Extensible Unification by Metastructures[*]

Ulrich Neumerkel

Institut für Praktische Informatik
Abteilung für Programmiersprachen und Übersetzerbau
Technische Universität Wien
`ulrich@vip.UUCP`

---Abstract----

Metastructures are a new way to extend Prolog's built-in unification by user definitions. While the behavior of ordinary terms during unification remains the same, the user can define the behavior of metastructures. Metastructures enable the user to implement efficiently many proposed enhancements of Prolog such as functional extensions, constraints according to the CLP scheme and meta-logical primitives in terms of Prolog, instead of relying on a highly specialized system. Metastructures can be implemented so efficiently that programs not using this extension are executed with a very small overhead. Metastructures have the same execution expense as an efficient implementation of `freeze/2`. Additionally, the system's garbage collector is able to detect and remove all unused metastructures without knowing their actual definition.

1 Introduction

In recent years the area of logic programming languages has evolved rapidly. The number of existing extensions of Prolog is very large. For all intents and purposes, in a specific Prolog implementation only a few extensions can be implemented because of the large development time required. The technique of metainterpreters is the only choice for realizing arbitrary languages in one system within a reasonable amount of time. While there already are sophisticated compiler-generation techniques for meta-interpreters e.g. [Neuma88], the compilation to Prolog cannot always remove the huge interpretation overheads, especially since extensions local to unification cannot be handled efficiently with this technique.

When considering existing approaches of extensions to syntactic unification, we observe: highly specialized implementations, where the unification algorithm cannot be manipulated by the user (e.g. [Col87,JaLa87,vH89]), approaches too general to be implemented efficiently and too general to allow the reuse of existing Prolog programs [Ko84], or extensions allowing definitions in a procedural language[CLiST89] only.

Our approach focuses on a sufficient abstract yet efficient interface, which permits to write implementations of constraints in Prolog, neglecting the actual representation of the constrained variables. Metastructures are applicable, but not restricted to the following areas:

- CLP-languages [JaLa87], equational theories[EmY87], new abstract data types consistent to unification, provided an unification algorithm exists, eg. associative-lists [Siek86].
- functional extensions, i.e. narrowing, term rewriting systems [BeLe86]

[*]This work was partially supported by the **Jubiläumsfonds** of the **Oesterreichische Nationalbank** under research contract Nr. 2791.

- control primitives e.g. `freeze/2` [vC86] and restricted forms of intelligent backtracking, support for negation [Nais86] and correct quantification for variables [Vas86]
- database support by set abstractions, and restricted forms of bottom up evaluation
- system programming
  + stream-based side effect free I/O-facilities, similar to committed choice languages
  + structure sharing in a structure copying system.
  + merging different representations for the same terms. In [KüNe87] character strings are an alternative representation for atoms within the same system. This extension is completely transparent to the user.

For the system implementor, metastructures have important advantages:

- Only small modifications are required for an existing Prolog implementation. The implementor needs only to concentrate on the efficient implementation of Prolog with metastructures. Many optimization schemes, e.g. tail-end recursion, may be reused. Application dependent problems (e.g. equation solving algorithms) are left over to the user.
- Metastructures make very few assumptions about an existing implementation. As a result a variety of possible implementations stands as an option.
- Due to the high efficiency and the small implementation effort, it seems probable that metastructures will be used within various Prolog systems. Constraint languages would be portable.
- Libraries of metastructures dedicated to specific problem domains can be provided.

In chapter 2 metastructures will be briefly presented and compared with the traditional approach to implement `freeze/2` and `dif/2`. In chapter 3 we will discuss metastructures from the (meta-) programming language point of view. Problems of consistent definitions will be discussed in chapter 5. Aspects of the implementation are covered in chapter 4. We will show that runtime overheads in a Prolog system enhanced by metastructures are insignificant. In addition the implementation effort to extend an existing Prolog machine will be shown to be considerably smaller than it is for a highly specialized constraint language.

2 Metastructures

A variable to be constrained needs an intermediate device to hold the description of the constraint. After solving or reducing the constraint, the intermediate device should be replaced by its "result". We introduce metastructures to serve this purpose. In the sequel a metastructure will be identified by the unique functor[1] `meta/`n, where $n > 0$. Although `meta/`n is not a conventional Prolog term, we will use this notation to describe a metastructure in detail. If we restrict for example a free variable `X` to be an even number, thus `X` $\in \{y|y \bmod\ 2 = 0\}$, we bind `X` as follows: `X = meta(Y,`"`Y` $\bmod 2 = 0$"`)`. In general, a metastructure consists of the following parts: `meta(`*value*`,`*property*$_1$`, ...,` *property*$_n$`)`

- *value-part*, represented by the first argument of `meta/`n. This is intended to represent later on a description of the constrained variable nearer or "narrowed" to the solution. The *value-part* determines the two possible states of a metastructure:
  + As long as the *value-part* is free, the metastructure serves to restrict of the unknown value. Such a metastructure is called *pending*.
  + If the *value-part* is bound to a non-variable term or to another metastructure, the metastructure has reached saturation and is then called *reduced*.

[1]In this paper we will follow this convention for sake of a simplified presentation. In Ch. 4.2 we describe our implementation in more detail allowing arbitrary names for metastructures.

`unify(T1,T2)`	Variable	Functor/Arity	Metastructure
Variable	$\surd$	$\surd$	$\surd$
Functor/Arity	$\surd$	$\surd$	`term_meta_unify/2`
Metastructure	$\surd$	`term_meta_unify/2`	`meta_meta_unify/2`

Abbildung 1: Unification table of Prolog terms and metastructures

Briefly, we term a metastructure as *pending* if the *value-part* is free; *reduced* if not free.

- *property-part*, represented by the other arguments (possibly empty). The arguments are arbitrary Prolog terms, which represent the meaning of the constraint.

At creation time the *value-part* is free. This means that the metastructure has no value yet and is hence *pending*. The whole metastructure with *property* and *value-part* is accessible. If the *value-part* becomes known, (i.e. if it is bound to a non-variable term or another metastructure), the *property-part* is no longer of importance. The reduced metastructure is replaced by the *value-part*. Thus the following equivalences are always true:

$$\texttt{meta(known\_value,}\mathit{property}_1\texttt{, ...)} \Longleftrightarrow \texttt{known\_value}$$
$$\texttt{meta(meta(V,}\mathit{property}_{11}\texttt{, ...),}\mathit{property}_{21}\texttt{, ...)} \Longleftrightarrow \texttt{meta(V,}\mathit{property}_{11}\texttt{, ...)}$$
$$\ldots$$

A metastructure is very similar to the *suspension*-data type as presented in [Carl87] however, its behavior during unification is slightly different.

In order to combine metastructures and ordinary Prolog terms, we extend unification as follows: The behavior of ordinary terms remains completely the same. When unified with a variable, metastructures behave in the same way as non-variable terms (see $\surd$'s in Fig. 2). All remaining cases where either or both of the terms are *pending*[2] need a more detailed treatment. The interface for the description of the desired behavior of metastructures comprises two user definable predicates—"methods":

- `term_meta_unify/2` for the unification with a non-variable term
- `meta_meta_unify/2` to unify two metastructures

Metastructures cannot be manipulated by unification directly. To prevent the call of the user definitions (the "evaluation"), we introduce the predicate

- `===/2` for syntactical unification.

It is equivalent to `=/2`, except for the cases of *pending* metastructures. `===/2` treats them like ordinary structures.

2.1 Implementation of `freeze/2`

Goal "`?- freeze(Variable,Goal).`" [vC86] delays execution of `Goal` until `Variable` is bound. We use metastructure `meta/2` to implement the suspension, refer to Fig. 2. The *property-part* (the second argument) contains the frozen goal. The predicate `freeze/2` unifies a new metastructure which contains the frozen goal and the variable responsible for suspension. The *value-part* of the new metastructure is a new variable (see `meta(_,...)`).

When such a suspension (i.e. a *pending* metastructure) is unified with a non-variable term, `term_meta_unify/2` is called. The frozen goal is fetched (see `===/2`), `Term` is unified with the *value-part*, `Term` is therefore the "result". In this manner, `meta/2` becomes *reduced* and the fetched `Goal` is finally called. When two suspensions are to be unified, `meta_meta_unify/2` is called, a new suspension which is the conjunction of both is constructed[3] and the two frozen

[2] *Reduced* metastructures have been already replaced by their *value-part*, or are "skipped".

[3] Attempts to unify a metastructure with itself are suppressed.

```
freeze(Variable,Goal) :-
    Variable = meta(_,Goal).        % create "suspension"

term_meta_unify(Term,Meta) :-       % "supspension" found
    Meta === meta(Term, Goal),      % fetch Goal, bind value-part
    Goal.                           % call frozen Goal

meta_meta_unify(Meta1,Meta2) :-
    Meta1 === meta(Var,Goal1),
    Meta2 === meta(Var,Goal2),
    Var === meta(_,(Goal1,Goal2)). % new "supension"
```

Abbildung 2: Implementation of `freeze/2` with metastructures

goals are reduced to the conjunction.

2.2 Metastructures vs. `delay/2`

The control primitive `delay/2` as defined in [Carl87] cannot be implemented with metastructures. As Carlsson states, a goal `g(Any)` is delayed by "`?- delay(X,g(Any)).`" *"until* `X` *is* **bound** *to any term, including a logical variable"*.[4] There are several reasons why such a predicate is undesirable:

- Semantically, the unification of a variable and any term being disjoint from it, whether the term be interpreted (constraint) or not (functor), must not fail. Even the unification of a free variable with another (free) variable may fail. By allowing to execute any goal upon unification of `X` with a free variable, a failure may consequently arise.
- The first action a goal delayed by `delay/2` will perform is to test in a loop whether it should be delayed again. Evidently, the delayed goal is called too often.
- Unification in ordinary Prolog-programs will be slowed down because, nearly every unification deals with variables. A test must be added for every comparison of variables. See chapter 4.1 for further details. Moreover, the event *"until* `X` *is bound"* is implementation dependent. `delay/2` may or may not detect all bindings of free variables.

It is apparent that Carlsson needs `delay/2` to implement `dif/2`. In chapter 2.3.2 we give another implementation of `dif/2` in terms of metastructures.

2.3 Implementation of `dif/2`

The predicate `dif/2`, already implemented in the first Prolog system *"Prolog 0"*[5], provides a sound implementation of disequality of terms. Our implementation of `dif/2` is different from other approaches [vC86,Boi88,Carl87].

2.3.1 Traditional implementation

Due to the symmetry of disunification only one variable needs to be constrained. If both arguments in "`?- dif(X,Y).`" are free, the constraint is added to one variable only:

```
dif(X,Y) :- reduce(X,Y,V), !, var(V), delay(V,dif(X,Y)).
dif(X,Y).
```

The predicate `reduce(X,Y,V)` realizes an iterator: It fails, if `X \= Y`. Itsucceeds otherwise and renders in `V` a variable of term `X` or `Y`, which then must be substituted in order to achieve unification. If `X == Y`, `V` holds an atom[6]. Any unification of `X` with another variable[7] will

[4][Carl87],§2.2, p.43

[5]See [vC86] for a historical overview

[6]as [vC86] we give no definition in Prolog. Under infinite trees such a definition is impossible see Ch. 5.

[7]Binding strategies with temporally ordered variables ([vC86], p. 91) may rule out several redundant calls.

cause a test for `dif/2`, whereas `Y` remains untouched. As long as `X` remains free, subsequent bindings of `Y` have no impact. Iterator `reduce/3` iterates over the two terms until the terms are definitely different[8].

2.3.2 Metastructure implementation

Metastructures are similar to non-variable terms during unification with free variables. This "inherited" property of metastructures prevents us from using the traditional approach. For "`?- dif(X,Y), X = Y.`" the unification `X = Y` cannot raise the execution of a related definition. Unfortunately, unification succeeds. Our implementation in Fig. 3 constrains *both* variables. The unification `X = Y` is detected, since both variables are bound to metastructures. The goal "`?- X = Y.`" entails "`?- meta_meta_unify(X,Y).`". No overheads are incurred for the unification of `X` or `Y` with a free variable.

The implementation in Fig. 3 uses an explicit passing of continuations to iterate over the two terms (argument 2 and 3 of `is_dif`). The quadratic cost for structured terms is avoided *a priori*. In this example, metastructures contain in their *property-part* a list of all other variables, i.e. metastructures from which they are to be different.

3 Language aspects

In this chapter aspects of programming metastructures are discussed. We focus on the meta-language to realize constraints rather than the usage of a constraint language.

The language for expressing how unification should be performed is only a small extension to the Prolog language. The built-in unification algorithm of Prolog is extended as presented previously. This new algorithm is used to perform unification in all programs. Modifications to unification are defined in Prolog by providing a new definition of the parts of ordinary unification.

To summarize, we have the following new elements which constitute the meta-language for metastructures: `term_meta_unify/2` and `meta_meta_unify/2` to control unification of metastructures, `===/2` to access the representation of metastructures via syntactic unification. This meta-language serves for the realization of constraints only and must not be visible to the user of constraints. The meta-language is very simple. It is the programmer defining unification who bears full responsibility for the correctness of his extensions. The definitions of metastructures cannot assert the equational rules. It is possible to break them (e.g. transitivity or symmetry; even for conventional terms) with an arbitrary definition of a metastructure For some guidelines refer to chapter 3.3.

The reason for the very primitive design lies in the various tasks metastructures should perform: On the one hand, we want to define meta-logical or extra-Prolog features as `freeze/2` whose semantics is defined operationally. On the other hand we want to define equational theories whose semantics is declaratively defined. As both of these tasks concern unification, it is desirable to have only one extension to a Prolog system to serve both. It should be evident to the reader that such diverging tasks result in only a very small compromise. The advantage of metastructures is that the compromise is settled at the level of Prolog and not at a lower level of a procedural language.

[8]In order to reduce the quadratic cost due to the search for a pair to be unified, the terms passed to the next examination may be simplified to the "frontier", i.e. the pairs of terms the result of `dif/2` depends on, as observes [Nais86], p. 9..

```
dif(X,Y) :-dif(X,Y,[],[]).

% dif/4: dif with additional continuations
dif(X,Y,Xs,Ys) :-
    X = meta(_,[is_dif(Y,Xs,Ys)]),
    Y = meta(_,[is_dif(X,Xs,Ys)]).

% reduce/2: search continuations for next possibly different pair
reduce([X|Xs],[Y|Ys]) :-
        var(X), var(Y), X == Y, !,
        reduce(Xs,Ys)               % pair identical, try next
    ;   (var(X); var(Y)), !,
        dif(X,Y,Xs,Ys)              % pair of "frontier"
    ;   X =.. [XF|XL], Y =.. [YF|YL],
        (   XF \= YF, !             % pair does not unify
        ;   append(XL,Xs,XN), append(YL,Ys,YN),
            reduce(XN,YN)           % functors identical, try next
        ).

term_meta_unify(T,M) :-
    M === meta(T,L),
    tm_dif(L,T).
meta_meta_unify(M1,M2) :-
    M1 === meta(V,L1), M2 === meta(V,L2),
    append(L1,L2,L3),
    mm_dif(L3,V),
    V = meta(_,L3).

tm_dif([],T).
tm_dif([is_dif(E,X,Y)|Ds],T) :-
    (var(E); reduce([T|X],[E|Y])),
    !,
    tm_dif(Ds,T).

mm_dif([],V).
mm_dif([is_dif(E,X,Y)|Ds],V) :-
    (nonvar(E); E \== V; reduce(X,Y)),
    !,
    mm_dif(Ds,V).
```

Abbildung 3: Implementation of `dif/2` with metastructures

3.1 Metastructures and object-oriented programming

From an object-oriented point of view, metastructures can be seen as a means to realize objects. In [KTMB86] various possible object-oriented extensions of logic programming languages are discussed. Concerning metastructures, the representation of an object is given by the metastructure's *property-part*. Just as a class defines a set of similar objects (i.e. a type), all definitions related to a metastructure define a new type of terms for Prolog. Messages are equivalent to predicates visible to the user which deal with metastructures, e.g. `write/1`. The corresponding methods (e.g. the extended implementation of `write/1`) access via `===/2` the internal representation. Also, `term_meta_unify/2` and `meta_meta_unify/2` can be seen as methods responding to the message of unification. Here, the relation of metastructures to a language like Vulcan is best expressed by [KTMB86]:

"A weakness of Vulcan ... is the inability to unify Vulcan objects. This could be dealt with by extending the underlying Prolog to send `equals` *messages when attempting to unify objects. This would not work in the*

standard concurrent logic programming languages, since messages cannot be sent from the guard."

In contrast, metastructures send `equals` messages. This is the point, at that, where the analogy to object-oriented programming does not hold convincingly: While message passing is unidirectional in object-oriented languages, unification works in both directions. Unification is more powerful than comparisons in a procedural language. Whereas an object's (temporal) state is a fundamental concept for object-oriented programming, equivalence is a fundamental concept for logic programming languages. It is interesting to observe the problems in object-oriented programming due to this lack of referential transparency. Specifically equality (procedurally: comparisons) of objects pose severe conceptual problems[KhoCo86]. Even if there is a large gap between the object-oriented and logic programming paradigm, both deal with abstract data types, each in its own manner.

The motivation of metastructures as a means to implement abstract data types should be obvious. The internal representation is hidden from the user (who uses ordinary unification). Only the behavior during unification and some operations (predicates) define the abstraction.

3.2 Hierarchies

We have intentionally excluded a hierarchical (dynamic) type system. One of the reasons is the conceptual mismatch incurred by hierarchies known form object-oriented techniques: As mostly only one hierarchy is provided, design decisions (i.e. designing conceptual hierarchies) must take into account implementation issues (e.g. using inheritance for *code sharing*) too. This area is still open to research. It is possible to support the dynamic part of a type system as [GoMe87,Smol88] with metastructures. At the level of metastructures this is considered as an application issue.

3.2.1 Modularization

The definition of metastructures can be reused in a Prolog system supporting modules where there is a *unique* relation between any functor and its predicate. Only a *name based* module system which supports modularization of terms is appropriate for combination with metastructures. Whereas, a *predicate based* system cannot preserve the unique relation (consider, for example, `freeze/2`).

3.3 Coding technique

Developing unification algorithms, even in Prolog, is a very difficult topic. Adopting an existing unification algorithm which uses a similar term representation as Prolog will only amount to a small programming effort. One should be aware of the following points, which help to avoid basic problems when starting programming. Some may be enforced by appropriate type checking and data flow-analysis.

- `term_meta_unify(T,M)` must either bind the metastructure `M` to the term `T` or fail. Otherwise the basic rules of equational reasoning would be broken.
- `meta_meta_unify/2` often needs to create a new metastructure replacing both old ones.
- `===/2` should be used with care. It is similar to CLU's **up** and **down** [LisGu86]: It allows you to access *representations.* Up to now we have used this predicate in the interfaces only and have not found any other meaningful application, i.e. in `term_meta_unify/2` and `meta_meta_unify/2` as well as for the redefinition of some "methods" e.g. `write/1` and `assert/1`.
- Typing has to be implemented by the user, no checks are made by the system.
- Side effects in the definition of unification should be avoided.

`unify(T1,T2)`	Variable	Functor/Arity	Constraint
Variable	‡	‡	†
Functor/Arity	‡	**[10]	†, *
Constraint	†	†, *	†, *

Abbildung 4: Unification overheads of `dif/2`: *,**) metastructures; †,‡) traditionally; one step tag decoding: †), *)

- Use shared variables within any metastructures with care. In the context of metastructures they can be abused like pointers in a procedural programming language. The user of a metastructure definition can get access to the representation. Special care must be taken for the *value-part*. However, it sometimes occurs, where different metastructures, being *friends*, share variables even in the *value-part* with one another.

Difficulties arise when several “types” have to be “coerced”. A lot of tedious and redundant code has to be written which is often much larger than the original definitions together. In fact, providing tools for such tasks is an interesting research topic. Many results could be reused from the field of automated deduction.

Formally, the notion of *conservative extensions* seems to be appropriate to describe the relation of correctly defined metastructures to ordinary Prolog terms[9]: metastructures, when applied correctly, do not change the meaning of ordinary terms. An ordinary term is still referentially transparent. In contrast, semantic unification lacks this property. Conventional terms are *reinterpreted.* The advantage of a conservative extension is that all the pre-existing sorts are completely independent of metastructures. This is not only true for the conceptual level but also for the (system-) implementation level as will be shown in the next chapter.

4 Implementation issues

4.1 Required modifications

Metastructures are an abstraction within the unification algorithm. They can be implemented with a variety of implementation techniques. In the sequel we will sketch techniques to be considered, if an existing Prolog system is to be modified. As metastructures are an extension local to unification, their impact on the design of the abstract machine will be rather small. For a WAM-implementation *get-* and *unify*-instructions are subject to extensions.

The only undesired overheads during unification appear, if metastructures are implemented close to ordinary structures. In such an implementation the overheads occur in the case where a structure is unified with a non-variable term (e.g. “`?- a = s(X).`”, see Fig. 4). If the attempt to unify these terms fails, an additional test is necessary.

During head-unification `term_meta_unify/2` or `meta_meta_unify/2` may be called. In most abstract machines such as the WAM or the abstract machine used in VIP-Prolog (Vienna Abstract Machine—VAM[Kral87]) the process of unifying goal and head (*get-* and *unify-*instructions) is considered to be an atomic action for efficiency reasons (i.e this process must not be interrupted by another inference). Goals backed up throughout an inference are to be executed thereafter. A straightforward implementation uses a register pointing to a list of pairs to be unified. Resetting the register and testing it after completion of an inference adds overheads far below 5% on conventional processors.

[9][EhrMa85]; “extensions and enrichment” Ch. 6.12

[10]Overheads in fail case only, e.g. `a = g(Z)`; but not `a = b`.

If the definition of a constraint is a completely deterministic Prolog program a compiler may replace the Prolog part by an appropriately compiled routine. It is thus possible to execute, e.g. `dif/2`'s, definition during the process of unifying goal and head, allowing earlier failure during the inference. The program's procedural behavior might be slightly different, as the early failure may prevent the execution of other constraints. The programmer must not make any assumptions about the order of unification, especially about the order of execution of additional goals. This feature is therefore implementation dependent.

4.2 Indexing different sorts of metastructures

In the presence of many different metastructures, *sorts*, indexing mechanisms improve the interface in retrieving appropriate definitions. In our implementation, any Prolog-structure has a uniquely associated metastructure. The search for appropriate rules is reduced to an indexed search over one functor for `term_meta_unify/2` and over two for `meta_meta_unify/2`. The effort is nearly independent from the number of existing *sorts*. However, as noted in chapter 3.2 the conceptual structurization within a type system still remains an open problem.

4.3 Built-in predicates

In a system supporting metastructures the I/O- and db-BIPs must detect *pending* metastructures. The decision of treatment depends largely on the meaning of the metastructure and should therefore be handled by the user. Raising and catching an appropriate exception allows the redefinition of these BIPs appropriately. There are essentially two ways to handle metastructures in this context:

- The metastructure represents a kind of abstract data type (e.g. associative lists). It is for the user a first class object like an ordinary Prolog term. In this case a syntactical representation should be displayed. Metastructures are asserted respectively.
- The metastructure is considered relevant to the actual computation only. It serves as a control primitive. The term will therefore be shown or asserted without the metastructure.

Constraints in the CLP-scheme are between these extremes. In [HeiMi89] a framework for dealing with such issues in CLP($\Re$) is presented.

4.4 Memory management, Garbage collection.

There are several detailed implementation descriptions of `dif/2` and `freeze/2` using an additional stack to manage frozen goals [vC86,Boi88]. As an additional stack increases both the size of choice points and the time required for backtracking, we believe that metastructures are represented best as ordinary structures.

During equation solving many intermediate results occur, each result being represented by a metastructure. Programs will now be more deterministic, as metastructures used as constraints help to avoid choice points. Already used metastructures, and in all likelyhood probably their *property-part* too, will have no significance for the ongoing computation[11]. The specific treatment during unification allows one to detect unused *reduced* metastructures without knowing the actual definitions of `term_meta_unify/2` and `meta_meta_unify/2`. *Reduced* metastructures can be removed except where there exists a choice point from which

[11]Consequently memory may be used up although traditional garbage collection is performed: *Reduced* metastructures may exhaust memory representing nothing at all. Even the order of computation time may be deteriorated. This phenomenon—called *space leaks*—is well known in lazy functional languages [Wad87].

```
unify(X,Y) :- (var(X); var(Y)), !, X = Y. % maybe occur-check
unify(X,Y) :- X =.. [F|Xs], Y =.. [F|Ys], unifylist(Xs,Ys).

unifylist([],[]).
unifylist([X|Xs],[Y|Ys]) :- unify(X,Y), unifylist(Xs,Ys).
```
Abbildung 5: Unification in Prolog with occur-check

the metastructure is reachable as *pending*. We will not go into details of the marking and reduction algorithm. Basically, reduced metastructures which will not become *pending* anymore are detected by means of *virtual backtracking* [Bru84,Pit85]. In contrast to Bruynooghe's algorithm, marking starts from the *oldest* choice point. Besides, this algorithm can eliminate redundant reference chains.

4.5 Reducing constraints

In [vH89], p.103 a *value trail* to overwrite the representation of constraints is used. In our experience it is preferable to reduce metastructures to *ordinary terms* by simply binding the *value-part* and applying a garbage collector later on. However, there are many classes of constraints which can be represented more efficiently than the way we have presented. Consider finite integer domains. Obviously a definition similar to `greater/2` in Fig. 6 can be used. However, in many cases this implementation is not very efficient. If we want to constrain a variable `X` to be an interval of integers, `X` may be specified as follows: "`..., Min < X, X < Max, ...`" Subsequent exclusions of elements (by `dif(X,E1)`) will lead to more complicated representations.

$$S_0 = \{x|min < x \wedge x < max\}$$
$$S_1 = \{x|min < x \wedge x < max\} - \{x|x = e_1\}$$
$$S_1 = \{x|min < x \wedge x < max \wedge (x > e_1 \vee x < e_1)\}$$

This process is very space consuming. Another choice is to enumerate explicitly the set of possible values. Subsequent exclusions reduce the set. Such operations are more efficiently implemented with destructive updates than by respecting referential transparency of Prolog data structures; i.e. creating modified copies. Such a representation is chosen in CHIP [vH89], p. 104. It is evident that such operations cannot (and must not) be supported for metastructures defined at the meta-language level directly. Yet metastructures can at least serve as the interface between the Prolog engine and low-level implementations, avoiding the redesign the kernel's abstract machine and garbage collector. The Prolog interface can then be used for rapid prototyping of new extensions.

5 Open problems

Nearly all Prolog implementations realize unification without occur-check. It is therefore possible to create infinite terms—*infinite trees*. While there is a consistent framework for *infinite trees*[Col82], they pose some problems for the design of new unification algorithms in general and for metastructures in particular. However, infinite data structures pose a problem *only* for the definition of metastructures which contain arbitrary subterms. As long as metastructures are *closed* they can be used consistently even in a system with infinite trees.

One uses the procedure in Fig. 5 as a skeleton for implementing new unification algorithms. In the case of a system with infinite trees there is no corresponding skeleton.

Furthermore the definition above will end up looping, if an infinite data structure is to be processed. It is consequently much more tedious to define consistent extensions to unification in a system without occur-check. By the way, this is the reason why no definition of `reduce/3` [vC86] is given in Prolog.

On the other hand an occur-check for ordinary terms *and* metastructures restricts the expressiveness of metastructures considerably and it would not be possible, to implement `freeze/2` anymore: For a variable attached to a frozen goal and occurring in the same goal, our definition in chapter 2.1 creates an infinite metastructure, since `X` is bound to `meta(V,f(X))` in "`?- freeze(X,f(X)).`". Anyway, with our definition of `freeze/2`, the user would never realize that infinite data structures are created, because the definition of `freeze/2` prevents falling into an endless loop. Also in CLP($\Re$), constraints corresponding to infinite data structures have a definite meaning. Given `f/1` as an interpreted functor, "`?- X = f(X).`" denotes the function's fix points.

In Fig. 6 an implementation of the well known successor functor based on metastructures is shown. It allows to mix (coerce) the successor functor with integer numbers. Given our implementation, we can create the infinite metastructure `X = meta(_,X)` by the goal "`?- X = s(X).`". The unification of an arbitrary variable with a metastructure always succeeds. Hence the creation of the infinite term is not prevented. A *fix* might be made by putting another metastructure into the argument. The successor functor `s(X)` could be represented by `meta_s(_,meta_check(_,X))`. We do not know whether such a fix can be made in general. However we believe that the current definition of metastructures needs no redefinition. In summary, our experiences with metastructures and occur-check are as follows:

- Infinite metastructures like `X = meta(X,...)` are to be definitely excluded.
- *Flat* metastructures containing no subterms pose no problems, even in a system with infinite trees.
- Metastructures implementing an abstract data type which has no or a very restricted connection to arbitrary terms, may consistently contain for themselves arbitrary structures, even infinite trees, or an explicit occur-check implemented in Prolog.
- Reasoning about ordinary terms (e.g. `dif/2`) is unsafe without occur-check.

6 Conclusion

Metastructures provide a simple yet efficient interface to extend unification by the user. The main advantages of our approach in comparison to other logic programming languages are: Unification is extensible by user defined Prolog predicates without changing the Prolog kernel. The extensions may be used for implementing new equational theories as well as for meta-logical programming. Ordinary Prolog programs are executed with a minimal overhead. The extension is easy to implement and may be well supported by the Prolog system.

Acknowledgements

I am very much indebted to eva Kühn for her fruitful guidance in project Nr. 2791 supported by the **Oesterreichische Nationalbank**. Metastructures originate from efficient low-level extensions [KüNe87] of unification which where developed in this project [Neume88]. Gustaf Neumann made many comments. Finally, I would like to thank Professor Manfred Brockhaus, who has always encouraged our work and the anonymous referees.

Literatur

[BeLe86] Bellia, M. & Levi, G. 'The Relation between Logic and Functional Languages: A Survey', *JLP 1986:3:217-236*, (1986).

[Bl87] Blair, H.A., 'Canonical Conservative Extensions of Logic Program Completions', *Proc. 4th IEEE Symp. Logic Programm.*, 154-161, (1987).

[Boi88] Boizumault, P., *PROLOG l'implantation*, Masson, Paris, (1988).

[Bru84] Bruynooghe, Maurice, 'Garbage Collection in Prolog Interpreters', *Implementations of Prolog,* Campbell (ed), Ellis Horwood, 259-267, (1984).

[Carl87] Carlsson, M., 'Freeze, Indexing and Other Implementation Issues in the WAM', *Proc. 4th Int. Conf. on Logic Programm., Melbourne*, Lassez, J.-L. (ed.), MIT Press, (1987).

[CLiST89] Crossley, J.N. & Lim, P. & Stuckey, P. 'Interface logic programming', *The Australian Computer Journal*, 21(2), (MAY 1989).

[Col87] Colmerauer, A., 'Opening the Prolog-III Universe', *BYTE*, 12(9), (AUGUST 1987).

[Col82] Colmerauer, A., 'Prolog and Infinite Trees', *Logic Programming*, Clark, W.K. & Tarnlund, S.A., (eds.), Academic Press, (1982).

[DL86] DeGroot, D. & Lindstrom, G., *Logic Programming, Functions, Relations, and Equations*, Prentice-Hall (1986).

[EhrMa85] Ehrig, H. & Mahr, B., *Fundamentals of Algebraic Specification 1*, Springer-Verlag, (1985).

[EmY87] Emden, M.H. & Yokawa, K., 'Logic Programming with Equations', *JLP*, 1987:4:265-288, (1987).

[GoMe87] Goguen, J.A. & Meseguer, J., 'Models and Equality for Logical Programming', *TAPSOFT '87* Vol.2, LNCS 250, 1-22, (MARCH 1987).

[HeiMi89] Heintze, N. & Michaylov, S., 'On Meta-Programming in CLP($\Re$)', *Proc. Logic Programm., North Americ. Conf.*, Lusk, E.L. & Overbeek, R.A. (ed.), MIT Press, (1989).

[Holz90] Holzbaur, Ch. 'Metastructures as a Basis for the Implementation of Constraint Logic Programming Techniques',*TR-90-2*, Austrian Research Institute for Artificial Intelligence, Vienna, (1990).

[JaLa87] Jaffar, J. & Lassez, J.-L., 'Constraint Logic Programming', *14th ACM POPL Symp.*, (JANUARY 1987).

[Ko84] Kornfeld, W.A., 'Equality for Prolog', *Proc. 8th IJCAI*, Karlsruhe, (AUGUST 1983); also in [DL86].

[KhoCo86] Khoshafian, S.N. & Copeland, G. P, 'Object Identity' *OOPSLA 86*, 406-416, (1986).

[Kral87] Krall, A., 'Implementation of a High-Speed Prolog Interpreter', *ACM SIGPLAN NOTICES, Conf. Interpr. and Interpretive Techn.*, 7(7), (JULY 1987).

[KüNe87] Kühn, eva & Neumerkel, U., 'A Freezy Way of Coupling the Prolog Universe with DBS', *VIP-TR 1802/87/2*, also [Neume88], TU-Wien, (1987).

[LisGu86] Liskov, B. & Guttag, J., *Abstraction and Specification in Program Development*, MIT-Press, (1986).

[Nais86] Naish, L., *Negation and Control in Prolog*, LNCS 238, (1986).

[Neuma88] Neumann, Gustaf, *Metaprogrammierung und Prolog*, Addison-Wesley, (1988).

[Neume88] Neumerkel, U., 'Metastrukturen in Prolog', *Abschlußbericht des Jubiläumsfondsprojektes Nr.2791 der Oesterr. Nationalbank*, (1988); also *VIP TR 1802/88/2-4*, TU-Wien.

[Pit85] Pittomvills, E., Bruynooghe, M., & Willems, Y.D. 'Towards a Real Time Garbage Collector for Prolog', *IEEE 1985 Symp. on Logic Programm.*, 185-198, (1985).

[Siek86] Siekmann, J., 'Unification Theory', *Proc. ECAI 86.*

[Smol88] Smolka, G., 'Logic Programming with Polymorphically Order-Sorted Types', *Proc. 1st PROTOS Workshop*, Appelrath, H.-J. et al. (eds.), Morcote, (1989).

[Vas86] Vasak, T., 'Universal Closure Operator for Prolog', *SIGPLAN Notices*, 21(6),61-62, (JUNE 1986).

[KTMB86] Kahn, K. & Tribble, E.D., & Miller, M. S. & Bobrow, D.G., 'Objects in Concurrent Programming Languages', *OOPSLA '86*, (1986).

[Wad87] Wadler, Ph., 'Fixing Some Space Leaks with a Garbage Collector', *Software-Practice and Experience*, 17(9), 595-608 (SEPTEMBER 1987).

[vC86] Caneghem, Michel van, *L'Anatomie de Prolog*, InterÉditions, Paris, (1986).

[vH89] Hentenryck, P. van, *Constraint Satisfaction in Logic Programming*, MIT-Press, Cambridge, (1989).

```
% successor-notation
% replace occurences of s(X) by meta(_,X)

term_meta_unify(T,M) :-
    int(T), T > 0, M === meta(V,P),
    S is T - 1,
    S = P.

meta_meta_unify(M1,M2) :-
    M1 === meta(V,P1),
    M2 === meta(V,P2),
    P1 = P2.
```

Abbildung 6: Example of coercion: `s(0) = 1`

```
Gr greater Sm :-
    Gr \== Sm,
    Gr = meta(_,[Sm],[]),
    Sm = meta(_,[],[Gr]).

term_meta_unify(T,M) :-
    M === meta(T,Ss,Gs),
    not ( get_member(smaller,S,Ss), nonvar(S), S >= T ),
    not ( get_member(greater,G,Gs), nonvar(G), G =< T ).

meta_meta_unify(M1,M2) :-
    M1 === meta(V,S1s,G1s), M2 === meta(V,S2s,G2s),
    consistent(S1s,G2s), consistent(S2s,G1s),
    append(S1s,S2s,S3s), append(G1s,G2s,G3s),
    V === meta(_,S3s,G3s).

consistent(Ss,Gs) :-
    not (
            get_member(smaller,S,Ss),
            get_member(greater,G,Gs),
            (    nonvar(S), nonvar(G), G =< S
            ;    S == G
            )
        ).

get_member(C,E,Es) :-
    member(T,Es),
    (   E === T
    ;   var(T),
        ismeta(T),
        get_meta(C,T,Ts),
        get_member(C,E,Ts)
    ).

get_meta(smaller,T,Ts) :- T === meta(_,Ts,_).
get_meta(greater,T,Ts) :- T === meta(_,_,Ts).
```

Abbildung 7: Comparing natural numbers