Integration of Extended Unification into a Prolog Interpreter

Christian Holzbaur
(email: christian@ai-vie.uucp)

Department of Medical Cybernetics and AI
University of Vienna, and
Austrian Research Institute for Artificial Intelligence
Freyung 6, A-1010 Vienna, Austria *

TR-90-09

Abstract

This paper reports on the practical experiences collected during the implementation of constraint logic programming techniques via metaterms and extended unification. During the actual implementation of extended unification in a concrete PROLOG implementation some minor refinements to a specification proposed earlier [3] had to be taken care of. Further, a convention related to the interpretation of the meaning of metaterms has been moved into the specification. This paper is also thought as a supplement to the `C-Prolog` manual [7], describing the additional predicates and conventions for the implementation of extended unification.

1 Introduction

The incorporation of semantic unification into PROLOG was inspired by the work of Kornfeld [5]. His early paper provoked some clarifications in [2]. In Kornfeld's outline a *failed syntactic* unification was patched by trying to prove the equality of the two terms in the extended theory. This gave rise to some problems which can be avoided altogether through the introduction of a new data type which receives special treatment during unification. This special treatment is user-defined. The user provides the semantic theory in form of PROLOG predicates. The approach is completely general, i.e., the semantic theory might be nondeterministic[1] or even refer to the extended theory itself. The implementation (fig. 1) of `freeze/2` [1] requires this generality. Although the approach allows the

*Thanks for feedback are due to Igor Mozetic who uses this code in the course he teaches

[1]interesting in commutative, associative theories for example

```
:- meta_functor($frozen/2).

metatermunify($frozen(Value,Goals),Value) :- call(Goals).

metametaunify($frozen(V,G1),$frozen(V,G2)) :-
  V = $frozen(_,(G1,G2)).

freeze($frozen(_,G),G).
```

Figure 1: Implementation of `freeze/2`

specification of such control related matters, it seems to the author, that their presence indicates the lack of a sound semantic theory in most cases. Delay mechanisms stretch the operationality only practically, in theory nothing is gained. The aim of this paper is *not* to give implementations of extended unification theories, but to present a common basis for their realization. The term 'theory' is used rather sloppy in this context – some unification extensions don't deserve this naming. Purists should read 'method' instead of 'theory' in the sequel.
Section 2 specifies the incorporation of extended unification through a metainterpreter. Later we show how this specification can be implemented directly. In section 3 some empiric evidence is given for the claim that the extensions incur virtually no overhead to programs that do not use them. Section 4 describes the predicates and conventions that are needed to implement a particular extended unification theory from the users point of view.

2 Implementation of extended unification

We present the incorporation of extended unification through a PROLOG metainterpreter in order to facilitate the transparency and the portability of the approach. Note that the following specification does not describe any *particular* extended unification theory. It gives an account of how such extensions can be incorporated, i.e., made operational.

2.1 Specification

The realization of extended unification is outlined through a PROLOG metainterpreter that makes the unification explicit. Although this is an elegant and precise means [8], it should be stressed that the actual implementation does not use this metainterpreter as such. In general we can get rid of the additional level of interpretation that is introduced by a metainterpreter through partial evaluation [6]. In this particular case we would end up with copies of the extended unification algorithm in every predicate. Even if we limit the depth up to which the extended unification algorithm is unfolded, the increase in code-size is unbearable. Therefore we propose a realization of the specification that is more

direct. As the interesting part of the operation of the metainterpreter is limited to the unification, we can expect a certain locality of necessary changes that have to be applied to the kernel of a PROLOG implementation, to achieve the desired behaviour directly. On this implementation level the incorporation of extended unification is quite similar to the implementation of `delay/2`[2]. Therefore many techniques presented in [1] can be applied more or less directly later when we get rid of the metainterpreter.
A simple metainterpreter that makes the unification explicit is given in figure 2. It handles just pure PROLOG. Control primitives as cut and if-then-else and metalogical predicates as `setof/3` are out of its scope. As the incorporation of extended unification does not need any changes in the control structure, we can inherit the corresponding functionality from PROLOG. Lines 3 to 7 in figure 2 take care of the interesting part of the computation. If a goal G is to be resolved with a predicate, the goal and the head of a clause of the matching predicate have to be unified. Matching predicate candidates are found by hashing based on the functor of the goal and its arity. The predicate `headunification/5` takes explicit care of the unification of the actual arguments of G with the formal arguments of candidate clauses. If the unification succeeds, the interpreter proceeds with the body of the selected clause of the selected predicate recursively (line 7 in figure 2). In order to prevent infinite regress to the extended unification theory during the implementation of the extended unification theory, one needs to refer to the syntactic unification of the host PROLOG implementation. This could be done through special built-in predicates. It turned out to be more convenient to have declarations that lead to syntactic headunification for certain predicates. The set of all predicates is split into three categories:

category 0: Extended, semantic headunification. This is the default.

category 1: Syntactic headunification. This is typically used for the implementation of the extended unification theory that has to refer to the components of metaterms.

category 2: Symmetric syntactic headunification. This is also used for the implementation of the extended unification theory. If more than one sort of metaterms is used to implement the extended theory, one has to specify how all possible pairs of them are to be unified via the user-defined predicate `metametaunify/2`. If a predicate is defined to be symmetric, the number of defining clauses is cut into half.

The predicate `headunification_category/2` returns the unification category for any functor. The predicate `headunification/5` in figure 3 dispatches on the unification category. Lines 1 to 4 treat extended unification. A most general copy of the goal G[3] is used to look for candidate clauses via `clause/2`. The most general copy contains just free variables as arguments. Therefore no significant unifications take place — we take care of them in line 4 of figure 3 by a call to `metaunify/2`. Syntactic headunifications are done directly by `clause/2` in lines 6 and 7 of figure 3. Symmetric, syntactic headunification is defined for predicates of arity two only. It might succeed in lines 9 and 10 or with switched arguments

[2]a primitive function in [1] to implement `freeze/2`

[3]the most general copy of `member(2,[2,3])` is: `member(_,_)`

```
solve(true)  :- !.
solve((A,B)) :- !, solve(A), solve(B).
solve(G) :-
  functor(G,Name,Arity),
  headunification_category(G,Cat),
  headunification(Cat,G,Name,Arity,Body),       % Fig. 3
  solve(Body).
```

Figure 2: Metainterpreter

```
 1 headunification(0,G,N,A,Body) :- % semantic
 2   functor(Gn,N,A),
 3   clause(Gn,Body),
 4   metaunify(Gn,G).              % Fig. 5
 5
 6 headunification(1,G,_,_,Body) :- % syntactic
 7   clause(G,Body).
 8
 9 headunification(2,G,N,2,Body) :- % syntactic , symmetric
10   clause(G,Body).
11 headunification(2,G,N,2,Body) :-
12   functor(Gn,N,2),
13   arg(1,G,A1), arg(2,Gn,A1),
14   arg(2,G,A2), arg(1,Gn,A2),
15   clause(Gn,Body).
```

Figure 3: Headunification

in lines 11 to 15 in figure 3. The specification for the syntactic, symmetric case is just an approximation. In the actual implementation there is a kind of `cut` after a successful headunification. We *don't* get multiple solutions because of symmetric headunification. This could be made explicit in the specification by using the definition of `clause/2`. Extended unification is oriented along the types of objects that are subject to the unification. Before we go into the details of how to unify each pair of these types we have to make a little deviation that is made necessary by the way how many extended theories work:

> A logical (PROLOG) variable gets assigned a value once and only once. There is no way to change this value as in imperative languages. Many extended theories *conceptually change* the value of variables *after* they got a first value. As an example take a theory called 'forward checking' [4] where a variable has an associated domain which is narrowed subsequently. This narrowing is reflected by 'reassignments' of the new domains.

As our extended theories are implemented in PROLOG, there is no way to actually reassign values to logical variables. The escape from this problem is the concept of 'open' datastructures. Instead of representing the value of a variable directly, we package it into a structure. The special property of this structure is that it can be extended, i.e., updated with subsequent values. A familiar example of this technique are open tailed lists [8]. Such lists can be extended indefinitely by appending any number of open tailed lists. If we assume the convention that the last element of such a list represents its 'value' we have a mechanism for nondestructive (re)assignments. Of course, the datastructure does not have to be a list — any structure that is extended by a value *and* an unbound variable for subsequent updates works. The process of scanning such a datastructure to find its 'value' is called 'dereferencing'.

In our extended unification theories we often need this functionality. Therefore there are two types of metafunctors: derefable metafunctors and other metafunctors. The first argument[4] of a derefable metafunctor might be a derefable metafunctor — we proceed the scan for its value recursively in analogy to the open tailed lists. If the first argument is an unbound variable the value of the metaterm is undetermined. Any other object in the first argument position of a derefable metafunctor represents the value of this metaterm. As a result of this coding scheme a very frequent operation in extended theories is dereferencing such chains of metaterms, as all operations usually apply to the current value of an object, which is found at the end of the chain.

Dereferencing of metaterms could be left to be coded by the author of the extended unification theory, but the pattern is so frequent that dereferencing has been made part of the specification and the kernel.

The predicate `meta_deref/4` in figure 4 lines 1 to 5 takes any object as its first argument and returns its *type*, the *last element* of the chain of metafunctors and the *first argument* of the last derefable metafunctor in the chain. There are five basic types of objects that can result when an arbitrary object is dereferenced:

type 1: There was a nonempty chain of derefable metafunctors. The last metafunctor in the chain has its first argument bound to something other than a metafunctor. We call such a metafunctor *instantiated*. Example:[5]

```
| ?- X=meta(const), meta_deref(X,T,D,A).

X = meta(const)
T = 1
D = meta(const)
A = const
```

[4]this is just another convention — it could be any argument

[5]assume that `meta/1` is defined as a derefable metafunctor and `+/2` as a non-derefable metafunctor

type 2: There was a nonempty chain of derefable metafunctors. The last metafunctor in the chain has its first argument unbound. The 'value' of the metafunctor is not yet completely determined. Example:

```
| ?- X=meta(Y), Y=meta(Z), meta_deref(X,T,D,A).

X = meta(meta(_9))
Y = meta(_9)
Z = _9
T = 2
D = meta(_9)
A = _9
```

type 3: The first argument to `meta_deref/4` was an unbound variable. An unbound variable has of course no first argument. Therefore the third and fourth argument are equal to the first argument. Example:

```
| ?- meta_deref(X,T,D,A).

X = _0
T = 3
D = _0
A = _0
```

type 4: The first argument to `meta_deref/4` was no metafunctor and no variable. As in the previous case the third and the fourth argument are equal to the first argument. Example:

```
| ?- meta_deref(const,T,D,A).

T = 4
D = const
A = const
```

type 5: After skipping an arbitrary number (including zero) of derefable metafunctors a non-derefable metafunctor was encountered. Example:

```
| ?- X=meta(Y), Y=meta(Z), Z=I+K, meta_deref(X,T,D,A).

X = meta(meta(_17+_18))
Y = meta(_17+_18)
Z = _17+_18
I = _17
```

```
 1 meta_deref(Term, Type, Last, Arg) :-
 2   meta_deref(Term, Term, T, L, A),
 3   Type = T,
 4   Last = L,
 5   Arg = A.
 6
 7 meta_deref(Last, Arg, 3, Last, Arg) :- var(Arg), !.
 8 meta_deref(A0, A1, T, Last, Arg) :-
 9   derefable_meta(A1), !,                            % builtin
10   arg(1,A1,A2),
11   meta_deref(A1,A2,Ta,Last,Arg),
12   map_type(Ta,T).
13 meta_deref(Last, Arg, 5, Last, Arg) :- ismeta(Arg), !.  % builtin
14 meta_deref(Last, Arg, 4, Last, Arg).
15
16 map_type(1,1).
17 map_type(2,2).
18 map_type(3,2).
19 map_type(4,1).
20 map_type(5,5).
```

Figure 4: `meta_deref/4`

```
K = _18
T = 5
D = meta(_17+_18)
A = _17+_18
```

Lines 2 to 5 in figure 4 ensure the appropriate flowpattern for `meta_deref/5`. The builtin predicates `derefable_meta/1` and `ismeta/1` succeed if the functor with which they are called has the corresponding attribute. Lines 7, 13 and 14 in figure 4 treat the base-cases. Skipping of metafunctor chains takes place in lines 8 to 12. As the recursive branch shares the code of the base-cases, the types returned from them have to be fixed in order to reflect the situation that a nonempty chain of derefable metafunctors has been skipped.
Now we are in the position to specify how each pair of objects of certain types should be unified. The predicate `metaunify/2` (lines 1 to 5 in figure 5) determines the types of its arguments via `meta_deref/4` and hands them together with the dereferenced arguments to `metaunify/6` iff they are not equal already (line 4 in figure 5). The given realization of the equality test via `==/2` is just an approximation. In the actual implementation only pointer equality is checked. If one of the two arguments to `metaunify/2` was an unbound variable (type 3), this argument is bound to the other argument (lines 7 and 8 in figure 5). Lines 10, 11, 20 and 21 deal with the unification of 'constants', i.e., non-metaterms. The predicate `metaunify_rec/2` in figure 6 recursively applies `metaunify/2` to the corresponding

```
 1 metaunify(X,Y) :-
 2   meta_deref(X,Tx,Dx,Ax),       % Fig. 4
 3   meta_deref(Y,Ty,Dy,Ay),
 4   (Ax == Ay ->
 5     true ; metaunify(Tx,Ty,Dx,Dy,Ax,Ay) ).
 6
 7 metaunify(3,_, Dx, Dy,  _,  _) :- Dx = Dy.
 8 metaunify(_,3, Dx, Dy,  _,  _) :- Dx = Dy.
 9
10 metaunify(1,1,  _,  _, Ax, Ay) :- metaunify_rec(Ax,Ay).         % Fig. 6
11 metaunify(1,4,  _,  _, Ax, Ay) :- metaunify_rec(Ax,Ay).
12 metaunify(1,2,  _, Dy, Ax,  _) :- metatermunify(Dy,Ax).
13 metaunify(1,5,  _, Dy, Ax,  _) :- metatermunify(Dy,Ax).
14
15 metaunify(2,1, Dx,  _,  _, Ay) :- metatermunify(Dx,Ay).
16 metaunify(2,4, Dx,  _,  _, Ay) :- metatermunify(Dx,Ay).
17 metaunify(2,2, Dx, Dy, Ax, Ay) :- metametaunify(Dx,Dy).
18 metaunify(2,5, Dx, Dy,  _,  _) :- metametaunify(Dx,Dy).
19
20 metaunify(4,1,  _,  _, Ax, Ay) :- metaunify_rec(Ax,Ay).         % Fig. 6
21 metaunify(4,4,  _,  _, Ax, Ay) :- metaunify_rec(Ax,Ay).
22 metaunify(4,2,  _, Dy, Ax,  _) :- metatermunify(Dy,Ax).
23 metaunify(4,5,  _, Dy, Ax,  _) :- metatermunify(Dy,Ax).
24
25 metaunify(5,1, Dx,  _,  _, Ay) :- metatermunify(Dx,Ay).
26 metaunify(5,4, Dx,  _,  _, Ay) :- metatermunify(Dx,Ay).
27 metaunify(5,2, Dx, Dy,  _,  _) :- metametaunify(Dx,Dy).
28 metaunify(5,5, Dx, Dy,  _,  _) :- metametaunify(Dx,Dy).
```

Figure 5: Extended unification

arguments of two structures, given that the name and arities of the functors match. The clauses in lines 12-16 and 22-26 in figure 5 unify metaterms with ordinary terms through calls to `metatermunify/2`, a user-defined predicate that implements one part of the extended unification theory. The second part of the definition of the extended theory is called from the remaining clauses in figure 5.

```
metaunify_rec(X,Y) :-
  functor(X,N,A),
  functor(Y,N,A),
  metaunify_args(0,A,X,Y).

metaunify_args(N,N,_,_) :- !.
metaunify_args(N,A,X,Y) :-
  N1 is N+1,
  arg(N1,X,Xa),
  arg(N1,Y,Ya),
  metaunify(Xa,Ya),
  metaunify_args(N1,A,X,Y).
```

Figure 6: Extended unification (continued)

2.2 Realization

As mentioned above, the specification in the form of a metainterpreter cannot be used directly for space and efficiency reasons. This section deals with the necessary modifications in a PROLOG implementation that allow the *direct* implementation of the specification. We applied the necessary changes to a widespread PROLOG interpreter: `C-Prolog` [7] — the resulting extended interpreter is called `MetaProlog`.

2.2.1 Unification

Metastructures are PROLOG terms with an arity > 0, quite similar to `susp/2` in [1]. The standard unification algorithm has to be modified in order to take the new data type 'metafunctor' into account. If the host PROLOG system does its type dispatching based on tags, we can provide metafunctors with a special tag. In systems that do type recognition by signed pointer comparisons we need an additional test in the branches that treat structures. Terms are tagged at their creation time — during `read/1` and through explicit calls to `functor/3` and `=../2`. The following table sketches the implementation of the extended unification algorithm:

unify(X,Y)	constant	variable	functor	metafunctor
constant	compare	bind	fail	metatermunify/2
variable	bind	bind	bind	bind
functor	fail	bind	compare, recurse	metatermunify/2
metafunctor	metatermunify/2	bind	metatermunify/2	metametaunify/2

Once the unification algorithm 'knows' that at least one of its two arguments is a metaterm, it conceptually calls one of the user-defined handlers `metametaunify/2` or `metatermunify/2`. Of course, no implementor would like to spoil his high-performance unification algorithm by suspending it and by calling a user-defined predicate with the remaining unification as continuation[6]. If we consider *syntactic* unification as an atomic operation and just collect calls to metaunifications and proceed for the moment as if they would succeed, the unification algorithm remains deterministic[7].
As in [1], we need a new machine register W to hold the pending metaunifications. The arguments to the pending unifications are stored on the heap. If the W register already contains a pending unification, we allocate a conjunction over pending unifications on the heap and assign it to W in turn. The W register needs *not* to be preserved across failures, i.e., the size of the choicepoint data structure remains the same. A typical fragment from the C code that implements unification is shown in figure 7.

[6]pending computation

[7]In the case of *deterministic* unification extensions it is possible to execute the extensions *inside* the unification algorithm

```
if ( IsAtomic(b) ) {
  if ( IsMeta(SkelP(a)->
    push_metaunify(pa,b)
    continue;
   }
   goto fail;
}
```

Figure 7: Fragment of unification code

2.2.2 Execution of pending metaunifications

The contents of the W register is executed at each inference step by calling the appropriate user-defined handlers — in our case `metametaunify/2` and/or `metatermunify/2`. On systems with an event handling mechanism the check for a nonempty W register imposes no additional overhead [1]. To wait with the semantic unifications until after success of the syntactic unification not only makes the implementation of nondeterministic unification possible, but can also lead to the detection of failure during syntactic unification, making semantic unification(s) superfluous. Further, variables inside metaterms might get bound by syntactic unification, providing *more* information to the semantic unification which is to follow.
As metaunifications are just *collected* during syntactic unification, they might mutate into ordinary unifications through later bindings. Example (assume that `meta/1` is a derefable metaterm):

```
f( 1, 2) = f( meta(X), X).
```

The unification of the number 1 with `meta(X)` is collected as a metaunification to be executed at the next inference step. The unification between the number 2 and `X` turns `meta(X)` into an instantiated metaterm. This has to lead to a unification between the numbers 1 and 2 — according to the specification in figure 5. In analogy, `metametaunify/2` calls can mutate into `metatermunify/2` calls. Therefore the main computation takes place when the pending metaunifications are executed based on the specification in figure 5 and not during collection. A fragment of this code is shown in figure 8. Now a further reason for making dereferencing a part of the specification becomes evident: The realization of the specification can be moved into the low level implementation, where it can be implemented more efficiently as opposed to the case when dereferencing is an explicit part of the extended unification theory. The C-code counterpart to `meta_deref/4` from figure 4 is in figure 9. It is reproduced for the interested reader who is familiar with C. The weak correspondence between the PROLOG specification and the C-code should be obvious. A detailed description of the C-code would require a lengthy interpretation of the internal `C-Prolog` datastructures.

```
t1 = meta_deref(a,x1,Addr(SkelP(g)->Arg1), &a,&ag,&ap, &aa,&aag,&aap);
t2 = meta_deref(b,x1,Addr(SkelP(g)->Arg2), &b,&bg,&bp, &ba,&bag,&bap);

switch (t1) {
  case 1: switch (t2) {
    case 1:
    case 4:
      TRY(unify(aa,x,aag,ba,x,bag));
    case 2:
    case 5:
      env = v1; GrowGlobal(2);
      *env = bp;
      *(env+1) = IsAtomic(aa) ? aa : aap;
      ConsMol(Addr(FunctorP(MetaTerm)->gtoffe),env,pg);
      goto icall;
  } break;
```

Figure 8: Fragment of code setting up a call to `metatermunify/2`

```
int
meta_deref(t,tg,tp, lt,ltg,ltp, la,lag,lap)
register PTR t;
PTR tg,tp, *lt,*ltg,*ltp,*la,*lag,*lap;
{ register PTR a; PTR ag, ap;
  int depth = 0;

  t = *tp;
  if (IsVar(t)) {
    tp = FrameVar(IsLocalVar(t) ? x : tg, VarNo(t));
    deref_macro(t,tp);
    if ( Undef(t) ) {
      t = tp;
    }
    else if ( IsComp(t) ) {
      tg = MolP(tp)->Env;
      t = MolP(tp)->Sk;
    }
  }
  a = t; ag = tg; ap = tp;
  while ( IsComp(a) && !Undef(*a) &&
          IsMeta(SkelP(a)->Fn) && !NoDeref(SkelP(a)->Fn) ) {
    depth++;
    t = a; tg = ag; tp = ap;
    ap = Addr(SkelP(a)->Arg1); a = *ap;
    if (IsVar(a)) {
      ap = FrameVar(IsLocalVar(a) ? x : ag, VarNo(a));
      deref_macro(a,ap);
      if ( Undef(a) ) {
        a = ap;
        break;
      }
      if ( IsComp(a) ) {
        ag = MolP(ap)->Env;
        a = MolP(ap)->Sk;
      }
    }
  }
  *lt = t; *ltg = tg; *ltp = tp;
  *la = a; *lag = ag; *lap = ap;
  if ( IsRef(a) && Undef(*a) ) return depth ? 2 : 3;
  if ( IsComp(a) && IsMeta(SkelP(a)->Fn) ) return 5;
  return depth ? 1 : 4;
}
```

Figure 9: C-code counterpart to `meta_deref/4`

3 Benchmarks

This section reports on two experiments conducted in order to provide a reference point for any further evaluation of the merits of semantic unification based on this implementation. It also gives some empiric evidence to the claim made in [3], namely that the *provision* of semantic unification is incurring virtually no overhead.

3.1 AI Expert June 1987

These benchmarks were written by Fernando Pereira of SRI to test different aspects of a PROLOG implementation. The benchmarks concentrate on 'basic execution' performance and do not perform any input/output or other operating system interface functions. The benchmarks were used for a comparison of different PC and MAC PROLOGs in the June 1987 issue of AI Expert magazine (published by Miller Freeman, 500 Howard st, San Francisco, CA 94105). AI Expert has also made versions available on various bulletin boards. The following table gives the name of the test[8] and the ratio $\frac{\texttt{MetaProlog}}{\texttt{C-Prolog}}$. The execution times were typically averaged over 2000 iterations per test.

Test	$\frac{\texttt{MetaProlog}}{\texttt{C-Prolog}}$
medium_unify	1.54
deep_unify	1.53
args(16)	1.19
args(8)	1.18
args(4)	1.17
args(2)	1.15
walk_term	1.13
walk_list	1.12
walk_term_rec	1.10
walk_list_rec	1.10
args(1)	1.10
assert_unit	1.00
choice_point	0.99
tail_call_atom_atom	0.98
setof	0.98
pair_setof	0.98
double_setof	0.98

Test	$\frac{\texttt{MetaProlog}}{\texttt{C-Prolog}}$
cons_term	0.98
cons_list	0.98
binary_call_atom_atom	0.98
arg(1)	0.98
trail_variables	0.97
slow_access_unit	0.97
integer_add	0.97
floating_add	0.97
bagof	0.96
arg(8)	0.96
arg(4)	0.96
arg(2)	0.96
arg(16)	0.96
access_unit	0.96
index	0.95
deep_backtracking	0.95
shallow_backtracking	0.93

3.1.1 Summary

The ratio $\frac{\texttt{MetaProlog}}{\texttt{C-Prolog}}$ ranges from 1.54 to 0.93, i.e., the *provision* for semantic unification costs as much as 54% overhead in execution time in extreme cases. As `C-Prolog` implements type recognition by signed pointer comparisons, one needs an additional test in some places of the unification algorithm. If a benchmark would test these branches exclusively, the ratio $\frac{\texttt{MetaProlog}}{\texttt{C-Prolog}}$ would be ≈ 2. For some tests `MetaProlog` is even slightly faster than

[8] if you are curious what each one does, please refer to the original literature

`C-Prolog`. Partially this can be attributed to noisy measurements, but the real cause is that some test sequences in the original unification algorithm had to be rearranged in order to accommodate for the new tests for metafunctors — as an unintended side effect this gave us some percents of performance gain in some cases.

3.2 Chat

CHAT-80 [9] is a classic PROLOG program developed as a part of a research project at the University of Edinburgh by Fernando C.N. Pereira and David H.D. Warren. CHAT-80 is a good medium-size PROLOG program that does something interesting and can be used for demonstrations and performance measurements. It has been taken as an example for a 'realistic' PROLOG program, as opposed to the previous benchmark set (section 3.1), which tests very specific parts of a PROLOG implementation in isolation.
The following table gives the number of the query, the ratio $\frac{\texttt{MetaProlog}}{\texttt{C-Prolog}}$ for the execution times for various subtasks, and the ratio for the total query evaluation times.

Test	Parse	Semantics	Planning	Reply	Total
1	1.00	1.00	0.80	1.02	1.00
2	0.93	0.94	0.75	1.00	0.92
3	1.03	0.93	1.00	1.33	1.00
4	1.03	0.94	0.97	0.99	0.99
5	1.04	1.00	1.00	0.94	0.95
6	1.03	0.96	1.00	1.00	1.00
7	1.06	0.97	0.93	0.95	0.95
8	1.10	0.97	0.95	0.94	0.96
9	1.01	0.98	1.04	0.94	0.96
10	1.00	0.94	0.96	0.95	0.96
11	1.02	0.96	0.95	0.96	0.97
12	1.05	0.91	1.00	0.95	1.00
13	1.00	1.11	0.98	0.95	0.98
14	1.02	0.94	0.98	0.94	0.94
15	1.03	0.95	0.97	0.95	0.95
16	1.00	0.92	0.95	0.92	0.95
17	1.02	0.96	1.00	0.96	0.99
18	1.00	1.12	1.00	0.94	0.96
19	0.96	1.08	0.96	0.91	0.92
20	1.00	0.96	1.00	0.91	0.92
21	1.00	0.96	0.93	0.91	0.94
22	1.02	0.98	0.96	0.99	0.99
23	1.04	0.96	1.00	0.94	0.95
Total	1.02	0.98	0.96	0.97	0.96

3.3 Summary

Averaged over all queries, subtasks and totals `MetaProlog` and `C-Prolog` can be considered equal on this benchmark.

For all realistic PROLOG programs operational at our site, we found that unification does not seem to dominate the 'composition' of these programs, as they executed in `MetaProlog` with the same speed as in `C-Prolog`.

4 Usage - extensions to C-Prolog

The principal difference between `MetaProlog` and standard `C-Prolog` is that an extended unification algorithm works in place of the standard one. After some functors are defined as metafunctors *and* instances of them have to be considered during unification, the system needs to refer to the definition of the extended unification theory. This reference is made via two *user-defined* predicates. If their definitions are missing, all metaunifications will fail.

4.1 Additional builtin predicates

The following predicates allow for

- the definition of metaterms
- to check for presence of metaterms
- the explicit removal of redundant metastructures
- syntactic unification, i.e., considering metaterms as ordinary terms during unification. This allows us to construct metaterms and to take them apart.

4.1.1 `meta_functor/1` (essential)

This predicate is used to define derefable metafunctors. Its single argument is of the form N/A, where N is an atom and $A > 0$ the arity of the functor. The effect of this predicate is dynamic. It does not matter if instances of this functor already exist or if this definition comes first. Use `ordinary_functor/1` to remove the property of being a metafunctor from any functor.

```
Example:
        :- meta_functor(meta/2).
```

4.1.2 `nonderefable_meta_functor/1` (rarely needed)

This predicate is used to define non-derefable metafunctors. Its single argument is of the form N/A, where N is an atom and $A > 0$ the arity of the functor. The effect of this predicate is dynamic. It does not matter if instances of this functor already exist or if this definition comes first. Use `ordinary_functor/1` to remove the property of being a metafunctor from any functor.

```
Example:
        :- nonderefable_meta_functor((+)/2).
```

4.1.3 `ordinary_functor/1` (rarely needed)

This predicate is used to remove the property of being a metafunctor from any functor. You will rarely need it. Its single argument is of the form N/A, where N is an atom and $A > 0$ the arity of the functor. The effect of this predicate is dynamic. It does not matter if instances of this functor already exist or if this definition comes first. Use `meta_functor/1` to define metafunctors.

```
Example:
        :- ordinary_functor(meta/2).
```

4.1.4 `syntactic_headunification/1` (essential)

This predicate tells the system that calls to the specified (user-defined) predicate have to take place with syntactic headunification in force. The possibility of refering to syntactic unification is a strict necessity for the implementation of an extended unification theory. Otherwise unifications involving metaterms would just rise further metaunifications which could never be executed. The single argument of `syntactic_headunification/1` is of the form N/A, where N is an atom and $A > 0$ the arity of the predicate to be called with syntactic headunification. The effect of this declaration is dynamic. It does not matter if the predicate is already defined or if this definition comes first. Note that the bodygoals of the predicate get executed with semantic headunification in force. Use `semantic_headunification/1` to restore the default behaviour of any user-defined predicate.

```
Example:
        :- syntactic_headunification(metatermunify/2),
           syntactic_headunification(metametamunify/2).
```

4.1.5 `symmetric_syntactic_headunification/1` (essential)

This predicate tells the system that calls to the specified user-defined predicate have to take place with syntactic headunification in force *and* that the predicate is symmetric, i.e., $f(A, B) = f(B, A)$. This is very convenient if the semantic unification theory operates on many sorts of metafunctors. The definition of `metametamunify/2` has to cover all possible $(n(n-1)/2)$ pairs of metafunctors. If `metametamunify/2` is defined to be symmetric, the number of defining clauses is cut into half. The single argument of the declaration is of the form N/A, where N is an atom and $A = 2$ the arity of the predicate to be called with symmetric, syntactic headunification. The effect of this declaration is dynamic. It does not matter if the predicate is already defined or if this definition comes first. Note that the bodygoals of the predicate get executed with semantic headunification in force.

Use `semantic_headunification/1` to restore the default behaviour of any user-defined predicate.

```
Example:
       :- symmetric_syntactic_headunification(metametamunify/2).
```

4.1.6 `semantic_headunification/1` (rarely needed)

This declaration restores the default, i.e. semantic headunification, for any predicate. It reverts the effects of `syntactic_headunification/1` and `symmetric_syntactic_headunification/1`. You will rarely need this declaration. The single argument of the declaration is of the form N/A, where N is an atom and $A > 0$ the arity of the predicate to be called with symmetric, syntactic headunification. The effect of this declaration is dynamic. It does not matter if the predicate is already defined or if this definition comes first.

```
Example:
       :- semantic_headunification(metametamunify/2).
```

4.1.7 `headunification_category/2` (rarely needed)

Unifies the second argument with the headunification category of the first argument that must be a functor.

```
Example:
| ?- headunification_category(metametaunify(_,_),Category).
Category = 2

| ?- headunification_category(Var,Category).
no
```

4.1.8 `===/2` (essential)

This predicate functions *exactly* like the original `C-Prolog` `=/2` predicate, i.e., it tries to unify the two arguments *syntactically*. It is the predicate of choice to construct metaterms and to take them apart. Note that `=/2` operates with semantic unification in force. Through the use of `syntactic_headunification/1` any predicate can be made operating with syntactic headunification, but we recommend the use of `===/2`. It is defined as:

```
:- op(700,xfx,===), syntactic_headunification('==='/2).

A === A.
```

4.1.9 `ismeta/1` (rarely needed)

Succeeds if its single argument is a metafunctor. Note that this predicate does not skip redundant levels of metastructure and that it recognizes instantiated metaterms with a bound first argument still as metaterms. A predicate that is more powerful in this respect is `meta_deref/4`.

```
Example:
| ?- ismeta(X).
no

| ?- meta_functor(f/2),ismeta(f(1,1)).
yes

| ?- meta_functor(f/2),ismeta(f(_,1)).
yes
```

4.1.10 `derefable_meta/1` (rarely needed)

Succeeds if its single argument is a derefable metafunctor. Note that this predicate does not skip redundant levels of metastructure and that it recognizes instantiated metaterms with a bound first argument still as metaterms. A predicate that is more powerful in this respect is `meta_deref/4`.

4.1.11 `meta_deref/4` (essential)

The predicate `meta_deref/4` takes any object as its first argument and returns its type, the last element of the chain of metafunctors and the first argument of the last metafunctor in the chain. It *operates with syntactic headunification in force.* The PROLOG specification of `meta_deref/4` is in figure 4, the resulting types are described in section 2.1, page 5.

4.1.12 `current_reader_var/2` (rarely needed)

This is a very low-level predicate, and it should not be expected to be portable. Is was needed for the implementation of an alternative toplevel function (see section 4.4.2). Pretty printing of cyclic metaterms relies on the use of names to cut the loops. One could invent arbitrary names, of course. For some potential cut-points we have names already: the names used for the variables in the goal given to the PROLOG interpreter. Unfortunately, the correspondence between variables and their names is kept in the guts of `C-Prolog` only[9]. The predicate `current_reader_var/2` is the interface to this representation. It nondeterministically returns the variables and their names seen by the previous invocation of `read/1`.

[9]Some PROLOG implementations provide a `read/2` predicate that returns a list of variable-name pairs as second argument.

4.1.13 Relational predicates

The `C-Prolog` predicates `</2`, `=</2`, `>/2` and `>=/2` have been made redefinable. If they are not redefined they behave exactly as in `C-Prolog`. In the original `C-Prolog` implementation they were protected against redefinition. Their protected versions are now called `lt/2`, `le/2`, `gt/2` and `ge/2`. The unprotected versions have been made necessary by the implementation of a particular semantic theory.

4.2 Builtin predicates inherited from `C-Prolog`

The rule is that they don't 'know' anything about metaterms, i.e., they treat them as ordinary terms. Nevertheless they operate with semantic headunification in force. Inherited builtins are treated in this way because any extended unification theory induces a new meaning to all builtin predicates. As we cannot take care of any possible semantic theory in advance, we move the responsibility of proper builtin predicate applications to the designer of the extended unification theory. In practice this usually amounts to dereferencing metaterms prior to builtin applications. As most of the inherited builtin predicates are not prepared to cope with cyclic terms, this may lead to some additional effort. The builtin predicate `write/1` is an exception to this rule (see section 4.4).

4.3 Additional user-defined predicates

In order to implement an extended unification theory one has to define the following two predicates. *They are always called with all redundant levels of metastructure removed (dereferenced).* Please refer to figure 5 which specifies the circumstances that lead to calls to `metatermunify/2` and `metametamunify/2`.

4.3.1 `metatermunify/2` (essential)

This predicate is used to specify how a metaterm and an ordinary term should be unified under the extended unification theory. The first argument is the *dereferenced* metaterm, the second argument is any PROLOG term except a variable and except a metaterm. The first argument of the metaterm is guaranteed to be an *unbound* variable if the metaterm is derefable. The predicate is called with *syntactic* headunification in force by default.

`Example:`

```
  metatermunify($frozen(Value,Goals),Value) :- call(Goals).
```

4.3.2 `metametamunify/2` (essential)

This predicate is used to specify how two metaterms should be unified under the extended unification theory. The predicate is called with *syntactic* headunification in force by default. Both arguments are *dereferenced* metaterms. The first argument of any of the two metaterms is guaranteed to be an *unbound* variable if the metaterm is derefable.

Example:

```
  metametaunify($frozen(V,G1),$frozen(V,G2)) :-
    V = $frozen(_,(G1,G2)).
```

4.4 Printing of metaterms

The builtin function `write/1` does not do much about metaterms:

- Derefable metafunctors are dereferenced and the end of this chain is printed via `write/1`. As metaterms might be cyclic, a depth bound on printing metaterms inside other metaterms is enforced. The default value for this bound is 1. It might be set when `C-Prolog` is started from the unix command line (see section 4.4.1). If printing a metafunctor would exceed the depth bound, it is not printed as `functor(arg1, ... argN)`, but just as `arg1`. Operator definitions for metafunctors are ignored during printing. Metafunctors print in standard parenthesised prefix notation.
- Non-derefable metafunctors behave as ordinary functors during printing. Operator definitions for such a functor are honored. Printing of non-derefable metafunctors is never suppressed by the depth bound for metaterms, nor does it increase the current depth.

Metastructures are often complex or even cyclic structures — therefore they have to be printed with some care in order to be useful. You might want to use `portray/1` for fancy printing. Note that output from trace is produced by `write/1`.

4.4.1 Additional command line parameter to `C-Prolog`

The new command line switch `-m N` specifies the depth up to which nested metaterms are printed. The default for N is 1. If you set it to 0, uninstantiated derefable metafunctors print like unbound variables. Example:

```
% -m 0
| ?- freeze(X,write(X)).
X = _8

% -m 1
| ?- freeze(X,write(X)).
X = $frozen(_8,write(_8))

% -m 2
| ?- freeze(X,write(X)).
X = $frozen(_8,write($frozen(_8,write(_8))))
```

4.4.2 New toplevel

The ordinary `C-Prolog` toplevel uses `write/1` to show the bindings of goalvariables. The new `toplevel` uses `print/1` instead. This leads to calls of `portray/1`, which gives us the possibility to represent metaterms[10] more meaningful and readable. The new toplevel catches errors, but an `abort` throws you back to the standard toplevel. To re-enter it, just type `toplevel`. Inside a break the original toplevel is executed. This is convenient if you have to have a look at your metastructures without having them grinded by `portray/1`. If you decide to use the new toplevel, your definition of `portray/1` has to take care of cyclic metaterms. In particular that means, that if no clauses for `portray/1` are defined, and if there are cyclic metaterms, `print/1` will loop.

References

[1] Carlsson M.: *Freeze, Indexing, and Other Implementation Issues in the WAM*, in Lassez J.(ed.), Logic Programming - Proceedings of the 4th International Conference - Volume 1, MIT Press, Cambridge, MA, 1987.

[2] Elcock E.W., Hoddinott O.: *Comments on Kornfeld's "Equality for Prolog": E-unification as a Mechanism for Augmenting the Prolog Search Strategy*, in Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86), Morgan Kaufmann, Los Altos, CA, 1986.

[3] Holzbaur C.: *Metastructures as a Basis for the Implementation of Constraint Logic Programming Techniques*, Austrian Research Institute for Artificial Intelligence, Vienna, TR-90-2, 1990.

[4] Holzbaur C.: *Realization of Forward Checking in Logic Programming through Extended Unification*, Austrian Research Institute for Artificial Intelligence, Vienna, TR-90-11, 1990.

[5] Kornfeld W.A.: *Equality for Prolog*, in Proceedings of the 8th International Joint Conference on Artificial Intelligence, Morgan Kaufmann, Los Altos, CA, 1983.

[6] Neumann G.: *Metaprogrammierung und Prolog*, Addison-Wesley, Reading, MA, 1988.

[7] Pereira F.: *C-Prolog 1.5 Users Manual*, SRI International, Menlo Park, CA, 1982.

[8] Sterling L., Shapiro E.: *The Art of Prolog*, MIT Press, Cambridge, MA, 1986.

[9] Warren D.H.D., Pereira F.C.N.: *An Efficient Easily Adaptable System for Interpreting Natural Language Queries*, American Journal of Computational Linguistics, 8(3-4), 1982.

[10] and of course, ordinary ones too!