To appear in: *The Journal of Logic Programming, 18(2), pp. 123–148, Febr. 1994*

Run-time type computations in the Warren Abstract Machine

Christoph Beierle* and Gregor Meyer
IBM Germany, Scientific Center
Institute for Knowledge Based Systems
P.O. Box 10 30 68
D-6900 Heidelberg, Germany
e-mail: beierle@vnet.ibm.com, gmeyer@dhdibm1.bitnet

April 1992; revised April 27, 1993

Abstract

The type concept of the logic programming language PROTOS-L supports sorts, subsort relationships and parametric polymorphism. Due to the order-sortedness types are also present at run time, replacing parts of the deduction process required in an unsorted version by efficient type computations. Together with the polymorphism most of the flexibility of untyped logic programming carries over to the order-sorted approach. The operational semantics of PROTOS-L is based on polymorphic order-sorted resolution. Starting from an abstract specification, we show how this operational semantics can be implemented efficiently by an extension of the Warren Abstract Machine and give a detailed description of all instructions and low-level procedures responsible for type handling. Since the extension leaves the WAM's AND/OR structure unchanged, it allows for all WAM optimizations like last call optimization, environment trimming, etc. Moreover, the extension is orthogonal in the sense that any program part not exploiting the facilities of computing with subtypes is executed with almost the same efficiency as on the original WAM.

*current address from July 1, 1993: FernUniversität Hagen, Fachbereich Informatik, Bahnhofstr. 46-48, D-58084 Hagen, Germany; *e-mail:* Christoph.Beierle@fernuni-hagen.de

Contents

1 Introduction

When introducing types into logic programming there are various different points of view. One can distinguish between an *operational approach* where the notion of type is merely a syntactic one and the underlying logic is still untyped, and a *semantical approach* where a typed logic is used. For instance, [24] uses the notions of *descriptive* and *prescriptive* typing. For pragmatic reasons, one has to compromise between the expressive power of the type system and an efficient operational semantics. For instance, type systems integrated into logic programming that allow for unrestricted equational types are often considered to be very inefficient when it comes to implementing them. On the other hand a system like Turbo Prolog is much too inflexible in its type system and sacrifices much of the attractiveness of logic programming.

In this paper we show how the extension of logic programming to a powerful semantical type concept having sorts, subsort relationships and parametric polymorphism can be efficiently implemented on a WAM like architecture.

From an implementation point of view type concepts for logic programming can be classified according to how much of the type information is handled with at compile time and how much at run time. For instance, the polymorphic type system for Prolog presented in [21] that is based on the parametric polymorphism of ML [20] [9], is tailored such that compile time type checking is sufficient. This is not the case in [14] and [13] where some of the restrictions in [21] are dropped, effectively allowing also ad-hoc polymorphism which is then used for higher-order programming. For an implementation of λProlog [19] [22], types are also needed at run time through typed unification. In Gödel [15] types play the central role in providing a logical semantics for meta-programming constructs. For most Gödel programs static type checking is sufficient. The current Gödel system does not support dynamic type checking, but in some cases Gödel types also have to be considered at run time.

All approaches cited so far are *many-sorted* since they do not support hierarchical relationships between types. Such subtype relationships are the characteristic feature for the so-called *order-sorted* setting. Although the well-known slogan "well-typed programs do not go wrong" in the sense as used in [9] is still valid in order-sorted logic programs, unification failures may occur due to incompatible type constraints which is a consequence of the semantical approach to typing. In the order-sorted approach of Eqlog [11] with subsort relationships types are present at run time through typed unification. The same is true for e.g. the approach of [16] and for LOGIN [2]. [10] extends the polymorphic approach of [21] to an order-sorted setting, but poses some data flow restrictions to ensure that static type checking is sufficient. In TEL [26, 23] order-sorted types are

combined with parametric polymorphism, requiring also types to be present at run time; however, the current implementation of TEL is incorrect in the sense that solutions provided by the system are not always logical consequences of the input program. This is due to the fact that the implementation only uses the ordinary term unification of Quintus Prolog [26]. The type system of PROTOS-L [3] has been derived from TEL by disallowing explicit subtype relationships between polymorphic types. In addition to the concepts discussed in this paper PROTOS-L has modules, deductive database access, an object-oriented interface to OSF/Motif, and various built-ins and extra-logical features [5].

A complete inference engine for PROTOS-L has been implemented, called PROTOS Abstract Machine (PAM) [25, 7] extending the WAM [29, 1] by the required polymorphic order-sorted unification. Starting with an abstract specification of polymorphic order-sorted resolution derived from [27] and tailored towards the type concept of PROTOS-L, we show how the PAM can be derived from the WAM. This extension is orthogonal to the WAM concepts realizing the AND / OR structure. Moreover, it is also orthogonal in the sense that any program part not exploiting the facilities of computing with subtypes is executed with almost the same efficiency as on the original WAM: in this case, only the original WAM instructions are used and there is only a small low-level overhead in the tagging and trailing of value cells (cf. 5.2 and 7.1).

The major deviation from the WAM is in the term representation. The representation of terms is extended by modifying the WAM representation of unbound variables to hold the type restriction of the variables. Unification of terms can then still be carried out almost as in the WAM. Only when binding a variable `X` to another variable or non-variable term `t` the type restriction of `X` and also possibly the type restrictions of the variables occurring in `t` must be taken into account. This is reflected by a modification of the WAM low-level binding operation which in the PAM is also responsible for the type computations.

Using Gurevich's evolving algebras [12] and the WAM correctness proof in [8], a mathematical correctness proof for the PAM scheme of extending the WAM to run time type constraints is given in [4].

Most of the work on the implementation of typed logic programming, including all of the related work cited above, has put the emphasis on compile time type analysis and inferencing, and on methods of avoiding type considerations at run time. Our approach seems to be the first one to provide an implementation scheme based on an abstract machine for a logic programming language with polymorphic order-sorted typing. In the recent work of [17], a WAM based implementation scheme for a logic programming language with ML-style typing (with the possibility of ad-hoc polymorphism and thus the necessity for run time type checking) is presented, and an extension to a PROTOS-L implementation is

outlined. We will compare the PAM implementation to this approach in Section 7.5 after having described the PAM low-level type computation procedures.

The rest of this paper which revises and extends the work reported in [7], is organized as follows: In Section 2, we present the characteristic features of PROTOS-L related to types. In Section 3, the required changes to the WAM are summarized, and in Sections 4 and 5 the sort and symbol tables of the PAM are introduced. (Note: we will not make a clear cut distinction between sorts and types in this paper, and use both terms interchangeably.) All PAM instructions responsible for unification are discussed in Section 6. In Section 7, we complete the PAM description by presenting all low-level instructions which differ from the WAM, and Sections 8 contains some conclusions and points out possible extensions.

Acknowledgements: The work reported here has been carried out within the EUREKA project PROTOS ("Logic Programming Tools for Building Expert Systems", EU56). We would like to thank all members in the PROTOS project team at the IBM Institute for Knowledge Based Systems for their enthusiastic support of the project, in particular our former colleague Heiner Semle who implemented the first prototype of the PAM. Thanks also to the anonymous referees for their helpful comments.

2 Polymorphic order sorted types in PROTOS-L

2.1 Type definitions

In PROTOS-L there are monomorphic types which are defined by enumerating their constructor functions (generating the elements of the given type) and/or by the union of subtypes. Thus, the monomorphic types are ordered by a partial order, denoted by $\leq$. For reasons of completeness of the unification algorithm (for the order-sorted case without polymorphism see [28], for the polymorphic order-sorted case see [27]) it is required that the monomorphic sort hierarchy forms a meet semi-lattice - i.e. lower bounds exist - with a smallest element (the empty type). We will denote this type both by $\perp$ and `BOTTOM` in this paper. However, no program is allowed to contain `BOTTOM` explicitly.

We assume that the greatest lower bound (GLB) of two sorts denotes their intersection; thus, if the GLB of two sorts $\mathtt{s}_1$ and $\mathtt{s}_2$ is `BOTTOM`, then $\mathtt{s}_1$ and $\mathtt{s}_2$ denote disjoint sets. (For another interpretation of sort hierarchies see for instance [6]).

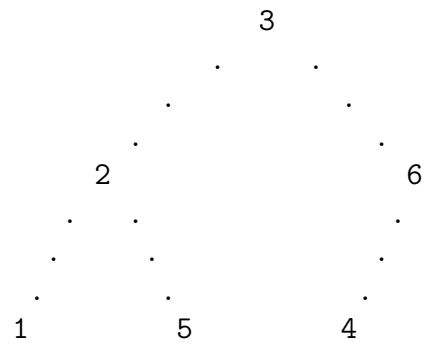
Polymorphic types are defined by enumerating their constructors, e.g. labelled binary trees which are generic in the type `T` of labels could be defined by:

```
bin_tree(T) := { leaf:  T,
                 left:  bin_tree(T) x T,
                 right:  T x bin_tree(T),
                 both:  bin_tree(T) x T x bin_tree(T) }.
```

Now the term

```
both(both(leaf(1), 2, leaf(5)), 3, left(leaf(4), 6))
```

is a term of type `bin_tree(nat)` where `nat` is the built-in type of natural numbers $\{0, 1, 2, 3, \ldots\}$. It represents the binary tree



All polymorphic types are monotonic functions with respect to the subtype relationships between monomorphic types, e.g. `bin_tree(nat)` is a subtype of `bin_tree(int)` if `nat` is a subtype of `int`. However, no subtypes of a polymorphic type may be defined explicitly in a PROTOS-L program - as opposed to TEL where this is allowed. A major advantage of this restriction in PROTOS-L is that the subtype relationship between all instances of polymorphic types can be reduced to the subtype relationship between monomorphic types. Another advantage is that any polymorphic type applied to a (with respect to the type hierarchy) maximal type yields again a maximal type which is of great operational importance because of the approximations of type terms (see below).

A PROTOS-L program consists of type and relation definitions. Each relation definition is a list of clauses together with a declaration of its argument domains. A part of a simple example program is the following:

```
nat    := { null } ++ posint.
posint := { succ : nat }.
list(S)     := { nil,
                 . : S x list(S) }.
double(S)   := { p : S x S }.
```

```
pair(S1,S2) := { mk_pair : S1 x S2 }.

rel plus: nat x nat x nat.
    plus(null,N,N).
    plus(succ(M),N,succ(MN)) <-- plus(N,M,MN).

rel transpose: double(S) x double(S).
    transpose(p(X,Y),p(Y,X)).

rel no_null: list(nat) x list(nat).
    no_null(L,L) <-- L : list(posint).
    no_null(L,L1) <-- treat_null(L, L1).
```

We will use the list constructor '.' as a right-associative infix operator. Of course, in the actual PROTOS-L system the types `nat`, `posint`, and `list(S)` are built-in types, but we will use their explicit definition as given above for illustration purposes in the examples of this paper.

2.2 Type checking and type inference

An important requirement is that every program must be well-typed. Essentially, this means that every clause of the program is well-typed for a mapping of the variables occurring in that clause to appropriate types. In [26] algorithms for the automatic type checking and type inference in TEL are given; these algorithms can be used for PROTOS-L as well.

Thus, for the rest of this paper we assume that every clause is well-typed and comes with a so-called *prefix* P which has been derived automatically by the compiler and which assigns a type to every variable occurring in the clause. For instance, we have

type prefix	*clause without types*
`{N:nat}`	`plus(null,N,N).`
`{M:nat & N:nat & MN:nat}`	`plus(succ(M),N,succ(MN)) :- plus(N,M,MN).`
...	
`{X:S & Y:S}`	`transpose(p(X,Y),p(Y,X)).`
...	
`{L:list(posint)}`	`no_null(L,L).`
`{L:list(nat) & L1:list(nat)}`	`no_null(L,L1) <-- treat_null(L,L1).`

for the clauses given above. Furthermore, [27] introduces the operationally important notion of *approximations* of type terms: type terms which are maximal

in the lattice of type terms can be neglected during run time unification. In particular, due to the notion of parametric polymorphism, every type variable can be considered maximal. Maximal types can be neglected in an operational semantics at run time since the type checking in the compiler ensures that the abstract machine will not produce an undetected type inconsistency. We also impose a restriction on the head of clauses already given in [21]: the arguments in the clause head must always be of the "most general type", i.e. a variant of the type in the declaration of the relation.

The approximation $\downarrow\tau$ of a type term τ is obtained by systematically replacing every maximal type by the special symbol $\top$ (which we will refer to also as `TOP`):

$\downarrow\tau$	$=$	$\top$	if τ is a sort variable
$\downarrow\tau$	$=$	$\top$	if τ is a sort constant that is maximal in the partial order on the sorts
$\downarrow\xi(\tau_1,\ldots,\tau_n)$	$=$	$\top$	if $\downarrow\tau_1 = \ldots = \downarrow\tau_n = \top$
		$\bot$	if $\xi(\tau_1,\ldots,\tau_n)$ cannot be instantiated (see below)
		$\xi(\downarrow\tau_1,\ldots,\downarrow\tau_n)$	otherwise

A type term can be instantiated or is *inhabited* if there is a ground term which belongs to that type. For instance, `list(`$\bot$`)` can be instantiated by the empty list, whereas `bin_tree(`$\bot$`)` (Section 2.1) can not be instantiated.

From now on we assume that every prefix of a clause has already been replaced by its approximation. In particular, this means that no program variable will have a type variable in its type restriction. For instance, `{X:S & Y:S}` is replaced by `{X:`$\top$ `& Y:`$\top$`}` because the compiler can ensure type correctness statically.

Now a goal P & F consists of a prefix P and a set of equations and relational literals F such that F is well-typed under P. P & F is in solved form if

$$P \texttt{ = } Y_1\texttt{: } tt_1 \texttt{ \& } \ldots \texttt{ \& } Y_n\texttt{: } tt_n$$
$$F \texttt{ = } X_1 \doteq t_1 \texttt{ \& } \ldots \texttt{ \& } X_m \doteq t_m$$

where we use the binary symbol $\doteq$ to couple two terms to be unified and where

1. the X_i and Y_i are pairwise distinct,
2. the X_i do not occur in $t_1,\ldots,t_m$, and
3. the type terms tt_i can be instantiated.

Note that compared to the ordinary unsorted situation F represents exactly the solution substitution, where P additionally assigns a type to the variables that are still free under the substitution F.

2.3 Operational semantics

The operational semantics of Prolog realized in the WAM is based on SLD resolution with term unification. Similarly, the operational semantics of PROTOS-L realized in the PAM is SLD resolution with polymorphic order-sorted unification. The resolution rule (RES) is

$$\frac{P \;\&\; F \;\&\; r(t_1,\ldots,t_n)}{P \;\&\; P' \;\&\; F \;\&\; t_1 \doteq t'_1 \;\&\; \cdots \;\&\; t_n \doteq t'_n \;\&\; B}$$

if $\{P'\}\ \ r(t'_1,\ldots,t'_n)$ <-- B is a variant of a program clause

and the rules for polymorphic order-sorted unification needed for PROTOS-L are given in Figure 1 (the case with polymorphic subtypes is given in [27]). The rules for elimination (E), decomposition (D), binding (B), and orientation (O) are exactly as in the unsorted case (cf. [18]) except that the binding rule refers to the prefix computation rules (ES) - (DS). (ES) is the first elimination rule for (monomorphic) types and arises e.g. from the binding rule (B) applied to

$$X{:}s \;\&\; X \doteq f(t_1,\ldots,t_n)$$

which requires that the target sort s' of the constructor f is a subsort of s. Similarly, the second elimination rule (ES') applies if in the situation above X has type restriction $\top$ (in which case f could belong to to a monomorphic or to a polymorphic type).

The merging rule (MS) arises from the binding rule (B) applied to

$$X{:}\tau \;\&\; Y{:}\tau' \;\&\; X \doteq Y$$

which requires to compute the *infimum* of two type terms. The infimum function generalizes the meet operation (i.e. greatest lower bounds) on the monomorphic sorts to the lattice of all type terms (with approximations):

$$\begin{array}{lcl}
inf(\top,\tau) & = & \tau \\
inf(\tau,\top) & = & \tau \\
inf(\tau,\tau') & = & \tau'' \\
 & & \quad \text{if } \tau \text{ and } \tau' \text{ are sort constants with} \\
 & & \quad \text{maximal common subsort } \tau'' \\
inf(\xi(\tau_1,\ldots,\tau_n),\xi(\tau'_1,\ldots,\tau'_n)) & = & \xi(inf(\tau_1,\tau'_1),\ldots,inf(\tau_n,\tau'_n)) \\
 & & \quad \text{if } \xi(inf(\tau_1,\tau'_1),\ldots,inf(\tau_n,\tau'_n)) \\
 & & \quad \text{can be instantiated} \\
inf(\tau,\tau') & = & \bot \\
 & & \quad \text{otherwise}
\end{array}$$

The decomposition rule (DS) arises from (B) e.g. when applied to

$$X{:}list(nat) \;\&\; Y{:}int \;\&\; L{:}list(int) \;\&\; X \doteq Y.L$$

(E) $$\frac{P \;\&\; E \;\&\; x \doteq x}{P \;\&\; E}$$

(D) $$\frac{P \;\&\; E \;\&\; f(t_1, \ldots, t_n) \doteq f(t'_1, \ldots, t'_n)}{P \;\&\; E \;\&\; t_1 \doteq t'_1 \;\&\; \cdots \;\&\; t_n \doteq t'_n}$$

(B) $$\frac{P \;\&\; x : \tau \;\&\; E \;\&\; x \doteq t}{P' \;\&\; \sigma(E) \;\&\; x \doteq t}$$

if x occurs in E but not in t, and where $\sigma = \{x/t\}$ and $P \;\&\; t : \tau$ reduces to the prefix P' using the rules (ES) ... (DS)

(O) $$\frac{P \;\&\; E \;\&\; t \doteq x}{P \;\&\; E \;\&\; x \doteq t}$$

if t is not a variable

(ES) $$\frac{P \;\&\; f(t_1, \ldots, t_n) : s}{P}$$

if $f : s_1 \ldots s_n \to s'$ and $s' \leq s$.

(ES') $$\frac{P \;\&\; f(t_1, \ldots, t_n) : \top}{P}$$

(MS) $$\frac{P \;\&\; x : \tau \;\&\; x : \tau'}{P \;\&\; x : inf(\tau, \tau')}$$

(DS) $$\frac{E \;\&\; f(t_1, \ldots, t_n) : \xi(\tau_1, \ldots, \tau_m)}{E \;\&\; t_1 : \downarrow\theta(\tau'_1) \;\&\; \cdots \;\&\; t_n : \downarrow\theta(\tau'_n)}$$

if $f : \tau'_1 \ldots \tau'_n \to \xi(\alpha_1, \ldots, \alpha_m)$ and where $\theta = \{\alpha_1/\tau_1, \ldots, \alpha_m/\tau_m\}$

Figure 1: The rules for polymorphic order-sorted unification: Term unification part (E) - (D), and type prefix computation part (ES) - (DS)

where the infix list constructor

$$. \ : \ \alpha \times list(\alpha) \rightarrow list(\alpha)$$

corresponds to the general constructor declaration

$$f \ : \ \tau'_1 \ldots \tau'_n \rightarrow \xi(\alpha_1, \ldots, \alpha_m)$$

in (DS). Note that the α_i are necessarily variables since ξ is a polymorphic type constructor whose declaration may only contain such general type variables for its arguments. In the given example, (DS) propagates the type restrictions given by the arguments of the polymorphic list type term to the arguments of the list constructor '.':

$$X{:}list(nat) \ \texttt{\&} \ Y{:}int \ \texttt{\&} \ L{:}list(int) \ \texttt{\&} \ Y{:}nat \ \texttt{\&} \ L{:}list(nat) \ \texttt{\&} \ X \doteq Y.L$$

Applying the rule (MS) twice and assuming $inf(int, nat) = nat$ we get

$$X{:}list(nat) \ \texttt{\&} \ Y{:}nat \ \texttt{\&} \ L{:}list(nat) \ \texttt{\&} \ X \doteq Y.L$$

In Section 6 we will show how these unification rules and in particular the prefix computation rules are reflected in the definition of the machine instructions of the PAM.

3 What had to be changed in the WAM

In order to adapt the Warren Abstract Machine [29] to polymorphic order-sorted unification as required in PROTOS-L most of its design could be left unchanged. This is true in particular for the realization of the AND/OR structure with choice points and environments, and for the general layout of the data areas consisting of the local stack, the global stack, the trail stack, and the various machine registers. Therefore, most WAM optimizations such as last call optimization and environment trimming carry over to the PAM. So we will not describe these parts of our abstract machine but refer to e.g. the WAM tutorial in [1].

Executing polymorphic order-sorted programs, there is a central task to handle sorts in the machine. This will be done with the aid of a *sort table* containing the static sort information available at compile time. In addition, we have a *symbol table* containing information about each constructor, e.g. its domain and target sort (see Section 4). The representation of terms in the WAM has to be extended in order to accommodate the type restrictions of variables (Section 5). Finally, the WAM instructions and low-level operations responsible for unification have to be adapted (Sections 6 and 7).

4 Sort and symbol table

4.1 Sort table

The sort table contains information about **built-in** sorts such as `int` and `list`, and about all **user defined** sorts. The purpose is to provide the unification procedures with access to the sort lattice defined by the program. Neglecting special representation aspects here, the sort table is introduced in an abstract way. Therefore, we only talk about 'sorts' although sometimes their indices into the sort table are meant.

The main functions provided by the sort table are:

- **sort_is_subsort: SORT $\times$ SORT $--> $ BOOL**

  `sort_is_subsort(s1,s2)` tests if sort `s1` is a subsort of `s2`. Both sorts are monomorphic.

- **sort_glb: SORT $\times$ SORT $--> $ SORT**

  `sort_glb(s1,s2)` computes the greatest lower bound of the monomorphic sorts `s1` and `s2`. This GLB may be `BOTTOM`, indicating that `s1` and `s2` are disjoint (cf. Section 2.1).

- **sort_arity: SORT $--> $ INT**

  `sort_arity(s)` is only used for polymorphic sorts `s`, e.g. `sort_arity(list)` $= 1$ and `sort_arity(pair)` $= 2$.

For polymorphic order-sorted unification we also need to know whether for a polymorphic type term there is a term belonging to that type. (For monomorphic sorts, the compiler ensures that they are not empty, i.e. there is a ground term for every monomorphic sort different from `BOTTOM`.) For instance, given the (standard) notions of `list(`α`)` and `pair(`α`,`β`)` as given in the example at the end of Section 2.1, `list(BOTTOM)` can be instantiated to the empty list `nil` as already pointed out in 2.2, while `pair(BOTTOM,INTEGER)` is empty since there is no pair without a first component. Therefore, the sort table also provides a function

- **inhabitation_mode: SORT $\times$ BOOL$^+$ $--> $ BOOL**

which tells whether polymorphic terms of the given sort `s` can be instantiated, depending only on the emptyness of the argument types, but not on the arguments themselves. More precisely, `inhabitation_mode(s,(`b_1`,...,`b_n`))` takes as

parameters a polymorphic sort `s` and n = `sort_arity(s)` flags $\texttt{b}_i$ marking empty and non-empty parameter sorts with `false` and `true`, respectively. The result is `true` if and only if there exists a non-empty sort term $\texttt{s(tt}_1\texttt{,...,tt}_n\texttt{)}$ where for for $\texttt{b}_i$ = `false`, $\texttt{tt}_i$ is `BOTTOM` and for $\texttt{b}_i$ = `true`, $\texttt{tt}_i$ is a non-empty sort term. For instance, for the type definitions

```
list(S)     := { nil,
                 . : S x list(S) }.
double(S)   := { p : S x S }.
pair(S1,S2) := { mk_pair : S1 x S2 }.
```

from Section 2.1 we have

```
inhabitation_mode(list, (true)) = true
inhabitation_mode(list, (false)) = true
inhabitation_mode(double, (true)) = true
inhabitation_mode(double, (false)) = false
inhabitation_mode(pair, (true,true)) = true
inhabitation_mode(pair, (false,true)) = false
inhabitation_mode(pair, (true,false)) = false
inhabitation_mode(pair, (false,false)) = false
```

Assuming the number of type parameters n to be relatively small, for each polymorphic type `s` and for each pattern $\texttt{b} = \texttt{(b}_1\texttt{,...,b}_n\texttt{)}$ the result of a call to `inhabitation_mode(s,b)` can be precompiled. A run time only a simple table look-up is performed.

4.2 Symbol table

As done with sorts, we will use the notion of 'constructor' or (Prolog) 'functor' and the related identifier in the machine with no distinction. To get information about the constructor symbols and their arguments defined in the program the symbol table provides the following functions.

- **constructor_arity: CONSTRUCTOR −− > INT**

  `constructor_arity(c)` returns the number of arguments of the constructor `c`.

- **target_sort: CONSTRUCTOR −− > SORT**

  `target_sort(c)` returns (a pointer to) the target sort of constructor `c`, i.e. the least sort to which `c` belongs.
  Note that the result of `target_sort` will be an index in the sort table, but as said before there is no need to elaborate this point here (see 4.1).

- **constructor_arg: CONSTRUCTOR × INT −− > SORTTERM_INDEX**

  `constructor_arg(c,i)` returns a pointer to the sort term which describes the i-th argument of `c`.

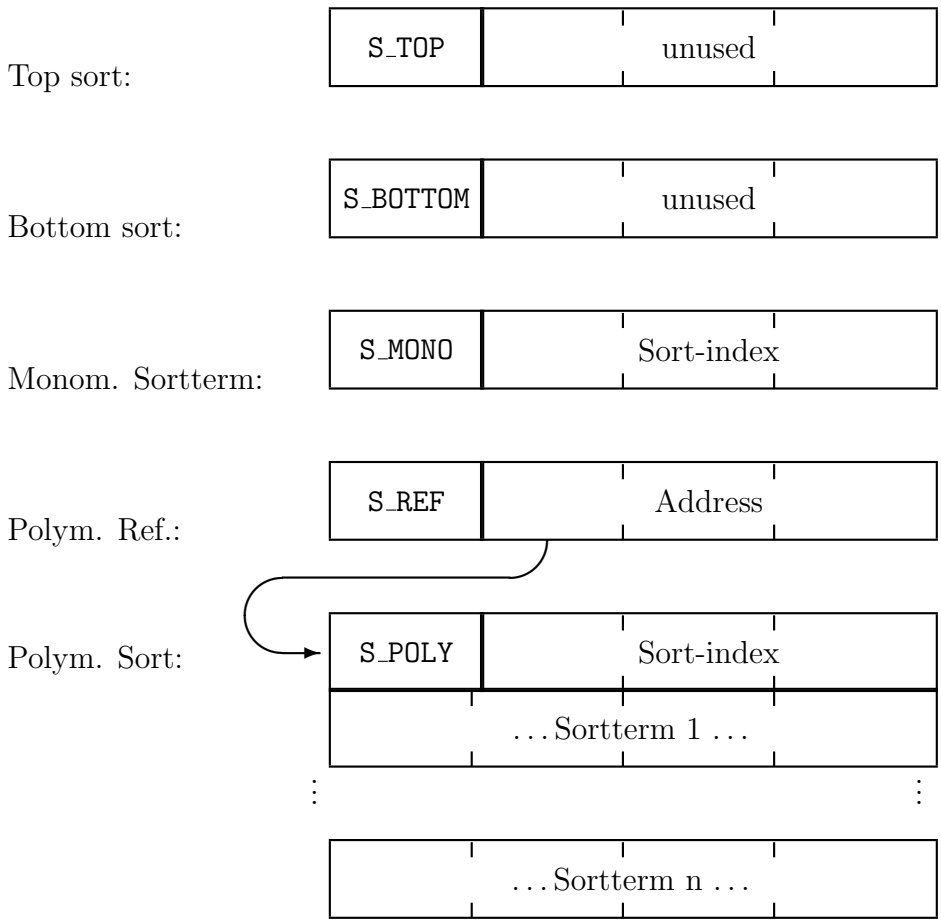
Constructors and their sort information are represented in the symbol table in a similar way as constructors in the WAM-stack. The argument domains of the constructors are represented by special sort terms which are introduced in the next section.

5 Term representation

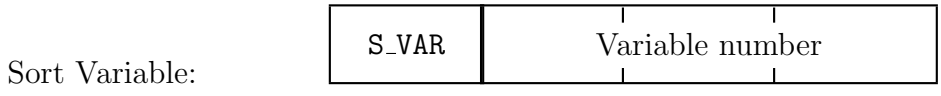
The term representation in the WAM stacks has to be changed in order to accommodate the sort information assigned to unbound variables. Apart from this change the term representation on the stack is as in the WAM: There are tags `CONST`, `STRUC`, and `REF` for constants, structures and references. Additionally, for various built-in types such as integers, lists, arrays, and strings there are special PAM tags which, however, will not be considered in detail. The current PAM implementation uses 4-byte words with one byte reserved for tags. Constructors only have a 1-bit tag which is set to '1' while all remaining bits are used to represent the constructor itself; the corresponding bit in every other tag is '0'.

5.1 Sort terms

The sort information attached to a variable (see below) may be a complex term; such sort terms are represented similar to ordinary terms. The top sort is simply indicated by the tag `S_TOP` while the value field is not used. In rare cases we need the sort `BOTTOM` in sort terms, indicated by the tag `S_BOTTOM`, e.g. if there is a variable restricted to `list(BOTTOM)` which may be instantiated later to `nil`. In the monomorphic case all sorts can be represented by the tag `S_MONO` combined with an index into the sort table. When polymorphic sorts appear, the representation of sorts becomes more complex, because these sorts have parameters. In analogy to constructors these sort terms are described by an index of the sort to which the sort term belongs (e.g. `list`) followed by n sort terms describing the argument sorts where n is the arity of the polymorphic sort.



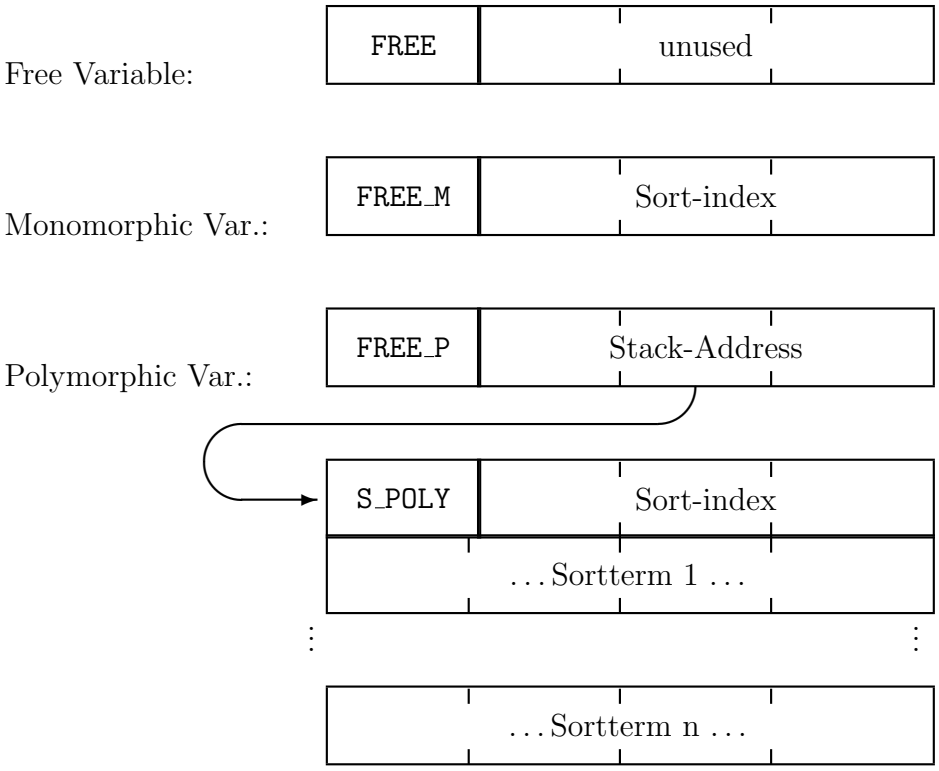
This representation of sort terms is also used in the PAM-code (see e.g. the instruction `get_x_poly` X_n`,` A_i`, sortterm` in Section 6.1). Additionally, this representation is used in the symbol table for the argument domains of constructors. In the case of a polymorphic constructor such an argument domain – for instance, pointed to by the result of a function call `constructor_arg(c,i)` – may still contain type variables. The compiler numbers the type variables occurring in a polymorphic type definition `s(T1,...,Tn)` from left to right. A variable in a sort term is then represented by the tag `S_VAR` followed by the variable number:



However, as already argued in Section 2, such sort variables do not occur in the run time stacks. They are replaced by ground sort terms when they are built up on the stack (cf. the PAM low-level procedure `propagate()` in Section 7).

5.2 Variables

As explained before, the variables of the WAM which are indicated by a REF-tag and a self-referencing pointer are replaced by variables with sort restrictions. For optimization a part of the sort information is coded into the tags FREE, FREE_M, FREE_P which denote no sort (i.e. TOP), a monomorphic sort, and a polymorphic sort restriction, respectively.



The value field of a FREE variable contains no information. In the monomorphic FREE_M case the value field holds the sort index of the variable, pointing to an entry in the sort table. Because polymorphic sort descriptions are bigger than a single value cell the sort-restriction of a polymorphic variable is given by the address of the complex sort term.

In [16] a WAM extension is proposed for the order-sorted case. Their treatment corresponds to our case where the only variable tag is FREE_M. Our approach not only admits additionally polymorphic sorts but is also an optimization because of the approximation realized by the tag FREE. If a program does not use any subsort relationship only FREE variables will occur at run time, thus disposing any sort processing, whereas in [16] the sort informations of an order-sorted program always have to be considered at run time.

6 Instructions for Unification

As argued above, the AND / OR structure of the WAM carries over to the PAM, but the term representation (and thus also e.g. the trailing of variables for backtracking) has to be changed. Since the essential difference in the term representation comes with the representation of the variables, all WAM instructions that

- create a new variable, or
- bind a variable

have to be modified. These instructions are the instructions responsible for unification, namely the

- get instructions to be used for the arguments in the clause head,
- put instructions to be used for the arguments in the body of a clause,
- unify instructions to be used for the nested arguments of a structure.

In the following subsections we will not present the PAM instructions grouped by these three classes because the differences from the WAM to the PAM can be better highlighted by separately looking at the instructions (1) for the *first* occurrence of a variable, (2) for the non-first occurrences of a variable, and (3) for constants and structures. As it will turn out only the instructions belonging to first group will get an additional sort parameter.

6.1 First occurrence of a variable in a clause

In each of the put, get, and unify instruction classes there is one WAM instruction that is generated by the compiler for the *first* temporary variable occurrence in that clause, namely:

(1) `get_x_variable` X_n, A_i
(2) `put_x_variable` X_n, A_i
(3) `unify_x_variable` X_n

In the PAM the type restrictions of variables have to be taken into account. The *statically* derived type of a variable that is available at compile time is therefore used when the variable occurs for the first time in a clause. Thus, depending on whether that type is the **maximal** type restriction `TOP`, a **monomorphic** type restriction `sort`, or a **polymorphic** type restriction `sortterm` the compiler for the PAM generates

```
(1.1)  get_x_free Xn, Ai
(1.2)  get_x_mono Xn, Ai, sort
(1.3)  get_x_poly Xn, Ai, sortterm
```

respectively, for the first occurrence of $\texttt{X}_n$ in an argument position of a clause head. Similarly, the `put_x_variable` $\texttt{X}_n$, $\texttt{A}_i$ and `unify_x_variable` $\texttt{X}_n$ instructions from the WAM are replaced in the PAM by

```
(2.1)  put_x_free Xn, Ai              (3.1)  unify_x_free Xn
(2.2)  put_x_mono Xn, Ai, sort        (3.2)  unify_x_mono Xn, sort
(2.3)  put_x_poly Xn, Ai, sortterm    (3.3)  unify_x_poly Xn, sortterm
```

respectively.

Here is a complete description of these nine PAM instructions where $\mathbf{X}_n$ is a temporary variable occurring the first time in a clause as the i-th argument and the register *nextarg* (simply called register S in [1]) points to the next argument subterm of a structure to be unified:

- **get_x_free ($\mathbf{X}_n$,$\mathbf{A}_i$)**

  Reset register $\mathbf{X}_n$ to the value given by $\texttt{A}_i$.

- **get_x_mono ($\mathbf{X}_n$,$\mathbf{A}_i$,sort)**

  Put a `FREE_M` value cell restricted to **sort** into $\mathbf{X}_n$. Unify register $\mathbf{X}_n$ with register $\mathbf{A}_i$. If unification doesn't succeed backtracking takes place.

- **get_x_poly ($\mathbf{X}_n$,$\mathbf{A}_i$,sortterm)**

  Put a `FREE_P` value cell restricted to **sortterm** into $\mathbf{X}_n$. Unify the register $\mathbf{X}_n$ with register $\mathbf{A}_i$. If unification doesn't succeed backtracking takes place.

- **put_x_free ($\mathbf{X}_n$,$\mathbf{A}_i$)**

  Create a new unrestricted `FREE` value cell on the global stack and reset $\mathbf{X}_n$ and $\mathbf{A}_i$ to reference this value cell.

- **put_x_mono ($\mathbf{X}_n$,$\mathbf{A}_i$,sort)**

  Create a new `FREE_M` value cell restricted to **sort** on the global stack and reset $\mathbf{X}_n$ and $\mathbf{A}_i$ to reference this value cell.

- **put_x_poly ($\mathbf{X}_n$,$\mathbf{A}_i$,sortterm)**

  Create a new `FREE_P` value cell restricted to **sortterm** on the global stack and reset $\mathbf{X}_n$ and $\mathbf{A}_i$ to reference this value cell.

- **unify_x_free ($\mathbf{X}_n$)**

  In *write mode* an unbound `FREE` value cell is inserted at *nextarg*. A reference to *nextarg* is put into $\mathbf{X}_n$ and *nextarg* is incremented.

  In *read mode* a reference to *nextarg* is put into $\mathbf{X}_n$ and *nextarg* is incremented.

- **unify_x_mono ($\mathbf{X}_n$,sort)**

  In *write mode* an unbound `FREE_M` value cell restricted to **sort** is inserted at *nextarg*. A reference to *nextarg* is put into $\mathbf{X}_n$ and *nextarg* is incremented.

  In *read mode* a new `FREE_M` value cell restricted to **sort** is created on the global stack. A reference to the new value cell is put into $\mathbf{X}_n$ and finally $\mathbf{X}_n$ and *nextarg* are unified and *nextarg* is incremented. If unification doesn't succeed backtracking takes place.

- **unify_x_poly ($\mathbf{X}_n$,sortterm)**

  In *write mode* an unbound `FREE_P` value cell restricted to **sortterm** is inserted at *nextarg*. A reference to *nextarg* is put into $\mathbf{X}_n$ and *nextarg* is incremented.

  In *read mode* a new `FREE_P` value cell restricted to **sortterm** is created on the global stack. A reference to the new value cell is put into $\mathbf{X}_n$ and finally $\mathbf{X}_n$ and *nextarg* are unified and *nextarg* is incremented. If unification doesn't succeed backtracking takes place.

Analogously to these instructions for the first occurrence of a temporary variable the WAM instructions for the first occurrence of a permanent variable are each split into three PAM instructions.

Note that get instructions for variables occurring in the head of a clause only need an additional sort argument if the statically derived type for the variable is different to (i.e. less than) the corresponding type in the declaration of the predicate. If these types are identical, the caller of the predicate already ensures the appropriate typing and thus a simple `get_x_free` instruction would be sufficient.

If any of the above instructions calls the low-level unification procedure, the calling context can be used to enter the unification routine at a point where redundant tests are not repeated. These obvious optimizations can be applied at several other places and will not be mentioned explicitly in the sequel.

6.2 Non-first occurrences of a variable in a clause

The instructions presented in the previous subsection are the only PAM instructions with an explicit sort argument in their parameters. For every non-first

occurrence of a variable the type restriction derived *statically* at compile time (which has already been taken into account at the first occurrence) is no longer relevant. Instead, the actual *dynamic* type restrictions of the variables which have been derived so far during run time have to be taken into account.

Apart from using the modified low-level functions, the PAM instructions for the non-first occurrence of a variable are as in the WAM with the following slight modification: When globalizing a variable with `put_unsafe_value` $\mathtt{Y}_n$`,` $\mathtt{A}_i$`,` `unify_x_local_value` $\mathtt{X}_n$, or `unify_y_local_value` $\mathtt{Y}_n$ the complete value cell must be copied in order to keep the sort restriction.

6.3 Constants and structures

For the constants and structures occurring in the head and the body of a clause the instructions `put_constant` *const*,$\mathtt{A}_i$ and `put_structure` *func*,$\mathtt{A}_i$ are as in the WAM. When binding a `FREE_M` variable to a term in the `get_constant` *const*,$\mathtt{A}_i$ and `unify_constant` *const* instructions the target sort of *const* must be a subsort of the variable's sort restriction. In the `FREE` and `FREE_P` cases no check is necessary. The only instruction with a major modification is:

- **get_x_structure (func,Ai)**

  If **Ai** references a structure labelled by the functor **func**, *nextarg* is reset to the first argument of the structure and the machine continues in *read mode*.

  Otherwise, if **Ai** references a variable we have three cases:

  1. **Ai** is unrestricted (`FREE`):
     A new structure with functor **func** is created on the global stack, and the machine continues in *write mode*.
  2. **Ai** is monomorphic (`FREE_M`):
     If the target sort of **func** is a subsort of **Ai**'s sort, then a new structure with functor **func** is created on the global stack and the machine continues in *write mode*, else backtracking is initiated.
  3. **Ai** is polymorphic (`FREE_P`):
     If the arguments of a skeleton structure with functor **func** can be restricted according to the declaration of **func** and **Ai**'s type restriction (this is called *propagation*, cf. the example below), then a new structure with functor **func** is created on the global stack with accordingly restricted new variables and the machine continues in *write mode*, else backtracking is initiated.

  Additionally, if any of the three steps above is executed without initiating backtracking, the value cell referenced by **Ai** is trailed and then reset

to reference the newly created structure, and *nextarg* is reset to the first argument of the structure.

Note that the rules (ES'), (ES), and (DS) (see Figure 1) are reflected in the `get_structure` instruction by the cases 1., 2., and 3., respectively. The difference to the corresponding WAM instruction is encountered when A_i references a free variable with a monomorphic or polymorphic type restriction.

As illustrated in the example given in Section 2 with respect to the decomposition rule (DS), the type restrictions for the arguments of a polymorphic type must be propagated to the arguments of the structure given in the clause head. For instance, if A_i's type restriction is `list(nat)` and *func* is the list constructor '.', then

`get_structure (., A`$_i$`)` (*)

creates a skeleton structure `X`$_1$`.X`$_2$ with the new variable `X`$_1$ restricted to `nat` and `X`$_2$ restricted to `list(nat)`. If A_i's type restriction is `list(BOTTOM)` (cf. 4.1) then (*) would cause backtracking since the type restriction `BOTTOM` can not be propagated onto a variable. The propagation is achieved by a call of the low-level PAM function `propagate` (see 7.4). The thus created n (= arity of the constructor) value cells with the correct type restriction will be unified in the following n unify instructions for the arguments of the structure when the machine continues in *read* mode.

In addition to the PAM get, put and unify instructions presented above there are also instructions which are the PAM equivalents of specialized WAM instructions for dealing with built-in lists and integers (e.g. `get_nil` or `put_list`).

7 Low-level instructions

There are a number of auxiliary WAM functions that are used in the definitions of the machine instructions responsible for unification. Here we will describe where the PAM low-level functions differ from the WAM, referring to the description in [1], and which additional operations are required for type handling. Thus, we will refer to the memory where the data is stored as `STORE`, and `STORE`[i] may contain a *tagged value cell*, written ⟨`t`, `v`⟩, such that `STORE[i].tag` = `t` and `STORE[i].value` = `v`.

```
procedure bind(a1, a2 : address)
    if STORE[a1].tag ∈ {FREE, FREE_M, FREE_P} and
      ((STORE[a2].tag ∉ {FREE, FREE_M, FREE_P}) or (a2 < a1))
      then bind_first_arg(a1, a2)
      else bind_first_arg(a2, a1)
```

Figure 2: The `bind` operation (part 1)

7.1 Dereferencing, unification, and trailing

The dereference function stops when the first non-`REF` tag is encountered since free variables in the PAM have a tag different from `REF`. The trailing operation not only saves the address `a` of a variable but also its contents `STORE[a]` containing its type restriction. Therefore, the WAM operations `trail`, `unwind_trail`, and `tidy_trail` must be modified accordingly.

The rules for polymorphic order-sorted unification as given in Figure 1 consist of the ordinary term unification rules plus additional type computation rules which are applied each time a variable is bound. Thus, the low-level unify operation is as described in [1]; only the binding procedure has to be modified. However, whereas the binding procedure in the WAM never causes backtracking, the PAM binding procedure (and also some additional auxiliary procedures called by it) may detect a unification failure and may thus initiate backtracking.

7.2 Binding

The complete PAM procedure `bind` (split into four subprocedures) is given in Figures 2 - 5. When `bind(a1,a2)` is called, at least one argument must be unbound. The condition in Figure 2 ensures that in the case where both arguments are unbound, the higher address is always bound to the lower one (for a discussion of this WAM binding discipline see [1]).

The binding of unrestricted `FREE` variables is the same as in the WAM (case `FREE` in Figure 3). When dealing with monomorphic variables (procedure `bind_FREE_M` in Figure 4), the subsort relationships must be taken into account. This is realized by using the interface functions to the sort and symbol tables, cf. Section 4. Note that due to the restriction that polymorphic types may not be subtypes of each other (see 2.1) and since only well-typed programs and queries

```
procedure bind_first_arg(a1, a2 : address)
    case STORE[a1].tag of
         FREE  :  trail(a1); STORE[a1] := ⟨REF, a2⟩;
         FREE_M:  bind_FREE_M(a1, a2);
         FREE_P:  bind_FREE_P(a1, a2);
    endcase
```

Figure 3: The `bind` operation (part 2)

```
procedure bind_FREE_M(a1, a2 : address)      % monomorphic variable
    v1 := STORE[a1].value;                    % tag at a1 is FREE_M
    v2 := STORE[a2].value;
    case STORE[a2].tag of
         FREE  :  trail(a2); STORE[a2] := STORE[a1];
                  trail(a1); STORE[a1] := ⟨REF, a2⟩;
         FREE_M:  glb := sort_glb(v1,v2);
                  if glb = BOTTOM
                    then fail_and_backtrack
                    else if v2 ≠ glb then
                           begin  trail(a2); STORE[a2].value := glb;
                           end
                  trail(a1); STORE[a1] := ⟨REF, a2⟩;
         FREE_P:                % not reached
         CONST :  s := target_sort(v2);
                  if not sort_is_subsort(s, v1)
                    then fail_and_backtrack;
                  trail(a1); STORE[a1] := STORE[a2];
         STRUC :  s := target_sort(STORE[v2]);
                  if not sort_is_subsort(s, v1)
                    then fail_and_backtrack;
                  trail(a1); STORE[a1] := STORE[a2];
    endcase
```

Figure 4: The `bind` operation (part 3)

are considered, the case of binding a FREE_M variable to a FREE_P variable (and vice versa) can not occur.

When binding a polymorphic variable (procedure bind_FREE_P in Figure 5), there are four cases similar to the monomorphic case, depending on the second argument a_2. When binding two variables with polymorphic type restrictions (case FREE_P in Figure 5) the function

```
function infimum(a1, a2 : address) : address
    % the tag at a1 must be a sort term tag (Sec. 5.1)
    % the tag at a2 must be a sort term tag
```

is used (realizing the *inf* function of Section 2; for its definition see Sec. 7.3). The returned address inf points to a type term representing the infimum of both given type terms. Thus, either STORE[inf].tag = S_POLY or STORE[inf].tag = S_BOTTOM, the latter indicating unification failure.

In the case of binding an address a_1 with polymorphic type restriction to a non-constant structured term in a_2 (case STRUC in Figure 5) the procedure

```
procedure propagate(a1, a2 : address)
        % the tag at a1 must be FREE_P
        % the tag at a2 must be STRUC
```

propagates the type restrictions enforced by the type restriction of the variable (given by address a_1) to the arguments of the given term. As in the get_structure instruction, the decomposition rule (DS) of Figure 1 is reflected here. Note that - as opposed to the monomorphic case - there is no need for a subsort test between the target sort of the constructor of the term and the (top-level) sort symbol of the variable, because all polymorphic sort symbols are maximal (cf. the discussion in Section 2). How the propagation is achieved is described in Sec. 7.4.

7.3 Polymorphic infimum

Computation of the infimum of two sort terms is achieved by the function infimum depicted in Figure 6. At the two addresses a_1 and a_2 given as input there must be sort terms. If either of them is TOP or BOTTOM the result can be obtained immediately. Otherwise, a switch on the sort tag is made. In all three arising cases (S_MONO, S_REF, and S_POLY) the computation follows the same structure:

1. Compute the infimum inf at the next lower level (by looking at the value parts v_1 and v_2 of the two given arguments a_1 and a_2).

```
procedure bind_FREE_P(a1, a2 : address)      % polymorphic variable
    v1 := STORE[a1].value;                    % tag at a1 is FREE_P
    v2 := STORE[a2].value;
    case STORE[a2].tag of
         FREE  :  trail(a2); STORE[a2] := STORE[a1];
                  trail(a1); STORE[a1] := ⟨REF, a2⟩;
         FREE_M:                % not reached
         FREE_P:  inf := infimum(v1,v2);
                  if STORE[inf].tag = S_BOTTOM
                    then fail_and_backtrack
                    else if v2 ≠ inf then
                           begin  trail(a2); STORE[a2].value := inf;
                           end
                  trail(a1); STORE[a1] := ⟨REF, a2⟩;
         CONST :                % no subsort test necessary
                  trail(a1); STORE[a1] := STORE[a2];
         STRUC :  propagate(a1, a2);
                      % propagate may call fail_and_backtrack
                  trail(a1); STORE[a1] := STORE[a2];
    endcase
```

Figure 5: The `bind` operation (part 4)

```
function infimum(a1, a2 : address) : address % called in bind_FREE_P
    % the tags at a1 and a2 must be sort term tags (Sec. 5.1)
if STORE[a1].tag = S_BOTTOM or STORE[a2].tag = S_TOP then return a1;
if STORE[a2].tag = S_BOTTOM or STORE[a1].tag = S_TOP then return a2;
v1 := STORE[a1].value;
v2 := STORE[a2].value;                                              } 1.
case STORE[a1].tag of
     S_MONO : inf := sort_glb(v1,v2);                               } 2.
              if inf = v1 then return a1;                           } 3.
              if inf = v2 then return a2;
              if inf = BOTTOM then return Bottom_Ref;               } 4.
              STORE[H] := ⟨S_MONO, inf⟩;
              H:= H + 1;                                                  } 1.
              return H - 1;
     S_REF  : inf := infimum(v1,v2);                                      } 2.
              if inf = v1 then return a1;                                 } 3.
              if inf = v2 then return a2;
              if STORE[inf].tag = S_BOTTOM then return inf;               } 4.
              STORE[H] := ⟨S_REF, inf⟩;
              H:= H + 1;
              return H - 1;                                                       } 1.
     S_POLY : arity := sort_arity(v1);
              declare inf[1,...,arity];
              for i = 1,...,arity do
                      inf[i] := infimum(a1+i,a2+i);                               } 2.
              if forall i ∈ {1,...,arity} . inf[i] = a1+i
                then return a1;
              if forall i ∈ {1,...,arity} . inf[i] = a2+i
                then return a2;
              declare b[1,...,arity] array of bool;                               } 3.
              for i = 1,...,arity do
                      b[i] := STORE[inf[i]].tag ≠ S_BOTTOM;
              if inhabitation_mode(v1,(b[1],...,b[arity]) = false
                then return Bottom_Ref;
              STORE[H] := STORE[a1];
              for i = 1,...,arity do                                              } 4.
                      STORE[H+i] := STORE[inf[i]];
              H:= H + arity + 1;
              return H - arity - 1;
endcase
```

Figure 6: The infimum operation for sort terms

2. Optimization step: If the infiumum `inf` is already given by $\mathtt{a}_1$, then return $\mathtt{a}_1$; if it is given by $\mathtt{a}_2$, then return $\mathtt{a}_2$. Thus, no additional heap space is needed.

3. If `inf` is `BOTTOM`, then return a (constant) reference `Bottom_Ref` to a `S_BOTTOM`-tagged value cell. (Note that this is also an optimization; one could just as well create a new sort term cell on the heap.)

4. Otherwise, create a new sort term with value `inf` on the heap. (The global variable `H` used in Figure 6 when creating a new sort term on the heap always points to the top of the heap; thus it is exactly the same as the variable `H` used in [1].)

For instance, in the case of a polymorphic sort term (case `S_POLY`) the infimum of all arguments of the polymorphic sort constructor must be determined (step 1). The two following if-conditions realize the optimization step 2: if the result is already given in $\mathtt{a}_1$ (resp. $\mathtt{a}_2$), then $\mathtt{a}_1$ (resp. $\mathtt{a}_2$) can be returned as result. If the resulting sort term is not inhabited (cf. Section 4.1), the result is `BOTTOM` (step 3). Otherwise, the new sort term is created on the heap and its address is returned as result (step 4).

7.4 Polymorphic propagation

Before discussing the `propagate` procedure in detail, let us first give an example. Consider the type definition

```
bin_tree(T) := { leaf:  T,
                 left:  bin_tree(T) x T,
                 right:  T x bin_tree(T),
                 both:  bin_tree(T) x T x bin_tree(T) }.
```

of binary trees from Section 2.1. Suppose that `B` is a variable of type `bin_tree(nat)`, and `E`, `T` are free variables with no (i.e. `TOP`) type restriction, and we want to bind `B` to the term `left(right(2,T),E)`. That is, in the presence of the type constraints

```
B : bin_tree(nat)  &  E : TOP  &  T : TOP
```

we want to perform the binding

```
B ≐ left(right(2,T),E)                                    (*)
```

The type restriction of `B` must be propagated to the data term `left(right(2,T),E)`. Now in general, the arguments of the data term (in the example `right(2,T)` and `E`) must be restricted to the respective argument domains of the top-level functor (here: the `bin_tree` constructor `left`) where each type variable in an argument domain in its declaration (here: `left : bin_tree(T) x T --> bin_tree(T)`) is replaced by the respective argument of the type term given by `B` (here: replacing `T` by `nat`, which yields `left : bin_tree(nat) x nat --> bin_tree(nat)`).

In an implementation, this can be achieved in two steps: First, a new term $\texttt{f(X}_1\texttt{,...,X}_m\texttt{)}$ (in the example: $\texttt{left(X}_1\texttt{,X}_2\texttt{)}$) is created with appropriately type-restricted new variables $\texttt{X}_i$ (here: $\texttt{X}_1\texttt{:bin\_tree(nat)}$ and $\texttt{X}_2\texttt{:nat}$), and second, this new term is unified with the term given by $\texttt{a}_2$. Thus, in the example the type constraint

```
left(right(2,T),E) : bin_tree(nat)
```

that has to be solved when performing the binding (*), is reduced to the unification problem

$$\texttt{left(X}_1\texttt{,X}_2\texttt{)} \doteq \texttt{left(right(2,T),E)}$$

with type-constrained new variables $\texttt{X}_1\texttt{:bin\_tree(nat)}$ and $\texttt{X}_2\texttt{:nat}$. (In fact, this is a slight simplification of the representation over the actual PAM implementation where the top-level functor (here: `left`) would not be generated since it is not needed; instead, the binding of the n argument variables of the new term can be called directly.)

When the procedure $\texttt{propagate(a}_1\texttt{,a}_2\texttt{)}$ (Figure 7) is called, the first argument is a polymorphically restricted variable (thus with tag `FREE_P`), while the second argument is a compound term (thus the tag at $\texttt{a}_2$ must be `STRUC`). In the example just given, $\texttt{propagate(a}_1\texttt{,a}_2\texttt{)}$ would be called with $\texttt{a}_2$ pointing to the (representation of the) term `left(right(2,T),E)`, and with $\texttt{a}_1$ pointing to the (representation of the) variable `B:bin_tree(nat)`.

The procedure $\texttt{propagate(a}_1\texttt{,a}_2\texttt{)}$ starts the generation of the new term by writing the top level functor on the heap and temporarily saving its address in `skeleton`. Then for every argument position `i` of the top level constructor, a new variable $\texttt{X}_i$ is written on the heap at the address *h(i)* = `skeleton + 1 + i`. The type restriction of $\texttt{X}_i$ is defined by the `i`-th argument domain in the declaration of the constructor (which is given by `constructor_arg(c,i)`) where type variables are instantiated by the corresponding type term arguments given by $\texttt{v}_1$. In order to create each variable $\texttt{X}_i$ in the skeleton with its correct type, $\texttt{X}_i$'s type restriction is put into the heap at address *h(i)* by the procedure

```
procedure propagate(a1, a2 : address);      % called in bind_FREE_P
      % the tag at a1 must be FREE_P
      % the tag at a2 must be STRUC
  v1 := STORE[a1].value;
  v2 := STORE[a2].value;
  c  := STORE[v2].value;                                % constructor
  skeleton := H;                        % address of new skeleton term

  STORE[H] := ⟨S_REF, H+1⟩;             % STEP 1: create skeleton term
  STORE[H+1] := c;
  H:= H + 2 + constructor_arity(c);
  let h(i) = skeleton + 1 + i;             % position of i-th argument
  for i = 1,...,constructor_arity(c) do
    put_sortterm(constructor_arg(c,i), v1, h(i));
    case STORE[h(i)].tag of
      S_TOP    : STORE[h(i)].tag := FREE;
      S_MONO   : STORE[h(i)].tag := FREE_M;
      S_REF    : STORE[h(i)].tag := FREE_P;
      S_BOTTOM : fail_and_backtrack;
    endcase;

  unify_skeleton(a2, skeleton);        % STEP 2: unify skeleton term
```

Figure 7: The propagate operation

```
procedure put_sortterm(d : sortterm_index, v,h : address)
    % d points to a sort term in the symbol table (Sec. 4.2)
    % the tag at v is S_POLY
    % h is the heap address where the sortterm is put
```

`put_sortterm(d,v,h)` 'copies' the sort term referenced by `d` from the symbol table to the heap and returns its heap address. However, instead of just copying, two additional modifications are carried out:

1. Type term instantiation:

   Each type variable ⟨`S_VAR, k`⟩ that is read from the symbol table representation, is replaced by `STORE[v+k]`, the `k`-th argument of the type term given by `v`. This simple replacement realizes the type term instantiation as it occurs in the decomposition rule (DS) in Figure 1.

   For instance, for the first argument of the `left` constructor in our `bin_tree` example above, `d` points to (the representation of) `bin_tree(`α_1`)` in the symbol table, `v` points to `bin_tree(nat)` whose first argument is `nat`, `h` is *h(1)*, and `put_sortterm(d,v,h)` writes `bin_tree(nat)` at the address *h(1)* on the heap. Thus, `STORE`[*h(1)*] will then contain ⟨`S_REF, p`⟩ with `STORE[p]` = ⟨`S_POLY, bin_tree`⟩ and `STORE[p+1]` = ⟨`S_MONO, nat`⟩.

   Similarly, for the second argument of the `left` constructor, `d` points to (the representation of) α_1 in the symbol table, `v` still points to `bin_tree(nat)`, `h` is *h(2)*, and `put_sortterm(d,v,h)` would thus write `nat`, yielding `STORE`[*h(2)*] = ⟨`S_MONO, nat`⟩.

2. Approximation of type terms:

   Moreover, the procedure `put_sortterm` computes the approximation (see Section 2.2) of the thus instantiated type term. Therefore, `put_sortterm(d,v,h)` may also write an `S_TOP` or `S_BOTTOM` tagged sort term. For instance, instantiating the type variable α_1 in `bin_tree(`α_1`)` by `TOP` yields `bin_tree(TOP)` whose approximation is `TOP`, and instantiating α_1 by `BOTTOM` yields `bin_tree(BOTTOM)` whose approximation is `BOTTOM` since `bin_tree(BOTTOM)` is not inhabited (see 2.2).

After `put_sortterm(...)` has written the type term for $\mathtt{X}_i$'s type restriction at the address *h(i)*, the procedure `propagate` converts the top level tag at *h(i)* to the corresponding tag for a variable. Note that indeed only the top-level type term tag at *h(i)* is changed; any deeper nested type terms remain unchanged: the sort term tag `S_TOP` is replaced by the tag `FREE` for an unrestricted free variable, `S_MONO` is replaced by `FREE_M`, and `S_REF` by `FREE_P`. If *h(i)*'s tag is `S_BOTTOM`, backtracking is initiated immediately since $\mathtt{X}_i$`:BOTTOM` is an inconsistent type constraint. This case occurs e.g. when trying to bind `X:list(BOTTOM)` (which itself

is *not* inconsistent - see the discussion in 4.1) to the term `Y.L`; then backtracking occurs since the type restriction of the first argument variable of the skeleton term would have to be `BOTTOM`.

When all argument variables have been written on the heap, in the second step of the `propagate` procedure `unify_skeleton(`a_2`, skeleton)` unifies a_2 with the newly generated term. The easiest way to express what happens in the procedure `unify_skeleton(...)` is to view it as a recursive call to the unify procedure. (In the actual implementation, the PAM exploits the fact that only a special part of the general unification is needed here. Indeed, for each argument, the respective binding procedure can be called directly since the second argument is guaranteed be an unbound variable.) Note that therefore also `unify_skeleton(...)` may cause a unification failure and initiate backtracking.

7.5 Comparison with an alternative abstract machine

After having presented the PAM procedures for type computations, let us now comment on the approach of [17] to an abstract machine implementation of a polymorphic order-sorted logic programming. The original goal of [17] is aimed towards an implementation of λProlog [22] which includes higher-order unification, but they also describe an extension of their machinery to cope with the type system of PROTOS-L and compare it to the PAM.

The basic machinery of [17] already gives a major difference in the representation of terms. Instead of just one value cell as used in the WAM or the PAM, each symbol in a term uses three value cells in [17] where the two additional cells hold pointers to a table of type skeletons and type environments, respectively, allowing for structure sharing of type terms. Thus, for each term and every subterm a type information is kept, instead of just for the variables as in the PAM. Moreover, when moving to the order-sorted case, all subterms of a structure corresponding to an argument with a polymorphic type variable get an additional level of indirection by the use of specially tagged variables. Therefore, the representation of `right(X,Y)` with `X:posint` and `Y:bin_tree(nat)` requires 20 heap cells in [17], as opposed to just 5 heap cells in the PAM. Moreover, if `int` is maximal and `Y:bin_tree(int)`, the optimized representation in the PAM requires only 3 heap cells, just as in the original WAM. No similar optimization in given in [17].

In the set of machine instructions in [17] there are type term unification instructions which are separated from the usual WAM unification instructions for (untyped) term unification. No difference is made between monomorphic and polymorphic type unification on the level of instructions. On the other hand, the PAM has specialized instructions that combine particular cases of typed unifi-

cation (e.g. `unify_x_mono` or `unify_x_poly` with ordinary term unification. For instance, [17] gives the code for compiling a clause of the form

```
p(f(X,Y)) <-- Y:ll(nat).
```

with `p:int`, `f: ll(T) x ll(T) --> ll(T)`, `X:ll(int)`, and `Y:ll(nat)`, which consists of 16 machine instructions. The corresponding PAM code just consists of the four instructions

```
get_structure f, A1
unify_x_poly X2, ll(int)
unify_x_poly X3, ll(nat)
proceed
```

where, assuming again that `int` is maximal, the second instruction would be replaced by

```
unify_x_free X2
```

The internal complexity of each of these PAM instructions does not seem to be greater than the complexity of the instructions in [17]. These specialized instructions are possible in the PAM since the type system of PROTOS-L is more specialized than the type system of [17]. A major difference between the approach of [17] and the PAM is the occurrence of type variables at run time. Since [17] aims at an implementation on λProlog, type variables are essential also at run time. On the other hand, as described in Section 2, the operational semantics of PROTOS-L does not deal with type variables at run time and therefore, type variables do not occur in the stacks of the PAM. This excludes the treatment of general higher-order unification as needed for λProlog, but enables various of the PAM optimizations described in this paper.

8 Conclusions and further work

Starting from an abstract specification of polymorphic order-sorted resolution, we have shown how the WAM can be extended accordingly, mainly by

- managing a sort table and extending the symbol table,
- adding types to data structures in the stacks,
- adding type parameters to the instructions handling the first occurrences of variables, and
- adding type computations to the low-level *bind* procedure.

This extension is orthogonal in the sense that any program part not exploiting the facilities of computing with subtypes is executed with almost the same efficiency as on the original WAM since none of the new PAM instructions is used; the only low-level overhead is in the PAM's extended tagging scheme and the value trailing. On the other hand, any typed program exploiting e.g. the possibilities of computing with subtypes can take advantage of the type constraint handling facilities in the PAM which would have to be simulated by additional explicit program clauses in an untyped version. One of the main principles of the WAM - to compile parts of the complex unification procedure into specialized instructions - has been applied to the type computation in the PAM where the instructions are further split according to a variable's (static) type restriction.

Among the features of PROTOS-L and the PAM not described here are various additional extensions and built-ins which are needed in a practical programming language [5]. The switch instructions are extended in order to use the type information of variables, providing further means to use types in guiding the inference process by avoidance of backtracking and support of determinacy detection. There are several built-ins related to sorts worth to be mentioned. In Section 2, we stated that a solution consists of bindings to variables together with sort restrictions for unbound variables. Using the information in the sort and symbol tables, there are built-ins for enumerating all possible instantiations of variables restricted to some sort. Furthermore, there are some meta programming like facilities to check and compare the types of variables without restricting them. We have also included built-in arrays which nicely fit into the polymorphic type concept of the language. The PROTOS-L system with the PAM is implemented on IBM RS/6000 workstations, IBM RT/PC 6150, and IBM PS/2 under the AIX operating system.

There are various possible extensions of the PAM, e.g. the integration of higher-order programming (cf. the discussion in the previous section), the dropping of the restriction that polymorphic types may not have explicitly defined subtypes, or the definition of types by a set of attributes in an object-oriented way. Another extension that has recently been carried out in the PROTOS project is the orthogonal integration of a finite domain constraint solver into the PAM.

References

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

[2] H. Aït-Kaci and R. Nasr. LOGIN: A logic programming language with built-in inheritance. *The Journal of Logic Programming*, 3:185–215, 1986.

[3] C. Beierle. Logic programming with typed unification and its realization on an abstract machine. *IBM Journal of Research and Development*, 36(3):375–390, May 1992.

[4] C. Beierle and E. Börger. Correctness proof for the WAM with types. In E. Börger, H. Kleine Büning, G. Jäger, and M. M. Richter, editors, *Computer Science Logic - CSL'91*, volume 626 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1992.

[5] C. Beierle, S. Böttcher, and G. Meyer. Draft report of the logic programming language PROTOS-L. IWBS Report 175, IBM Germany, Scientific Center, Inst. for Knowledge Based Systems, Stuttgart, 1991. Revised version: Working Paper 4, IBM Germany, Scientific Center, Inst. for Logics and Linguistics, Heidelberg, July 1994.

[6] C. Beierle, U. Hedtstück, U. Pletat, P. H. Schmitt, and J. Siekmann. An order-sorted logic for knowledge representation systems. *Artificial Intelligence*, 55(2–3):149–191, 1992.

[7] C. Beierle, G. Meyer, and H. Semle. Extending the Warren Abstract Machine to polymorphic order-sorted resolution. In V. Saraswat and K. Ueda, editors, *Logic Programming: Proceedings of the 1991 International Symposium*, pages 272–286, Cambridge, MA, 1991. MIT Press.

[8] E. Börger and D. Rosenzweig. The WAM - definition and compiler correctness. TR-14/92, Dipartimento di Informatica, Università di Pisa, 1992. (To appear in: C. Beierle, L. Plümer (Eds.), *Logic Programming: Formal Methods and Practical Applications*. Studies in Computer Science and Artificial Intelligence, North-Holland, 1994).

[9] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[10] R. Dietrich and F. Hagl. A polymorphic type system with subtypes for Prolog. In *Proceedings of the 2nd European Symposium on Programming*, Lecture Notes in Computer Science, pages 79–93, Berlin, 1988. Springer-Verlag.

[11] J. A. Goguen and J. Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pages 295–363. Prentice Hall, 1986.

[12] Y. Gurevich. Evolving algebras. A tutorial introduction. *EATCS Bulletin*, 43, February 1991.

[13] M. Hanus. Polymorphic higher-order programming in Prolog. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 382–397, Cambridge, MA, 1989. MIT Press.

[14] M. Hanus. Horn clause programs with polymorphic types: Semantics and resolution. *Theoretical Computer Science*, 89:63–106, 1991.

[15] P. M. Hill and J. W. Lloyd. The Gödel Report. TR-91-02, Dept. of Computer Science, University of Bristol, Bristol, UK, Revised Version, June 1992.

[16] M. Huber and I. Varsek. Extended Prolog for order-sorted resolution. In *Proceedings of the 4th IEEE Symposium on Logic Programming*, pages 34–45, San Francisco, 1987.

[17] K. Kwon, G. Nadathur, and D. S. Wilson. Implementing logic programming languages with polymorphic typing. Technical Report CS-1991-39, Duke University, October 1991.

[18] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[19] D. Miller and G. Nadathur. Higher-order logic programming. In E. Shapiro, editor, *Third International Conference on Logic Programming*, pages 448–462, Berlin, 1986. Springer-Verlag.

[20] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[21] A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[22] G. Nadathur and D. Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 810–827, Cambridge, MA, 1988. MIT Press.

[23] W. Nutt and G. Smolka. Implementing TEL. SEKI-Report, FB Informatik, Universität Kaiserslautern, 1993. (in preparation).

[24] U. S. Reddy. Notions of polymorphism for predicate logic programs. In K. Bowen and R. Kowalski, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Cambridge, MA, 1988. MIT Press.

[25] H. Semle. Extension of an abstract machine for order-sorted prolog to polymorphism. Diplomarbeit Nr. 583, Universität Stuttgart and IBM Deutschland GmbH, Stuttgart, April 1989. (in German).

[26] G. Smolka. TEL (Version 0.9), Report and User Manual. SEKI-Report SR 87-17, FB Informatik, Universität Kaiserslautern, 1988.

[27] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, FB Informatik, Univ. Kaiserslautern, 1989.

[28] C. Walther. Many-sorted unification. *Journal of the ACM*, 35(1):1–17, January 1988.

[29] D. H. D. Warren. An Abstract PROLOG Instruction Set. Technical Report 309, SRI, 1983.