Logic programming with typed unification and its realization on an abstract machine

by C. Beierle

Logic programming can benefit from a typing concept which supports many software engineering principles such as data abstraction, modularization, etc. From a computational point of view, the use of types can drastically reduce the search space. Starting from these observations, this paper gives a survey of many-sorted, order-sorted, and polymorphic approaches to type concepts in logic programming. The underlying unification procedures for ordinary term unification, order-sorted unification, and in particular for polymorphic order-sorted unification are given in the style of solving a set of equations, giving a common basis for comparing them. In addition, the realization of these unification procedures on a Warren Abstract Machine-like architecture is described. Special emphasis is placed on the abstract machine developed for PROTOS-L, a logic programming language based on polymorphic order-sorted unification.

1. Introduction

Logic programming has proven a powerful programming paradigm, representing the most successful tool that falls in the category of "declarative programming" and is also operationally feasible. In establishing a structural relationship between declarative and operational semantics, the central notion of unification is most important. As the heart of the so-called SLD resolution principle, it provides the basis for a simple albeit fully operational semantics that is compatible with its declarative counterpart [1]. In addition, with the pioneering work of D. H. Warren [2, 3], an efficient

implementation of unification and logic programming became available.

However, in comparing Prolog, the most prominent representative of the logic programming family, with other programming languages, one observes that certain major software engineering principles such as modularization and data abstraction are not directly supported in Prolog. Especially as applications grow larger and more complex, such software engineering principles become more important. The basis for many of these principles can be given by a powerful type concept. It allows many common programming errors to be detected at compile time which might otherwise be difficult to locate. Additionally, in AI applications such as theorem proving, it has been shown that the introduction of types with subtypes may drastically reduce the search space of a problem (e.g., [4]). On the other hand, types should not burden the programmer or knowledge engineer by requiring too narrow or too strict a discipline.

Starting from these observations, during the last few years there have been various attempts and suggestions to extend the logic programming paradigm by, among other things, the introduction of types; see, e.g., [5–13]. The purpose of this paper, which extends and revises the work presented in [14], is to investigate the extension of logic programming by types under three aspects, namely, to compare different classes of typing concepts, to present the unification procedures underlying them, and to show how the typed unification procedures can be realized on a suitable extension of the Warren Abstract Machine (WAM).

In Section 2, an overview of many-sorted, order-sorted, and polymorphic type concepts is given. Each of the various approaches can be classified as to whether it suffices to do static type checking, or whether types are also present at run time. It is also interesting to note that in the cases where types must be considered at run time, the type information can be dealt with completely within the unification.

In Section 3, the underlying unification procedures are investigated and compared to one another. As a common base allowing for easy comparison, we present each of them in the form of solving a set of equations (cf. [15]). Special emphasis is put on the case of an order-sorted type concept with polymorphism (cf. [16]), which has been incorporated into the logic programming language PROTOS-L [14, 17] and for which an abstract machine implementation has been developed.

The realization of the underlying unification procedures is the subject of Section 4, where in particular the PROTOS Abstract Machine (PAM) developed for PROTOS-L is described as an extension of the original WAM. Section 5 contains some conclusions and points out some further work.

2. Type concepts in logic programming

• *A many-sorted approach*

In unsorted predicate logic, unary predicates define a subset of the universe for any interpretation. For instance, a Prolog program containing the clauses

```
car(opel).
car(ford).
car(mercedes).
```

where car does not occur in the head of any other clause will have the three constants opel, ford, and mercedes in the set assigned to car in its standard interpretation (i.e., its minimal Herbrand model).

Within Prolog it is not possible to reason on this more abstract level of "cars"; rather, the level of the individuals of this set must be used. However, within the framework of algebraic abstract data type specifications [18, 19], the declaration

```
sort        car.
operations  opel:  → car.
            ford:  → car.
            mercedes:  → car.
```

within a specification yields essentially the same effect. Provided there are no other operations with target sort car and no equations are imposed on the constants, the set assigned to car in the standard interpretation of the specification (i.e., its initial model) contains the three constants opel, ford, mercedes.

A first extension of Prolog to sorts could thus be to allow sort declarations together with the operations that yield objects of the respective sorts. Following terminology used in abstract data type specifications, we call these operations *constructors* because they construct the elements of the given sort. Thus, the fragment of a Prolog program given above would correspond to

```
sort          car.
constructors  opel, ford, mercedes.
```

in a sorted version. In the following we use the more compact notation

```
sort  car := { opel, ford, mercedes }.
```

Similarly,

```
sort  boat := { ferry, steamer, sailing_boat }.
```

introduces the sort boat with its elements ferry, steamer, sailing_boat.

Given such sort declarations we can attach to every predicate p a declaration stating how many arguments p takes and of which sorts the arguments of p must be. For instance, the declaration

```
predicate  go_from_to_with:  city × city × vehicle.
```

would require the first two arguments of go_from_to_with to be of sort city and the third argument to be of sort vehicle.

Requiring such declarations for all predicates in a logic program and insisting on well-typed terms and literals already has some important consequences that lead us away from the usual Prolog situation:

- The distinction between declarations and clauses defining a predicate, together with a notion of well-typedness, requires a *type checker*.
- Since type checking takes place with respect to a set of declarations, the unit for type checking cannot be a single clause or declaration, but should be a program part containing both a set of declarations and a set of clauses (e.g., a module).
- Since only well-typed programs should be executed, there is a distinction between the type checking phase and the execution phase of a program. In analogy to classical typed programming languages, we refer to these phases as *compile time* and *run time*, respectively. If all type checking can be done at compile time we have *static* type checking, whereas type checking done at run time is called *dynamic*.

In Prolog there is essentially no type checking, and every clause added to a program file can be given to the interpreter immediately. However, the third distinction listed above can be exploited in terms of efficiency. In the situation discussed so far, type checking is completely a compile-time activity, and no type information whatsoever need be present at run time. Crucial to this observation is the fact that up to now we have only considered sorts with no possibility of having subsort relations between them. Again using abstract data type terminology, we refer to this case as the *many-sorted* case. The possible gains in run-time efficiency are based upon the fact that on the one hand no overhead due to type information occurs at run time, and on the other hand, because of well-typedness, any argument of a predicate is guaranteed to be of the specified sort. Therefore, the underlying unification need not deal with the whole universe of terms, but only with the terms which are of the given argument sort. An example for a logic programming language based on the many-sorted approach is Turbo-Prolog.

Generally one can say that a many-sorted approach does require only a static type-checking phase—which can be realized by a preprocessor—and that every implementation for the unsorted case also works well for the many-sorted case. In particular, an abstract machine such as the Warren Abstract Machine (WAM) [2, 3] does not have to be modified for the many-sorted case; however, it could be modified for efficiency reasons, as has been indicated above.

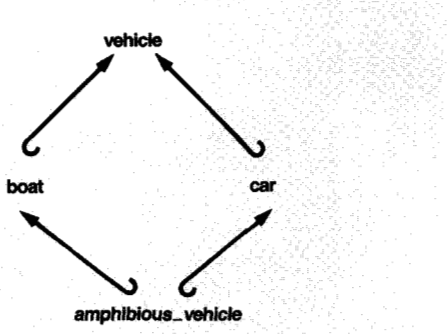

Figure 1

A simple sort hierarchy with subsorts.

- *An order-sorted approach*

An extension to the many-sorted approach is to allow subsort relationships between sorts. For instance, the sorts car and boat could both be seen as subsorts of a common supersort vehicle. Likewise, there could also be a sort amphibious_vehicle as a common subsort to both car and boat, yielding the sort hierarchy shown in **Figure 1**.

Order-sorted approaches allowing such sort hierarchies have been suggested in areas such as abstract data type specifications, automatic theorem proving, and logic programming. An order-sorted logic was given in 1962 by Oberschelp [20]. Order-sorted algebra originated with Goguen [21] and was developed further by Goguen et al., e.g., [22, 23]; it is the basis for the specification language OBJ [24] and the logic programming language Eqlog [7]. Gogolla [25] studied and extended order-sorted algebra, in particular with respect to error-handling approaches. Walther [26, 27] has investigated order-sorted unification and has given correctness and completeness results for resolution and paramodulation with order-sorted unification, and Huber and Varsek [10] have used this approach for an extended Prolog with order-sorted resolution. Schmidt-Schaub [28] has developed an order-sorted deduction calculus with polymorphic function definitions, and several other order-sorted approaches have been described in the literature.

Extending the approach described in the preceding subsection, we add to every sort declaration an enumeration of its (direct) subsorts separated by ++, if there are any. For instance, the sort hierarchy shown in **Figure 2** could be defined as follows:

```
sort  vehicle    := airplane ++ boat ++ car.
sort  airplane   := { bo747, dc10, airbus }.
sort  car        := amphibious_vehicle
                    ++ { opel, ford, mercedes }.
```

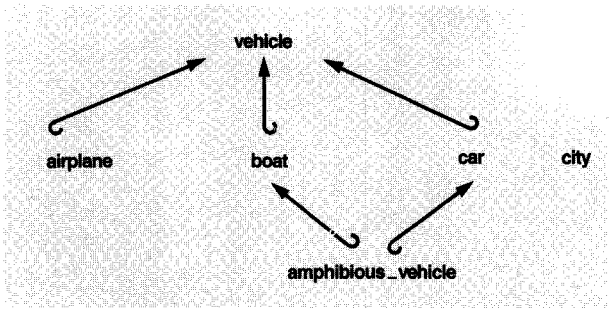

Figure 2

A sort hierarchy with subsort relationships and maximal sort vehicle and city.

```
sort  boat          := amphibious_vehicle
                       ++ { ferry, steamer, sailing_boat }.

sort  amphibious_vehicle   := { amphi1, amphi2, amphi3 }.

sort  city          := { stuttgart, frankfurt,
                         london, calais, dover }.
```

The following are two predicate definitions using these sort definitions:

```
predicate  go_direct:  city × city × vehicle.
    go_direct(stuttgart, frankfurt, X) :– X:airplane.
    go_direct(frankfurt, london, X) :– X:airplane.
    go_direct(stuttgart, calais, X) :– X:car.
    go_direct(dover, london, X) :– X:car.
    go_direct(calais, dover, X) :– X:boat.

predicate  go_from_to_with:  city × city × vehicle.
    go_from_to_with(From, To, With) :–
        go_direct(From, To, With).
    go_from_to_with(From, To, With) :–
        go_direct(From, Over, With) &
        go_from_to_with(Over, To, With).
```

go_direct(a, b, v) means that there is a direct way of getting from city a to city b using only vehicle v. The predicate go_from_to_with takes the same arguments as go_direct. The intended meaning of go_from_to_with(a, b, v) is that there is a way of going from city a to city b using only vehicle v, but possibly going via some other cities. Correspondingly, there are two clauses defining the predicate go_from_to_with as given above.

Now consider the query

```
?– go_from_to_with(stuttgart, london, V)        (1)
```

asking for a vehicle V which can be used to go from stuttgart to london.

We assume the usual Prolog evaluation strategy (i.e., left-to-right and depth-first search), but instead of ordinary term unification we now use order-sorted unification [14].

The original sort restriction of the variable V in query (1) is vehicle due to the declaration of go_from_to_with. The first solution to (1) leaves the variable V uninstantiated but sharpens its sort restriction to airplane:

V = X:airplane.

Obviously, this solution represents a more abstract answer than enumerating all elements of the sort airplane as would have to be done in the unsorted case. Enforcing backtracking first resets the sort of V to vehicle. Resolving with the third clause of go_direct restricts the sort of V to car. Further resolving the remaining subgoal with the fifth clause of go_direct leads to the unification of the variable V with sort restriction car and a variable X_2 with sort restriction boat. Order-sorted unification yields a variable whose sort restriction is the greatest common subsort of the two sorts in the given sort hierarchy, i.e., amphibious_vehicle in this case. After one more resolution step, the second solution

V = X:amphibious_vehicle

to query (1) is given.

By comparing the order-sorted case to the unsorted case, where each sort would correspond to a unary predicate and each sort restriction to a subgoal calling this predicate, we can observe that the sorted version provides a higher level of abstraction, thus providing more compact solutions and avoiding possibly expensive backtracking. As far as the last point is concerned, the left-to-right, depth-first search strategy may be even more complete in the order-sorted case if there are infinitely many backtracking points in the unsorted version which are covered by a subsort in the order-sorted version.

Whereas the observations made for the many-sorted case above about type checking at compile time, type-checkable units, etc., essentially still hold in the order-sorted case, this is not true with respect to the absence of type information at run time.

However, when looking at the degenerate order-sorted case without any subsort relationship at all, one obviously can neglect the sort information completely at run time, since this is just the many-sorted case presented above. More generally, all sort information concerning maximal sorts (vehicle or city in the example above) can be neglected at run time (see Section 3).

- *Polymorphic type concepts*

A serious drawback of both a many-sorted approach and an order-sorted approach as discussed so far is the fact that it is not possible to achieve the effect of parameterizing a structured sort over some other sort. For instance, in the many-sorted setting lists must be defined explictly for every sort. To give an example, the sort list_nat of lists over natural numbers would be given by

sort list_nat := { []_nat,
•_nat: nat × list_nat }.

If we wish additionally to use lists over the sort car, we must define a new sort and new constructors:

sort list_car := { []_car,
•_car: car × list_car }.

Moreover, for every predicate operating on lists we must have an extra version for every list variant, e.g.,

predicate append_nat: list_nat × list_nat × list_nat.
predicate append_car: list_car × list_car × list_car.
...

where every predicate version must be defined explicitly.

One method of overcoming this difficulty is to introduce *parameterization*. This concept has been studied extensively in abstract data type theory (cf. [19]) and has been applied to logic programming in Eqlog [7]. For instance, in Eqlog one can introduce a parameterized module

module LIST(X::ELEM)

which refers to a formal parameter specification ELEM. The functions and predicates operating on lists are defined by referring to the names introduced in ELEM. Instances of a parameterized module are obtained by replacing the formal parameter X::ELEM with an actual parameter, e.g., NAT or CITY, yielding the resulting instance, such as LIST(NAT) or LIST(CITY). Thus, there need be only one definition for every list predicate such as append, namely in the parameterized LIST module.

An advantage of such a parameterized concept is that apart from formal sort, function, and predicate names the parameter specification may also contain requirements for these; for instance, a parameterized specification for sets would require an equality relation on the elements. However, every instance of such a parameterized module must be generated explicitly, and for all names stemming from the parameterized specification different new names must be used in every instance.

A related way of avoiding multiple definitions for the same structure is the concept of polymorphism [29, 30]. As with the parameterized specifications outlined above, parameter variables also exist in the polymorphic case. However, they range over *all* sorts and it is not possible to restrict them by a parameter specification. Thus, there are no function or predicate symbols in the formal parameters, nor any constraints. On the other hand, in the polymorphic case it is not necessary to generate any instance explicitly, and the same names from a polymorphic definition are used in every instance.

In the following, we denote such sort variables by α and β. A polymorphic sort declaration for lists (here and later, we use the standard Prolog notation for lists) would then be

sort list(α) := { [], [_|_]: $\alpha \times$ list(α) }. (2)

Similarly, the polymorphic sort definition

sort pair(α,β) := { mkpair: $\alpha \times \beta$ }

defines ordered pairs over two sorts.

With such polymorphic sort functions at hand, all instances of them are automatically available, for example,

list(city)
list(airplanes)
list(boat)
list(list(city))
pair(boat,city)
pair(car,list(city))
...

In [5], a polymorphic type system for Prolog is proposed that allows parametric polymorphism (for a discussion of different polymorphism concepts, see [31]). In this approach the user must give sort declarations for the function and predicate symbols in a way similar to that for the many-sorted setting (see the preceding subsection), but with the additional polymorphic sort declarations. Mycroft and O'Keefe [5] define a notion of well-typedness such that (again as in the many-sorted case) static type checking is sufficient and no run-time type information is needed. Using (2) and the syntax of Section 2, the polymorphic definition of the append predicate is

predicate append: list(α) × list(α) × list(α).
append([], L, L).
append([H|T], L, [H|TL] :– append(T, L, TL). (3)

There have been several other approaches to a polymorphic type system for logic programming. Dietrich and Hagl [9] extend the approach of Mycroft and O'Keefe [5] to an order-sorted setting. To ensure that static type checking is sufficient, data-flow information within the program clauses is required in certain cases. Such data flow could be provided by a mode system, or in some cases it could be provided by global analysis, but the necessity for such information restricts the generality of this approach.

Another extension of [5] which deals with subsorts is reported in [32]. However, in this case dynamic type checking is needed. The dynamic type checking is not achieved by using a special unification algorithm but by a reduction to the ordinary Prolog case: A preprocessor inserts system-defined literals into the body of a clause in order to ensure sort-correct instantiations, reporting a run-time error otherwise. For instance, given the predicate declarations

Table 1 Polymorphically order-sorted unification.

t1	*t2*	*Result*
X:list(car)	[ford, opel]	[ford, opel]
X:list(car)	[ford, airbus, opel]	*fail*
X:list(car)	[ford, Y:vehicle, opel]	[ford, Y:car, opel]
X:list(car)	Y:list(boat)	Z:list(amphibious_vehicle)

```
predicate  p:  car.
    ...
predicate  q:  vehicle.
    ...
```

the clause

```
p(X) :- q(X).
```

in the definition of p is translated to

```
p(X) :- instantiated(X) & q(X).
```

Therefore, computing with uninstantiated sort-restricted variables, as demonstrated in the travel world example in the section on the order-sorted approach, is not possible; the advantages of order-sortedness are thus severely restricted in this framework.

The three approaches to a polymorphic type system discussed so far are all operational approaches: They do not provide a semantic notion of a type, but only a syntactic one. Their aim is to guarantee by static type checking that no type error can occur at run time, and the operational semantics is the same as the semantics of the untyped version (except for the third approach, as indicated by the example given above). There are approaches to a polymorphic type system for logic programming that also provide a semantic notion of a type [11, 12], but in contrast to the syntactic approaches discussed above, they have to take the type information into account at run time, requiring a special unification algorithm as in the order-sorted case.

The approach of Hanus [11] extends the polymorphic concept of Mycroft and O'Keefe [5]. By removing the restrictions of Mycroft and O'Keefe [5] on the use of type expressions, Hanus also allows for *ad hoc* polymorphism [31]. Besides a model-theoretic semantics, sound and complete deduction relations involving a special polymorphic unification are given. For instance, consider the polymorphic append predicate given in (3). Adding the specialized clause

```
append([opel, ford], [mercedes], [opel, ford, mercedes]).        (4)
```

is possible in this approach. It is not allowed in [5], since in that framework clause (4) is not well-typed because the types of the arguments in the head of (4)—list(car)—are not of the most general type—list(α)—declared for the predicate append in (3).

Whereas Hanus [11, 33] uses *ad hoc* polymorphism and exploits it, for example, for higher-order programming techniques but does not consider subsorts, the approach of Smolka [12] combines parametric polymorphism with an order-sorted approach. In this framework a sound and complete deduction relation for the corresponding model-theoretic semantics is defined that uses polymorphically order-sorted unification. To give an example, let us extend the travel world program in Figure 2 by the standard definition of polymorphic lists as given in (3). As in the polymorphic cases above, we now have not only a finite set of sorts but infinitely many sorts denoted by sort terms which show up in the unification procedure: Besides a sort constant such as car or vehicle, the sort restriction of a variable can be a sort term such as list(car) or list(list(vehicle)). To illustrate polymorphically order-sorted unification, we list in **Table 1** the unification results for different values of the terms t1 and t2.

The partial order on the (monomorphic) sort constants induces a partial order on sort terms as shown in the table. For instance, list(amphibious_vehicle) is a subsort of list(car) because amphibious_vehicle is a subsort of car.

In addition to this induced sort relationship, the approach of Smolka [12] allows explicit definitions of subsort relationships between polymorphic sorts such as

```
sort  lp(α,β) := list(α) ++ pair(α,β).
```

Thus, in this case the computation of the greatest lower bound of two sort restrictions requires taking into account those explicit subsort relationships between polymorphic sorts which can be handled by sort-rewriting systems. For instance, in trying to unify X:lp(car,airplane) and Y:list(boat), the sort restriction of X would have to be rewritten first as list(car). Several conditions for such sort-rewriting systems must be satisfied [12], and their use in a logic programming language could slow down the unification procedure. On the other hand, the important optimization of neglecting any maximal sort information is blocked in many cases (cf. Section 3): For instance, if list(α) has no supersort, then list(vehicle), list(city), list(list(city)), etc. will also be maximal, and such sort information can be neglected as well. However, if list(α) is a subsort of, e.g., lp(α,β), then no list instance at all will be maximal.

The type system of the logic programming language PROTOS-L [14, 17] is derived from the type system in TEL [12, 16]. However, PROTOS-L does not allow explicit subsort relationships between polymorphic types for the reasons discussed above. On the other hand, PROTOS-L supports the explicit definition of *maximal* sorts in order to reduce the sort information that must be considered at run time; these points are addressed in more detail in the next

$$\text{(E)}\quad \frac{E \;\&\; x \doteq x}{E} \qquad \text{if } x \text{ is a variable}$$

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

$$\text{(B)}\quad \frac{E \;\&\; x \doteq t}{\sigma(E) \;\&\; x \doteq t} \qquad \text{if } x \text{ is a variable, } t \text{ is a variable or a nonvariable term, and } x \text{ occurs in } E \text{ but not in } t, \text{ and where } \sigma = \{x/t\}$$

$$\text{(O)}\quad \frac{E \;\&\; t \doteq x}{E \;\&\; x \doteq t} \qquad \text{if } x \text{ is a variable and } t \text{ is not a variable}$$

Figure 3

The rules for ordinary term unification.

section and in Section 4, which presents a WAM extension for the polymorphic type concept of PROTOS-L.

3. Unification and typed unification

- *Term unification*

The rules for ordinary term unification are given in **Figure 3** in the style of equation-solving ([15]; for a survey on work in unification theory, see [34]), with the rules for elimination (E), decomposition (D), variable binding (B), and orientation (O). The idea is to start with a set of equations $E = \{t_1 \doteq t'_1, \cdots, t_n \doteq t'_n\}$ representing the pairs of terms to be unified, and to transform E into a set of equations E' that is in *solved form* by using the four given rules.

There are two conditions which indicate that no solution exists:

C1 If there is an equation $f(t_1, \cdots, t_n) \doteq g(t'_1, \cdots, t'_m)$ in E with $f \neq g$, then no solution exists.

C2 If there is an equation $x \doteq t$ in E such that $x \neq t$ and x occurs in t, then no solution exists.

E' is in solved form if it is of the form

$$E' = \{z_i \doteq t_i \mid i \in \{1, \cdots, n\}\},$$

where z_i are variables that do not occur elsewhere in E'. In this case the substitution represented by $\{z_1/t_1, \cdots, z_n/t_n\}$ is the most general unifier of the unification problem given by the original set of equations E.

- *Order-sorted unification*

The rules for order-sorted unification are given in **Figure 4**. Note that the rules for elimination, decomposition, and orientation look exactly as in the unsorted case. However, now every variable is of a particular fixed sort, and the difference from the unsorted case occurs in rules (B1) and (B2) when binding a variable.

A precondition for these rules to work correctly is that the sort structure must satisfy the following conditions:

- There are only finitely many sorts, and the subsort relationship is a partial order such that two different sorts have at most one common maximal subsort.

Additionally, there are the following requirements:

- There is no overloading of function symbols; i.e., for every function symbol f with arity n, there is exactly one arity declaration $f: s_1 \cdots s_n \rightarrow s$.
- Sorts are not empty; i.e., for every sort there is a ground term of that sort.
- The unification problems considered involve only well-sorted terms; i.e., the argument sort of t_i in $f(t_1, \cdots, t_n)$ must be of a subsort of s_i for $f: s_1 \cdots s_n \rightarrow s$.

Note that the first condition could be weakened by allowing certain cases of overloading; see for instance Waldmann [35].

(E) $$\frac{E \,\&\, x \doteq x}{E}$$ if x is a variable

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

(B1) $$\frac{E \,\&\, x \doteq t}{\sigma(E) \,\&\, x \doteq t}$$ if x is a variable of sort s, t is a variable or a nonvariable term of sort s', $s' \leq s$, and x occurs in E but not in t and where $\sigma = \{x/t\}$

(B2) $$\frac{E \,\&\, x \doteq y}{\sigma(E) \,\&\, x \doteq z \,\&\, y \doteq z}$$ if x is a variable of sort s, y is a variable of sort s', $x \neq y$, $s' \not\leq s$, z is a new variable of sort s'' where s'' is the greatest common subsort of s and s', and where $\sigma = \{x/z, y/z\}$

(O) $$\frac{E \,\&\, t \doteq x}{E \,\&\, x \doteq t}$$ if x is a variable and t is not a variable

Figure 4

The rules for order-sorted unification.

Now there are two situations in addition to those for the unsorted case such that there is no solution:

C3 If there is an equation $x \doteq t$ in E such that the sort of t is strictly greater than the sort of x, then no solution exists.

C4 If there is an equation $x \doteq y$ in E such that the sorts of x and y do not have a common subsort, then no solution exists.

If none of the conditions *C1–C4* is satisfied and E' is in solved form, $E' = \{z_i \doteq t_i \mid i \in \{1, \cdots, n\}\}$, then

$\{z_i/t_i \mid z_i \doteq t_i \in E'$ and z_i occurs in the original set of equations $E\}$

represents a most general unifier of the original unification problem E.

Consider the rules for order-sorted unification as given in Figure 4. Comparing the rules to the unsorted version (Figure 3) reveals two places where sort information is involved, namely (B1) and (B2). In (B1), a subsort test $s' \leq s$ is required, involving the sort s of the variable x and the sort s' of t. Thus, if t is a variable, s' is its sort, and if t is the nonvariable term $f(t_1, \cdots, t_n)$, s' is the target sort of the top-level function symbol $f: s_1 \cdots s_n \rightarrow s'$ of t.

To give an example, binding the term opel (of sort car) to variable Y:amphibious_vehicle would not be possible under rule (B1), but binding the term amphi1 of sort amphibious_vehicle to Y:car would be possible.

However, it is interesting to note that under the requirements given above (in particular, no overloading and only well-sorted unification problems) when trying to apply (B1), the rules will never try to bind a variable of sort city to a term of sort car, nor, conversely, a variable of sort vehicle to a term of sort city. Therefore, whenever the variable X involved in (B1) is either of sort vehicle or of sort city (i.e., the sort of X is maximal with respect to the subsort relationship), the subsort test will always be successful.

A similar observation is true for the other binding rule (B2). Under the given requirements, if either of the involved variables x and y in the equation $x \doteq y$ is of a maximal sort (vehicle or city in our example), the other one is either of the same sort or of a subsort thereof. If the sort s of x is maximal, rule (B2) cannot be applied, and (B1) must be used instead. Otherwise, if in (B2) the sort s' of y is maximal and s is nonmaximal, we have $s < s'$, and the sort of the resulting variable z (i.e., s) is determined completely by the nonmaximal sort. In fact, we could simplify (B2) in this case by not introducing a new variable

but by orienting the equation to $y \doteq x$ and just doing the binding as in (B1).

We conclude that in both places where sort information is involved, the sort information for variables of maximal sorts is redundant. Thus, if we leave out the redundant sort tests for variables of maximal sorts as indicated above, both (B1) and (B2) reduce to special cases of the binding rule (B) in the unsorted case.

Thus, we could do the following optimization for a given well-sorted unification problem $E = \{t_1 \doteq t_1', \cdots, t_n \doteq t_n'\}$:

1. Transform E into E' by replacing every variable x of a maximal sort with an "unsorted" variable x_u.
2. Transform E' into E'' using the rules in Figure 4 with the modified version of (B1) and (B2) [i.e., effectively applying rule (B)] when binding an "unsorted" variable.
3. Replace every "unsorted" variable x_u from E' with the original sorted variable x.

In the degenerate order-sorted case where all sorts are maximal, all sort information therefore becomes redundant. This means that in step 2 above we do just ordinary unsorted unification. Of course, this comes as no surprise, since that case is just the many-sorted case where, for well-sorted unification problems, the unsorted unification rules work well. In [12] this optimization is suggested and elaborated in detail for the more general polymorphic order-sorted case discussed below.

• *Polymorphic order-sorted unification*

We now present the rules for polymorphically order-sorted unification as it is used in the logic programming language PROTOS-L; the general case with additional explicit subsort relationships between polymorphic sorts is given in [12].

First, let us recall the order-sorted unification rules discussed in the previous section. There, we considered every variable to be of a fixed sort. However, one could also use unsorted variables and introduce the sort restrictions on the variables by a special predicate. This yields certain technical advantages; for instance, when unifying two variables X:car and Y:boat one can produce a new sort restriction for the variables (i.e., X:amphibious_vehicle) instead of introducing a new variable as done by the rules in Figure 4. We have already used this approach informally in our discussion in Section 2.

Thus, in addition to the set E of equations we now also have a set $P = \{t_1:\tau_1, \cdots, t_n:\tau_n\}$ of sort restrictions with value terms t_i and sort terms τ_i. We call P a *prefix* if all t_i are variables that are pairwise distinct. As in the order-sorted case, we consider only well-sorted unification problems P & E, which now means that P is a prefix having a sort restriction for every variable in E and that for every equation $t = t'$ in E there exists a sort term τ such that both t and t' belong to τ under the sort restrictions given in P.

For the optimized version of the unification rules, such a well-sorted unification problem P & E is first transformed such that maximal sort information is neglected: Every maximal sort is replaced recursively by the special symbol $\top$. For a sort term τ the *approximation* of τ is defined by

$$
\begin{array}{lll}
\downarrow\tau = \top & & \text{if } \tau \text{ is a sort variable;} \\
\downarrow\tau = \top & & \text{if } \tau \text{ is a sort constant that is maximal in the partial order on the sorts;} \\
\downarrow\xi(\tau_1, \cdots, \tau_n) = \top & & \text{if } \downarrow\tau_1 = \cdots = \downarrow\tau_n = \top; \\
\qquad\qquad\quad \bot & & \text{if } \xi(\tau_1, \cdots, \tau_n) \text{ cannot be instantiated (see below);} \\
\xi(\downarrow\tau_1, \cdots, \downarrow\tau_n) & & \text{otherwise.}
\end{array}
$$

The approximation $\downarrow P$ of a prefix P is obtained by replacing all sort terms in P with their approximations.

Analogously to the computation of greatest common subsorts in the order-sorted case, one now needs to compute the infimum of two sort terms. For instance, the infimum of list(car) and list(boat) yields list(amphibious_vehicle). However, whereas there is no common subsort of, e.g., airplane and car, the infimum of list(airplane) and list(car) is well-defined, i.e., the set consisting of exactly the empty list.

Thus, another special symbol $\bot$ is introduced above that denotes the "empty" sort which becomes a subsort of every other sort. Then the infimum of list(airplane) and list(car) is list($\bot$), since there is a ground term of sort list($\bot$), namely the empty list [ ]. In other words, list($\bot$) can be instantiated or is *inhabited*. However, given the definition of standard pairs as above, the infimum of, e.g., pair(airplane,city) and pair(car,city) is not inhabited, since pair($\bot$,city) cannot be instantiated. Thus, the infimum of the two sort terms is $\bot$.

In general the infimum $\inf(\tau, \tau')$ of two sort term approximations τ and τ' is given by

$$
\begin{array}{lll}
\inf(\top, \tau) = \tau, & & \\
\inf(\tau, \top) = \tau, & & \\
\inf(\tau, \tau') = \tau'' & & \text{if } \tau \text{ and } \tau' \text{ are sort constants with maximal common subsort } \tau'', \\
\inf(\xi(\tau_1, \cdots, \tau_n), \xi(\tau_1', \cdots, \tau_n')) & & \\
\qquad = (\xi(\inf(\tau_1, \tau_1'), \cdots & & \text{if this term} \\
\qquad\quad \inf(\tau_n, \tau_n')) & & \text{can be instantiated,} \\
\inf(\tau, \tau') = \bot & & \text{otherwise.}
\end{array}
$$

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

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

(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) $\cdots$ (DS)

(O) $$\frac{P \,\&\, E \,\&\, t \doteq x}{P \,\&\, E \,\&\, x \doteq t}$$ if t is not a variable

---

(ES) $$\frac{P \,\&\, f(t_1, \cdots, t_n) : s}{P}$$ if $f : s_1 \cdots s_n \rightarrow s'$ and $s' \leq s$

(ES′) $$\frac{P \,\&\, f(t_1, \cdots, t_n) : \top}{P}$$

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

(DS) $$\frac{E \,\&\, f(t_1, \cdots, t_n) : \xi(\tau_1, \cdots, \tau_m)}{E \,\&\, t_1 : \downarrow\theta(\tau'_1) \,\&\, \cdots \,\&\, t_n : \downarrow\theta(\tau'_n)}$$ if $f : \tau'_1 \cdots \tau'_n \rightarrow \xi(\alpha_1, \cdots, \alpha_m)$ and where $\theta = \{\alpha_1/\tau_1, \cdots, \alpha_m/\tau_m\}$

Figure 5

The rules for polymorphic order-sorted unification.

Using this infimum operation, the rules for polymorphic order-sorted unification are given in **Figure 5**, again under the assumption that there is no overloading of function symbols and that only well-sorted unification problems are considered. Let $P \,\&\, E$ be a well-sorted unification problem and let these rules transform the approximation $P' \,\&\, E$ into the form $P'' \,\&\, E''$. The $P'' \,\&\, E''$ is in solved form and presents a solution if E'' is in solved form (in the sense of the order-sorted approach), P'' is a prefix that does not contain $x : \bot$, the variables in P'' do not occur on the left-hand sides of an equation in E'', and every right-hand side of an equation in E'' is well-sorted under the prefix P''.

Analogously to the unsorted case (compare the three steps at the end of the subsection on the order-sorted approach), one can now solve a well-sorted unification problem by

1. Transforming P into the approximation P'.
2. Transforming $P' \,\&\, E$ into $P'' \,\&\, E''$ using rules (E)–(O) and (ES)–(DS) from Figure 5.
3. Transforming P'' into P''' by replacing the approximations in P'' with the actual sort information from the original prefix P and the substitution defined by E''.

The third step uses the *retract* operation, which transforms an approximation τ and a sort term τ' that is an upper bound of τ into the retraction $\tau \uparrow \tau'$ as follows:

$\top \uparrow \tau = \tau$,

$\tau \uparrow \tau' = \tau$ if τ and τ' are sort constants and τ is a subsort of τ',

$\xi(\tau_1, \cdots, \tau_n) \uparrow \xi(\tau'_1, \cdots, \tau'_n) = \xi(\tau_1 \uparrow \tau'_1, \cdots, \tau_n \uparrow \tau'_n)$;

$\tau \uparrow \tau'$ undefined otherwise.

The upper bounds for the sort term approximations in P'' are obtained by applying the substitution $\sigma_{E''}$ defined by E'' to the original prefix P and transforming $\sigma_{E''}(P)$ again into a prefix, say $P_{E''}$. If $x:\tau$ is now in P'' and $x:\tau'$ is in $P_{E''}$, then $x:\tau \uparrow \tau'$ will be in P'''.

In this section we have discussed three unification procedures related to untyped, order-sorted, and polymorphic order-sorted logic programming, respectively. By presenting them in the uniform framework of equation-solving transformations, we have obtained a common basis for comparing them. Using SLD resolution with the correspondingly extended unification yields for any of the three cases an operational semantics. This operational semantics can be realized on a suitable abstract machine, and in the following section we will show how the different transformation rules are reflected in such an abstract-machine implementation.

4. Abstract machines

As noted in the Introduction, an efficient implementation of logic programming and its required term unification became available with the Warren Abstract Machine. In this section we present an extension of the WAM to order-sorted and polymorphically order-sorted unification. We start with a short presentation of the original WAM.

- *The Warren Abstract Machine*

For a detailed description of the WAM we refer to [2, 3] or [36]. Here, we give only a brief overview of the machine model that allows us to describe the extensions and modifications later on.

The machine model of the WAM consists essentially of the following:

1. The *code area* containing the machine code of the program.
2. Three stacks:
   - The *global stack*, containing all structures that are generated during program execution. Structures are represented by the top-level function symbol followed by the arguments of the structure.
   - The *local stack*, containing information on the execution structure of the program and backtracking information. In particular, the local stack contains *choice points* for alternative clauses for a goal and *environments* containing the status information needed for evaluating the rest of a clause.
   - The *trail stack*, containing the addresses of the variables that have been bound during program execution and that have to be reset upon backtracking.
3. A set of *registers* defining the current machine state, e.g., a next-instruction pointer, pointers to the tops of the three stacks, a pointer to the last choice point, etc., and a special set of argument registers.

An actual implementation will also contain additional components, such as a *symbol table* containing the arity and the print names for every function and predicate symbol occurring in the program.

The instruction set of the WAM can be divided into five classes:

1. The *get* instructions are used for the arguments in the head of a clause in order to unify them with incoming arguments.
2. The *put* instructions are used for the arguments in the body of a clause in order to build up the arguments of the subgoals.
3. The *unify* instructions are used for the deeper-nested arguments within a structure.
4. The *environment* and *choice* instructions are used for the management of procedure calls, choice points, and environments.
5. The *switch* instructions are used to select only a subset of all alternative clauses from the set of all clauses of a predicate, depending on the value of the given arguments.

Additionally, there are some lower-level instructions that are called by certain instructions above (e.g., there is a low-level *unify* instruction, and the low-level instruction *fail* is called in order to initiate backtracking).

- *Extending the WAM to typed unification*

As pointed out in Section 2, the many-sorted case does not need special treatment at run time; therefore, the WAM needs no modification. However, in the order-sorted case, types are present at run time.

Order-sorted WAM extensions have been suggested in [10, 37], and in [38] an abstract machine implementation of PROTOS-L without polymorphism is given. The subsort relationships between the sorts are compiled into a square matrix such that both a subsort test and the computation of the greatest lower bound of two sorts can be done in constant time. However, the matrix implementation is not easily extendible to separate compilation of modules, and it requires space that is quadratic in the number of sorts.

Instead of going into the details of the order-sorted WAM extensions, we move directly to the polymorphic order-sorted case as it has been incorporated in PROTOS-L. We call this WAM extension the PROTOS Abstract Machine (PAM) [14, 39], and in the following, we describe its differences from the WAM.

The first difference is that the sort structure of a program must be stored in the machine. This is done

primarily in a new *sort table* containing the following information:

- The sort constants and their subsort relationships (e.g., airplane and city having no subsorts; vehicle having subsorts airplane, boat, and car).
- The arity of the polymorphic sorts [e.g., list(α) has arity 1, while pair(α,β) has arity 2].
- Information on how sort terms can be instantiated. [For example, the instantiation information for the sort term list(α) states that α may be the "empty" sort $\perp$ because [] is a term of sort list($\perp$). For the sort term pair(α,β) it is required that both α and β be instantiated by a sort other than $\perp$, since otherwise there would be no term of that pair sort.]
- Additionally, the symbol table must contain the arity of the function symbols in order to determine the target sort of the top-level function symbol of a term [e.g., opel is of sort car, stuttgart of sort city, and the list constructor "•" has list(α) as its target sort].

The second major difference is the representation of variables. In the WAM variables are represented by pointers. Binding a variable simply requires pointing to the value (e.g., a structure in the global stack), and a free variable is a pointer to itself.

In the polymorphic PAM the variables are divided into three classes:

- The *free* variables, which are treated in the same manner as the free variables in the ordinary Prolog case. These are the variables for which no sort information is relevant at run time; i.e., the approximation of their sort restriction is $\top$.
- The *mono* variables, whose sort restriction is a (monomorphic) sort constant. These are represented as in the nonpolymorphic PAM; i.e., each *mono* variable has a sort constant attached to it.
- The *poly* variables, whose sort restriction is a nonconstant sort term. The sort restriction of these variables is a pointer to the respective sort term. The sort terms themselves are represented as regular terms, i.e., in the global stack where the polymorphic sort symbol is followed by its arguments.

The trail stack must record the previous sort restrictions of the variables. For the *free* variables this information is void; for the *mono* variables it is a sort constant to be stored on the trail stack, and in the case of *poly* variables it is a pointer to the global stack.

To determine the changes for the instruction set, consider the unification rules given in Figure 5. Because rules (E)–(O) are effectively the same as in the unsorted case (Figure 3), they do not cause any changes, except that the binding rule (B) refers to rules (ES)–(DS). Rules (ES)–(DS) manipulate the sort restrictions. For the elimination of a monomorphic sort restriction (ES), the target sort of the function symbol (which is recorded in the symbol table) must be in the subsort relationship (documented in the sort table) to the required sort. For the other elimination rule (ES′), no action is required at all.

The merging of two sort restrictions (MS) requires the computation of the infimum of the two sort terms. The information required is the subsort relationship for the monomorphic sorts and the instantiability of polymorphic sorts, which are both contained in the sort table.

The decomposition of a sort restriction (DS) requires that the arguments of a polymorphic sort term, such as pair(car,airplane), be propagated to the arguments of a value term, such as mkpair(X:vehicle,Y:vehicle), yielding mkpair(X:car,Y:airplane) in the given example.

The instructions that must be modified because of these observations are the *get*, *put*, and *unify* instructions, which are the instructions used for the unification of terms. They are generated by the compiler, depending on the occurrences of a variable or a term in a clause.

For example, suppose that the ith argument of the head of a clause is a (temporary) variable represented by X_n. All one must do in the unsorted case is set X_n to the value of the ith incoming argument A_i because at this point X_n is guaranteed to be free. This is achieved by the WAM instruction

get_x_variable X_n, A_i .

However, in the PAM case the sort restriction of X_n must be taken into account. Following the distinctions made above, X_n is either a *free*, a *mono*, or a *poly* variable. Accordingly, the WAM instruction get_x_variable is replaced by three PAM instructions:

1. get_x_free X_n, A_i.
2. get_x_mono X_n, A_i, s.
3. get_x_poly X_n, A_i, st.

The first instruction corresponds exactly to the WAM case because no sort-related action need be performed. The second instruction is generated for a variable with the sort constant s as its restriction. The execution of this instruction creates a new value cell on the global stack with sort restriction s, which is then unified with A_i in order to ensure that A_i is of sort s (possibly by further restricting the sort of A_i if it is a variable). Analogously, the third instruction generates a new value cell with the polymorphic sort restriction st, where st is a pointer to a sort term on the global stack. The unification with A_i then ensures that A_i satisfies the sort restriction.

Thus, the three PAM instructions represent three special cases of the general unification procedure where at compile

time it is guaranteed that one of the two terms (namely X_n) to be unified is an unbound variable. On the other hand, for the second and any subsequent occurrence of X_n in the head of a clause, the full unification is required. In the WAM case this is achieved by the instruction

get_x_value X_n, A_i,

and the only difference for the PAM case is that polymorphic order-sorted unification is used instead of ordinary term unification.

Analogous modifications are made for all other *get*, *put*, and *unify* instructions: If a WAM instruction is used for the first occurrence of a variable (i.e., get_y_variable, put_x_variable, put_y_variable, unify_x_variable, unify_y_variable), it is replaced by three PAM instructions for a *free*, a *mono*, and a *poly* variable, respectively. For these as well as for all remaining instructions, the extended unification is used.

However, except for the extended trail information, the entire execution control including the management of choice points and environments can be treated as in the WAM case. For the *switch* instructions, an additional optimization is possible. For the exclusion of alternative clauses it is now also possible to take into account the sort restrictions for the variables. For instance, having the clauses

```
p(opel,...)   :- ...
p(mercedes,...)   :- ...
p(bo747,...)   :- ...
p(ferry,...)   :- ...
```

for the predicate p and a call to p with uninstantiated variable X in its first argument, indexing (on the first argument) would not yield any selective effect in the WAM case. However, if X has a sort restriction, indexing does gain something in the sorted case: For instance, the sort restriction X:airplane would switch directly to the third clause and rule out all other clauses immediately.

The remaining sort-related modifications required in the PAM case are instructions for the retraction of the actual sort of a variable from its original sort and its approximation, as well as some additional lower-level instructions, e.g., for the computation of the infimum of two sort terms.

- *Implementation*

There are numerous WAM implementations based more or less directly on D. H. Warren's original design [2, 3]. Within the European EUREKA project PROTOS ("Logic Programming Tools for Building Expert Systems"), the PAM with the features described above has been implemented in C and is available on the IBM RT Personal Computer® 6150, the IBM RISC System/6000™ processor, and the IBM PS/2® under the AIX® operating system [39, 40]. A compiler for PROTOS-L that generates PAM code has been developed using the TEL system [16, 41]. Static type checking and type inferencing done at compile time could be derived from this system. However, TEL's run-time unification is not correct because the ordinary term unification of the underlying Prolog system is used. The PAM provides the first abstract machine implementation for polymorphic order-sorted unification; several applications written in PROTOS-L and running on the PAM have already been developed. Within the PROTOS project, PROTOS-L is being used for the development of a knowledge-based planning system in the chemical production area.

5. Conclusions and further work

There are two main motivations for the introduction of types into the logic programming paradigm. One stems from a software engineering point of view, since a typing concept can support various software engineering principles such as data abstraction, modularization, static consistency checks, etc., which are vital in large applications. On the other hand, from a computational point of view, the use of types with subtypes can drastically reduce the search space, especially in AI applications.

Starting from these observations, in this paper we have given a survey of many-sorted, order-sorted and polymorphic type concepts as used in logic programming. When the typing concept supports computations with subtypes, types are also present at run time through typed unification. In order to compare the related unification procedures, we have presented them in the uniform framework of equation-solving transformations. Both for an order-sorted and for a polymorphic order-sorted type concept, we have argued that the classical Warren Abstract Machine can be extended to the corresponding typed unification, putting special emphasis on polymorphic order-sorted unification as it has been realized in the PAM, the core implementation component of the logic programming language PROTOS-L.

Additional features of the PAM, some of which have been added since this paper was written (summer 1990) and are not described here, include a module concept allowing for separate compilation, the integration of a deductive database component, and an object-oriented interface to the OSF/Motif system [14, 17, 42, 43]; all of these features have already been used successfully in various planning applications [44].

In [40] the PAM is described in more detail. It is shown that representing the three classes of typed variables (i.e., *free*, *mono*, and *poly*) in the PAM by special tags leads to the situation that the type extension in the PAM is truly orthogonal to that in the WAM. Any untyped program is carried out in the PAM with the same efficiency as in the WAM: Adding the trivial one-sorted type information to

such a program reveals that the PAM code will contain only the *free* case for variables. Apart from the minor difference of representing a free (unconstrained) variable not by a self-reference (as in the WAM) but by a special tag, the generated and executed code is thus exactly the same for both the WAM and the PAM. 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. Moreover, in [45] a mathematical correctness proof for the PAM, based on the WAM correctness proof in [46, 47], is given.

In order to allow "extra-logical" features, in particular meta-programming, the PROTOS-L type system as discussed in this paper would need to be extended. Meta-primitives like negation-as-failure, set-of, etc., as well as the concept of call-back procedures needed in the object-oriented interface to OSF/Motif, have already been built into the current prototype system. In order to allow general meta-calls the type system still has to be extended, e.g., toward higher-order logic; an approach in this direction is, for instance, λProlog [8]. In the recent work of Kwon et al. [48], which aims at developing an abstract machine for λProlog, a WAM-based implementation scheme for a first-order logic programming language with ML-style typing (with the possibility of *ad hoc* polymorphism and the necessity for run-time type checking) is presented, and an extension to an alternative PROTOS-L implementation is outlined. In Gödel [13] types play the central role in providing a logical semantics for meta-programming constructs. For most Gödel programs static type checking is sufficient, but in some cases Gödel types also have to be considered at run time.

Among the additional extensions of PROTOS-L we are currently investigating is the extension of the type concept by descriptions of types with attributes. As in frame- or object-oriented approaches, the elements of a type are not defined by enumerating them or by defining the constructors generating them, but by giving a set of attributes characterizing them. This yields a much more flexible type description method, since (for instance) adding an attribute to an already existing type should not invalidate any existing program part. A term of such an attributed type would consist of a list of attribute-value pairs (cf. [49]). The design of an efficient abstract machine for such an extended type concept is the subject of current research.

Acknowledgments

For their contributions and support toward the development and realization of the PROTOS-L system, whose kernel is embodied by the PAM, I would like to thank all members of the PROTOS project team at IBM in Stuttgart, in particular Gregor Meyer and Heiner Semle. Harald Ganzinger, Michael Hanus, and Gert Smolka made helpful comments on an earlier version of this paper. Thanks also to the anonymous referees of this paper for their valuable suggestions, and to Birgit Wendholt for some additional hints. The work reported here was carried out in PROTOS ("Logic Programming Tools for Building Expert Systems"), an international EUREKA project (EU56).

References

1. J. W. Lloyd, *Foundations of Logic Programming. Symbolic Computation*, Springer-Verlag, Berlin, 1984.
2. D. H. D. Warren, "Compiling Predicate Logic Programs," *D. A. I. Research Report,* University of Edinburgh, Edinburgh, Scotland, U.K., 1977.
3. D. H. D. Warren, "An Abstract PROLOG Instruction Set," *Technical Report 309*, SRI International, Menlo Park, CA, 1983.
4. C. Walther, "A Mechanical Solution of Schubert's Steamroller by Many-Sorted Resolution," *Artif. Intell.* **26**, 217–224 (1985).
5. A. Mycroft and R. A. O'Keefe, "A Polymorphic Type System for Prolog," *Artif. Intell.* **23**, 295–307 (1984).
6. *Functional and Logic Programming*, D. DeGroot and G. Lindstrom, Eds., Prentice-Hall, Inc., Englewood Cliffs, NJ, 1986.
7. J. A. Goguen and J. Meseguer, "Eqlog: Equality, Types, and Generic Modules for Logic Programming," *Functional and Logic Programming*, D. DeGroot and G. Lindstrom, Eds., Prentice Hall, Inc., Englewood Cliffs, NJ, 1986, pp. 295–263.
8. G. Nadathur and D. Miller, "An Overview of λProlog," *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, K. Bowen and R. Kowalski, Eds., MIT Press, Cambridge, MA, 1988, pp. 810–827.
9. R. Dietrich and F. Hagl, "A Polymorphic Type System with Subtypes for Prolog," *Proceedings of the 2nd European Symposium on Programming: Lecture Notes in Computer Science*, Volume 300, Springer-Verlag, Berlin, 1988, pp. 79–93.
10. M. Huber and I. Varsek, "Extended Prolog for Order-Sorted Resolution," *Proceedings of the 4th IEEE Symposium on Logic Programming*, San Francisco, 1987, pp. 34–45.
11. M. Hanus, "Horn Clause Specifications with Polymorphic Types," Dissertation, FB Informatik, Universität Dortmund, Dortmund, Germany, 1988.
12. G. Smolka, "Logic Programming over Polymorphically Order-Sorted Types," Dissertation, FB Informatik, Universität Kaiserslautern, Kaiserslautern, Germany, 1989.
13. P. M. Hill and J. W. Lloyd, "The Gödel Report (Preliminary Version)," *TR-91-02*, Dept. of Computer Science, University of Bristol, Bristol, UK, March 1991.
14. C. Beierle, "Types, Modules and Databases in the Logic Programming Language PROTOS-L," *Sorts and Types for Artificial Intelligence. Proceedings of the Workshop, Eringerfeld, April 1989*, K. H. Bläsius, U. Hedtstück, and C.-R. Rollinger, Eds., *Lecture Notes in Artificial Intelligence*, Volume 418, Springer-Verlag, Berlin, 1990.
15. A. Martelli and U. Montanari, "An Efficient Unification Algorithm," *ACM Trans. Programming Lang. & Syst.* **4**, 258–282 (1982).

16. G. Smolka, "TEL (Version 0.9), Report and User Manual," *SEKI-Report SR 87-17*, FB Informatik, Universität Kaiserslautern, Kaiserslautern, Germany, 1988.
17. C. Beierle, S. Böttcher, and G. Meyer, *Draft Report of the Logic Programming Language PROTOS-L*, IWBS Report 175, IBM Germany, Scientific Center, Institute for Knowledge-Based Systems, Stuttgart, 1991.
18. J. A. Goguen, J. W. Thatcher, and E. G. Wagner, "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types," *Current Trends in Programming Methodology IV: Data and Structuring*, R. Yeh, Ed., Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978, pp. 80–144.
19. H. Ehrig and B. Mahr, "Fundamentals of Algebraic Specification 1—Equations and Initial Semantics," *EATCS Monographs on Theoretical Computer Science*, Volume 6, Springer-Verlag, Berlin, 1985.
20. A. Oberschelp, "Untersuchungen zur mehrsortigen Quantorenlogik," *Math. Ann.* **145,** 297–333 (1962).
21. J. A. Goguen, "Order Sorted Algebra," *Semantics and Theory of Computation Report No. 14*, University of California, Los Angeles, 1978.
22. J. A. Goguen and J. Meseguer, "Order-Sorted Algebra I: Partial and Overloaded Operators, Errors and Inheritance," Technical Report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1987.
23. G. Smolka, W. Nutt, J. A. Goguen, and J. Meseguer, "Order Sorted Equational Computation," *Resolution of Equations in Algebraic Structures*, H. Ait-Kaci and M. Nivat, Eds., Academic Press, Inc., New York, 1989.
24. K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer, "Principles of OBJ2," *Proceedings of the 12th ACM Conference on Principles of Programming Languages*, B. Reid, Ed., Association for Computing Machinery, New York, 1985, pp. 52–66.
25. M. Gogolla, "Über partiell geordnete Sortenmengen und deren Anwendung zur Fehlerbehandlung in abstrakten Datentypen," Dissertation, Technische Universität Braunschweig, Braunschweig, Germany, 1986.
26. C. Walther, "A Many-Sorted Calculus Based on Resolution and Paramodulation," *Research Notes in Artifical Intelligence*, Pitman, London, and Morgan Kaufmann, Los Altos, CA, 1987.
27. C. Walther, "Many-Sorted Unification," *J. ACM* **35,** 1–17 (1988).
28. M. Schmidt-Schaub, "A Many-Sorted Calculus with Polymorphic Functions Based on Resolution and Paramodulation," *Proceedings of the 9th International Conference on Artificial Intelligence*, Morgan Kaufmann, Los Altos, CA, 1985, pp. 1162–1168.
29. R. Milner, "A Theory of Type Polymorphism in Programming," *J. Computer Syst. Sci.* **17,** 348–375 (1978).
30. L. Damas and R. Milner, "Principal Type-Schemes for Functional Programs," *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, 1982, pp. 207–212.
31. L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction and Polymorphism," *ACM Comput. Surv.* **17,** 471–522 (1985).
32. G. Dayantis, "Types, Modules and Abstraction in Logic Programming," *Algebraic and Logic Programming*, J. Grabowski, P. Lescanne, and W. Wechler, Eds., Akademie-Verlag, Berlin, 1988, pp. 127–136.
33. M. Hanus, "Horn Clause Programs with Polymorphic Types: Semantics and Resolution," *Theor. Comput. Sci.* (Netherlands) **89,** 63–106 (1991).
34. J. Siekmann, "Unification Theory," *J. Symbolic Computation* **7**, 207–274 (1989).
35. U. Waldmann, "Unification in Order-Sorted Signatures," *Forschungsbericht Nr. 298*, FB Informatik, Universität Dortmund, Dortmund, Germany, 1989.
36. H. Aït-Kaci, *Warren's Abstract Machine: A Tutorial Reconstruction*, M.I.T. Press, Cambridge, MA, 1991.
37. H. J. Bürckert, "Extending the WARREN Abstract Machine to Many-Sorted Prolog," *SEKI-Memo-85-07*, FB Informatik, Universität Kaiserslautern, Kaiserslautern, Germany, 1985.
38. B. Müller, "Design and Implementation of an Abstract Machine for Order-Sorted Logic Programs," *Studienarbeit Nr. 711*, Universität Stuttgart und IBM Deutschland GmbH, Stuttgart, Germany, October 1988 (in German).
39. H. Semle, "Extension of an Abstract Machine for Order-Sorted Prolog to Polymorphism," *Diplomarbeit Nr. 583*, Universität Stuttgart und IBM Deutschland GmbH, Stuttgart, Germany, April 1989 (in German).
40. C. Beierle, G. Meyer, and H. Semle, "Extending the Warren Abstract Machine to Polymorphic Order-Sorted Resolution," *Logic Programming: Proceedings of the 1991 International Symposium*, V. Saraswat and K. Ueda, Eds., MIT Press, Cambridge, MA, pp. 272–286.
41. W. Nutt and G. Smolka, "Implementing TEL," *SEKI-Report*, FB Informatik, Universität Kaiserslautern, Kaiserslautern, Germany, 1992, in preparation.
42. G. Meyer, "Rule Evaluation on Databases in the PROTOS-L System," *Diplomarbeit Nr. 630*, Universität Stuttgart und IBM Deutschland GmbH, Stuttgart, Germany, December 1989 (in German).
43. H. Jasper, "A Logic-Based Programming Environment for Interactive Applications," *Proceedings of Human Computer Interaction International*, Stuttgart, 1991.
44. C. Beierle, "Knowledge Based PPS Applications in PROTOS-L," in *Logic Programming in Action, Lecture Notes in Artificial Intelligence*, Vol. 636, G. Comyn, N. E. Fuchs, and M. J. Ratcliffe, Eds., Springer-Verlag, Berlin, 1992.
45. C. Beierle and E. Börger, "Correctness Proof for the WAM with Types," *Computer Science Logic—CSL'91*, E. Börger, H. Kleine Büning, G. Jäger, and M. M. Richter, Eds., *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1992 (in press).
46. E. Börger and D. Rosenzweig, "From Prolog Algebras Towards WAM—A Mathematical Study of Implementation," *Computer Science Logic*, E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, Eds., *Lecture Notes in Computer Science*, Volume 533, Springer-Verlag, Berlin, 1991, pp. 31–66.
47. E. Börger and D. Rosenzweig, "WAM Algebras—A Mathematical Study of Implementation, Part II," *Russian Conference on Logic Programming '91, Lecture Notes in Computer Science*, Springer-Verlag, 1992 (in press).
48. K. Kwon, G. Nadathur, and D. S. Wilson, "Implementing Logic Programming Languages with Polymorphic Typing," *Technical Report CS-1991-39*, Duke University, Durham, NC, October 1991.
49. H. Aït-Kaci and R. Nasr, "LOGIN: A Logic Programming Language with Built-in Inheritance," *J. Logic Program.* **3,** 185–215 (1986).

Received November 20, 1990; accepted for publication June 18, 1992

Christoph Beierle *IBM Germany Scientific Center Heidelberg, Institute for Knowledge Based Systems, P.O. Box 80 08 80, D-7000 Stuttgart 80, Germany (BEIERLE at DS0LILOG.BITNET).* Dr. Beierle received a Diploma degree in computer science from the University of Bonn in 1980. From 1981 to 1986 he was a research associate at the Universities of Bonn and Kaiserslautern, where he worked primarily in the area of formal foundations of software development and verification. In 1985, he received his Ph.D. in computer science from the University of Kaiserslautern, with a thesis on algebraic implementation techniques. From 1986 to 1987, he held an IBM Postdoctoral Fellowship in the LILOG (Linguistic and Logic Methods) project at IBM Germany, doing work on knowledge representation formalisms used for natural-language processing. Since 1988 Dr. Beierle has been with the Institute for Knowledge Based Systems of IBM Germany. He is project leader for the international EUREKA project PROTOS (Logic Programming Tools for Building Expert Systems), in which advanced extensions of logic programming are developed and applied to knowledge-based planning applications. In 1991, Dr. Beierle received an IBM Outstanding Innovation Award for his work on the PROTOS-L system.