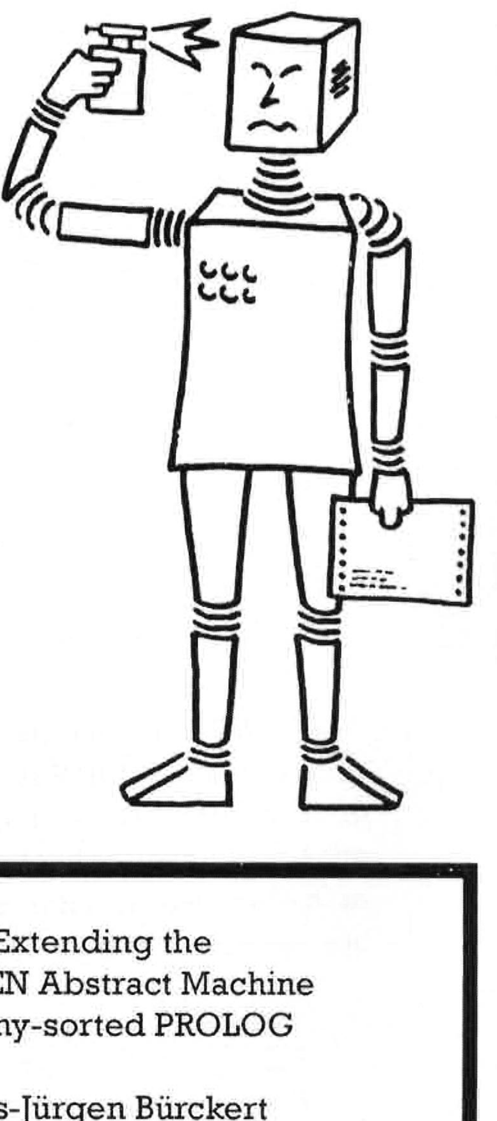
SEKI-PROJEKT

SEKI MEMO

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany



Extending the WARREN Abstract Machine to Many-sorted PROLOG

Hans-Jürgen Bürckert

MEMO SEKI-85-VII-KL

PIPE-Universität-KL-0003

Extending the WARREN Abstract Machine to Many-sorted PROLOG

Hans-Jürgen Bürckert

Universität Kaiserslautern
Fachbereich Informatik
Postfach 3049
D-6750 Kaiserslautern
W. Germany

Abstract:
This report describes an abstract Prolog instruction set, known as WARREN Abstract Machine, for a many-sorted version of MPROLOG. The Prolog inference mechanism - based on SLD-resolution - is extended to many-sorted SLD-resolution, which reduces the deduction tree and helps avoiding unnecessary backtracking. Building in those sorts is also known as introducing a type mechanism into Prolog, which helps detecting and avoiding syntax errors.

This work was supported by the Bundesministerium für Forschung und Technologie of the FR·Germany and by the NIXDORF Computer AG within the framework of the Joint Research Project ITR-8501 A.

Introduction and Motivation

In the following paper we describe an extension of MPROLOG /HU 82/ with sorts (=restricted quantification) and a corresponding abstract Prolog machine based on the WARREN Abstract Machine (WAM), see /War 83/.
Our goal was not to design a new logic programming language but to extend an available Prolog version, such that a modification of present implementations should easily be possible. Moreover, already existing programs should run without any modifications or run time worsening.
We chose MPROLOG, since this paper is part of the PIPE-project (**P**arallel **I**nferencing **P**ROLOG **E**nvironment)/PIPE 84+85/. The aim of this project is to build a parallel MPROLOG machine - based on the WAM - to speed up present and future MPROLOG programs. Hence our sort extension should be able to be implemented on this machine. On the other hand no difficulties should arise in adapting this sort mechanism to other Prolog dialects.

The variables in logic programming languages can be regarded to be universally quantified (first order logic) variables, that is they might be instantiated by every individual of the underlying universe. On the other hand, if one wants to describe for example some mathematical propositions or natural problems, the variables frequently are restricted to special domains of the universe (we will call them sorts):

$\forall x \in \mathbf{N}: x > 0.$
All men are mortal.

Here the quantifiers range only over the domains `N´ (natural numbers) and `men´. Clearly both propositions can be transformed into normal first order logics (without sorts):

$\forall x: x \in \mathbf{N} \Longrightarrow x > 0.$
All x: (x is man) implies (x is mortal).

But it is well known in the field of automated reasoning systems, that using socalled many sorted first order calculi will shorten the search space and hence lead to faster deductions /Wal 84/, /Sch 85/. This will be a main reason for designing sorted Prolog versions /Fr 85/, /GM 85/. Another reason will be, that restricted quantification can be regarded as the introduction of a typing mechanism into logic programming languages. This will help to respectively avoid and detect syntax errors in logic programs /Mi·84/, /MK 84/. The following examples - written in (sorted) MPROLOG - should demonstrate those

advantages:

Example1a /Fr 85/:

```
vehicle(X) :- bicycle(X).
vehicle(X) :- car(X).
bicycle(b1).
  . . .
bicycle(bn).
car(c1).
  . . .
car(cm).
car(mycar).
has(X,tires) :- vehicle(X).
has(X,doors) :- car(X).
owns(alan,mycar).

?- has(X,tires),has(X,doors),owns(alan,X).
```

The program will consider all bicyles and all cars to succeed the first goal `has(X,tires)´. Then it always has to backtrack for solving the next goals until it finally will succeed with `X = mycar´. In the following sorted version the variable will only be instantiated by the sort `vehicle´ and later on by the sort `car´, before it finally will succeed with the constant `mycar´.

Example1b:

```
csort(b1,bicycle).
  . . .
csort(bn,bicycle).
csort(c1,car).
  . . .
csort(cm,car).
csort(mycar,car).
subsort(bicycle,vehicle).
subsort(car,vehicle).
has(X:vehicle,tires).
has(X:car,doors).
owns(alan,mycar).

?- has(X,tires),has(X,doos),owns(alan,X).
```

The following errorneous definition of the reverse and append relation for lists will lead to a staticly recognizable error, if sorts are used.

Example2a /Mi 84/:

```
revbad([ ],[ ]).
revbad([X | L1 ],L2) :- revbad(L1,L),     app(L,X,L2).
                                          ------------

app([ ],L,L).
app([X | L1 ],L2, [X | L3]) :- app(L1,L2,L3).
```

The underlined goal of the reverse-definition should correctly be `app(L,[X],L2)´, but the error might only arise dynamicly by returning `fail´, if we for example call it with a list and its reversed list, instead of outputting `true´. But using sorts `list´ and `any´ (restriction to the sort `any´ is equivalent to unrestricted quantification) an error will arise (staticly), because the `X´ in the underlined goal can be instantiated to a non-list, while `app´ requires lists only as arguments.

Example2b:

(Note, that the sort of a variable must only be declared at its first occurrence; `any´-sorted variables need no declaration.)

```
psort(revbad(list,list)).
psort(app(list,list,list)).
fsort([any|any],list).
csort([ ],list).

revbad([ ],[ ]).
revbad([X|L1:list],L2:list) :- revbad(L1,L:list),        app(L,X,L2).
                                                          ___________

app([ ],L:list,L).
app([X|L1:list],L2:list, [X|L3:list]) :- app(L1,L2,L3).
```

Sorted Prolog

There are several approaches to introduce types or sorts in Prolog (or logic programming in general) /Mi 84/,/MK 84/,/Fr 85/,/GM 85/. Some of them deal with a typing mechanism used only for type checking, as in common programming languages, and not for computation. But since Prolog implementations can be regarded as automated reasoning systems, a logic based approach would be to introduce restricted quantification like in many sorted first order logics as it is done in automated theorem proving /MKRP 84/. Therefore, we require a signature of finite sets of sort, constant, function, predicate and variable symbols. The set of sorts is partially ordered by a subsort ordering with exactly one top sort `any´, such that every two sorts will have at least one maximal common subsort (this is sometimes called a complete supremum semilattice).

Every variable and every constant will be declared to have a fixed sort. A range sort and n (maximal) argument sorts will be assigned to every n-ary function symbol and every n-ary predicate will get n (maximal) argument sorts. We agree upon the default sort to be \`any´ in all cases.
Terms, literals, clauses and so on are defined as common in logic programming /Ll 83/, but with the following restrictions:
Let the sort of a structure (= complex term) be the range sort of its function symbol. Then a term or a literal will be well sorted, iff its subterms will have a sort less than or equal to the declared argument sorts of the leading symbol. Other terms and literals are not allowed. In a program the sort relations and declarations of the symbols must be given before the first occurrence of those symbols.
The deduction will be done by usual SLD-resolution, but the unification procedure /Ro 65/ will be restricted by:
(1) A variable and a non-variable are unifiable, iff the sort of the non-variable is a subsort of that of the variable.
(2) Two variables are unifiable, iff there is a maximal common subsort of their sorts (note, that this is unique by the definition of the sort hierarchy). If this subsort is not the sort of one of the variables, the unifier is represented by binding both variables to a \`new´ variable with that subsort.
The soundness and completeness of the many sorted SLD-resolution is an immediate consequence of the results of /Wal 84/ and /Sch 85/ (if unification is done with occur-check).
The following example demonstrates the use of sorts (see also the former examples):

Example3:

```
 standard MPROLOG              |   sorted MPROLOG
---------------------------    |---------------------------
 person(john).                 |   csort(john,person).
 person(mary).                 |   csort(mary,person).
 dog(bill).                    |   csort(bill,dog).
 animal(X) :- dog(X)           |   subsort(dog,animal).
 likes(john,mary).             |   likes(john,mary).
 likes(john,bill).             |   likes(john,bill).
                               |
 ?- likes(john,X),animal(X).   |   ?-likes(john,X:animal).
```

The variable \`X´ is restricted to the sort \`animal´, and hence it will not unify with \`mary´, but with \`bill´, who is of sort \`dog´, a subsort of \`animal´.
In the examples we have already seen some of the new built-in predicates for sort declarations. The formal definition of their syntax might be (see also the MPROLOG manual /HU 82/):

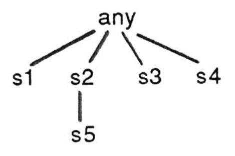
```
<sort declaration> ::=
            subsort(<name>,<name>)<full stop>/
            csort(<name>,<name>)<full stop>/
            fsort(<name>(<sort list>),<name>)<full stop> /
            psort(<name>(<sort list>))<full stop>

<sort list> ::=
            <name>/
            <name>,<sort list>
```

\`subsort(s1,s2).´ declares its first argument to be a subsort of its second one.
\`csort(c,s).´ assigns the sort \`s´ to the constant \`c´.
\`fsort(f(s1,...,sn),s).´ declares the n-ary function symbol \`f´ to have range sort \`s´ and argument sorts \`s1,...,sn´.
\`psort(p(s1,...,sn)).´ restricts the predicate \`p´ to the argument sorts \`s1,...,sn´.
In a program such declaration have to be given before the first occurrence of the symbols in any clause. The variable sort declaration will be given in the clauses.

```
<term> ::=
            <number>/
            <name>/
            <variable>/
            <variable>:<name>/
            <compund term>/
            (<term>)
```

Example4:

```
subsort(s5,s2).
psort(p(s2,s1,any)).
fsort(f(s1,s3,s4),s5).
csort(a,s4).
p(f(X:s1,Y:s3,a),X,Z).
```



The second occurence of the variable \`X´ needs no further sort restriction, while the variable \`Z´ is regarded to have (default) sort \`any´. Since \`s5´ is a subsort of \`s2´ and \`f´ has range sort \`s5´, the first argument of \`p´ has a correct sort.

As side effects these sort declarations will influence the unification procedure and the syntax checker. Terms and literals, which are not well sorted, will lead to a syntax error. A subsort ordering, which will not be a complete supremum semilattice, will also produce an error.

The Sort Unification Instructions

In this section we define the sort unification instructions extending the WARREN Abstract Machine (WAM) to sorts. We use the same notation as in /Be·85,1/.
The WAM is defined by an abstract instruction set for the compilation of Prolog programs. The idea behind it is to transform the unification procedure for two unknown terms into several special unification procedures determined by the structures of the clause headers. Those structures are completely specified when the program is created. Hence they can be compiled into special unification code, that can only unify terms with an analoguous structure, but can do this very efficiently.
Additionally there are some instructions for memory management and Prolog procedure calls. If we have for example a clause of the following form,

```
h( . . . ) :- g1( . . . ), g2( . . . ).
```

the instruction code will do something like the following /Be 85,1/,

```
allocate environment
unify h( . . . ) with the calling goal
initialize argument registers for g1
call g1
initialize argument registers for g2
call g2
deallocate environment
return from clause.
```

For this several registers and memory stacks need to be assigned. /Be 85,1/ contains a detailed description. The reader is also referred to the original papers /War 77/, /War 83/ and the WAM tutorial /GLLO 84/.
Our extension needs only to guarantee that the unification codes respect the sort hierarchies. So we have to introduce some new GET, PUT and UNIFY instructions simulating the unification of goal and header arguments with non-`any´ sorts; the original instructions will work with `any´-sorted arguments. The allocation of the environments will also be somewhat modificated, because we may have to store some sort information about the terms and literals. Variables with a non-`any´ sort now contain as value their sort instead of a self-reference. Unrestricted variables do however contain a self-reference like in the original machine.

S_PUT_Y_VARIABLE   Yn , Ai , S
This instruction represents a goal argument that is an unbound variable with sort S. It puts the address of variable Yn into register Ai . The variable is determined by an offset in the current environment on the local stack, containing the sort S and tagged by S_UNBOUND.

S_PUT_X_VARIABLE   An , Ai , S
This instruction represents an argument of the final goal that is an unbound variable with sort S. It creates an unbound variable on the global stack with sort S as content and puts references to it into the registers An and Ai .

The PUT....VALUE instructions need no equivalent for the sort case, since the sort of a bound variable is the sort of its value being available after dereferencing. But remember, that in opposite to the WAM an unbound (sorted) variable now will not have a reference to itself but to its sort, hence the execution of this instructions has to watch this case.

S_PUT_CONSTANT   C , Ai , S
This instruction represents a goal argument that is a constant with sort S. It pushes C and S onto the heap and puts a corresponding sorted-constant pointer into Ai .

S_PUT_STRUCTURE   F , Ai , S
This instruction marks the beginning of a structure with sort S occuring in a goal literal. The functor F together with its sort S is pushed onto the global stack and a sorted-structure pointer is put into Ai . Execution then proceeds in "write" mode.

In the next instructions the sort unification procedure is used:

S_GET_Y_VARIABLE   Yn , Ai , S
This instruction represents a head argument that is an unbound variable with sort S. It stores the sort S in Yn and unifies this variable with the content of register Ai.

S_GET_X_VARIABLE   An , Ai , S
Same as S_GET_Y_VARIABLE but An is a register.

We need no S_GET_VALUE instructions. To garantee the speed, if there are no sort declarations, the unification procedure should only do the sort checks, if

the sort table is not empty. This might be controlled by special sort/no-sort modes.

S_GET_CONSTANT C , Ai , S
This instruction represents a head argument being a constant with sort S. It gets the value of Ai and dereferences it. If the result is a variable with a greater sort than S, the constant and its sort is pushed onto the heap and the variable is bound to it. Otherwise, if the result is not the constant C, backtracking occurs.

S_GET_STRUCTURE F , Ai , S
This instruction marks the beginning of a structure with sort S occurring as head argument. It gets the value of Ai and dereferences it. If the result is a variable with a greater sort than S, the functor and its sort is pushed onto the heap and the variable is bound to it. Execution proceeds in "write" mode. Otherwise, if the result is a structure and its functor is F, the register NEXTARG is set to point to the arguments of the structure, and execution proceeds in "read" mode. Otherwise backtracking occurs.

S_UNIFY_Y_VARIABLE Yn , S
This instruction represents an unbound variable with sort S occurring as a structure argument. If it is executed in "read" mode, it gets the next argument from NEXTARG and unifies it with the variable Yn. If execution is done in "write" mode, it pushes a new unbound variable with sort S onto the heap and stores a reference to it in Yn.

S_UNIFY_X_VARIABLE An , S
Same as S_UNIFY_Y_VARIABLE , but the variable is a temporary register variable.

Again no S_UNIFY....VALUE instructions are needed. The original instruction must do sort unification, if the sort table is not empty.

S_UNIFY_CONSTANT C , S
This instruction represents a structure argument being a constant of sort S. If it is executed in "read" mode, it gets the next argument from NEXTARG and dereferences it. If the result is a variable with greater sort than S, the constant C and its sort are pushed onto the heap and bound to this variable. Otherwise, if the result is not C, backtracking occurs. If execution is in "write" mode, the constant C and its sort are pushed onto the heap.

Since lists are built-in structures, they need no special sort instructions. But we would like to assert a special built-in sort `list´ and to define the constant `nil´ and the list constructor [.|.] having (range) sort `list´ and argument sorts `any´. The original list instructions then should be changed to work with the sort `list´. The integers and the arithmetic functions might also get special inbuilt sorts.

In the following we give the compile code of the entrance examples, both the standard and the sorted version.

Example5a:

```
vehicle/1     try_me_else V2
              execute bicycle/1
V2            trust_me_else fail
              execute car/1

bicycle/1     switch_on_term Bv,Bc,fail,fail
Bc            switch_on_constant ...,(b1 B1, ... , bn Bn, fail)
Bv            try_me_else B2a
B1            get_constant b1,A1
              proceed
B2a ...        . . .
Bna           trust_me_else fail
Bn            get_constant bn,A1
              proceed

car/1         switch_on_term Cv,Cc,fail,fail
Cc            switch_on_constant ...,(c1 C1, ... , cm Cm, mycar C , fail)
Cv            try_me_else C2a
C1            get_constant c1,A1
              proceed
C2a ...        . . .
Cma           try_me_else Ca
Cm            get_constant cm,A1
              proceed
Ca            trust_me_else fail
C             get_constant mycar,A1
              proceed

has/2         try_me_else H2
              get_constant tires,A2
              execute vehicle/1
H2            trust_me_else fail
              get_constant doors,A2
              execute car/1

owns/2        trust_me_else fail
              get_constant alan,A1
              get_constant mycar,A2
              proceed
```

```
query/0        allocate 1
               put_y_variable Y1,A1
               put_constant tires,A2
               call has/2
               put_y_value Y1,A1
               put_constant doors,A2
               call has/2
               put_constant allan,A1
               put_y_value Y1,A2
               execute owns/2
```

Example5b: The sorted version (example1b) will be encoded this way.

```
csort/2        switch_on_term Cv,Cc,fail,fail
Cc             switch_on_constant ...,(b1 B1,....,bn Bn,c1 C1,...,cm Cm, mycar C,fail)
Cv             try_me_else B2a
B1             get_constant b1,A1
               get_constant bicycle,A2
               proceed
B2a ... Cm       . . .
Ca             trust_me_else fail
C              get_constant mycar,A1
               get_constant car,A2
               proceed

subsort/2      switch_on_term Sv,Sc,fail,fail
Sc             switch_on_constant ...,(bicycle S1, car S2,fail)
S1a            try_me_else S2a
S1             get_constant bicycle,A1
               get_constant vehicle,A2
               proceed
S2a            trust_me_else fail
S2             get_constant car,A1
               get_constant vehicle,A2
               proceed

has/2          try_me_else H2
               s_get_y_variable Y1,A1,vehicle
               get_constant tires,A2
               proceed
H2             trust_me_else fail
               s_get_y_variable Y1,A1,car
               get_constant doors,A2
               proceed
owns/2         trust_me_else fail
               get_constant A1
               s_get_constant mycar,A2,car
               proceed

query/0        allocate 1
               put_y_variable Y1,A1
               put_constant tires,A2
               call has/2
               put_y_value Y1,A1
               put_constant doors,A2
               call has/2
               put_constant allan,A1
               put_y_value Y1,A2
               execute owns/2
```

Example6: We give an encoding of the (correct) reverse and append definitions with sorts (see example2b).

```
psort/1     switch_on_term Pv,fail,fail,Ps
Ps          switch_on_structure ..,(rev P1,app P2,fail)
Pv          try_me_else P2a
P1          get_structure rev/2,A1
            unify_constant list
            unify_constant list
            proceed
P2a         trust_me_else fail
P2          get_structure app/3,A1
            unify_constant list
            unify_constant list
            unify_constant list
            proceed

fsort/2     trust_me_else fail
            get_list A1
            unify_constant any
            unify_constant any
            get_constant list,A2
            proceed

csort/2     trust_me_else fail
            get_nil A1
            get_constant list,A2
            proceed

rev/2       switch_on_term Rv,Rc,Rl,fail
Rv          try_me_else R2
Rc          get_nil A1
            get_nil A2
            proceed
R2          trust_me_else fail
Rl          allocate 3
            get_list A1
            unify_y_variable Y2
            s_unify_x_variable A1,list
            s_get_y_variable Y3,A2,list
            s_put_y_variable Y1,A1,list
            call rev/2
            put_unsafe_value Y1,A1
            put_list A2
            unify_y_value Y1
            unify_nil
            put_y_value Y3,A3
            deallocate
            execute app/3

app/3       switch_on_term Lv,Lc,Ll,fail
Lv          try_me_else L2,3
Lc          allocate 1
            get_nil A1
            s_get_y_variable Y1,A2,list
            get_x_value Y1,A3
            proceed
L2          trust_me_else fail
Ll          allocate 1
            get_list A1
```

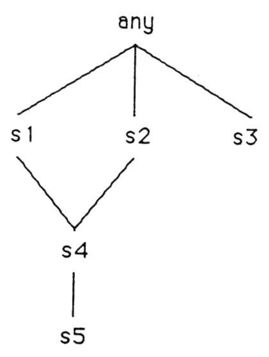
```
unify_x_variable A4
s_unify_x_variable A1,list
s_get_y_variable Y1,A2,list
get_list A3
unify_x_value A4
s_unify_x_variable A3,list
execute app/3
```

Since the sorts are a finite supremum semilattice, we can store them in a square table. Let $\{s_i : 1 \leq i \leq n\}$ be the set of sorts. The contents of the table are $inf(s_i,s_j)$, the uniquely determined maximal common subsort of s_i and s_j $(1 \leq i,j \leq n)$, if it exists, otherwise `fail´. The entrances of the table will be the different sorts. Since the table is symmetric, only half the storage is needed. The following example should demonstrate this.

Example7:

The minimal information to be stored is the marked part of the table.

sort hierarchy



sort table

	any	s1	s2	s3	s4	s5
any	any	s1	s2	s3	s4	s5
s1	s1	s1	s4	∅	s4	s5
s2	s2	s4	s2	∅	s4	s5
s3	s3	∅	∅	s3	∅	∅
s4	s4	s4	s4	∅	s4	s5
s5	s5	s5	s5	∅	s5	s5

Conclusion

During the writing of this report a similiar approach by M. Huber /Hub 85/ to designing a sorted Prolog machine has reached us. It will be an extension of the L-machine /Nie 85/, a sequential Prolog machine. It is similiar to our machine, because the L-machine is also a WARREN Abstract Machine and the sort extension is based on the same papers as our extension (/Wal 84/, /Sch 85/). The main difference apart from the syntax is, that for the sorted L-machine the old unification instructions are changed, while we define new sort unification instructions. Our approach therefore might result in a better run time behavior, if no sorts are used.

Several further extensions will be part of our future investigations. It might be possible to incorporate the sort restrictions into the indexing mechanism. In addition the many sorted calculi are also defined for more general sort hierarchies and for socalled polymorphic functions. But then the most general unifier of two given terms will no longer be uniquely determined (/Wal 84/, /Sch 85/). This will also occur, if unification is extended to unification under equational theories /Si 84/.

It is not known, if and how these extensions can be introduced into present Prolog control mechanism, especially how this might be realized in an abstract Prolog machine.

Literatur

/Be 85,1/	Beer J. Comments on Compiling PROLOG Programs Using Warren´s Abstract PROLOG Instruction Set PIPE-Report, GMD-FIRST, Berlin 1985
/Be 85,2/	Beer J. An Extended PROLOG Instruction Set for the PIPE PIPE-Report, GMD-FIRST, Berlin 1985
/Fr 85/	Frisch A.M. An Investigation into Inference with Restricted Quantification and a Taxonomic Representation Logic Progeramming Newsletters 6, 1984/85
/GLLO 84/	Gabriel J., Lindholm T., Lusk E.L., Overbeek R.A. A Tutorial on the WARREN Abstract Machine for Computational Logic Mathematics and Computer Science Division Argonne National Laboratory 1984/85
/GM 85/	Goguen J.A., Meseguer J. EQLOG: Equality, Types; and Generic Modules for Logic Programming Functional and Logic Programming 1985
/HU 82/	MPROLOG-Manual Institute for Coordination of Computer Techniques Budapest 1982
/Hub 85/	Huber M. L-Maschine: Maschinenmodell mit Sorten Univerity of Karlsruhe 1985

/Ll 83/	Lloyd J.W. Foundations of Logic Programming Berlin-Heidelberg-New York-Tokyo 1983
/Mi 84/	Mishra P. Towards a Theory of Types in PROLOG Proc. IEEE Internat. Symp. Logic Programming 1984
/MK 84/	Mycroft a., O'Keefe R.A. A Polymorphic Type System for PROLOG Artificial Intelligence 23, 1984
/MKRP 84/	Karl Mark G Raph The Markgraf Karl Refutation Procedure Memo-Seki, Universities of Kaiserslautern & Karlsruhe 1984
/Nei 85/	Neidecker B. L-Maschine: Maschinenmodell Universität Karlsruhe 1985
/PIPE 84,85/	Parallel Inferencing PROLOG Environment Workshops and Reports of the PIPE-Project Berlin-Kaiserslautern-Paderborn 1984/85
/Ro 65/	Robinson J.A. Computational Logic: The Unification Computation Machine Intelligence 6, 1965
/Sch 85/	Schmidt-Schauß M. A Many-sorted Calculus with Polymorphic Functions Based on Resolution and Paramodulation Internal Report, Universität Kaiserslautern, 1985
/Si 84/	Siekmann J. Universal Unification Proc. Conf. Automated Deduction, Los Angeles 1984
/Wal 84/	Walther C. Unification in Many-sorted Theories Dissertation, Universität Karlsruhe, 1984
/War 77/	Warren D.H.D. Compiling Predicate Logic Programs D.A.I. Research Report, University of Edinburgh, 1977
/War 83/	Warren D.H.D. An Abstract PROLOG Instruction Set SRI Technical Note, Stanford, 1983