Metastructures vs. Attributed Variables in the Context of Extensible Unification

—

Applied for the Implementation of CLP Languages

Christian Holzbaur
Austrian Research Institute for Artificial Intelligence, and
Department of Medical Cybernetics and Artificial Intelligence
University of Vienna
Freyung 6, A-1010 Vienna, Austria
email: christian@ai.univie.ac.at

Abstract

We relate two mechanisms which aim at the extension of logic programming languages. The first mechanism directly extends syntactic unification through the introduction of a data type, whose (unification) semantics are specified through user-defined predicates. The second mechanism was utilized for the implementation of coroutining facilities, and was independently derived with optimal memory management for various Prolog extensions in mind. Experience from the application of both mechanisms to the realization of CLP languages, without leaving the logic programming context, enables us to reveal similarities and the potential with respect to this task. Constructive measures that narrow or close the gap between the two conceptual schemes are provided.

1 Introduction

As a serious user of two rather similar mechanisms — as far as their applications are concerned — we think that it is useful to expose this similarity in some detail. Both mechanisms provide means for the extension of logic programming languages.

Metastructures as introduced by [Neumerkel 90] aim at extensions to Prolog's builtin unification through user-defined behavior of metastructures during unification. A refined version of the concept of metastructures was used in [Holzbaur 90] for the specification and

implementation of a variety of instances of the general CLP scheme [Jaffar & Michaylov 87].

More or less at the same time, the data type *attributed variable* was introduced by Hoitouze. Memory management issues as *early reset* and *variable shunting* by the garbage collector were addressed in [Hoitouze 90]. The behavior of attributed variables during unification was not mentioned. However, regarding applications, Hoitouze also proposed the use of attributed variables for the implementation of delayed computations, reversible modification of terms, variable typing, and others.

Earlier, [Carlsson 87] used a data type *suspension*, which was incorporated into SICStus Prolog [Carlsson & Widen 90] for the implementation of coroutining facilities. As far as we can tell — as a third party — the data structures attributed variable and suspension are the same. The difference between Hoitouze's and Carlsson's exposition is that the former put some emphasis on the data type as such and on memory management. The latter used it as a low level primitive for the implementation of mechanisms that necessitated the specification of the behavior of the data type during unification.

In the following sections we will have a closer look on metastructures and attributed variables. In particular, we compare them with regard to their behavior during unification and their potential for the implementation of CLP languages. In fact, the quality and the availability of SICStus Prolog motivated the reiteration of the experiments in [Holzbaur 90], which were originally based on metastructures, incorporated by the author into the C-Prolog [Pereira 82] interpreter. The migration to SICStus Prolog, entailing a compiler, garbage collector, better arithmetics, and, although hidden from the ordinary user, the suspension data type, led to this comparison.

2 Metastructures

Metastructures are ordinary, non-variable Prolog terms with the sole difference that they can be detected as members of this special sort. Metastructures are introduced by a declaration `:- meta_functor N/A`, where `N/A` denotes *any* functor. In [Neumerkel 90], metastructures are restricted to be of the form `meta/N`, $N > 0$, which prevents the declaration of e.g. the usual arithmetic functors `+/2,*/2`... as being 'meta' — which is a disadvantage if we are about to implement CLP($\Re$) via metastructures. Simple source transformations from arbitrary functors to `meta/N` functors are *not*, as one might think, sufficient for an emulation of the effects of the proposed declaration. Think of already existing structures, built from a given functor, prior to the declaration, and think of dynamically produced structures (via `read/1,functor/3,=../2`).

The behavior of metastructures during unifications can be specified precisely through a Prolog meta interpreter which makes unification explicit [Holzbaur 90]. The meta interpreter implements the unification table from [Neumerkel 90] *and* makes some further conventions integral parts of the specification.

- Unifications between variables and metastructures just produce a binding as usual. If a metastructure is to be unified with an ordinary term, the reaction to this event is given by a user-supplied predicate `meta_term_unify/2`. Similarly, unifications between two metastructures are covered by the user-supplied predicate `meta_meta_unify/2`, the arguments being the two metastructures involved.

- Once extensible unification is put into force, we have a problem passing metastructures to the user-supplied predicates `meta_term_unify/2` and `meta_meta_unify/2`, without triggering further calls to them in a nonterminating fashion. Neumerkel solved the problem by the introduction of a builtin predicate `===/2`, which behaves as `=/2`, but treats metastructures as ordinary structures. In addition he has to rely on the programmers discipline: Nothing but variables may be used as formal arguments in the definition of the two user-supplied predicates. Access to the components of metastructures is via `===/2`. The disadvantages of this solution are that the user has to be very careful, and that indexing does not apply.

  Therefore, we specified the following mechanism: Calls to the two user-supplied predicates `meta_term_unify/2` and `meta_meta_unify/2` are made with syntactic unification in force. The encapsulation effect of this solution is at least as strong as the one with `===/2`, as the only means to get access to the 'internals' of a metastructure.

- The last part of the specification stemmed from a typical application of metastructures in the context of the implementation of CLP languages. It is covered in detail in the next section.

2.1 Metastructures and reversible modification

The implementations of many CLP instances require the functionality that is achieved with destructive updates in traditional, procedural realizations. In logic programming, this functionality is provided in a sound fashion by either copying or *modifying by variable substitution.*

The latter option can be applied to metastructures through the convention that one particular argument of the structure is a free variable, which will eventually be bound to another metastructure, obeying the same convention. Therefore, sequences of modifications lead to metastructure chains. The current 'value' of a metastructure is to be found at the end of the chain. Traversing this chains could of course be left to the user, but it is so common a pattern, that is has been made part of our specification. The additional convention is that the user-supplied predicates `meta_term_unify/2` and `meta_meta_unify/2` are called with the *current* metastructures, and that the *first* argument of metastructures is used for modifying by variable substitution. In [Neumerkel 90] this convention is also exploited by the garbage collector, which can therefore reclaim useless metastructures.

From our experience with the implementation of CLP instances we conclude: If metastructures are to be employed in serious applications, a garbage collecting scheme which 'knows'

about metastructures is strongly implied!

One step beyond, but in the same direction: To wait for the garbage collection to occur in order to reclaim space and to shorten metastructure chains (much like variable shunting) is a bad idea! Because of the need for traversal, access and modification operations are of $O(n^2)$[1]. After recognizing the metastructure traversal as being nothing but the *find* operation of the well known *union-find* algorithm, which is of $O(n)$ through *path compression* [Aho et al. 83], path compression was applied for metastructure access [Holzbaur 90]. Although it is realized in a fashion transparent to the user, preserving logical soundness, we will also present the Prolog version of the *find* operation with and without path compression, because it nicely demonstrates the power of the logical variable:

```
find( Current, Last) :-
  arg( 1, Current, Next),
  ( var(Next) ->
     Last = Current
  ;
     find( Next, Last)
  ).
```

The *find* operation without path compression

```
find( Current, Last) :-
  arg( 1, Current, Next),
  ( var(Next) ->
     Last = Current
  ;
     setarg( 1, Current, Last),
     find( Next, Last)
  ).
```

The *find* operation with path compression

The first argument to the predicate `find/2` is the (meta)structure to be traversed, the second parameter will be unified with the last element of the (meta)structure chain. The predicate `setarg/3` is the SICStus Prolog [Carlsson & Widen 90] primitive for reversible modification.

3 Attributed variables

Attributed variables are variables with an associated attribute, which is a term. Attributes are attached to variables, and attributes are referred to, through built-in predicates. As far as the rest of a given Prolog implementation is concerned, attributed variables behave like variables — they can be considered as a subtype of type variable. The indexing mechanism treats variables and attributed variables the same way. Built-in predicates observe attributed variables as if they were ordinary variables. Special treatment for attributed variables applies:

[1]Each modification extends the chain by one element, which has to be skipped on the next access. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

- During memory management, as proposed in [Hoitouze 90], i.e, early reset and variable shunting.
- During unification. [Carlsson 87] describes the data type *suspension* as:

  > "A suspension is an unbound variable with a reference to a suspended goal, represented as a record on the heap. A suspended goal is *woken* when the suspension is unified with another term."

  Three observations:

  - The attribute associated with a suspension variable is supposed to be executed eventually — therefore the name *suspension*.
  - The behavior of the suspension during unification is specified. In particular, an explicit WAM extension for the wakeup mechanism is presented.
  - From the remaining text of the reference cited above, we conclude that the suspension data type was not meant to be made available as such to the user.

Present versions of SICStus Prolog perform the proposed memory management of attributed variables alias suspensions, and implement the wakeup mechanism quoted. Given the conceptual proximity of metastructures and attributed variables, we decided to repeat experiments from [Holzbaur 90], i.e., the implementation of some CLP instances, which were originally based on metastructures, in the framework of attributed variables.

In order to be able to describe the problems we encountered with the original wakeup mechanism of SICStus Prolog, and the remedies, we sketch it here:

When a suspension — to be exact: the value part of a suspension — gets bound during unification to a term or to another suspension, the goal part of the bound suspension is prepended to the current continuation[2] of the execution state. At the next inference step, i.e., call or execute in WAM terminology, the woken goals will be run.

This mechanism is perfectly sufficient for the implementation of `freeze/2` and `dif/2` — no wonder, this was the intention behind the introduction of suspensions. In a wider context, however, the attribute of an attributed variable needs not to be executed. The narrow interpretation of attributed variables was the prime cause for the problems we encountered in more complex applications.

The problem description with the help of an example

From the point of view of the of the code which implements, say, a CLP($\Re$) solver via attributed variables, any number of binding and aliasing events may take place between two inference steps. Once the corresponding goals are woken, they are in the execution

[2] Scheme slang

state of the WAM only. The association between the attributed variables and the attributes (goals) is lost, as the attributed variables are now *transparent* — we only see the objects bound to them.

This is a dilemma, because the bindings which changed the situation, temporarily invalidating invariants of the CLP($\Re$) implementation, conceptually took place at once (because there was no inference step in between).

Each executing woken goal has to find out for which bindings it is responsible, take a sort of 'repair' action, leaving the data structures in a partially repaired state, assuming that the remaining woken goals will repair the rest. This is not only cumbersome, but also a source of incompleteness. In our CLP($\Re$) example, we want to fix the linear equation system after a binding or aliasing event, which might turn some of the variables of the system into constants (numbers). Binding a variable potentially triggers actions of the nonlinear equation solver or the inequality solver. We want to postpone these actions until the invariants about the linear equation system (being in solved form) are true again. Making the first woken goal to repair everything at once does not work either, as some of the relevant data structures are in the execution state only, and therefore inaccessible.

Note that this observations also destroy the hope for a *complete* CLP language implementation via `freeze/2`.

The solution

These problems can be avoided when the attributed variables are not bound immediately, but one by one, as part of the execution of the woken goals. This gives the (user)code a chance to look at the data structures *before* anything changes and, more important, conceptually there is always *one* and only one binding event taking place in isolation, and the (user)code has control over this event.

This treatment of attributed variables during unification leads to a situation that is equivalent to the one for metastructures.

4 Proposal for the treatment of attributed variables during unification

Carlsson's wakeup mechanism requires only minor changes, i.e., generalizations, to perform as proposed above. First we generalize suspensions in the sense that there is no need that the associated 'goal' will be executed eventually — this is why we prefer the name *attributed variable* instead of suspension. Next, the wakeup mechanism is supposed to work as follows:

- When an attributed variable is about to be bound during unification, the attributed variable and the value it should be bound to are recorded in some internal data

structure[3].

- If there is more than one such event between two inference steps, a list of attributed variable–value pairs is collected in some internal data structure.
- The fact that such an event took place is recorded.
- At the next inference step, the abstract machine takes measures to feed the attributed variable-value pairs to the two user-supplied predicates, in analogy to the specification for metastructures. The data structures for the representation of the list of variable-value pairs can be reclaimed at this point.

The memory management of attributed variables remains unchanged, except that we plan to repeat the incorporation of path compression on access and update of attributed variables.

4.1 Pragmatics

In this section we describe the user's point of view of a SICStus Prolog clone, providing extensible unification via attributed variables.

Source transformation

If we want to get extensible unification through attributed variables, we have to transform terms with interpreted functors into terms with attributed variables with the interpreted functors as attributes. Static occurrences of interpreted functors can be dealt with through source transformations. Dynamically introduced interpreted terms require changes in built-in predicates as `read/1`,`functor/3`,`=../2`.

Builtin predicates

The following predicates provide for the introduction, detection, and manipulation of attributed variables.

`get_attribute(X,C)`
: If X is an attributed variable, unify the corresponding attribute with C.

`attach_attribute(X,C)`
: Turn the free variable X into an attributed with attribute C.

`detach_attribute(X)`
: Remove the attribute from an attributed variable, turning it into a free variable.

[3] on the heap, in fact

`update_attribute(X,C)`
: Change the attribute of the attributed variable X to C. Acts as if defined below, but might be more (memory) efficient.

```
update_attribute( X, C) :-
  detach_attribute( X),
  attach_attribute( X, C).
```

User-defined predicates

The following two predicates have to be supplied by the user. They specify the behavior, i.e., the meaning of attributed variables during unification.

`verify_attribute(C,T)`
: This predicate is called when an attributed variable with attribute C is about to be unified with the non-variable term T.

`combine_attributes(C1,C2)`
: This predicate is called when two attributed variables with attributes *C1,C2* are about to be unified.

Note that the two predicates are are *not* called with the attributed variables involved, but with the corresponding attributes instead. The reasons are:

- There are applications which only refer to the attributes. Example:

  > In an implementation of a type system with, say, two disjunct types `odd` and `even`, we are interested in the detection of type (in)compatibility. Let the atoms `odd` and `even` be the attributes attached to the variables to be typed.

- If the application wants to refer to the attributed variables themselves, they can be made part the attribute term. The implementation of `freeze/2` below utilizes this technique. Note that this does *not* lead to cyclic structures, as the connection between an attributed variable and it's attribute is invisible to the pure parts of a given Prolog implementation.

- If attributed variables were passed as arguments, the user's code would have to refer to the attributes through an extra call to `get_attribute/2`.

- As the/one attribute is the first argument to `verify_attribute/2` and `combine_attributes/2`, indexing applies. Note that attributed variables themselves look like variables to the indexing mechanism.

4.2 Example

In order to show that both metastructures and attributed variables are equally capable to serve for the implementation of `freeze/2`, to demonstrate that the proposed wakeup mechanism for attributed variables is strictly more general that the old one, and to allow for the comparison of two solutions to the same task, we present two implementations of `freeze/2` side by side:

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

freeze( frozen(_,Goal), Goal).


meta_term_unify( frozen(Value,Goal), Value) :-
   call( Goal).


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

```
freeze( X, Goal) :-
  attach_attribute( V, frozen(V,Goal)),
  X = V.

verify_attribute( frozen(Var,Goal), Value) :-
  detach_attribute( Var),
  Var = Value,
  call(Goal).

combine_attributes( frozen(V1,G1), frozen(V2,G2)) :-
  detach_attribute( V1),
  detach_attribute( V2),
  V1 = V2,
  attach_attribute( V1, frozen(V1,(G1,G2))).
```

The left encoding of `freeze/2` with metastructures assumes the specification from [Holzbaur 90] being in force. The solution on the right builds on the semantics of the SICStus clone, as proposed in this paper.

5 Summary

Both metastructures and attributed variables are attractive and powerful concepts and are very similar to each other. From the inspection of the previous example, and from many others which do not fit into *one* paper, we conclude:

- The use of metastructures, together with the proposed conventions, are logically 'cleaner'. Their application for reversible modification has a sound declarative semantics, without any reference to alien concepts as destructive updates, and still allows for the transparent utilization of such methods in the implementation.
- Attributed variables are available in a 'raw' version in an attractive, State of the Art Prolog implementation — SICStus. It would be irresponsible to ignore this potential, given that minor changes can produce the intended functionality. A slight aftertaste remains because of the explicit modification predicate for the attribute part of attributed variables and because of the need for the transformation of interpreted terms into attributed variables.

Acknowledgements

This work was supported by the Austrian Federal Ministry of Science and Research.

References

[Aho et al. 83] Aho A.V., Hopcroft J.E., Ullman J.D.: *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

[Carlsson 87] Carlsson M.: Freeze, Indexing, and Other Implementation Issues in the WAM, in Lassez J.L.(ed.), *Logic Programming - Proceedings of the 4th International Conference - Volume 1*, MIT Press, Cambridge, MA, 1987.

[Carlsson & Widen 90] Carlsson M., Widen J.: Sicstus Prolog Users Manual, Swedish Institute of Computer Science, SICS/R-88/88007C, 1990.

[Holzbaur 90] Holzbaur C.: Specification of Constraint Based Inference Mechanisms through Extended Unification, Dept. of Medical Cybernetics & Artificial Intelligence, University of Vienna, Dissertation, 1990.

[Hoitouze 90] Huitouze S.le: A new data structure for implementing extensions to Prolog, in Deransart P. and Maluszunski J.(eds.), *Programming Language Implementation and Logic Programming*, Springer, Heidelberg, 136-150, 1990.

[Jaffar & Michaylov 87] Jaffar J., Michaylov S.: Methodology and Implementation of a CLP System, in Lassez J.L.(ed.), *Logic Programming - Proceedings of the 4th International Conference - Volume 1*, MIT Press, Cambridge, MA, 1987.

[Neumerkel 90] Neumerkel U.: Extensible Unification by Metastructures, *Proc. META90*, 1990.

[Pereira 82] Pereira F.: C-Prolog 1.5 Users Manual, SRI International, Menlo Park, CA, 1982.