Dynamic attributes, their hProlog implementation, and a first evaluation

Bart Demoen

Report CW 350, October 2002

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Dynamic attributes, their hProlog implementation, and a first evaluation

Bart Demoen

Report CW 350, October 2002

Department of Computer Science, K.U.Leuven

Abstract

Most systems implement statically declared attributes for performance reasons. This paper gives a first indication that also for dynamic attributes whose implementation is not based on static declarations, performance can be very good, while offering more flexibility. Their implementation in hProlog is shown. The differences with existing systems are indicated. Small benchmarks make the claim hard.

Dynamic attributes, their hProlog implementation, and a £rst evaluation

Bart Demoen *

October 28, 2002

Abstract

Most systems implement statically declared attributes for performance reasons. This paper gives a £rst indication that also for dynamic attributes whose implementation is not based on static declarations, performance can be very good, while offering more ¤exibility. Their implementation in hProlog is shown. The differences with existing systems are indicated. Small benchmarks make the claim hard.

1 Introduction

We assume knowledge of Prolog and its implementation. For a good introduction to the WAM [10], see [1]. We will use hProlog in this paper: it is a WAM based system and a successor of dProlog [3]; it is available from the author. hProlog is meant to become a back end to HAL [2]. Knowledge about attributed variables is also assumed, in particular having read the parts about attributed variables in the manuals of SICStus Prolog and ECLiPSe might prove helpful.

The basic concept of attributed variables can be traced back to several authors: Christian Holzbaur [6], Ulrich Neumerkel [8], Serge Le Houitouze [7] ... They bear similarities and there are also differences. In terms of successful implementations of attributed variables, it is probably fair to say that there is SICStus Prolog and ECLiPSe [1]. When reading their manuals, the following citation from the ECLiPSe manual related to pre_unify is striking:

> *This handler* **[pre_unify]** *is provided only for compatibility with SICStus Prolog and its use is not recommended, because it is less ef£cient than the* **unify** *handler and because its semantics is somewhat unclear, there may be cases where changes inside this handler may have unexpected effects.*

The SICStus Prolog manual says about the goal **verify_attributes(-Var, +Value, -Goals)**:

*Dept. of Computer Science, K.U. Leuven, Belgium, bmd@cs.kuleuven.ac.be

[1] Yap implements attributed variables following SICStus Prolog, XSB is different - see later

verify_attributes/3 may invoke arbitrary Prolog goals, but Var should not be bound by it. Binding Var will result in unde£ned behavior.

and

Var might have no attributes present in Module; the uni£cation extension mechanism is not sophisticated enough to £lter out exactly the variables that are relevant for Module.

We tried to understand these notes by looking at implementation source code, by asking around and trying out alternative implementations. We implemented within hProlog £rst the model as in the SICStus Prolog manual - with verify_attributes/3 - but without any of the unde£ned or unsophisticated behavior alluded to in the SICStus Prolog manual. Later we committed to the ECLiPSe model, but without support for pre_unify: The implementation of pre_unify in ECLiPSe uses a daring trick which we are not ready to mimic.

At the moment we started this work, hProlog had already for some time the predicate freeze/2 with its usual semantics. The prime reason for turning it into more general attributed variables, was their future use in hProlog as a backend for HAL.

In Sections 2, 3 and 4, we present our basic choices. Section 5 describes some features not (yet) implemented. Section 6 describes the implementation of our attributed variables. Section 7 describes some benchmarks and their results. Section 8 shortly deals with how to mimic one model in the other. Section 9 expresses some preliminary ideas about how to combine modules with attributes. Section 10 £nishes with a conclusion. The appendix contains the user manual for our attributed variables, and the essential parts of the benchmark source code.

2 Basic choice I: dynamic attributes

In Prolog systems like SICStus Prolog, ECLiPSe, XSB, Yap ... a program using attributed variables contains a declaration of the used attributes and in some of these systems also must use the atts module. ECLiPSe and the other systems differ in the declaration: in ECLiPSe one declares an atom to be the name of the attribute, and one can attach any term to it as its value. In SICStus Prolog, one declares in a module (which has a name that is an atom) a set of functors as attribute wrappers (which can have arity 0 [2]) and associated to this module one can have any set of values so that their principal functors are drawn from this set of wrappers: each wrapper can occur only once per module. There is also a similarity: the atom name in ECLiPSe and the module name in SICStus Prolog are used as the module in which to activate the unify hook.

The same module that declares the attributes, contains calls to predicates like get_atts and put_atts (SICStus Prolog) which contain the functor wrapper of a value as a manifest argument, as in e.g.

```
get_atts(X, lin(LinX))
```

and

```
put_atts( X, [type(t_Lu(Bound,U)),strictness(Strict)])
```

[2] but ./2 does not work

The preprocessing of such a module (by means of term_expansion) uses these manifest arguments (and the declarations) to generate (more) ef£cient code. We will see later what happens when non-manifest arguments are used.

We name a model in which attributes must be declared, *static attributes*.

It is clear that there are advantages to requiring the declaration of attributes: it becomes easier optimize installing attributes and retrieving them, by common preprocessing techniques. Joachim Schimpf actually con£rmed to us that this declaration and preprocessing business *is just for ef£ciency reasons*. These declarations also give some compile time type checking, because programs with typos in the names of the functor wrappers produce errors during compilation [3].

Another model for attributed variables does not require any declarations: at any moment (and in any module) a variable can be given any attribute. We name this *dynamic attributes*. Apparently, SICStus Prolog used to have dynamic attributes - but the oldest release we have is 3.7 and it already needs declarations. One can image some JIT optimizations when a new attribute is created that have the same effect as a compile time optimization.

We have chosen to implement dynamic attributes because hProlog is meant to be a backend for HAL; the HAL compiler will compile code to hProlog code and HAL itself is typed. hProlog should rely on the HAL compiler to do the preprocessing. On the whole, the concept of preprocessing is rather alien to hProlog at this moment. Another reason is that the ¤exibility of dynamic attributes appeals to the untyped-minded Prolog programmer, as long as there is no huge performance loss, even if no JIT optimization is attempted. One purpose of this paper is to show that performance loss is not a necessary consequence of dynamic attributes.

3 Basic choice II: one attribute value per module

Another common characteristic of implementations of attributed variables is that they are module aware: in SICStus Prolog, an attribute belongs to a module, which means that the verify_attribute/3 predicate from the corresponding module is activated on uni£cation of an attributed variable. In ECLiPSe it means that calls to add_attribute/3 can be replaced by calls to add_attribute/2 because the compiler is module aware. For SICStus Prolog, at the implementation level it means that together with the attribute name (principal functor) also the module must be separately stored [4]. In ECLiPSe, the name of the module serves as the name of the attribute and is enough to retrieve the value of the attribute.

There is simplicity and elegance in using an atom as the name of an attribute - and the value is associated to it - and at the same time using that atom as the name of the module in which the (equivalent of) verify_attributes is called. But that also imposes the restriction of having at most one term per module as attribute value for a variable. Still, this is not a restriction in practice as ECLiPSe seems to live happily with it and it makes life considerably simpler.

We have chosen the model of one module-one attribute value, i.e. the ECLiPSe model. It is easy to mimic more than one attribute value per module in this model.

[3] Introducing attributed variables in a typed language has more severe problems than that

[4] this is also a consequence of the fact that SICStus Prolog does not have an atom based module system

4 Basic choice III: no *pre_unify*

The most profound difference between attributed variables in SICStus Prolog and ECLiPSe is the state of the variable at the moment the uni£cation handler is called. As an example: assume the attributed variable X has one attribute A (with module user) and X is uni£ed with the integer 666. In SICStus Prolog, verify_attributes is called with X still being unbound. In ECLiPSe, the unify handler is called with A as an argument - X has been bound to 666 already. The SICStus Prolog behavior follows [6] which claims that it is important to have the variables as if not yet bound. ECLiPSe people on the other hand (Joachim Schimpf) claim that this is not needed. We do not want to take a stand in this issue: we had implemented at some point the full SICStus Prolog behavior (but on dynamic attributes of course) and were not convinced that the extra implementation complexity was worth the gain. However, we were not impressed by the manuals of SICStus Prolog and ECLiPSe, which claim that binding the X during a pre_unify yields unexpected results: in our opinion (and experience), it just shows that full support for pre_unify is too cumbersome to implement and maintain in a full system like SICStus Prolog or ECLiPSe. So we switched to the ECLiPSe model, but without the compromise to support pre_unify for compatibility reasons: Occam's razor is a blessing :-)

5 Not yet implemented

We have currently not implemented

1. a hook for printing attributed variables
2. a hook for getting the goal representation of attributed variables
3. a hook for copying attributed variables
4. a hook for comparing attributed variables
5. a test_unify hook (as in ECLiPSe)
6. a predicate for getting the attributed variables created since a particular execution point (as with '$constraint_list'/2 in SICStus Prolog)
7. time stamps for avoiding multiple trailing (this is generally not yet available in hProlog)

The £rst two are easy. The others would require some more work, especially the compare hook - it is not clear to us why one would want this hook.

Less visible to the user, but potentially very important for performance, are path compression (see [6]) and variable chain shunting ([9]). The chosen representation of attributed variables implies that during dereferencing, one can't know whether this chain is in fact a chain of repeatedly bound attributed variables. This means that path compression is done for all chains, or for none. No experiments have yet been done on this issue. On the other hand, exactly because of the chosen representation, the garbage collector does not need to be aware of attributed variables: plain variable chain shunting seems suf£cient. These issues deserve further investigation.

The deletion of an attribute - currently done by del_attr/2 - is not implemented in a satisfactory way yet.

6 Implementation

We start by presenting in Section 6.1 the old hProlog implementation of freeze/2: it was implemented in January 2001. The main point is the representation of a suspension, which is unusual as far as we know. The implementation of attributed variables is just a small variant of this freeze implementation and is shown in Section 6.2. Section 6.3 shows how currently freeze/2 is implemented just using the general interface to attributes. Section 6.4 shows a specialized implementation of freeze/2, which uses a more low-level interface to attributes, but which is safe and available to the user: the benchmarks will show that it performs much better than the general implementation. Section 6.5 explains some issues related to the representation of a suspension. Section 6.6 delves a bit deeper in the low-level details.

6.1 The old implementation of freeze/2

We introduced a new internal data type: the frozen variable. The internal representation of a frozen variable used to be two consecutive cells:

1. one cell containing an encoding of a self-pointer with a tag == DELAY
2. one cell containing the list of frozen goals

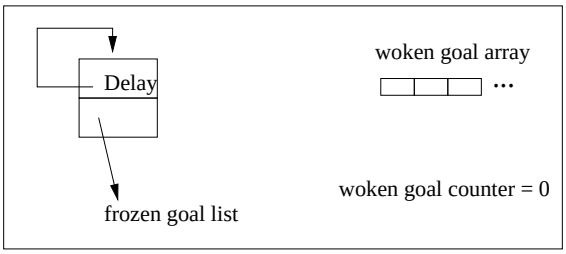


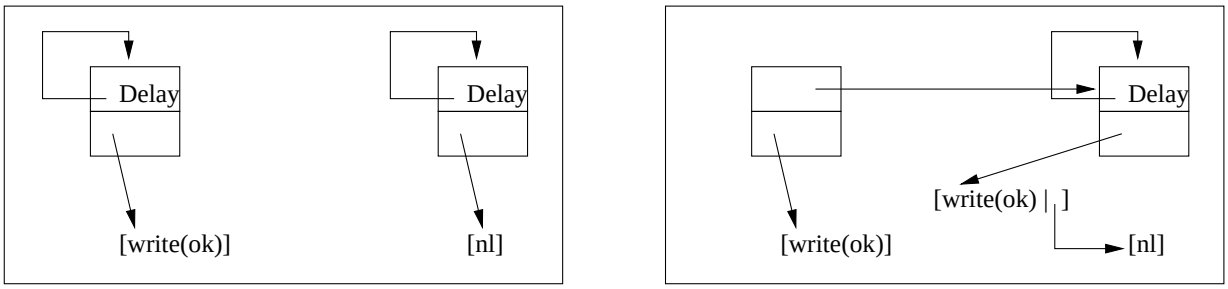
Figure 1: Heap representation of a suspension

When two frozen variables are uni£ed, the uni£cation routine immediately concatenates the frozen goals lists (which is linear in the size of one of them) and one DELAY-self-ref into a reference to the other DELAY-self-ref.

When a frozen variable is bound, its goal list is added to a woken-goal-array which is external to the heap (but contains pointers into the heap); a global ¤ag [5] is set; there is a global counter indicating how many woken goal lists there are in the woken-goal-array; the counter is reset to zero on backtracking and at the start of activating the woken goal lists. See Figure 3.

At the next call port (heap over¤ow check actually) the goals in the woken-goal-array are put on the heap as a list and given as £rst argument to a predicate named deal_with_wakeup/2; its second argument is the continuation, i.e. the computation that was interrupted to execute the woken goals. Wakeup is simply implemented as:

[5] the common trick is to abuse the heap limit



(a) Before unifying two frozen variables

(b) After their uni£cation

Figure 2: Illustration of the transformation

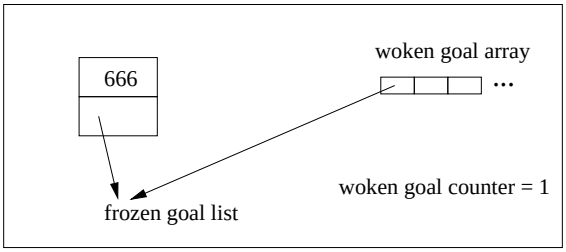


Figure 3: After binding the frozen variable to 666

```
deal_with_wakeup([],G) :- call(G).
deal_with_wakeup([LG|LGs],G) :-
          wakeups(LG),
          deal_with_wakeup(LGs,G).

wakeups([]).
wakeups([G|Gs]) :- call(G), wakeups(Gs).
```

This proved to be a simple implementation with good performance.

6.2 The implementation of dynamic attributes

Basically the same idea was used for implementing attributed variables; an attributed variable is represented by two cells:

1. one cell containing an encoding of a self-pointer with a tag == ATT
2. one cell containing (a reference to) the attributes; we name it the *attribute term*

One important difference with the previous implementation of freeze/2 is that when two attributed variables are uni£ed, their unify handler must be called, while previously this did not result in a co-routing action. This change was minimal.

The attribute term is open for experimenting: since we have dynamic attributes - i.e. no attribute declarations and no preprocessing - we have considerable freedom. At £rst we implemented just four basic predicates:

1. attvar/1: similar to var/1, but succeeds only of the variable is an attributed variable
2. make_new_attvar/1: turns a variable (that is not an attributed variable) into an attributed variable (without attributes)
3. get_attr(X,A): uni£es A with the attribute term of X (or fails)
4. put_attr(X,A): replaces the attribute term of X by A

The user-level predicates are simply get_attr(Var,Mod,Val) and put_attr(Var,Mod,Val) [6], but it turned out to be necessary (for performance) to have a low-level implementation of these predicates - see Section 6.6.

Everything one needs can be implemented on top of that, given some conventions. In hProlog, an attribute is any term associated with an atom name - this atom is used to trigger in a module (with the same name) *attr_unify_hook/2* which is called when an attributed variable is bound to a term (or touched, i.e. bound to another attributed variable). Binding (or touching) an attributed variable X results in two items put in the woken-attribute-array: the attribute term of the attributed variable X and the term it is bound to (never a variable, but possibly another attributed variable). The deal_with_wakeup/2 predicate gets these in a list (and as second argument the continuation of course). The convention is that the attribute term looks like: [Mod1,Val1,Mod2,Val2,...], so that the implementation of wakeup becomes:

```
deal_with_wakeup(UnificationList,Continuation) :-
        wake_attvars(UnificationList),
        call(Continuation).

wake_attvars([]).
wake_attvars([Atts,Value|Rest]) :-
        call_all_attr_uhooks(Atts,Value),
        wake_attvars(Rest).

call_all_attr_uhooks([],_).
call_all_attr_uhooks([Name,AttVal|R],Value) :-
        Name:attr_unify_hook(AttVal,Value),
        call_all_attr_uhooks(R,Value).
```

Note that the attr_unify_hook is only called for attributes that are really present, independent of how many modules have attributes (contrary to what happens in ECLiPSe) and it is clearly de£ned (contrary to the remark in the SICStus Prolog manual).

One can wonder whether the implementation of dynamic attributed variables can compete with an implementation built on preprocessing. We will show that this is not impossible.

[6] there is also a delete predicate

One advantage of the above choices is that one can quite easily build a specialized implementation on top of the chosen primitives, which uses another convention for the attribute term: we have done so for the freeze library (which when used does no longer allow to use other attributes than the freeze-attribute) and a oneatt library (which allows at most one module-value to be associated to an attributed variable). The performance of both is discussed later in Section 7.

6.3 The general implementation of freeze/2 with attributed variables

Following folklore (see the SICStus Prolog manual), hProlog has currently the following implementation of freeze/2:

```
freeze(X,Goal) :-
        (attvar(X) ->
            (get_attr(X,freeze,G2) ->
                put_attr(X,freeze,(G2,Goal))
            ;
                put_attr(X,freeze,Goal)
            )
        ;
            (var(X) ->
                put_attr(X,freeze,Goal)
            ;
                call(Goal)
            )
        ).

freeze:attr_unify_hook(Goal,Y) :-
        (attvar(Y) ->
            (get_attr(Y,freeze,G2) ->
                put_attr(Y,freeze,(G2,Goal))
            ;
                put_attr(Y,freeze,Goal)
            )
        ;
            call(Goal)
        ).
```

In Section 7 we will compare this implementation with the old one described in Section 6.1.

6.4 A specialized implementation of freeze/2

Next follows another implementation of freeze/2 based on attributed variables, but the general form of the attribute term is not respected. This means that the predicates put_attr/3, get_attr/3 and del_attr/2, which use this general form, can no longer be used (on variables with the freeze attribute) and in fact, when the following module is loaded [7], the freeze attribute cannot be mixed

[7] hProlog allows to rede£ne many *builtin* predicates

with other attributes. The purpose of this module is mostly to show the kind of speedup one could get from a specialized implementation: see Section 7.

```
:- module(freeze, [freeze/2, deal_with_wakeup/2]).

freeze(X,Goal) :-
        (attvar(X) ->
            get_attr(X,G2),
            put_attr(X,(G2,Goal))
        ;
            (var(X) ->
                make_new_attvar(X),
                put_attr(X,Goal)
            ;
                call(Goal)
            )
        ).

deal_with_wakeup(Woken,G) :-
        wake_attvars(Woken),
        call(G).


wake_attvars([]).
wake_attvars([Goal,X|Rest]) :-
        (attvar(X) ->
            get_attr(X,G2),
            put_attr(X,(G2,Goal))
        ;
            call(Goal)
        ),
        wake_attvars(Rest).
```

6.5 About the tagged self-reference

The introduction of the tagged self-reference for representing a delayed or attributed variable is a deviation from having tag-on-pointers as is usual in WAM implementations to tag-on-data - as for instance in BinProlog. At £rst sight, it might look ugly to have in hProlog both tag-on-data and tag-on-pointers (for LIST and STRUCT), but the representation was chosen on purpose:

1. the deref routine does not need any changes: normally, in a tag-on-data schema, one needs a deref routine that returns both the address of the end of the chain and the value at the end of the chain; in a tag-on-pointer schema, one needs only the value at the end of the chain; with a tagged self-reference, the deref routine can still just return the end value of the chain; untagging gives its address; see [4] for more on different tagging schemas

2. when an attributed variable is bound, its tagged self-reference cell, is overwritten - by an atom for instance or a list tagged pointer ... In this way, the deref routine does not need to be aware of a possible difference between a bound and a free attributed variable and the attributed variable has no slot indicating this. In XSB for instance, the deref loop was changed to a double loop because of the representation of attributed variables. Also, in other descriptions of attributed variables - e.g. [5] - one can see this double loop: on top of the normal abstract machine dereferencing, there are explicit calls to *meta_deref*.

There is one caveat regarding that representation (and it is common to all tag-on-data except for single cell terms): one cannot copy such a tagged self-reference to an argument register or a cell in the local stack. And neither can one bind a variable to an attributed variable by copying the tagged self-reference; one must untag £rst.

6.6 Low-level details

We choose the representation of the set of attribute values together with their names, to be a list with an alternating name and value: [N1,V1,N2,V2,...]. At £rst we had the more obvious representation [N1=V1, N2=V2,...], but this takes both more space and is less ef£cient.

When at £rst we only had the low-level predicates get_attr/2 and put_attr/2, we implemented quite obviously put_attr/3 as follows

```
put_attr(X,Mod,Val) :-
          get_attr(X,AllAtts),
          replace_att(AllAtts,Mod,Val,NewAtts),
          put_attr(X,NewAtts).

replace_att([],Mod,Val,NewAtts) :- NewAtts = [Mod,Val].
replace_att([M,V|R],Mod,Val,NewAtts) :-
          (M == Mod ->
                NewAtts = [Mod,Val|R]
          ;
                NewAtts = [M,V|N],
                replace_att(R,Mod,Val,N)
          ).
```

Similarly, the implementation of get_attr/3 was based on the member predicate.

It turned out that about half of the time spend in some benchmarks, was in either member or replace_att. That was enough reason to implement put_attr/3 and get_attr/3 directly at the C-level.

7 Experiments

We have two types of benchmarks: (1) for testing freeze/2 (Section 7.1) and (2) for testing more generally attributed variables (Section 7.2). Unfortunately, all benchmarks are arti£cial: realistic benchmarks with attributed variables seem not to exist, unless one takes a solver from e.g. the

SICStus Prolog libraries. The latter approach has its value (it could show scaling properties for instance), but the outcome will depend too much on other factors than the particular implementation of attributed variables. The artificial benchmarks have the advantage that they just (or mainly) test the implementation of attributed variables.

We have subtracted dummy loops from the test, except in the *sieve* benchmark.

7.1 Experiments with freeze/2

Our goal is to compare the following:

1. hProlog 1.6 - with the original freeze implementation described earlier
2. hProlog 1.9 - with the freeze implementation based on the general attributed variable implementation (Section 6.3)
3. hProlog 1.9 - with the freeze implementation based on the specialized freeze library (Section 6.4)
4. SICStus Prolog with its native implementation of freeze/2
5. SICStus Prolog with an implementation of freeze/2 along the lines of the example in its own manual and which is similar to the one described in Section 6.3

We could have included other systems in this experiment: ECLiPSe, Yap and XSB. The reasons for not doing so are: ECLiPSe has an elaborated suspension mechanism about which we found it difficult to make sure that our use of it would be fair. Yap and XSB can't run some benchmarks because of memory management bugs.About Yap and XSB we will mention some results in passing though.

There are three benchmark programs for freeze/2:

1. unifreeze(N): a list of N variables is frozen on an atomic goal w, and then these N variables are unified - there is no wake up of the goals, since the variables are not instantiated; tested with N == 500
2. multiplefreeze(N): one variable X is subject N times (in conjunction) to the goal freeze(X,w); no wake up; tested with N == 500
3. sieve(N): the program computes the first N prime numbers with a classical implementation of the sieve algorithm using freeze; tested with N = 10000

In the first two benchmarks, the number of freeze operations is $O(N^2)$, while a natural way to introduce a wake up in these benchmarks would do just N wake ups. That is why they were not done.

Yap was giving segmentation faults for unifreeze and sieve, and incorrect behavior during multiplefreeze.

XSB and Yap had memory management problems for sieve (already with $N > 1000$).

SICStus Prolog had a *Resource error: insuf£cient memory* for ?- multiplefreeze(2000), while hProlog has no problems with this query.

We report the sum of the timings for 3 runs in milliseconds on a PentiumPro, 1.8 MHz. The £gure between brackets is the garbage collection time (if zero, it is not mentioned): for SICStus Prolog, we did *export GLOBALSTKSIZE=36909859* and for hProlog we gave accordingly 36909860 bytes of initial heap. This £gure was chosen because when started with a smaller heap size, SICStus Prolog expands the heap up to that point and this contorts the £gures.

	hProlog1.6	hProlog1.9 general	hProlog1.9 specialized	SICStus 3.9.0	SICStus 3.9.0 with attributes
unifreeze	960	1100	360	660	1700
multiplefreeze	90	360	110	170	1860
sieve	1670	3110	1650	3530 (330)	7550 (2170)

Table 1: Performance for freeze/2

As an interpretation of these £gures, we offer the following:

1. the basic operations of freeze are about twice as fast in SICStus Prolog than in hProlog1.9; in a benchmark where other work is done, this seems to play not a big role
2. the specialized freeze/2 implementation for hProlog1.9 based on attributes outperforms all other implementations
3. the implementation of freeze with (general) attributes in SICStus Prolog is signi£cantly slower than in hProlog1.9; it also uses more heap space

One conclusion is that reserving a dedicated freeze slot in every attributed variable, seems a good idea: it would (in unifreeze) avoid the need for going to the unify handler all the time.

7.2 Experiments with general attributes

The SICStus libraries contain some solvers (clpb, clpr, clpfd, clpqr, clpr and chr) that declare in total 21 attributes, spread over 9 modules. So it seems that the number of attributes (and modules) in any particular application is rather low. We expect hProlog to perform worse if there are more attributes, so our arti£cial benchmarks use always 7 attributes, which seems to be more than usual. Their value is atomic - see the code in the appendix - which favours SICStus Prolog as for atomic values, simple arithmetic is enough to decide whether the attribute is present or not. On the other hand, the benchmark programs do not contain calls to get/put_atts with in the second argument a list if attributes: the preprocessing combines such accesses very ef£ciently. In this respect, the benchmarks favour hProlog.

There are essentially three tests:

1. repeatedly getting all 7 attributes (one at a time) from one attributed variable to which these 7 attributes were put once

2. repeatedly putting 7 attributes (one at a time) to a new variable

3. repeatedly unifying two attributed variables with just one attribute; the handler does nothing - in the case of SICStus Prolog, verify_attributes returns an empty list as third argument

For the £rst two test, the repeat factor was 1000000: it must be very high or otherwise the timings become too small. On the other hand, memory management can start dominating for lager repeat factors. One can run these two tests with the attribute name manifest: in that case preprocessing can improve the access as in SICStus Prolog, but hProlog does not. Or one can run the tests with the attribute non-manifest: that case is no different for hProlog, but for SICStus Prolog, it means that the preprocessing is shifted to the run time. SICStus Prolog has another inconvenience here: when the attribute name is not manifest, one must wrap goals like get_atts(X,A) in a call/1, otherwise preprocessing complains. In Yap, this is solved more gracefully. SICStus Prolog incurs quite some penalty from this metacall+shifting to runtime.

In the third test, the uni£cation is performed 10 million times and we measure essentially the overhead over doing a uni£cation of two free variables, i.e. the cost of setting up the handler and executing it. We name this test *trigger*.

	hProlog1.9	SICStus Prolog 3.9.0
put manifest	930	1460 (80)
get manifest	930	1120
put non-manifest	1140	667900 (19750)
get non-manifest	1210	392420 (12240)
trigger	8430	12300

Table 2: Performance of attributes

The table shows that for hProlog1.9 there is a small difference between manifest and non-manifest code; it is due to a slightly different setup of the benchmark. For SICStus Prolog, there is a huge difference, also in the heap consumption. Part of this is due to the implementation of the metacall. Showing these results does some injustice to the overall qualities of SICStus Prolog. Moreover, in ECLiPSe, one can do a non-manifest put, but not a non-manifest get. As Joachim Schimpf says: *What would you do with it?*

More importantly, the manifest part of the table indicates that with 7 attributes, dynamic attributes are not slower than static ones.

XSB could not run the benchmarks with the above repeat factor. With a 10 times lower repeat factor, the timings for manifest and non-manifest get, were very close to each other, but about 80 times slower than in hProlog. The put non-manifest had memory management problems.

Yap was twice as fast as hProlog1.9 for the manifest get, but busted on all other tests with the repeat factor of 1000000. A lower repeat factor showed that manifest get is about 50 times faster than non-manifest get - but the reliability of these tests is questionable: depending on previous queries in the same session, Yap would either bust or not.

The trigger benchmark shows that the triggering mechanism in SICStus Prolog and hProlog are rather similar. Surprisingly, Yap takes 6 times as much time for it as SICStus Prolog. XSB takes about 10% less time than hProlog. This last result prompted some investigation: it seems

that depending on the version of gcc used, XSB performs 10% better or 20% worse than hProlog. It might mean that differences between systems of the order of 10% are not meaningful.

8 Mimicking the other models

8.1 Pre_unify in hProlog

If the basic implementation does not provide pre_unify, it is probably not feasible to mimic it at the source level. But since one can't bind the £rst argument of verify_attributes anyway, the following rewrite of the predicate wake_attvars (see Section 6.2) comes close:

```
wake_attvars([]).
wake_attvars([Atts,Value|Rest]) :-
        put_attr(New,Atts),
        call_all_attr_uhooks(Atts,New,Value),
        wake_attvars(Rest).

call_all_attr_uhooks([],_,_).
call_all_attr_uhooks([Name,_|R],New,Value) :-
        Name:verify_attributes(New,Value,GoalList),
        call_all_attr_uhooks(R,New,Value),
        call_goal_list(GoalList).
```

The difference that remains is that when verify_attributes(X,Val,G) is activated in SICStus Prolog and Val should contain a reference to X, than Val sees the unbound attributed variable X, while with the above code in hProlog, Val would contain the bound value of X and the £rst argument of verify_attributes is a copy of the original unbound X.

It is not clear what the advantage in SICStus Prolog is: since Val can contain X, one can in general not reliably bind any attributed variable inside Val either.

8.2 More than one attribute per module in hProlog

Mimicking more than one attribute per module (as in SICStus Prolog) can be done easily in hProlog or ECLiPSe, because the predicates put_attr/2 and get_attr/2 allow for any term as the value of the attribute. A low-level implementation of some frequent and time-consuming operations will of course help performance, but this does not necessarily mean that the idea of dynamic attributes should be buried.

8.3 XSB and attributed terms

XSB has a different view on what uni£cation of attributed variables should do: from the users perspective, XSB postpones completely such uni£cation. This can be illustrated best with a small example:

```
:- import get_atts/2, put_atts/2 from atts.
:- attribute floop/1.
verify_attributes(X,Y) :-
        write(X-Y), nl.

| ?- put_atts(X,floop(a)), f(X,X) = f(1,2).
_h254 - 1
_h254 - 2
X = _h254
```

In all other Prolog systems we tried the similar code, the uni£cation fails and no uni£cation handler is called.

The implementation of the XSB model - at the low-level - is quite easy in hProlog: during general uni£cation, instead of binding an attributed variable immediately, it is put in the woken goal list not-yet-uni£ed. In XSB, the user is supposed to force the uni£cation inside the handler by using the attv_unify predicate.

The XSB implementation has (at least) two (easy to £x) bugs: attv_unify does not check whether its £rst argument is an attributed variable, and add_interrupt does not check for over¤ow of the interrupt array.

9 Is there a methodology for combining attributes ?

The ideas presented here are preliminary: we have asked around about a methodology for combining modules with attributes, but got no answers. Still the issue seems important, in view of the fact that solvers can be based on attributed variables, and that the wish for combining solvers is strongly present.

The main difference between the model in ECLiPSe (and hProlog) on one hand and SICStus Prolog (and XSB) on the other hand, is that in the latter systems, the uni£cation handler has access to ALL the attributes of the uni£ed attributed variable, while in the former model, a handler in a particular module only sees its own attribute. At £rst sight, there seems to be an advantage in seeing all the attributes, because this means that a handler can base its actions on the non-local attributes as well and in this way, composition of solvers might be achieved. Here is a quote from the SICStus Prolog manual:

> *More complicated interactions are likely to be found in more sophisticated solvers. The corresponding verify_attributes/3 predicates would typically refer to the attributes from other known solvers/modules via the module pre£x in Module:get_atts/2.*

From the software engineering point of view [8] it seems a bad idea to combine solvers by giving them knowledge about each other. One problem with this approach is that the order in which unify handlers from different modules are called, cannot be relied upon. Still, it might look like the only possibility. Suppose we have two solvers, CLP(color) and CLP(style), and an application about clothes could use both. The application programmer could decide to use only one, or a

[8] I can't believe I wrote that :-(

different one in different parts of the program or both all the time. She would still like to *see* these two separate solvers, knowingly deploying them when she thinks this is necessary. These solvers could work independently, but a combined effort might make the application perform better. So the approach in which both solvers know about each other seems £ne and the question is whether there is a better way.

There is certainly a different way, but whether it is better ... we make a new solver - CLP(looks) - which intercepts all posting of constraints and it calls the old solvers who do not know they are abused. The application programmer must now see only the new solver. The latter seems inconvenient: there should be a way for the applications programmer to use the two old solvers, and for the new solver take over invisibly. One declaration by the application programmer
:- combine [CLP(color),CLP(style)] into CLP(looks) should be enough. One necessary feature to get this off the ground, would be the possibility for the master solver to intercept calls to the slave solvers.

10 Discussion

Attributes variables are different in different systems, and every time we take a closer look (by reading the manuals once more, or by writing some small programs) more differences pop up. hProlog adds its own attributed variables to the confusion. We have followed ECLiPSe more closely than we have followed SICStus Prolog. However, we do not plan to provide compatibility with SICStus Prolog as an option. We deviate from both by making attributes dynamic: nothing needs to be known at compile time about attributes and we can still get decent performance. But the £rst full ¤edged application with attributed variables in hProlog still needs to be built [9], so a £nal conclusion on the performance issue must be postponed. We believe that this dynamic model makes programming with attributed variables easier and more fun. There are three issues to explore now for us: £rst we should introduce some technique for avoiding multiple trailings of the same cell during a deterministic computation; secondly, path compression and/or variable chain shunting should probably be introduced; thirdly, optimizing the dynamic attributes should be investigated, but still without requiring declarations. Luckily, these three are orthogonal.

Acknowledgements

We are grateful to Joachim Schimpf for enlightening e-mails on the subject of attributed variables in ECLiPSe, and Bert Van Nuffelen for explaining his use of attributed variables. We also thank Tom Schrijvers for comments on an early version of this paper and Henk Vandecasteele for his work on the ilProlog compiler used within hProlog.

References

[1] H. Ait-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990 See also: http://www.isg.sfu.ca/~hak/documents/wam.html.

[9] Tom Schrijvers started one.

[2] B. Demoen, M. García de la Banda, W. Harvey, K. Mariott, and P. Stuckey. An overview of HAL. In J. Jaffar, editor, *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *LNCS*, pages 174–188. Springer, 1999.

[3] B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *Lecture Notes in Arti£cial Intelligence*, pages 1240–1254. ALP, Springer, 2000.

[4] D. Gudeman. Representing type information in dynamically-typed languages. Technical Report TR93-27, Department of Computer Science, The University of Arizona, Tucson, Arizona, 1993.

[5] C. Holzbaur. Realization of forward checking in logic programming through extended uni£cation. Technical Report TR-90-11, Oesterreichisches Forschungsinstitut fuer Arti£cial Intelligence, Wien, Austria, 1990.

[6] C. Holzbaur. Meta-structures vs. Attributed Variables in the Context of Extensible Uni£cation. In M. Bruynooghe and M. Wising, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, number 631 in Lecture Notes in Computer Science, pages 260–268. Springer-Verlag, Aug. 1992.

[7] S. Le Houitouze. A New Data Structure for Implementing Extensions to Prolog. In P. Deransart and J. Maluszynski, editors, *Proceedings of the Second International Symposium on Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 136–150. Springer-Verlag, Aug. 1990.

[8] U. Neumerkel. Extensible uni£cation by metastructures. In *Proceedings of the second workshop on Metaprogramming in Logic (META'90)*, pages 352–364, Apr. 1990.

[9] D. Sahlin and M. Carlsson. Variable Shunting for the WAM. Technical Report SICS/R-91/9107, SICS, 1991.

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

Appendix - a short manual for attributes in hProlog

What is an *attribute* ?

An attribute consists of an identi£er and an associated value, also named the attribute value. hProlog restricts identi£ers to atoms. The associated value can be any term.

How to attach an attribute to a term

Attributes can only be attached to terms that can still bind to any term, i.e. to variables only. One usually names a variable to which an attribute was attached, an attributed variable, but one can think of any variable as an attributed variable, possibly without any attribute attached to it. An attribute is attached to a variable by the predicate attr_put/3. Here are some examples:

```
?- put_attr(X,foo,[coco]).

X = AttVar538{[foo,[coco]]}


?- put_attr(X,foo,[coco]), put_attr(X,gee,floris(Z,Z)).

X = AttVar921{[gee,floris(_807,_807),foo,[coco]]}
Z = _807


?- put_attr(X,foo,[coco]), put_attr(X,gee,floris(Z,Z)), put_attr(X,foo,bl

X = AttVar1069{[gee,floris(_807,_807),foo,bla]}
Z = _807
```

The default printing of an attributed variable can be seen above. This output is at the moment not readable by read/1.

The second example shows that two attributes can be attached to the same variable - actually, any number is possible.

The third example shows that if one attaches an attribute with the same identi£er twice, one actually overwrites the previous value associated to that identi£er. Such overwriting is backtrackable:

```
?- put_attr(X,foo,[coco]), (put_attr(X,foo,bla) ; true).

X = AttVar910{[foo,bla]}
Yes  | ;

X = AttVar910{[foo,[coco]]}
```

Note that the var/1 builtin succeeds on attributed variables ! To test whether a term is an attributed variable, use attvar/1 (see later).

How to get the value of an attribute, given its identi£er

Simply by get_attr/3:

```
?- put_attr(X,foo,[coco]), put_attr(X,gee,floris(Z,Z)), get_attr(X,foo,A)

X = AttVar1066{[gee,floris(_807,_807),foo,[coco]]}
Z = _807
A = [coco]
```

If the identi£er does not occur in the attributes of X, get_attr simply fails. (some error messages in case of abuse might be a good idea)

All attributes of a variable can be gotten in one go by get_attr/2:

```
?- put_attr(X,foo,[coco]), put_attr(X,gee,floris(Z,Z)), get_attr(X,A).

X = AttVar878{[gee,floris(_288,_288),foo,[coco]]}
Z = _288
A = [gee,floris(_288,_288),foo,[coco]]
```

How to delete an attribute

Use del_attr/3:

```
?- put_attr(X,foo,[coco]), put_attr(X,gee,floris(Z,Z)), del_attr(X,foo).

X = AttVar881{[gee,floris(_288,_288)]}
Z = _288
Yes

?- put_attr(X,foo,[coco]), put_attr(X,gee,floris(Z,Z)),
                               del_attr(X,foo), del_attr(X,gee).

X = AttVar1438{[]}
```

The last example shows that an attributed variable can have an empty list of attributes - it is not clear whether it is a good idea and reasonable to turn an attributed variable with an empty list of attributes (at least behaviorly) into an ordinary variable. hProlog currently does not do so.

What happens if an attributed variable X is uni£ed with T ?

We assume that X and T are not identical: unifying a variable with itself is a void operation. There are still two cases to consider:

1. T is a plain variable, i.e. no attributes where ever attached to it: T is bound to X and no further action occurs

2. otherwise: X is bound to T and for each identi£er M that X has in its attributes, the goal M:attr_unify_hook(MAtt,T) is scheduled as soon as possible; MAtt will be bound to the attribute value associated to M for X, as if gotten by get_attr(X,M,MAtt).

Several points can be noted:

- in hProlog, *as soon as possible* means at the next entry of a general Prolog predicate; some builtins are not treated by hProlog as a Prolog predicate: cut (!/0) is the most notable of them, but also most of arithmetic, functor/3, arg/3 ... if the correctness of a program relies on activating the unify_hook before a cut, it most probably is a bad program; if (only) performance suffers if unify_hook is not called early enough, one can insert a dummy call at speci£c program points; the HAL compiler will be responsible for inserting dummy calls
- if X and Y are both attributed variables, and they are uni£ed, one has no control - and one should not rely - on whether M:attr_unify_hook(AttM,Y) is called with M in the identi£ers of X or M:attr_unify_hook(AttM,X) with M in the identi£ers of Y; unify_hook should deal with this appropriately
- the order in which M:attr_unify_hook for different M is called, should not be relied upon; if it is important, it is better to use only one attribute
- M:attr_unify_hook/2 is called, even if it does not exist: this results in an error message and failure (later perhaps an exception)
- at this point, one should be able to see that uni£cation of an attributed variable with an empty list of attributes, when uni£ed with a bound term, results in exactly the same behaviour as a plain variable
- one should also be able to see that uni£cation of an attributed variable with an empty list of attributes, when uni£ed with an attributed variable with a non-empty list of attributes, might result in a different behaviour ...

Two more predicates

Implementors might £nd it useful to have at their disposal the following low-level, but safe predicates:

1. make_new_attvar/1: to be called with a plain variable; turns the argument into an attributed variable with an empty list of attributes
2. attvar/1: similar to var/1, but succeeds only if its argument is an attributed variable, i.e. it was subject to make_new_attvar/1 at some point and it is not yet instantiated

Some remarks/questions

The following became apparent to us while writing the above documentation.

ECLiPSe seems to call the unify hook of every module that de£nes an attribute, even if no value for that module (= identi£er) was ever put: the attribute value is then returned as a free variable. It is not clear to us why this is a good thing. If many independent modules declare attributes, this is bound to cause some unnecessary slow down. SICStus Prolog does not do that. It is apparent from the code above that hProlog only calls the hook(s) in the module(s) for which there is an attribute and that the interpretation on whether there is an attribute for a particular module is completely in the hands of the application implementor.

When the unify hook is called in module M, in hProlog and ECLiPSe, one only gets the attribute value associated to M. In SICStus Prolog, verify_attributes is called with the uni£ed attributed variable still free, so one has access to all its attributes. In hProlog it would be easy to give access to all attributes as well - even if the uni£ed attributed variable is already bound - by slightly modifying the implementation of call_all_attr_uhooks: all attributes instead of just one could be given to attr_unify_hook or as an extra argument. It is not clear to us whether having access to all attributes is helpful when using several attribute de£ning modules. The good news is that it is easy to rede£ne the behaviour of call_all_attr_uhooks, even for an applications programmer. This ease is a side effect of dynamic attributes.

In ECLiPSe, the conjunction

```
add_attribute(X,a,m), add_attribute(X,b,m)
```

is actually treated as

```
add_attribute(X,a,m), add_attribute(NewX,b,m), X = NewX
```

The latter uni£cation activates the unify handler for module m. A consequence is that if this handler is of the form:

```
handler(X,Y) :-
     compute_attributes(X,Y,Atts),
     add_attribute(Y,Atts,m).
```

the above conjunction loops. However, it seems a nice feature to intercept the addition (or replacement) of an attribute. Such feature can be easily implemented by local program transformation.

The benchmarks

attest.pl

The code below is the one of the manifest put and get benchmark as reported on in Section 7.2 for hProlog. The *implicit* versions are omitted.

```
get_explicit(N) :-
        put_attr(X,a,a),
        put_attr(X,b,b),
        put_attr(X,c,c),
        put_attr(X,d,d),
        put_attr(X,e,e),
        put_attr(X,f,f),
        put_attr(X,g,g),
        get_explicit(N,X).
```

```
get_explicit(N,X) :-
        (N > 0 ->
            get_attr(X,a,_),
            get_attr(X,b,_),
            get_attr(X,c,_),
            get_attr(X,d,_),
            get_attr(X,e,_),
            get_attr(X,f,_),
            get_attr(X,g,_),
            M is N - 1,
            get_explicit(M,X)
        ;
            true
        ).
```

```
put_explicit(N) :- put_explicit(N,X).

put_explicit(N,X) :-
        (N > 0 ->
            put_attr(X,a,a), put_attr(X,b,b),
            put_attr(X,c,c), put_attr(X,d,d),
            put_attr(X,e,e), put_attr(X,f,f),
            put_attr(X,g,g),
            M is N - 1,
            put_explicit(M,_)
        ;
            true
        ).
```

sattest.pl

The following is the SICStus Prolog version of the above attest.pl.

```
:- use_module(library(atts)).

:- attribute a/0, b/0, c/0, d/0, e/0, f/0, g/0.

get_explicit(N) :-
        put_atts(X,a),
        put_atts(X,b),
        put_atts(X,c),
        put_atts(X,d),
        put_atts(X,e),
        put_atts(X,f),
        put_atts(X,g),
        get_explicit(N,X).

put_explicit(N) :- put_explicit(N,X).

put_explicit(N,X) :-
        (N > 0 ->
            put_atts(X,a), put_atts(X,b),
            put_atts(X,c), put_atts(X,d),
            put_atts(X,e), put_atts(X,f),
            put_atts(X,g),
            M is N - 1,
            put_explicit(M,_)
        ;
            true
        ).

get_explicit(N,X) :-
        (N > 0 ->
            get_atts(X,a),
            get_atts(X,b),
            get_atts(X,c),
            get_atts(X,d),
            get_atts(X,e),
            get_atts(X,f),
            get_atts(X,g)
            M is N - 1,
            get_explicit(M,X)
        ;
            true
        ).
```

unifreeze.pl

```
unifreeze(N) :- s(N,N).

s(I,N) :-                                ss(N,L) :-
        I > 0,                                   (N > 0 ->
        ss(N,L),                                     freeze(X,w),
        tt(L,_),                                     M is N - 1,
        II is I - 1,                                 L = [X|R],
        s(II,N).                                     ss(M,R)
                                                 ;
w.                                                   L = []
                                                 ).
tt([],_).
tt([X|R],X) :- tt(R,X).
```

multiplefreeze.pl

```
multiplefreeze(N) :- s(N,N).

s(I,N) :-                              ss(N,X) :-
        I > 0,                                 (N > 0 ->
        ss(N,_),                                   freeze(X,w),
        II is I - 1,                               M is N - 1,
        s(II,N).                                   ss(M,X)
                                               ;
                                                   true
                                               ).
```

sieve.pl

```
sieve(N) :-                            int_list(N,[N|L]) :-
        freeze(L,sieve(L,N)),                  M is N + 1,
        int_list(2,L).                         int_list(M,L).

sieve([X|L],N) :-
%       write(X), write(' '),
        X < N,
        freeze(L,filter(X,L,L1)),
        freeze(L1,sieve(L1,N)).

filter(F,[X|L],L1) :- multiple(X,F), !, freeze(L,filter(F,L,L1)).
filter(F,[X|L],Z) :- Z = [X|L1], freeze(L,filter(F,L,L1)).

multiple(X,F) :- 0 is X mod F.
```