LiTH-IDA-R-88-01 ISSN-0281-4250

INSTITUTIONEN FÖR DATAVETENSKAP
UNIVERSITETET OCH TEKNISKA HÖGSKOLAN
LINKÖPING

Logic Programming with External Procedures: Introducing S-Unification

by

Jacek Leszczylowski*

Staffan Bonnier

Jan Maluszynski

** Department of Computer Science*
Iowa State University
Ames, Iowa 50011-1040, USA

RESEARCH REPORT
LOGPRO, January 1988

Postadress:
Institutionen för datavetenskap
Universitetet i Linköping och
Tekniska Högskolan
581 83 Linköping

Mailing address:
Department of Computer and
Information Science
Linköping University
S-581 83 Linköping, Sweden

Computer and Information Science Department
Linköping University

PhD theses:

* Available at: University of Microfilms Intl., 300 N. Zeeb Road, Ann Arbor, MI 48106, USA.

(*Linköping Studies in Science and Technology. Dissertations.*)

*No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981.

*No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981.

*No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982.

*No 94 **Hans Lunell:** Code Generator Writing Systems, 1983.

*No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983.

*No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation, 1984.

*No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons. 1987.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning. 1987.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems. 1987.

Licentiate of engineering Theses:

(*Linköping Studies in Science and Technology. Theses.*)

No 48 **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching. 1985.

No 52 **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems. 1985.

No 60 **Johan Fagerström:** Simulation and Evaluation of an Architecture based on Asynchronous Processes. 1985.

No 71 **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System. 1987.

No 72 **Tony Larsson:** On the Specification and Verification of VLSI Systems. 1986.

No 73 **Ola Strömfors:** A Structure Editor for Documents and Programs. 1986.

No 74 **Christos Levcopoulos:** New Results about the Approximation Behaviour of the Greedy Triangulation. 1986.

No 104 **Shamsul I. Chowdhury:** Statistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology. 1987.

No 108 **Rober Bilos:** Incremental Scanning and Token-based Editing. 1987

No 111 **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments. 1987.

No 113 **Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge. 1987.

No 118 **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs. 1987

LiTH-IDA-R-88-01 ISSN-0281-4250

Logic Programming with External Procedures: Introducing S-Unification

by

Jacek Leszczylowski

Staffan Bonnier

Jan Maluszynski

Abstract: A motivation for this work is the problem of re-usability of existing traditional software in logic programs. It can be viewed in an abstract way as the problem of amalgamation of Horn clause logic with a term reduction system whose rewrite rules are not accessible and thus cannot be used for construction of E-unifiers. Therefore we introduce a new unification algorithm, called S-unification, which is a special incomplete case of E-unification. It has the property that whenever it succeeds the result is a singleton complete set of E-unifiers of the arguments. It may also fail or report that it is not able to solve the problem of E-unification for given arguments. If the algorithm fails the actual arguments have no E-unifier. The paper discusses the problem of amalgamation of external functional procedures in a logic program and gives a characterization of a class of amalgamated programs for which S-unification is complete.

Postadress:
Institutionen för datavetenskap
Universitetet i Linköping och
Tekniska Högskolan
581 83 Linköping

Mailing address:
Department of Computer and
Information Science
Linköping University
S-581 83 Linköping, Sweden

1. INTRODUCTION

Though the Horn clause logic provides a universal computational paradigm [Kow,Tär] it seems often quite unnatural to express functions in the relational formalism. In recent years there have been a number of suggestions concerning combination of functional and logic programming in a unique framework (see e.g. [GrLi] or [BeLe] for a survey).

The approaches can be classified as:

(1) integrating existing programming languages and logic programs (well-known examples of this type are LOGLISP [RoSi], QLOG [Kom], POPLOG [MeHa] and APPLOG[Coh]);

(2) construction of new languages which allow to define functions and relations and to combine functional and relational definitions. (well-known examples are EQLOG [GoMe],LEAF [BBLM] and FUNLOG [SuYo]).

The main objective within the first approach is often to give access from logic programs to specific features of the underlying programming language, or programming environment. This aspect is usually more important than concern about the declarative semantics of the amalgamation. It may be rather difficult to give such a semantics if low-level features of the underlying system are accessible in the resulting language. On the other hand, many of the languages defined within the second approach have both declarative and operational semantics and some completeness results are also presented.

Our general perspective is different. We assume that we are given a logic programming language and a set of functional procedures written in some (not necessarily functional!) language. It is our belief that a good combination of these two systems should result in a language which is as "traditional" as possible. Our intention is that both the evalutation and the meaning of the functional procedures should be preserved in the combination, which opens for reusability of existing programs. On the other hand, we would like to give new programs written in the logic programming language access to those functional procedures. A basis for construction of an interface between the two systems is the assumption that terms are their common data structure. By imposing some more restrictions on those functional procedures we allow to be used, we can then conceptually think of the set of functional procedures as being defined by a canonical term rewriting system. This view allows us to give a clean declarative semantics to the amalgamated language in accordance to the theory of logic programming with equality (see e.g. [JLM]). However, from an operational point of view we can not use the rewrite rules for constructing E-unifiers, since we are not specific about the language of the functional procedures and hence have no access to the rewrite rules used by the system. In this paper we present a way for overcoming this difficulty. It is a new unification algorithm, called S-unification, which is a special incomplete case of E-unification. It has the property that whenever it succeeds the result is a singleton complete set of E-unifiers of the arguments. It may also fail or report that it is not able to solve the problem of E-unification for given arguments. If the algorithm fails the actual arguments have no E-unifier. As a side-effect of our work we get a generalization of Prolog "evaluable predicates" with well defined semantics.

2.PRELIMINARIES

2.1. Basic assumptions

Our intention is to re-use existing functional procedures in a logic programming system based on Horn clause logic with SLD resolution. We do not want to be too specific about the language in which the functional procedures are written. Ideally we want to be able to use procedures written in any language. There are however some minimal assumptions about the

syntax and the semantics of this language.
The logic programs we deal with are collections of Horn clauses. Those terms that appear as arguments of the litterals may or may not contain calls to functional procedures, which when evaluated yield terms as results. Thus our task is to construct an amalgamted system $\mathcal{TP}$ from a system $\mathcal{T}$, having terms as language and a semantics given a priori by the functional procedures, and a predicate system $\mathcal{P}$, having Horn clauses as sentences and a semantics based on Least Herbrand Model/SLD resolution. We require that the construction of $\mathcal{TP}$ preserves the semantics of $\mathcal{T}$ and $\mathcal{P}$ when considered separately.

We assume that the alphabets of $\mathcal{P}$ and $\mathcal{T}$ have the set *CSTR* of constructors in common, each constructor having arity zero or more. In addition, the alphabet of $\mathcal{P}$ includes the sets *VAR* and *PRED* of variables and predicates respectively, and the alphabet of $\mathcal{T}$ includes the set *FUNCT* of names of functional procedures, each having arity one or more. All these sets are disjoint. The alphabet of $\mathcal{TP}$ is *VAR, CSTR, FUNCT* and *PRED* $\cup$ $\{=\}$. We will also use the notation *TERM* for the set of all terms (built in the standard way over *VAR, CSTR*, and *FUNCT*), and *SUBS* for the set of functions from *VAR* to *TERM*. The terms built over just *VAR* and *CSTR* are called ***structures***. A term which begins with a procedure name is called a ***functional call***.

2.2. The Term Machine

As mentioned before, we assume that the semantics of $\mathcal{T}$ is given by some existing programming system which is able to evaluate ground functional calls. We view this system as an abstract term machine *TM*, on which we make two demands, the first of which is :

(1) For each ground (i.e. variable free) functional call t, $TM(t)$ is a ground structure.

This requirement guarantees that each procedure terminates when evaluated with all its arguments ground, producing a completely reduced term as value. In order to extend *TM* to arbitrary terms we define the procedure *RED* by :

```
RED (t) =
   if t∈VAR
   then t
   else let t=f(t1, ... ,tn) in
         if f∈FUNCT
         then if [for i=1 to n Ground(ti)]
                 then TM(f(t1, ... ,tn))
                 else f(RED(t1), ... ,RED(tn))
         else f(RED(t1), ... ,RED(tn))
```

We consider two terms t and t' to be equivalent, denoted $t =_E t'$, iff for every ground substitution δ of t and t', $RED(t\delta)$ and $RED(t'\delta)$ are syntactically equal. This equivalence relation extends to substitutions by $\theta =_E \sigma$ iff $v\theta =_E v\sigma$ for each $v \in VAR$. The second demand we make on *TM* is now :

(2) For each n-ary element f of *FUNCT*, $TM(f(t_1, \ldots, t_n)) = TM(f(t_1', \ldots, t_n'))$ whenever $t_i =_E t_i'$ and t_i, t_i' are ground for $1 \leq i \leq n$.

According to this requirement each procedure associated with some name in *FUNCT* will act like a function, and it is now possible to prove the following facts :

Fact :
For any terms t, t' and substitutions θ, θ':

(1) $t=_E t'$ and $\theta=_E \theta'$ implies $t\theta=_E t'\theta'$.

(2) $RED(t)\theta=_E RED(t\theta)$.

It is worth noticing that since we do not make any assumption about how the terms are going to evaluated, the term machine may very well be lazy. In order not to exclude untyped functional procedures, we relinquish to impose a type system on *TM*, although this could both increase efficiency [Nil] and facilate programming [Myc], [KoMa].
In the literature the term machine *TM* is often defined by a confluent and terminating set of equational axioms used as rewrite rules e.g. [GoTa]. It is easy to see that our approach admits such machines, but also many others. Indeed, we can now conceptually view the semantics of $\mathcal{T}$ as being defined by a set of nonaccessible equations, which when viewed as rewrite rules form a canonical rewrite rule system. In the sequel we will therefore allow ourselves to borrow some notation from the theory of term rewriting systems.

2.3. Amalgamation of Logic Programs and Functional Procedures

Introduction of the equivalence classes on terms opens for introduction of the equality predicate =, interpreted as the relation $=_E$. The question arises how to extend this interpretation for definite clauses employing the interpreted terms, and how to include the equality among the other predicates. We adopt the standard solution, which is the initial model as defined in e.g. [GoMe]. The next problem is how to represent and how to compute the elements of the model. For any term t, let $[t]$ denote the equivalence class to which t belongs. By the confluence and termination property of *RED* every element $([t_1], \ldots, [t_n])$ of an n-ary relation can uniquely be represented by the n-tuple of terms $(RED(t_1), \ldots, RED(t_n))$. It is also well known (see e.g. [JLM]), that the SLD-resolution can be extended for computing the relations specified by the extended logic programs. For this to be realized the unification should be replaced by a general equation solving, called E-unification. More precisely, an *E-unifier* of terms t and t' is a substitution θ such that the relation $t\theta=_E t'\theta$ holds. A set S of E-unifiers of terms t and t' is *complete* iff for each E-unifier θ of t and t' there exists some $\sigma \in S$ and substitution γ such that $\theta=_E \sigma\gamma$. In the case when *TM* is defined by a confluent set of rewrite rules there are some techniques based on narrowing, which sometimes make it possible to construct a complete set of E-unifiers or to decide that the terms are not E-unifiable. Generally every complete set of E-unifiers of two terms has more than one member, each of which should be handled separately.

3. AMALGAMATION

This section presents our approach to amalgamation of logic programs and functional procedures. As mentioned above, the programs of the language $\mathcal{TP}$ are finite sets of clauses of the form :

$$h \leftarrow b_1, \ldots, b_n$$

where h and all the b_i:s are atomic formulas over the alphabet of $\mathcal{TP}$, with h not being an equality. Concerning the operational semantics of $\mathcal{TP}$ we would generally like to follow the

traditional way, but the use of *TM* makes it impossible to give a general algorithm which given two terms enumerates a complete set of *E*-unifiers. In fact, by our premises we must confine ourselves with finding at most one *E*-unifier of any pair of *E*-unifiable terms. Since we are only interested in complete sets of *E*-unifiers, our approach is to carefully select some special cases when there exists a singleton complete set of *E*-unifiers of two terms. We also identify some other cases when the existence of any *E*-unifier is impossible. We use these cases to develop an incomplete *E*-unification algorithm which we call *S*-unification.

4. UNIFICATION

This section develops the *S*-unification algorithm. First we discuss the special cases of *E*-unification. Later we state and prove its correctness.

4.1. Special cases of *E*-unification

By the *disagreement set*, $DIS(t, t')$ of two terms t and t', we shall mean the set defined by :

$DIS(t, t') =$
  if $t \in VAR$ or $t' \in VAR$
  then if $t = t'$
    then ϕ
    else $\{(t,t')\}$
  else let $t = f(t_1, \ldots, t_n)$, $t' = f'(t_1', \ldots, t_m')$ **in**
    if $f \in FUNCT$ or $f' \in FUNCT$
    then if $t = t'$
      then ϕ
      else $\{(t,t')\}$
    else if $f = f'$
      then $\cup_i DIS(t_i, t_i')$
      else $\{(t, t')\}$

Notice that a term occuring in some pair of $DIS(t, t')$ *never* can derive its origin from a proper subterm of some functional call in t or t'.
Let us now study the relation between existence of *E*-unifiers of the terms t, t' and the set $D = DIS(RED(t), RED(t'))$ by considering the possible cases of elements (u, u') occuring in this set :

We first consider those cases when $u = f(t_1, \ldots, t_n)$ and $u' = f'(t_1', \ldots, t_m')$.

a) f and f' are constructors.

Since (u, u') are in the disagreement set, f and f' must be distinct. Moreover, by the definition of *DIS* u and u' can not be subterms of functional calls, and hence, no matter how the variables of t and t' are instantiated, we can never hope for them to reduce to the same value, i.e. no *E*-unifier of t and t' can exist. We call this case $Failcond1(D)$.

b) At least one of f and f' is a procedure name.

Since we applied *DIS* to the reduced versions of t and t', the one of u and u' which is a functional call must be nonground. Otherwise it would have been reduced to a ground

structure. Even though, in this case a purely syntactic unification may very well be possible [e.g. $(u, u')=(X+1, Y+Z)$], we have relinquished to use this information, since the unifier obtained for t and t' by unifying these terms rarely is unique. Instead we confine ourselves with the observation that this case is not a sufficient condition for t and t' *not* to be E-unifiable.

Now we consider those cases when one of u and u' is a variable X and the other one is any term τ.

c) X does not occur in τ.

In this case the substitution X/τ must be included in any E-unifier of t and t', possibly with τ further instantiated, and/or more reduced. This case is called $Substcond(X, \tau, D)$.

d) X does occur in τ.

If X occurs in τ outside every functional call, this reduces to the classical occur check failure which we call $Failcond2(D)$. If X occurs only inside functional calls being subterms of τ, there may still exist an E-unifier of t and t'. For example, let $\tau = s(f(X))$, where s is a constructor and f is a procedure name with $TM(f(s(a)))=a$ for some constant a, then $\{X/s(a)\}$ will E-unify X and τ. We call this a *fixpoint error*, since any E-unifier of t and t' must bind X to a fixpoint of the function $\lambda X.\ \tau(X)$, and generally we have no way of telling whether such a fixpoint exists.

From the definition of *DIS*, *Failcond1*, *Failcond2* and *Substcond* the following facts follows :

Fact :
For any terms t and t', with E-unifier θ :

(3) neither $Failcond1(DIS(t, t'))$ nor $Failcond2(DIS(t, t'))$ applies.

(4) if $(\tau, \tau')\epsilon DIS(t, t')$, then $\tau\theta=_E\tau'\theta$.

4.2. S-unification

Next we present the algorithm, which can be viewed as a function from $TERM \times TERM$ into $SUBST \cup \{FAIL\} \cup \{ERROR\}$:

$S\text{-}unify(t,\ t') =$
$\quad [\sigma_0 := \epsilon;$
$\quad u_0 := RED(t);$
$\quad u_0' := RED(t');$
$\quad D_0 := DIS(u_0,\ u_0');$
$\quad k := 0]$
$\quad$ **while** $D_k \neq \phi$ **do**
$\qquad$ **if** $Failcond1(D_k)$ or $Failcond2(D_k)$
$\qquad$ **then** $FAIL$
$\qquad$ **else if** $\exists(X,\ \tau)\ Substcond(X,\ \tau,\ D_k)$
$\qquad\quad$ **then** $[\sigma_{k+1} := \sigma_k\{X/\tau\};$
$\qquad\qquad u_{k+1} := RED(u_k\{X/\tau\});$
$\qquad\qquad u'_{k+1} := RED(u_k'\{X/\tau\});$
$\qquad\qquad D_{k+1} := DIS(u_k,\ u_k');$
$\qquad\qquad k := k+1]$
$\qquad\quad$ **else** $ERROR$

Before proving the correctness of the alogrithm we need the following lemma :

Lemma :

Let $t,\ t'$ be terms, and let σ_k, u_k and u_k' be defined as in $S\text{-}unify(t,\ t')$, then :

(1) $u_k =_E t\sigma_k$, $(u_k' =_E t'\sigma_k)$

(2) for any substitution γ, if θ is an E-unifier of t and t', $\theta =_E \sigma_k\gamma$ implies $u_k\gamma =_E u_k'\gamma$.

Proof :

Let $\mathcal{IH}(k) \Leftrightarrow u_k =_E t\sigma_k$.

From fact 2 follows that $\tau =_E RED(\tau)$, for any term τ. Hence :

I if $k = 0$, then $u_k = RED(t) =_E t = t\sigma_k$.

II Suppose $\mathcal{IH}(k)$ holds for some $k \geq 0$, and that $S\text{-}unify(t,\ t')$ defines σ_{k+1}, u_{k+1} and u'_{k+1}, then for some variable v and term τ, by fact 1 we have :

$$u_{k+1} = RED(u_k\{v/\tau\}) =_E u_k\{v/\tau\} =_E t\sigma_k\{v/\tau\} = t\sigma_{k+1}.$$

hence $\mathcal{IH}(k)$ holds for all $k \geq 0$, which proves (1). Now for any substitution γ, $\theta =_E \sigma_k\gamma$ implies :

$$u_k\gamma =_E t\sigma_k\gamma =_E t\theta =_E t'\theta =_E t'\sigma_k\gamma =_E u_k'\gamma,$$

which proves also (2).

Proposition :

$S\text{-}unify(t,\ t')$ terminates on every pair of terms $(t,\ t')$, moreover :

(1) If $S\text{-}unify(t,\ t') = \sigma$, with σ being a substitution, then $\{\sigma\}$ is a complete set of E-unifiers of t and t'.

(2) If $S\text{-}unify(t,\ t') = FAIL$ then t and t' have no E-unifier.

Proof :

At the k:th iteration for any $k \geq 0$, if *Substcond* does not apply, then the algorithm halts. Otherwise, since we do not allow functional procedures with nonground output, the number of distinct variables occuring in $t\sigma_{k+1}$ and $t'\sigma_{k+1}$ is one less than the number occuring in $t\sigma_k$ and $t'\sigma_k$. Since this number is initially finite the algorithm must terminate after a finite number of iterations.

Assume t and t' are E-unifiable terms such that *S-unify*(t, t') is not *ERROR*, and let θ be any of their E-unifiers. Under this assumption, define $I\!H(k)$ by :

$I\!H(k) \Leftrightarrow$ If σ_k, u_k and u_k' are defined by *S-unify*(t, t') then :

(a) there is a substitution γ_k such that $\theta =_E \sigma_k\gamma_k$.

(b) both *Failcond1*(D_k) and *Failcond2*(D_k) are false.

First we prove by induction on k that $I\!H(k)$ holds for every integer $k \geq 0$.

I For $k=0$ it suffices to let $\gamma_k=\theta$ which satifies (a), moreover since $D_k=DIS(u_k, u_k')$, lemma 2 together with fact 3 implies (b).

II Suppose $I\!H(k)$ holds for some $k \geq 0$, and that *S-unify*(t, t') defines σ_k, u_k and u_k'. If $u_k=u_k'$, the algorithm terminates and none of σ_{k+1}, u_{k+1} or u'_{k+1} is defined. Therefore, assume that $u_k \neq u_k'$. By (b) in the inductionhypothesis *Failcond1*(D_k) and *Failcond2*(D_k) are both false. Moreover, by our initial assumption *S-unify*$(t, t') \neq ERROR$, and hence there must be some variable v and term τ such that *Substcond*(v, τ, D_k) applies, i.e. $\sigma_{k+1}=\sigma_k\{v/\tau\}$. Since one of (v, τ) and (τ, v) is in $D_k=DIS(u_k, u_k')$, it follows from (a), lemma 2 and fact 4 that there exists a substitution γ_k such that $v\gamma_k =_E \tau\gamma_k$. By letting $\gamma_{k+1}=\gamma_k - \{v/v\gamma_k\}$ we then have :

$$\gamma_k=\{v/v\gamma_k\}\cup\gamma_{k+1} =_E \{v/\tau\gamma_k\}\cup\gamma_{k+1}=\{v/\tau\gamma_{k+1}\}\cup\gamma_{k+1}=\{v/\tau\}\gamma_{k+1},$$

where the next to last equality follows from the fact that v does not occur in τ. Thus $\theta =_E \sigma_k\gamma_k =_E \sigma_k\{v/\tau\}\gamma_{k+1}=\sigma_{k+1}\gamma_{k+1}$ which proves (a) for $I\!H(k+1)$. Again by lemma 2 and fact 3, (b) then follows.

By lemma 1 σ_k is an E-unifier of t and t' whenever $u_k=u_k'$, that is, whenever *S-unify*$(t, t')=\sigma_k$. Hence, by $I\!H(k)$ (a), $\{\sigma_k\}$ is a complete set of E-unifiers of t and t'. Moreover, since σ_k, u_k and u_k' are all defined whenever D_k is defined, $I\!H(k)$ (b) implies that *S-unify*(t, t') fails only if *S-unify*$(t, t')=ERROR$ or t and t' are not E-unifiable. However, if *S-unify*(t, t') fails it does certainly not give *ERROR* as result, and hence t and t' can not have any E-unifier.

Now we can use S-unification instead of the usual unification for interpretation of amalgamated programs. A program is *valid* for a given class of queries iff the interpreter will not give run-time error message during the execution of such queries. The above proposition indicates that SLD resolution combined with S-unification is complete for amalgamated programs being valid for submitted queries.

5.EXAMPLE

This section shows a simple example of a logic program that employes an external functional procedure. Our intention is to illustrate our idea of logic programming with functional procedures rather than to describe non-trivial problems.

Consider the following axioms for multiplication of integers, where + is a predefined function for addition :

multiplied(0, *X*, 0).

multiplied(*X*+1, *Y*, *Z*+*Y*) ← *multiplied*(*X*, *Y*, *Z*)

The program cannot be used for direct multiplication of integers: it is not valid with the queries of the form :

← *multiplied*(*c*, *d*, *X*)

where *c* and *d* are integers, since *c* gives ERROR when unified with *X*+1. However, the program still can be used to compute the relation *multiplied*. For this one can use queries of the form :

← *multiplied*(*X*, *c*, *Y*)

where *c* is an integer. Now it can be proved that the program is valid. The computation will give the answers :

X=0 and *Y*=0
or
X=1 and *Y*=*c*
or
X=2 and *Y*=*c*+*c*
etc...

6.CONCLUSIONS

In this paper we presented a way of integrating existing programming languages in a logic programming environment. We also introduced a new structured-oriented *S*-unification and we defined a class of programs for which (although not complete in general) it turns out to be a complete one. Let us also mention that our approach gives well defined semantical basis for "evaluable predicates" in Prolog. Our equality predicate is a generalization of many of the built-in features of Prolog. For example assume that the system $\mathcal{T}$ provides arithmetic operations on integers. In this case the equality *t*=*t'* in an amalgamated program may correspond to different constructs of Prolog, like *t is t'*, *t' is t*, *t*=*t'* and in some cases its effect cannot be simulated by a single Prolog construct (take for example 2+2 = 3+1: *S*-unification will reduce the terms and will succeed).

Let us compare our work with the two kinds of approaches mentioned in Section 1. The distinction between our work and the first approach is that the underlying programming system is considered as a black box: low-level features can be used in the underlying programs but not on the logical level. This makes it possible to give a relatively simple declarative semantics of the amalgamation. The difference between our work and the second approach is that we are primarily interested in re-using functional procedures written in other languages, regardless of the type of the language (be it a pure functional language, or an algorithmic language which admits functional procedures, like Fortran, Pascal or Ada) in a logic

programming environment. In contrast to the systems of this category we assume existence of the term machine and we are able to use it in the top-level computational mechanism without being specific about its construction.

In particular, comparing our work with [GoMe] we see that on one hand our approach is more general - a term language can be a fully higher order language like ML (see [GMW]). This means an indirect treatment of higher order computations (hidden in the term machine) in the amalgamated system. On the other hand, when limited to first-order languages, our approach preserves the completeness result for a subclass of programs and uses $\mathcal{S}$-unification employing the underlying term machine (instead of narrowing) as operational semantics.

Comparing our approach with the one of [SuYo] (which is the closest one to ours) the advantage we have is the ERROR value indicating the inability of telling whether two terms are E-unifiable, in contrast to the semantic unification which will answer "not E-unifiable" in such cases. The approach of [BBLM] seems to build too many levels of (difficult to work with) abstraction while our has a relatively simple construction.

Future work which can originate from our studies includes:

1. Sufficient conditions for validity which can be checked in compile time.

2. Exploration of other ways of amalgamation, based on the same principle of re-using the components.

3. Polymorphic-like type discipline for the amalgamated language.

4. "Modular" semantics - defined in terms of the semantics of the components - this will include the semantics of higher order functions.

7.REFERENCES

[BBLM] R.Barbuti, M.Bellia, G.Levi, M.Martelli "LEAF: a Language which integrates Logic, Equations and Functions", in Logic programming, functions, relations and equations, DeGroot D., Ed., Prentice-Hall, 1986.

[BeLe] M. Bellia, G. Levi, The Relation between Logic and Functional Languages: A Survey, J.Logic Programming, 1986:3:217-236

[Coh] S.Cohen "The APPLOG Language", in Logic programming, functions, relations ; and equations, DeGroot D., Ed., Prentice-Hall, 1986.

[GoMe] J.Goguen, J.Meseguer "EQLOG: Equality, Types and Generic Modules for Logic Programming", in Logic programming, functions, relations and equations, DeGroot D., Ed., Prentice-Hall, 1986.

[GMW] M.Gordon, C.Wadsworth, R.Milner "Edinburgh LCF: A Mechanized Logic of Computation", Springer LNCS 78 (1979).

[GoTa] J.Goguen, J.Tardo "An Introduction to OBJ: A Language for Writing and Testing Software Specifications", in Specification of Reliable Software, IEEE Press, 1979.

[GrLi] Logic programming, functions, relations and equations, DeGroot D. Lindstrom G., Eds. Prentice-Hall, 1986.

[JLM] J.Jaffar, J.L.Lassez, M.J.Maher, A theory of complete logic programs with equality, J.Logic Programming 1984:3:211-223

[Kom] J.Komorowski "QLOG - The programming environment for PROLOG

in LISP", in Logic Programming, Clark, K., Tarlund, S-A., Eds., Prentice-Hall, 1982.

[KoMa] J.Komorowski, J.Maluszynski , Logic programming and rapid prototyping, Rep. TR-01-86, Harvard Univ. Aiken Computation Lab. (to appear in Science of Computer Programming).

[Kow] R.A.Kowalski, Predicate logic as a programming language, in Information Processing 74, J.Rosenfeld,Ed., North-Holland 1974, 556-574

[MeHa] C.Mellish, S.Hardy "Integrating PROLOG in POPLOG environment", in Implementations of PROLOG, Campbell, J., Ed., Ellis Horwood, 1984.

[Myc] A. Mycroft and R.A. O'Keefe, A polymorphic type system for Prolog, Artificial Intelligence, 23: (1984) 295-307.

[Nil] J.F. Nilsson, On the compilation of a domain-based Prolog, in: R.E.A. Mason, Ed., Information Processing 83 (North Holland, 1983) 293-298.

[RoSi] J.Robinson, E.Sibert "LOGLISP: motivation, design and implementation", in Logic Programming, Clark, K., Tärnlund, S-A., Eds., Prentice-Hall, 1982.

[SuYo] P.Subrahmanyam, J-H.You "FUNLOG: a Computational Model integrating Logic programming and Functional Programming", in Logic programming, functions, relations and equations, DeGroot D., Ed., Prentice-Hall, 1986.

[Tär] S.-Å Tärnlund, Horn clause computability, BIT 17 (1977) 215-226

Avdelning, institution, fakultet
Division, department, faculty

Department of Computer and
Information Science
Linköping University

ISBN:

ISSN: ISSN-0281-4250

Rapportnr: Report no: LiTH-IDA-R-88-01

Upplagans storlek: Number of copies:

Datum: Date: January 1988

Projekt: Project:

Titel: Title: Logic Programming with External Procedures: Introducing S-Unification

Författare: Author Jacek Leszczylowski*), Staffan Bonnier and Jan Maluszynski

*)Department of Computer Science, Iowa State University
Ames, Iowa 50011-1040, USA

Uppdragsgivare: Commissioned by:

Dnr.: Call no:

Rapporttyp: Kind of report:

- ☐ Examensarbete/Final project
- ☐ Delrapport/Progress report
- ☐ Reserapport/Travel report
- ☐ Slutrapport/Final report
- ☒ Övrig rapport/Other kind of report

Rapportspråk: Language:

- ☐ Svenska/Swedish
- ☒ Engelska/English
- ☐ _____

Sammanfattning (högst 150 ord): Abstract (150 words):

A motivation for this work is the problem of re-usability of existing traditional software in logic programs. It can be viewed in an abstract way as the problem of amalgamation of Horn clause logic with a term reduction system whose rewrite rules are not accessible and thus cannot be used for construction of E-unifiers. Therefore we introduce a new unification algorithm, called S-unification, which is a special incomplete case of E-unification. It has the property that whenever it succeeds the result is a singleton complete set of E-unifiers of the arguments. It may also fail or report that it is not able to solve the problem of E-unification for given arguments. If the algorithm fails the actual arguments have no E-unifier. The paper discusses the problem of amalgamation of external functional procedures in a logic program and gives a characterization of a class of amalgamated programs for which S-unification is complete.

Nyckelord (högst 8): Keywords (8):

Logic Programming, Functional Programming, Amalgamation, E-unification

Bibliotekets anteckningar

A Selection of Previous Research Reports.

LiTH-IDA-R-87-18 **Henrik Nordin:** Reuse and Maintenance Techniques in Knowledge-Based Systems.

LiTH-IDA-R-87-17 **Tony Larsson:** Specification and Verification of VLSI Systems Actional Behaviour This is a close version of a paper presented at *the 8th international conference on Computer Hardware Description Languages, CHDL,* 87.

LiTH-IDA-R-87-16 **Shamsul I. Chowdhury:** State of the Art in Statistical Expert Systems. Also in *Proc. of the Conference on Expert Systems and their Applications*, Avignon, May 1987.

LiTH-IDA-R-87-15 **Nils Dahlbäck:** Kommunikation med datorer i naturligt språk - vad är det och vem behöver det?

LiTH-IDA-R-87-14 **Peter Fritzson:** Window System Architectures - an Overview. Also presented at *A Tutorial on Software Development Environments,* Linköping University, Nov 24-25 1986.

LiTH-IDA-R-87-13 **Mats Wirén:** A Comparison of Rule-Invocation Strategies in Context-Free Chart Parsing. Also in *Proc. of the Third Conference of the European Chapter of the Association for Computational Linguistics,* Copenhagen, April 1—3, 1987.

LiTH-IDA-R-87-11 **Sture Hägglund, Christer Hansson, Tomas Sokolnicki:** Knowledge-Based Training of Case Management Routines and Emergency Procedures. Also in *Proc. of the 3rd Int. Conf. on Expert Systems*, London, 1987.

LiTH-IDA-R-87-10 **Andrzej Lingas:** On Parallel Complexity of the Subgraph Isomorphism Problem.

LiTH-IDA-R-87-09 **Andrzej Lingas, Marek Karpinski:** Subtree Isomorphism and Bipartite Perfect Matching are Mutually NC Reducible.

LiTH-IDA-R-87-08 **Christos Levcopoulos, Andrzej Lingas, Jörg-R. Sack:** Nearly Optimal Heuristics for Binary Search Trees with Geometric Generalizations. Also in *Proc of ICALP'87*, Karlsruhe, July 1987.

LiTH-IDA-R-87-07 **Peter Haneclou:** A Formal Approach to Reason-maintenance Based on Abstract Domains.

LiTH-IDA-R-87-06 **Johan Hultman:** COPPS - A Software System for Defining and Controlling Actions in a Mechanical System.

LiTH-IDA-R-87-05 **Christer Bäckström:** Logical Modelling of Simplified Geometrical Objects and Mechanical Assembly Processes.

LiTH-IDA-R-87-03 **Mikael R.K. Patel:** A Threaded Interpretive Language Supporting Programming in the Large. Also in *Proc. of The sixth Rochester Forth Conference*, University of Rochester, Rochester, New York, June 11-14, 1986.

LiTH-IDA-R-87-02 **Rober Bilos:** A Token-Based Syntax Sensitive Editor. Also presented at the *Workshop on Programming Environments - Programming Paradigms*, Roskilde, Denmark, October 22-24, 1986.

LiTH-IDA-R-87-01 **Wlodzimierz Drabent:** Do Logic Programs Resemble Programs in Conventional Languages? Also in *Proc of the Fourth IEEE Symposium on Logic Programming*, San Francisco, Aug. 31- Sept. 4, 1987.

LiTH-IDA-R-86-44 **Erik Sandewall et al.:** Department of Computer and Information Science Annual Research Report 1986.

LiTH-IDA-R-86-43 **Ola Strömfors:** Editing Large Programs Using a Structure-Oriented Text Editor. Also presented at *the International Workshop on Advanced Programming Environments*, Trondheim, Norway,June 16-18, 1986.

LiTH-IDA-R-86-42 **Nils Dahlbäck, Arne Jönsson:** A Method for Studying Human-Computer Dialogues in Natural Language.

The Department of Computer and Information Science
Linköping University

organizes undergraduate and graduate studies in *Computer Science*, *Telecommunication and Computer Systems*, and *Administrative Data Processing*. Research activities have an emphasis on advanced software technology and computer systems design and are organized in a number of research laboratories:

- *ACTLAB - Laboratory for Complexity of Algorithms*, which is concerned with the design and analysis of efficient sequential and parallel algorithms, and complexity theory, especially in the areas of computational geometry, data structures on bounded domains and graph algorithms.
- *AIELAB - Artificial Intelligence Environments Laboratory*, which conducts research on representation and manipulation of knowledge, problem solving techniques and expert systems with mechanical engineering applications.
- *ASLAB - Application Systems Laboratory*, which studies design of advanced support systems for interactive use of computers, including tools for automated construction of applications software.
- *CADLAB - Laboratory for Computer-Aided Design of Electronics*, which concentrates its research activities around tools for integrated development of hardware and software, graphics-based modelling and simulation techniques.
- *LIBLAB - Laboratory for Library and Information Science*, which studies methods for access to documents and the information contained in the documents, concentrating on catalogs and bibliographic representations, and on the human factors of library use.
- *LOGPRO - Laboratory for Logic Programming*, which concentrates its research activities around foundations of logic programming, relations to other programming paradigms and methodology.
- *NLPLAB - Natural Language Processing Laboratory*, which conducts research related to the development and use of natural language interfaces to computer software.
- *PELAB - Programming Environments Laboratory*, which works with design of tools for software development, specific functions in such tools and theoretical aspects of programs under construction.
- *RKLLAB - Laboratory for Representation of Knowledge in Logic*, which covers issues and techniques such as non-monotonic logic, probabilistic logic, modal logic and truth maintenance algorithms and systems.

Research Reports 1987 and 1988

LiTH-IDA-R-88-01 **Jacek Leszczylowski, Staffan Bonnier, Jan Maluszynski:** Logic Programming with External Procedures: Introducing S-Unification. A shorter version is to appear in *Information Processing Letters*, North Holland.

LiTH-IDA-R-87-26 **Jonas Löwgren:** Applying a Rapid Prototyping System to Control Panel Dialgues.

LiTH-IDA-R-87-24 **Sven Moen:** Drawing Dynamic Trees.

LiTH-IDA-R-87-22 **Lars Ahrenberg, Arne Jönsson:** An Interactive System for Tagging Dialogues. To appear in *Proc. of 14th Conference of the Association for Literary and Linguistic Computing*, Göteborg 1-5 June, 1987.

LiTH-IDA-R-87-21 **Harold W. Lawson, Jr.:** Challenges and Directions in Computers and Education.

LiTH-IDA-R-87-20 **K**[illegible]sm Extraction from Se[illegible]s paper is to appear in *M*[illegible]*romicro Journal*, 1988.

LiTH-IDA-R-87-19 **Ze**[illegible]current Systems from th[illegible]*the 10th World Co*[illegible]1986.