots, t_n)),\sigma(p(s_1, \ldots, s_n))) = \mathcal{R}(p, q) \wedge (\bigwedge_{i=1}^{n} \mathcal{R}(\sigma(t_i), \sigma(s_i))) = \beta \wedge (\bigwedge_{i=1}^{n} \lambda_i) = \lambda$ and it is possible the WSLD step $\mathcal{S}$.

Proposition 3. *Given a logic program Π with a similarity relation $\mathcal{R}$, let Π' be the transformed program obtained by applying Definition 7. If there exists a derivation $\mathcal{D}' = (\langle(\leftarrow \mathcal{Q}'_0), \theta_0, \alpha_0\rangle \Longrightarrow_{SWAM}{}^* \langle\leftarrow \mathcal{Q}'_n, \theta_n, \alpha_n\rangle)$ in Π', then there exists a derivation $\langle(\leftarrow \mathcal{Q}'_0), \theta_0, \alpha_0\rangle \Longrightarrow_{WSLD}{}^* \langle\leftarrow \mathcal{Q}'_n, \theta_n, \alpha_n\rangle$ in Π, which computes the same state.*

Proof. By induction on the length of the derivation $\mathcal{D}'$. Without lost of generality we can assume that the steps in derivation $\mathcal{D}'$ are conveniently ordered to allow the application of Lemma 2.

The last proposition jointly with Proposition 2 state the equivalence of both operational mechanisms and the correctness of our implementation.

Although derivations in Π' have more steps than equivalent derivations in Π, observe that the unification effort and the number of choices in a choice point are reduced when executing a goal in the transformed program Π'. Hence, the SWAM operational mechanism is more efficient than a naive, direct implementation of the WSLD resolution rule.

7 Conclusions and Future Work

In this paper we have investigate how to incorporate the weak unification algorithm of [21] into the WAM, leading to a system well suited to be used for approximate reasoning and flexible query answering. We have presented the technical details that allow us to solve this problem:

1. We have designed a new pre-compilation phase, called the Adapter, which introduces some adaptations into the source code to facilitate the translation task. Also, the Adapter translates the original program into a transformed program, with explicit information about the similarity degree of predicates, that helps us to manage similarity relations properly.
2. We have appropriately modified some machine instructions to carry out the weak unification process. Mainly: `try_me_else`, `retry_me_else`, `trust_me`, `get_structure` and the procedure `unify`. A global register, the `AD` register, stores the result of computing the current approximation degree step by step.

As a result, we obtain a Prolog implementation based on similarity relations that we call S-Prolog. To the best of our knowledge this is the first WAM implementation that supports similarity-based SLD resolution.

At the present time, the SWAM is a prototype implementation useful to essay new compilation techniques. We have introduced algorithms to manage similarity relations, although this was not an objective of this work and we did not present them in this paper. However, the treatment of similarity relations is rather naive and it is necessary to implement more efficient algorithms to solve the transitive closure problem, what is left as a future work. Also we want to study how to combine, in our setting, the WSLD resolution rule with a concrete instance of the multi-adjoint logic programming framework described in [16–18].

References

1. H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, Cambridge, MA, 1991.

2. K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall, Englewood Cliffs, NJ, 1997.
3. J. F. Baldwin, T. P. Martin, and B. W. Pilsworth. *Fril- Fuzzy and Evidential Reasoning in Artificial Intelligence.* John Wiley &amp; Sons, Inc., 1995.
4. Francesca Arcelli Fontana and Ferrante Formato. Likelog: A logic programming language for flexible data retrieval. In *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC'99), February 28 - March 2, 1999, San Antonio, Texas, USA*, pages 260–267, 1999.
5. Francesca Arcelli Fontana and Ferrante Formato. A similarity-based resolution rule. *Int. J. Intell. Syst.*, 17(9):853–872, 2002.
6. Ferrante Formato, Giangiacomo Gerla, and Maria I. Sessa. Extension of logic programming by similarity. In Maria Chiara Meo and Manuel Vilares Ferro, editors, *APPIA-GULP-PRODE*, pages 397–410, 1999.
7. Ferrante Formato, Giangiacomo Gerla, and Maria I. Sessa. Similarity-based unification. *Fundam. Inform.*, 41(4):393–414, 2000.
8. S. Guadarrama, S. Muñoz, and C. Vaucheret. Fuzzy Prolog: A new approach using soft constraints propagation. *Fuzzy Sets and Systems, Elsevier*, 144(1):127–150, 2004.
9. M. Ishizuka and N. Kanai. Prolog-ELF Incorporating Fuzzy Logic. In Aravind K. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI'85). Los Angeles, CA, August 1985.*, pages 701–703. Morgan Kaufmann, 1985.
10. Frank Klawonn and Rudolf Kruse. A Łukasiewicz logic based Prolog. *Mathware & Soft Computing*, 1(1):5–29, 1994.
11. J.-L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca., 1988.
12. R.C.T. Lee. Fuzzy Logic and the Resolution Principle. *Journal of the ACM*, 19(1):119–129, 1972.
13. Deyi Li and Dongbo Liu. *A fuzzy Prolog database system.* John Wiley & Sons, Inc., 1990.
14. J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second edition.
15. A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.
16. J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Multi-adjoint logic programming with continuous semantics. In *Proc. of Logic Programming and Non-Monotonic Reasoning, LPNMR'01*, pages 351–364. Springer-Verlag, LNAI 2173, 2001.
17. J. Medina, M. Ojeda-Aciego, and P. Vojtáš. A procedural semantics for multi-adjoint logic programming. In *Proc. of Progress in Artificial Intelligence, EPIA'01*, pages 290–297. Springer-Verlag, LNAI 2258, 2001.
18. J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Similarity-based unification: a multi-adjoint approach. *Fuzzy Sets and Systems*, 146(1):43–62, 2004.
19. H.T. Nguyen and E.A. Walker. *A First Course in Fuzzy Logic.* Chapman & Hall/CRC, Boca Ratón, Florida, 2000.
20. B. Schweizer and A. Sklar. *Probabilistic Metric Spaces.* North-Holland, New York, 1983.
21. Maria I. Sessa. Approximate reasoning by similarity-based sld resolution. *Theoretical Computer Science*, 275(1-2):389–426, 2002.
22. P. Vojtas. Fuzzy Logic Programming. *Fuzzy Sets and Systems*, 124(1):361–370, 2001.

23. Peter Vojtas and L. Paulík. Soundness and completeness of non-classical extended SLD-resolution. In R. Dyckhoff et al, editor, *Proc. ELP'96 Leipzig*, pages 289–301. LNCS 1050, Springer Verlag, 1996.
24. David H. D. Warren. An Abstract Prolog Instruction Set. Technical note 309, SRI International, Menlo Park, CA., October, 1983.
25. L. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.
26. L. Zadeh. Calculus of fuzzy restrictions. *Fuzzy Sets and Their Applications to Cognitive and Decision Processes*, pages 1–39, 1975.