LiTH-IDA-R-88-03

ISSN-0281-4250

INSTITUTIONEN FÖR DATAVETENSKAP

UNIVERSITETET OCH TEKNISKA HÖGSKOLAN LINKÖPING

Towards Clean Amalgamation of Logic Programs with External Procedures

by

Staffan Bonnier

Jan Maluszynski

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-88-03 ISSN-0281-4250

Towards Clean Amalgamation of Logic Programs with External Procedures

by

Staffan Bonnier

Jan Maluszynski

The paper presents a clean approach to the amalgamation of logic programming with external functional procedures. Both the logical semantics and the operational semantics of the amalgamated language are outlined. The operational semantics is based on an incomplete E-unification algorithm which we call S-unification. It is suggested to use the abstract interpretation technique for identifying classes of goals for which the approach is complete. For this purpose a domain of abstract terms is defined, and an abstract unification algorithm used for a compile-time check is developed.

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

TOWARDS A CLEAN AMALGAMATION OF LOGIC PROGRAMS WITH EXTERNAL PROCEDURES

Staffan Bonnier, Jan Maluszynski
Department of Computer and Information Science, Linköping University
S - 581 83 Linköping, Sweden

ABSTRACT

The paper presents a clean approach to amalgamation of logic programs with external functional procedures. Both the logical semantics and the operational semantics of the amalgamated language are outlined. The operational semantics is based on an incomplete E-unification algorithm which we call S-unification. It is suggested to use the abstract interpretation technique for identifying classes of goals for which the approach is complete. For this purpose a domain of abstract terms is defined, and an abstract unification algorithm used for a compile-time check is developed.

1. INTRODUCTION

This paper presents a systematic approach to the problem of amalgamating logic programs with external procedures. The motivation for this is twofold :

① To allow for re-using of existing (possibly imperative) software while still preserving the declarative nature of the top-level logic programs.

② To allow for use of functions in logic programming whenever it is more natural than expressing a function as a relation.

In recent years there have been a number of suggestions concerning combination of functional and logic programming in a single framework (see e.g. [GL] or [BL] 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 [RS], QLOG [Kom], POPLOG [MH] and APPLOG [Coh]).

② Construction of new languages which allow one to define functions and relations and to combine functional and relational definitions. (well-known examples are EQLOG [GM 86], LEAF [BBLM] and FUNLOG [SY]).

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, within the second approach the objective is often to have a clean logical semantics.

This paper combines the objectives of both approaches by putting some restrictions. The assumption is that the language of functional procedures is given, but we are not specific about its definition, so that our approach could be applied for any language. The task is to integrate the external procedures with logic programs in such a way that the amalgamation has a clean logical semantics and a reasonably efficient operational semantics, as complete as possible.

The basis for construction of an interface between the two different systems is the assumption that terms are their common data structures. A call of a functional procedure is itself a term. Since its execution is assumed to return a term, we can view the underlying programming system as a term rewriting system. This permits the use of the theory of logic programming with equality (see e.g. [JLM]) to give a clean declarative semantics of the amalgamated language, and for application of E-unification in its interpreter. However, since we are not specific about the language of the functional procedures, we have no access to the rewrite rules used by the system, and we cannot use them for construction of E-unifiers.

Our solution to this problem is an incomplete E-unification algorithm, called S-unification, outlined first in [LM]. It has the property that whenever it succeeds, the result is a complete set of E-unifiers of the arguments (which due to our restrictions is a singleton). It may also fail or report that it is not able to solve the problem of E-unification for the given arguments. If the algorithm fails the actual arguments have no E-unifier. In [LBM] we give a formal presentation of the S-unification algorithm and we prove its properties stated above.

This paper uses the results to develop a sufficient condition for the safe use of amalgamated programs. A compile-time check is presented which for a given amalgamated program P and a class of goals can show that the execution of P with any goal of the class will not abort due to the "do not know" outcome of S-unification.

2. THE LANGUAGE

The main feature of the syntax is the partition of functors and predicate symbols into

internal and external symbols. The role of the external symbols is to denote the external procedures to be used in logic programs. The alphabet is, as usual, the union of three disjoint sets of symbols :

① Σ, the set of *function symbols* (or *functors*).

② Π, the set of *predicate symbols*.

③ *VAR*, the set of *variables*.

To each function symbol and predicate symbol we assign an *arity*. Σ partitions into two disjoint sets : one of *internal functors* (or *constructors*) and one of *external functors*. Similarly Π partitions into the three sets of *internal predicate symbols*, *external predicate symbols* and { = }, where '=' is a distinguished *equality symbol* of arity two. The *terms* are built in the standard way over *VAR* and Σ. Terms built only from variables and constructors are called *structures*. The *atoms* are constructed by filling the argument places of the predicate symbols with terms. In particular, the atoms constructed from '=' are called *equations*. By a *program clause* we mean a construction of the form H :- B_1 , . . . , B_n where H, called the *head* of the clause, is an atom built from an internal predicate symbol, and B_1 , . . . , B_n is a possibly empty sequence of atoms, called the *body* of the clause. By a *goal clause* we mean a program clause with no head. Finally we define an (amalgamated) *program* to be a finite set of program clauses.

3. THE LOGICAL SEMANTICS

Our main idea is to allow the use of functional procedures written in *any* language (be it assembler or be it ML) in logic programs. However, in order to assign a clean formal semantics to the amalgamation and to make it work operationally, some restrictions on the permitted procedures are needed. First of all we restrict ourselves to procedures working on ground (i.e. variable free) terms. It is assumed that whenever a functional procedure is called with all its arguments instantiated to ground terms of the appropriate sort, it terminates and returns a ground term as result. In order for this term to be completely reduced, we also require it to be a structure. In the sequel we distinguish a special class of functional procedures which return truth values (succeed or fail). They will be called *tests*. Each external functor f in Σ is the name of one non-test procedure, with *n* arguments, where *n* is the arity of f. Similarly the external predicate symbols are names of test procedures.

Hence s(t_1 , . . . , t_n) denotes the application of the procedure named by s to the terms t_1 , . . . , t_n, while η(s(t_1 , . . . , t_n)) denotes the result of its evaluation.

The restriction of η to non-test procedures will be denoted by η_f, and its restriction to test procedures will be denoted by η_t. In the rest of the paper we will consider the extension of η obtained by extending η_f to *all* ground terms, including structures : For every n-ary constructor c and ground terms $t_1, \ldots, t_n$, $\eta_f(c(t_1, \ldots, t_n))$ is defined to be $c(\eta_f(t_1), \ldots, \eta_f(t_n))$.

Two ground terms t_1 and t_2 are considered to be *equivalent*, denoted by $t_1 \sim t_2$, iff $\eta_f(t_1) = \eta_f(t_2)$. As a further restriction on the non-test procedures, we require that $\sim$ is a congruence, that is, for each functor f,
$\eta_f(f(t_1, \ldots, t_n)) = \eta_f(f(t_1', \ldots, t_n'))$ whenever $\eta_f(t_i) = \eta_f(t_i')$ for $1 \leq i \leq n$. From a computational point of view, this means that the result obtained by evaluating a term does not change if its subterms are reduced first.

Similarly we require that the relations computed by the test procedures respect $\sim$, i.e. whenever $\eta_f(t_i) = \eta_f(t_i')$ for $1 \leq i \leq n$ and P is an external predicate symbol, $\eta_t(P(t_1, \ldots, t_n))$ succeeds iff $\eta_t(P(t_1', \ldots, t_n'))$ succeeds.

Now the standard construction may be used to obtain the model M_P for each amalgamated program P (see e.g. [GM87]) :

First divide the ground term algebra by $\sim$ (and get a model for the non-test procedures) and then extend this quotient to the least model for
$\{\forall(B_1 \wedge \ldots \wedge B_n \rightarrow H) \mid H :- B_1, \ldots, B_n \in P\}$ which interprets '=' as the identity relation and which, for each external predicate symbol P, satisfies $P(t_1, \ldots, t_n)$ iff $\eta_t(P(t_1, \ldots, t_n))$ succeeds. The model so obtained is called the *initial* model for P.

4. S-UNIFICATION

This section is primarily concerned with equation solving or *E-unification* in the presence of a set E of equations. For this purpose we develop an incomplete E-unification algorithm, called S-unification.

4.1. E-unification

A set E of equations generates a number of congruence relations on terms, the finest of which is denoted by $=_E$. By an *E-unifier* of two terms t_1 and t_2 we mean a substitution σ such that $t_1\sigma =_E t_2\sigma$. A set S of E-unifiers of t_1 and t_2 is *complete* iff every E-unifier σ of t_1 and t_2 factors into $\sigma =_E \theta\gamma$ for some element θ of S and some substitution γ. Here $=_E$ is defined for substitutions σ_1 and σ_2 by $\sigma_1 =_E \sigma_2$ iff $X\sigma_1 =_E X\sigma_2$ for each variable X. Traditionally the problem of E-unification is solved by transforming the (finite set of) equations defining the functions into *rewrite rules*. By applying a

technique called *narrowing* [H] or some variant thereof, it is then sometimes possible to construct a complete set of E-unifiers for the given terms, or to decide that the terms are not E-unifiable. As the central object for our studies, we will use the set E of equations defined by :

$$E = \{ t_1 = t_2 \mid t_1\gamma \sim t_2\gamma \text{ for } \textit{every} \text{ ground instantiation } \gamma \text{ of } t_1, t_2 \}$$

This set is closed, in the sense that $t_1 =_E t_2$ iff $t_1 = t_2 \in E$. Clearly, in our case we only have access to an infinite number of ground equations which define the functions, and it is therefore impossible to give a reasonably efficient general algorithm, which enumerates a complete set of E-unifiers of two terms. Our approach is instead to carefully select some cases when it is possible to construct a complete *singleton* of E-unifiers for the given terms, or to decide that no E-unifier of the terms can exist. The incomplete E-unification algorithm obtained in this way we call S(tructure oriented) unification.

4.2. The Disagreement set

The *reduced form* of a term t is the term obtained by substituting each ground subterm u of t by $\eta_f(u)$. If for example c and the numerals are constructors, X is a variable and + is an external functor with its usual interpretation, then the reduced form of $c(X,(2+5)+X)$ is $c(X,7+X)$.

Let t_1 and t_2 be terms and let r_1, r_2 be their respective reduced forms. We define the *disagreement set* of t_1 and t_2, denoted by $D(t_1, t_2)$, as the least set which satisfies the following three conditions :

① If the top symbol of r_1 and r_2 differs, then $\{ r_1, r_2 \} \in D(t_1, t_2)$.

② If r_1 and r_2 are distinct but have the same external functor as top symbol, then $\{ r_1, r_2 \} \in D(t_1, t_2)$.

③ If $r_1 = c(u_1, \ldots, u_n)$ and $r_2 = c(u_1', \ldots, u_n')$ where c is a constructor, then $D(u_i, u_i') \subseteq D(t_1, t_2)$ for $1 \le i \le n$.

<u>Example :</u>

Let $t_1 = c(X*X, c(2+2, X+2))$ and $t_2 = c(Y*Y, c(7-2, X+2))$ where c and the numerals are constructors, X and Y are variables and +, - and * are external functors with their usual interpretations.

The reduced forms of t_1 and t_2 are $c(X*X, c(4, X+2))$ and $c(Y*Y, c(5, X+2))$ respectively. By two applications of ③ we see that $D(t_1, t_2)$ is the union of

D(X*X, Y*Y), D(4, 5) and D(X+2, X+2). From ② then follows that $\{ X*X, Y*Y \} \in D(t_1, t_2)$, and from ① it follows that $\{ 4, 5 \} \in D(t_1, t_2)$, i.e. $D(t_1, t_2) = \{ \{ X*X, Y*Y \}, \{ 4, 5 \} \}$.

□

It is not hard to see that the condition : $\forall \{ u_1, u_2 \} \in D(t_1, t_2) : u_1\sigma =_E u_2\sigma$, is both necessary and sufficient for σ to be an E-unifier of t_1 and t_2. Note that this would no longer be true if we choose to skip condition ② in the definition of the disagreement set, and instead handled external functors as constructors (this is done in e.g. [SY]). As an example, { X / -Y } is an E-unifier of X*X and Y*Y but it is *not* an E-unifier of X and Y, although D(X*X, Y*Y) would be { { X, Y } } with the altered definition. If we skipped ② we would therefore wind up with an algorithm which sometimes produced an *incomplete* singleton of E-unifiers for the given terms, and hence we would not achieve our goal. Our next step is to analyze $D(t_1, t_2)$, and by means of this analysis formulate three different conditions. The first condition tells us how any E-unifier of t_1 and t_2 necessarily must bind the variables, and the following two conditions ensures that it is safe to fail :

① If $\{ X, u \} \in D(t_1, t_2)$ where X is a variable which does not occur in u, then any E-unifier of t_1 and t_2 must bind X to a term equivalent to an instance of u. We therefore name the fact that X is a variable which does not occur in u by *Substcond*(X, u).

② If $\{ u_1, u_2 \} \in D(t_1, t_2)$ and u_1, u_2 both have constructors as top symbols, then no E-unifier of u_1 and u_2 can exist since these constructors must be distinct, and hence the above condition implies that no E-unifier of t_1 and t_2 exists. We call the existence of such an element in $D(t_1, t_2)$ by $Failcond_1(D(t_1, t_2))$.

③ If $\{ X, u \} \in D(t_1, t_2)$, X is a variable and X has at least one occurence in u, which is outside every subterm of u having an external functor as top symbol, then no E-unifier of u_1 and u_2 can exist, and consequently no such exists for t_1 and t_2 either. This case, which we call $Failcond_2(D(t_1, t_2))$, corresponds to the usual occur-check failure.

Note that the fact that X occurs in u is *not* sufficient for X and u not to be E-unifiable, even if the top symbol of u is a constructor. Consider for example the case where c is a constructor, f is an external functor and t is a ground term such that $\eta_f(f(c(t))) = t$, then { X / c(t) } is an E-unifier of X and c(f(X)).

4.3. Unification

Next we present the S-unification algorithm. The algorithm may *succeed* and return a substitution as result, it may *fail*, or it may give an *error message* :

```
S-unify( t1, t2 ):
begin
   σ := ε;
   D := D( t1, t2 );
   while D ≠ ∅ do
      if Failcond1(D) or Failcond2(D)
      then FAIL
      else  if ∃ { X, u } ∈ D : Substcond( X, u )
            then  begin
                     σ := σ{ X / u };
                     D := D( t1σ, t2σ );
                  end
            else ERROR
   else return σ
end .
```

An error message indicates that the terms are not sufficiently instantiated, and hence that the algorithm can not decide whether they are E-unifiable. We do however have the following result [LBM] :

<u>Theorem :</u>

For any terms t_1 and t_2, S-unify(t_1, t_2) terminates. Moreover :

① If S-unify(t_1, t_2) succeeds and returns σ,then $\{ \sigma \}$ is a complete set of E-unifiers of t_1 and t_2.

② If S-unify(t_1, t_2) fails, then t_1 and t_2 have no E-unifier.

□

It should be mentioned that the S-unification algorithm reduces to Robinsons unification algorithm [R] whenever it is called with both its arguments instantiated to structures.

<u>Example :</u>

Let t_1 = s(X*X , t) and t_2 = s(X , X), where s and the numerals are constructors, X is a variable and * is an external functor with its usual interpretation. S-unify(t_1, t_2)

will succeed and return { X / t } iff t is 0 or 1, it will fail iff t is any other numeral, and it will give an error message iff t is nonground.

□

5. THE OPERATIONAL SEMANTICS

In this section an interpreter for the amalgamated language is outlined, and some simple examples of amalgamated programs are presented. Throughout this and the following section, internal predicate symbols will be treated as constructors.

5.1. The Interpreter

We start out from a standard Prolog interpreter with its left-to-right computation rule, and exchange its purely syntactic unification for *S*-unification. Hence the operational semantics of a program *P*, which defines how an answer substitution is to be computed from a goal clause, may be given in terms of the least relation O_P between goal clauses and substitutions, satisfying :

① □ O_P ε, where □ is the empty goal.

② If *S*-unify(t_1, t_2) succeeds and returns σ, then :- $t_1 = t_2$ O_P σ.

③ If P is an external predicate symbol and t_1 , . . . , t_n are ground terms such that η_t(P(t_1 , . . . , t_n)) succeeds, then :- P(t_1 , . . . , t_n) O_P ε.

④ If A is an atom built from an internal predicate symbol, H :- B_1 , . . . , B_n is a (renamed) clause from *P* such that *S*-unify(A, H) succeeds and returns σ_1, and :- (B_1 , . . . , B_n)σ_1 O_P σ_2, then :- A O_P ($\sigma_1\sigma_2$ | *V*(A)), where the last expression denotes the restriction of $\sigma_1\sigma_2$ to the variables occuring in A.

⑤ If :- B_1 , . . . , B_i O_P σ_1 and :- (B_{i+1} , . . . , B_n)σ_1 O_P σ_2 where $i \leq n$, then :- B_1 , . . . , B_n O_P $\sigma_1\sigma_2$.

5.2. Admitted Goals of a Program

If, during the execution of a program *P* with a goal *G*, no runtime error caused by *S*-unification occurs and no test procedure is called with some nonground argument, then it is not possible at this execution to distinguish our interpreter from SLD-resolution combined with a complete *E*-unification algorithm. From the fact that such a combination is a complete deduction system for Horn clause logic with equality [JLM], we may then conclude that we for each program have a class of goals for which our interpreter is complete. If *P* is a program and *G* is such a goal, we say that *P admits G*. More precisely, let A be a body atom of *P* and *G*, and let C_G(A) be the set of *call*

instantiations of A, i.e. the set of all possible instantiations of A whenever A becomes the actual subgoal during the processing of G. Now P admits G iff, for each bodyatom A of P and G, and $\sigma \in C_G(A)$, the following three conditions are satisfied :

① If $A = t_1 = t_2$, then S-unify($t_1\sigma$, $t_2\sigma$) does not give an error message.

② If $A = P(t_1 , \ldots , t_n)$ where P is an external predicate symbol, then $t_i\sigma$ is ground for $1 \leq i \leq n$.

③ If A is built from an internal predicate symbol and $H :- B_1 , \ldots , B_n$ is any (renamed) clause of P, then S-unify($A\sigma$, H) does not give an error message.

5.3. Examples

Next we present two simple examples of amalgamated programs :

Example 1 :

Suppose + and - are external functors, and let ≤ be an external predicate symbol, all with their standard interpretations. The predicate `less` has the same formal semantics as ≤ on nonnegative integers, but may be called with its first argument uninstantiated :

```
less(0,X).
less(Y+1,X) :- 1≤X, less(Y,X-1).
```

Hence `:-less(X,2)` would yield the enumeration $X = 0$, $X = 1$ and $X = 2$. However, the program does not admit goals of the form `:-less(n,m)` where n and m are integers, since S-unification of `less(n,m)` and `less(Y+1,X)` would give *ERROR* as result. Neither does it admit goals where the second argument is uninstantitated, due to the symbol ≤ occuring in the second clause.

□

Example 2:

By a *partition* of a nonnegative integer n, we mean a sequence of positive integers which sums up to n. The following program, which uses Prologs standard notation for lists, enumerates all partitions of a given integer :

```
part([],0).
part([H|T],X) :- less(H,X), H≠0, part(T,X-H).
```

The program uses `less` from the previous example and the external predicate symbol

≠ with its obvious interpretation. As an example, the goal `:- part(X,2)`, would yield the following enumeration : `X = [1,1]`, `X = [2]`.

□

6. STATIC ANALYSIS OF AMALGAMATED PROGRAMS

This section outlines a technique for static analysis of amalgamated programs. The problem is whether a given program admits a given class of goals. This problem is undecidable. This can be proved by reducing it to the halting problem of Turing machines, in a similar way as it was done in [DM] for the occur-check problem. Therefore a sufficient condition is suggested, which results in a relatively simple check procedure. The approach is similar to that of abstract interpretation, see e.g. [M], [JS], [B], [N]. For the infinite domain of terms, a finite family of subdomains is defined. Its elements, called *kinds* are the following :

① The empty set, denoted by **e**.

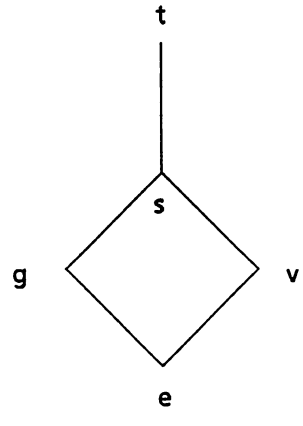
② The set of variables, denoted by **v**.

③ The set of all ground terms, denoted by **g**.

④ The set of all terms having a structure as reduced form, denoted by **s**.

⑤ The set of all terms, denoted by **t**.

The concretization function Δ on the set **K** (= { **e**, **g** , **v** , **s** , **t** }) of kinds, associates with each kind the corresponding set of terms. By the assumption that the external procedures always terminate and return a structure, **K** forms a lattice which is illustrated in the following figure :



An *abstract substitution* π, characterizes a set of substitutions **S**(π) by specifying :

① D_π : a finite set of variables, the *domain* of π.

② $\kappa_\Pi : D_\Pi \rightarrow \mathbf{K}$: the *kind assignment* which specifies the kind of terms which are allowed to be assigned to the variables, i.e. for every $X \in D_\Pi$ and $\sigma \in \mathbf{S}(\Pi)$, $X\sigma \in \Delta(\kappa_\Pi(X))$.

③ $\alpha_\Pi \subseteq D_\Pi \times D_\Pi$: the *aliasing relation* ; $X\ \alpha_\Pi\ Y$ iff there is a substitution σ in $\mathbf{S}(\Pi)$ such that $X\sigma$ and $Y\sigma$ have a common variable.

It is assumed that the classes of goals checked for admission are characterized by abstract substitutions. Thus, the starting point for static analysis is a program P, a goal $G = :- B_1, \ldots, B_n$ and an abstract substitution Π_0 defined on the variables of B_1. Π_0 specifies all bindings of the variables of B_1 which are allowed at the invokation time, i.e. when the execution of B_1 starts. Assume now that an abstract substitution is given not only for B_1 but also for $B_2, \ldots, B_n$ and for every body literal of P. The abstract substituion Π_L assigned to a literal L is said to be a *correct annotation* for P, G and Π_0 iff at every invokation of L during the execution of $:- B_1\sigma, \ldots, B_n$ where $\sigma \in \mathbf{S}(\Pi_0)$, the binding of the variables of L are in $\mathbf{S}(\Pi_L)$. Correct annotations will be generated by abstract interpretation, but this is outside the scope of this paper. Here we confine ourselves with the definition of an abstract unification algorithm which will be used for checking admissibility of goals. The algorithm works on *abstract terms*. By an abstract term we mean a pair $\langle t, \Pi \rangle$, where t is a term and Π is an abstract substitution on a domain including all variables of t. $\langle t, \Pi \rangle$ denotes all terms $t\sigma$ such that $\sigma \in \mathbf{S}(\Pi)$. For each abstract term $\langle t, \Pi \rangle$, we define the kind $\kappa_\Pi(t)$ as follows :

① If t is a constant, then $\kappa_\Pi(t) = \mathbf{g}$.

② If t is a variable X, then $\kappa_\Pi(t) = \kappa_\Pi(X)$.

③ If $t = c(t_1, \ldots, t_n)$ where c is a constructor, then $\kappa_\Pi(t) = \bigvee\{\mathbf{g}, \kappa_\Pi(t_1), \ldots, \kappa_\Pi(t_n)\}$.

④ If $t = f(t_1, \ldots, t_n)$ where f is an external functor, then $\kappa_\Pi(t) = \mathbf{g}$ if $\kappa_\Pi(t_1) = \ldots = \kappa_\Pi(t_n) = \mathbf{g}$, and $\kappa_\Pi(t) = \mathbf{t}$ otherwise.

The abstract unification algorithm, which we call AT-unification, applies to two abstract terms $\langle t_1, \Pi \rangle$ and $\langle t_2, \Pi \rangle$. Our design objective for AT-unify is that it should be able to recognize whenever there exists a substitution σ in $\mathbf{S}(\Pi)$ for which S-unify($t_1\sigma$, $t_2\sigma$) gives an error message. Due to the observation that the term component of an abstract term describes the top level structure of all its instances, we start by constructing the disagreement set D(t_1, t_2). In this way we do exactly the

same job as S-unification would have done for any terms in the denotations of $\langle t_1, \pi \rangle$ and $\langle t_2, \pi \rangle$. First we note that if $Failcond_1(u_1, u_2)$ or $Failcond_2(u_1, u_2)$ applies for some $\{u_1, u_2\} \in D(t_1, t_2)$, then no instance of t_1 and t_2 will S-unify, and hence AT-unify may safely fail. However, now $D(t_1, t_2)$ contains pairs of *sets* of terms rather than pairs of terms. Therefore the analysis of $D(t_1, t_2)$ has to be done in a quite different way than for S-unification. Our intention is to pick elements $\{u_1, u_2\}$ from $D(t_1, t_2)$ which are *safe* in the sense that for *no* σ in $S(\pi)$, S-unify$(u_1\sigma, u_2\sigma)$ gives an error message. It is not hard to see that each of the following two conditions, which we call $Substcond_1(u_1, u_2)$ and $Substcond_2(u_1, u_2)$ respectively, are sufficient for this to be true :

① One of u_1 and u_2 is a variable X, i.e. $\{u_1, u_2\} = \{X, u\}$, and either $\kappa_\pi(X), \kappa_\pi(u) \leq \mathbf{s}$ and X does not occur in u, or $\kappa_\pi(X) = \mathbf{v}$ and for no variable Y that occurs in u, $X\ \alpha_\pi\ Y$.

② One of u_1 and u_2 is a nonvariable term to which κ_π assigns the kind $\mathbf{s}$, and the other is a term, with an external functor as top symbol, to which κ_π assigns the kind $\mathbf{g}$.

Now we assume that u_1 and u_2 are instantiated by a $\sigma \in S(\pi)$ in such a way that S-unify$(u_1\sigma, u_2\sigma)$ succeeds with a substitution θ as result. Of course this assumption may be wrong. We do know however that S-unify$(u_1\sigma, u_2\sigma)$ will not give an error message. Our next step is then to simulate the composition of $t_1\sigma$ and $t_2\sigma$ with θ. This is done by the construction of two new abstract terms, denoted $\langle t_1, \pi \rangle + \{u_1, u_2\}$ and $\langle t_2, \pi \rangle + \{u_1, u_2\}$ and defined as follows :

① If one of u_1 and u_2 is a variable X, so that $\{u_1, u_2\} = \{X, u\}$, then $t_i + \{u_1, u_2\} = t_i\{X / u\}$ for $i = 1, 2$, and $D_{\pi + \{u_1, u_2\}} = D_\pi - \{X\}$.

② If both u_1 and u_2 are nonvariable terms, then $t_i + \{u_1, u_2\} = t_i$ for $i = 1, 2$, and $D_{\pi + \{u_1, u_2\}} = D_\pi$.

In order to construct $\kappa_{\pi + \{u_1, u_2\}}$ and $\alpha_{\pi + \{u_1, u_2\}}$, four different conditions are defined. A variable X in D_π may or may not satisfy some of them :

Condition 1 :For no variable Y that occurs in u_1 or u_2, $X\ \alpha_\pi\ Y$ holds.
Condition 2 :$\kappa_\pi(u_1) = \mathbf{v}$ and X occurs in u_2.
Condition 3 :X occurs in u_1 or u_2 and at least one of $\kappa_\pi(u_1)$ and $\kappa_\pi(u_2)$ is $\mathbf{g}$.
Condition 4 :None of condition 1 to 3 holds for X.

Now $\kappa_{\Pi + \{u_1, u_2\}}(X)$ is defined for each $X \in D_{\Pi + \{u_1, u_2\}}$ by :

① If *Condition 1* or *Condition 2* holds for X, then $\kappa_{\Pi + \{u_1, u_2\}}(X) = \kappa_\Pi(X)$.

② If *Condition 3* holds for X, then $\kappa_{\Pi + \{u_1, u_2\}}(X) = \mathbf{g}$.

③ If *Condition 4* holds for X, then $\kappa_{\Pi + \{u_1, u_2\}}(X) = \kappa_\Pi(X) \vee \kappa_\Pi(u_1) \vee \kappa_\Pi(u_2)$.

Finally, we let S be the subset of $D_{\Pi + \{u_1, u_2\}}$, consisting of all variables X such that $\kappa_{\Pi + \{u_1, u_2\}}(X)$ is **v**, **s** or **t**. $\alpha_{\Pi + \{u_1, u_2\}}$ is defined to be the least equivalence relation on S which satisfies :

① If $X\ \alpha_\Pi\ Y$ then $X\ \alpha_{\Pi + \{u_1, u_2\}}\ Y$.

② If $u_1 = Y$ and X satisfies *Condition 2*, then $X\ \alpha_{\Pi + \{u_1, u_2\}}\ Y$.

③ If *Condition 4* holds for X, and Y occurs in u_1 or u_2, then $X\ \alpha_{\Pi + \{u_1, u_2\}}\ Y$.

Now the AT-unification algorithm may be defined. At each iteration it picks one element $\{u_1, u_2\}$ from $D(t_1', t_2')$ for which one of the substitution conditions applies, and constructs $\langle t_i', \Pi' \rangle + \{u_1, u_2\}$ for $i = 1, 2$. If $Substcond_1(u_1, u_2)$ holds, then $\{u_1, u_2\}$ will not occur again in $D(t_1' + \{u_1, u_2\}, t_2' + \{u_1, u_2\})$. If however $Substcond_2(u_1, u_2)$ applies, then $\{u_1, u_2\}$ will be in $D(t_1' + \{u_1, u_2\}, t_2' + \{u_1, u_2\})$ with the kind **g** assigned to all the variables occuring in u_1 and u_2. Therefore we can not have $D(t_1', t_2') = \varnothing$ as termination condition. If however $\kappa_{\Pi'}(u_1) = \kappa_{\Pi'}(u_2) = \mathbf{g}$ holds for each $\{u_1, u_2\} \in D(t_1', t_2')$, then we may safely stop without giving an error message. We call this condition *Stopcond*(D).

```
AT-unify(⟨ t1, π ⟩, ⟨ t2, π ⟩):
  begin
    ⟨ t1', π' ⟩ := ⟨ t1, π ⟩
    ⟨ t2', π' ⟩ := ⟨ t2, π ⟩
    D := D(t1', t2');
    while not Stopcond(D) do
      if Failcond1(D) or Failcond2(D)
      then FAIL
      else  if ∃ { u1, u2 } ∈ D : Substcond1( u1, u2 ) or Substcond2( u1, u2 )
            then  begin
                    ⟨ t1', π' ⟩ := ⟨ t1', π' ⟩ + { u1, u2 }
                    ⟨ t2', π' ⟩ := ⟨ t2', π' ⟩ + { u1, u2 }
                    D := D(t1', t2');
                  end
            else ERROR
  end .
```

For the AT-unification algorithm we have the following theorem :

<u>Theorem :</u>
For any term patterns $\langle t_1, \pi \rangle$ and $\langle t_2, \pi \rangle$, AT-unify($\langle t_1, \pi \rangle, \langle t_2, \pi \rangle$) terminates. Moreover :

① If AT-unify($\langle t_1, \pi \rangle, \langle t_2, \pi \rangle$) fails, then S-unify($t_1\sigma$, $t_2\sigma$) fails for all $\sigma \in \mathbf{S}(\pi)$.

② If, for some $\sigma \in \mathbf{S}(\pi)$, S-unify($t_1\sigma$, $t_2\sigma$) = $ERROR$, then AT-unify($\langle t_1, \pi \rangle, \langle t_2, \pi \rangle$) = $ERROR$.

□

<u>Example :</u>
Let $t_1 = c(X, 3+Y)$, $t_2 = c(Z+Z, 3)$ and let $D_\pi = \{ X, Y, Z \}$, $\kappa_\pi(X) = \kappa_\pi(Z) = \mathbf{v}$ and $\kappa_\pi(Y) = \mathbf{g}$. If $X\ \alpha_\pi\ Z$ then AT-unify($\langle t_1, \pi \rangle, \langle t_2, \pi \rangle$) will terminate with $ERROR$, otherwise the algorithm will stop without error message.

□

Let P be a program, let G be a goal and let π_0 be the initial abstract substitution. If each body atom A of P and G is correctly annotated for P , G and π_0 then it suffices to check the following for each body atom A of P and G, in order to be sure that P admits each goal defined by G and π_0 :

① If $A = t_1 = t_2$, then AT-unify($\langle t_1, \pi_A \rangle, \langle t_2, \pi_A \rangle$) does not give an error message.

② If $A = P(t_1, \ldots, t_n)$ where P is an external predicate symbol, then π_A assigns the kind **g** to each variable in A.

③ If A is built from an internal predicate symbol and $H :- B_1, \ldots, B_n$ is any (renamed) clause of P, then AT-unify($\langle A, \pi_A' \rangle, \langle H, \pi_A' \rangle$) does not give an error message.

Here π_A' is an extention of π_A which includes all variables in H, so that no two distinct variables of H are aliases and all of the variables in H are assigned the kind **v**.

7. CONCLUSION

The paper presents a clean approach to amalgamation of logic programs with external functional procedures. The logical semantics of the amalgamated language is outlined as well as the operational semantics, based on an incomplete E-unification algorithm which we call S-unification. It is suggested to use the abstract interpretation technique for identifying classes of goals for which the approach is complete. For this purpose a domain of abstract terms is defined, and an algorithm of abstract unification, which is used for a compile-time check is developed.

An idea close to S-unification appeared independently in Le Fun [ALN] but in a purely operational setting. The idea of residuations used in Le Fun, and the idea of delays used in CLP [JM] makes it possible to avoid some errors of S-unification. Implicitly it is present also in our description, but only in the limits of one unification : notice that our abstract presentation of the algorithm concerns selection of certain pairs in the disagreement sets. A more realistic version should probably use left-to-right selection with residuation. Extending this technique beyond the scope of one unification is also possible, but a more elegant solution would be some kind of static analysis of the program leading to some program transformation.

The logical semantics of the language is based on well-known results presented in many papers, e.g. [GM87], [JLM]. However, our objectives are different. EQLOG attempts to exploit the narrowing technique, but we cannot do that since our functional procedures are black-boxes. Therefore our approach is inherently incomplete.

Since the language of functional procedures is left unspecified, we deal as a matter of fact with a family of logic programming languages. In that sense there is a similarity to

the CLP family of languages. However, in contrast to the CLP project we do not plan to use additional information about the domains during the execution. Thus our approach is "less complete" but applies to any domain. The future work includes the following topics :

- Efficient algorithms for S-unification.
- Methods for static analysis of amalgamated programs, including automatic generation of annotations, and program transformation for improving admissibility of goals.
- Relaxation of the restrictions on external procedures, e.g. considering lazy evaluation.
- Application of the idea of S-unification to higher-order procedures.
- Amalgamated programs with negation.

8. REFERENCES

[ALN] H. Aït-Kaci, P. Lincoln, R. Nasr, Le Fun: Logic, Equations and Functions, Proc. Symp. on Logic Programming, 1987, San Francisco, California.

[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.

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

[B] M. Bruynooghe, A Framework for the Abstract Interpretation of Logic Programs, Report CW 62, Katholieke Universiteit, Leuven, 1987.

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

[DM] P. Deransart and J. Maluszynski, Logic programs and attribute grammars, J. Logic Programming 2 (1985) 119-155.

[GM86] 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.

[GM87] J.Goguen, J.Meseguer, Models and Equality for Logical Programming, Proc. TAPSOFT' 87, Pisa, Italy 1987.

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

[Hul] J-M. Hullot, Canonical Forms and Unification, Proc., CADE-5, 1980, 318-334.

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

[JM] J.Jaffar, S.Michaylov, Methodology and Implementation of a CLP System, Proc., 4:th Int. Conf. on Logic Programming, Melbourne, Australia.

[JS] N. Jones, H. Søndergaard, A Semantics-Based Framework for the Abstract Interpretation of Prolog, in Abstract Interpretation of Declarative Languages, ed. Abramsky and Hankin, Ellis Horwood (1987).

[Kom] J.Komorowski "QLOG - The programming environment for PROLOG in LISP", in Logic Programming, Clark, K., Tarlund, S-A., Eds., Prentice-Hall, 1982.

[KM] 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.

[LM] J. Leszczylowski, J.Maluszynski, Logic programming with external procedures : Introducing S-Unification, Rep. TR-86-21, Dept. of Computer Sc., Iowa State University, Ames 1986

[LBM] J.Leszczylowski, S.Bonnier, J.Maluszynski, Logic programming with

external procedures : Introducing S-Unification, Rep. R-88-01, Dept. of Comp. and Inf. Sc., Linköping University, Linköping, Sweden, a shortened version also to appear in Information Processing Letters.

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

[Mel] C.S. Mellish, Abstract Interpretation of Prolog Programs, in Abstract Interpretation of Declarative Languages, ed. Abramsky and Hankin, Ellis Horwood (1987).

[Nil] U. Nilsson, Inferring Restricted AND-Parallelism in Logic Programs using Abstract Interpretation, Rep. R-88-02, Dept. of Comp. and Inf. Sc., Linköping University, Linköping, Sweden

[R] J.A. Robinson, A Machine-Oriented Logic Based on the Resolution Principle. JACM 12: (1965), 32-41.

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

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-88-03

Upplagans storlek:
Number of copies:

Datum:
Date: January 1988

Projekt:
Project:

Titel:
Title: Towards Clean Amalgamation of Logic Programs with External Procedures

Författare:
Author: Staffan Bonnier, Jan Maluszynski

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):

The paper presents a clean approach to the amalgamation of logic programming with external functional procedures. Both the logical semantics and the operational semantics of the amalgamated language are outlined. The operational semantics is based on an incomplete E-unification algorithm which we call S-unification. It is suggested to use the abstract interpretation technique for identifying classes of goals for which the approach is complete. For this purpose a domain of abstract terms is defined, and an abstract unification algorithm used for a compile-time check is developed.

Nyckelord (högst 8):
Keywords (8):

Bibliotekets anteckningar

A Selection of Previous Research Reports.

LiTH-IDA-R-87-20 **Krzysztof Kuchcinski, Zebo Peng:** Parallelism Extraction from Sequential Programs for VLSI Applications. This paper is to appear in *Microprocessing and Microprogramming, the Euromicro Journal*, 1988.

LiTH-IDA-R-87-19 **Zebo Peng:** Construction of Asynchronous Concurrent Systems from their Behavioral Specifications. Also in *Proc. of the 10th World Computer Congress*, Dublin, Ireland, Sept. 1-5, 1986.

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.

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-88-03 **Staffan Bonnier, Jan Maluszynski:** Towards a Clean Amalgamation of Logic Programs with External Procedures.

LiTH-IDA-R-88-02 **Ulf Nilsson:** Inferring Restricted AND-Parallelism in Logic Programs using Abstract Interpretation.

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 Dia[illegible] *of the Association for* [illegible] 1-5 June, 1987.

LiTH-IDA-R-87-21 **Ha**[illegible] ions in Computers and [illegible]