Progress Report: Term Dags Using Stobjs [⋆]

J.-L. Ruiz-Reina, J.-A. Alonso, M.-J. Hidalgo and F.-J. Martín-Mateos
`http://www.cs.us.es/{~jruiz, ~jalonso, ~mjoseh, ~fmartin}`

Departamento de Ciencias de la Computación e Inteligencia Artificial.
Escuela Técnica Superior de Ingeniería Informática, Universidad de Sevilla
Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain

Abstract. We explore in this paper the use of efficient data structures to implement operations on first-order terms, that can be formally verified. Specifically, we present the status of our work on defining and verifying a unification algorithm acting on terms represented as directed acyclic graphs (*dags*). This implementation is done using single threaded objects (*stobjs*) to store a dag representing the unification problem.

Introduction

In [6], we described a formal approach, using ACL2, to the meta-theory of equational reasoning and term rewriting systems. As a by-product of this work, we obtained verified implementations of some basic operations on first-order terms, including matching, anti–unification, unification, normal forms with respect to term rewriting systems and critical pair computation.

In that formalization, terms are represented as lists, using prefix notation. Such representation is specially well suited for the automation of reasoning about terms (in particular, in proofs by induction on the structure of terms) but from the execution efficiency point of view it is not the best solution: due to the applicative nature of the ACL2 language, some operations (for example, the application of a substitution to a term) needs to rebuild terms in order to compute their results.

The standard approach to deal with this problem is to store terms as *directed acyclic graphs (dags)*, using pointer structures allowing some amount of structure sharing. Thus, operations never build new terms but merely updates pointers [1]. In ACL2, destructive updates can be implemented using *single threaded objects (stobjs)*. These are structures with the usual applicative semantics of ACL2 but for which updates are implemented destructively. To ensure that these destructive updates are consistent with its applicative semantics, ACL2 enforces some syntactic restrictions on the use of a stobj, which ensures that only one reference to the object needs ever exist. See [2] for a detailed description of stobjs in ACL2 or [7, 8] for two case studies where stobjs are used to define and verify efficient programs.

Our intention is to explore the use of stobjs to represent term dags in ACL2, in order to define and verify efficient operations on first-order terms. As a first attempt, we report the status of our work trying to implement and verify a unification algorithm on term dags. In this progress report, we describe an implementation based on stobjs. Although we have not yet verified this unification algorithm, we think it is interesting to present the current status of our work.

1 Unification on term dags

In the following, we will assume that the reader is familiar with unification theory [1]. The unification algorithm we have implemented is based on the well-known set of transformation rules given by Martelli and Montanari and shown in figure 1. This set of rules act on pairs of systems of equations of the form $S; U$. The system S can be seen as a set of pairs of terms to be unified

[⋆] This work has been supported by MCYT: Project TIC2000–1368–C03–02

and the system U as a (partially) computed unifier. The symbol $\perp$ represents unification failure. Starting with a pair of systems $S; \emptyset$, these rules can be (non-deterministically) applied until either a pair of systems $\emptyset; U$ or $\perp$ is obtained. It can be proved that these rules are terminating and that S is unifiable if and only if $\perp$ is not obtained; in that case, U is a most general unifier (*mgu*) of S. Thus, an algorithm to unify two terms t_1 and t_2 can be designed by choosing a strategy to exhaustively apply the rules starting with the pair of systems $\{t_1 \approx t_2\}; \emptyset$.

Delete:	$\{t \approx t\} \cup R; U$	$\Rightarrow_u R; U$
Check:	$\{x \approx t\} \cup R; U$	$\Rightarrow_u \perp$
		if $x \in \mathcal{V}(t)$ and $x \neq t$
Eliminate:	$\{x \approx t\} \cup R; U$	$\Rightarrow_u \theta(R); \{x \approx t\} \cup \theta(U)$
		if $x \in X$, $x \notin \mathcal{V}(t)$ and $\theta = \{x \mapsto t\}$
Decompose:	$\{f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n)\} \cup R; U$	$\Rightarrow_u \{s_1 \approx t_1, \ldots, s_n \approx t_n\} \cup R; U$
Clash :	$\{f(s_1, \ldots, s_n) \approx g(t_1, \ldots, t_m)\} \cup R; U$	$\Rightarrow_s \perp$, if $n \neq m$ or $f \neq g$
Orient:	$\{t \approx x\} \cup R; U$	$\Rightarrow_u \{x \approx t\} \cup R; U$
		if $x \in X$, $t \notin X$

Fig. 1. Transformation rules

As part of an ACL2 library with formal proofs of the lattice theoretic properties of first-order terms, we defined and verified a unification algorithm based on the above ideas. This work was translated to ACL2 from a previous formalization done in Nqthm, described in [5]. In this library, we represent first-order terms in prefix notation using lists. For example, the term $f(x, g(y), h(x))$ is represented as `'(f x (g y) (h x))`. Every `consp` object can be seen as a term with its `car` as its top function symbol and its `cdr` as the list of its arguments. Variables are represented by `atom` objects. Substitutions are represented as association lists and equations as dotted pairs of terms.

A naive implementation of the above unification algorithm can be inefficient in some situations. Consider, for example, the following standard unification problem:

$$p(x_1, \ldots, x_n) \approx p(f(x_0, x_0), f(x_1, x_1), \ldots, f(x_{n-1}, x_{n-1}))$$

A mgu of this example is $\{x_1 \mapsto f(x_0, x_0), x_2 \mapsto f(f(x_0, x_0), f(x_0, x_0)), \ldots\}$, mapping each x_i to a complete binary tree of height i. This unifier can be obtained by repeatedly applying the **Eliminate** rule of figure 1. Using lists to represent terms, a reconstruction of the terms is needed every time the rule is applied, making the algorithm inefficient.

The standard approach to deal with this source of inefficiency is to represent terms as directed acyclic graphs (*dags*). For example, in figure 2 we represent the unification problem given by the equation $f(h(z), g(h(x), h(u))) \approx f(x, g(h(u), v))$. Nodes are labeled with function and variable symbols, and outgoing edges connect every node with dags representing its immediate subterms. We can naturally identify the root node of a dag term with the whole term. Note also that there is also a certain amount of *structure sharing*, at least for the repeated variables.

To implement a unification algorithm with this term representation, the main idea is never build new terms but only create pointers. In particular, the **Eliminate** rule can be implemented adding a pointer linking the variable with the term to which this variable is bound; in that way no reconstruction of the term is required in the application of a substitution. In figure 2, these pointers are represented by dashed arrows. The binding for a variable can be determined by following the pointers traversing the graph depth first, from left to right. In this case, the substitution represented is $\{x \mapsto h(z), u \mapsto h(z), v \mapsto h(h(z))\}$, which is a mgu of $f(h(z), g(h(x), h(u)))$ and $f(x, g(h(u), v))$.

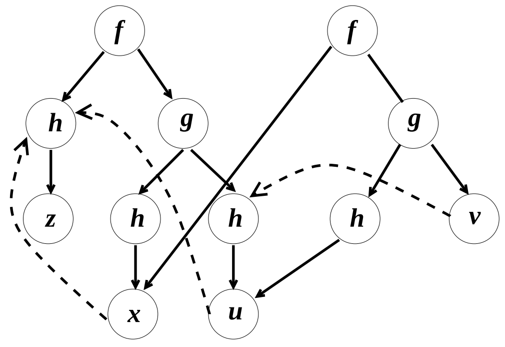

Fig. 2. Dag representation of $f(h(z), g(h(x), h(u))) \approx f(x, g(h(u), v))$

2 An ACL2 implementation

We describe now how we use a stobj to store first-order terms as directed acyclic graphs (we will assume that the reader is familiar with stobjs [2]). We define the following stobj containing one array field:

```
(defstobj terms-dag
  (dag :type (array t (1000)) :resizable t))
```

Each node in the graph is represented by a cell in the `dag` array of the stobj. Thus, a node in the graph can be identified with an array index. Each cell stores the label and the neighbors of each node, in the following way:

- If node `i` represents an unbound variable x, then `(dagi i terms-dag)` contains a dotted pair of the form `(`x `. t)`.
- If node `i` represents a bound variable, then `(dagi i terms-dag)` contains an index n pointing to the term for which the variable is bound.
- If node `i` is the root node of a non-variable term $f(t_1, \ldots, t_n)$, then `(dagi i terms-dag)` is a dotted pair of the form `(`f `.` l`)`, where l is the list of the indices corresponding to $t_1, \ldots, t_n$.

In this way, we can store a unification problem using the `terms-dag` stobj. For example, if we store the term $equ(f(h(z), g(h(x), h(u))), f(x, g(h(u), v)))$ (i.e., the unification problem of figure 2) the significant cells of the `dag` array are[1]:

```
((EQU 1 9) (F 2 4) (H 3) (Z . T) (G 5 7) (H 6) (X . T) (H 8) (U . T)
           (F 10 11) 6 (G 12 14) (H 13) 8 (V . T))
```

We can naturally identify an array index with the whole term whose root node is pointed to by that index. We exploit this intuitive idea in the definition of a function that applies one step of the transformation $\Rightarrow_u$ given in figure 3.

Let us now describe that function, called `dag-transform-mm`. In addition to the stobj, this function receives as input a (non-empty) system `S` of equations to be unified and a substitution `U` (partially) computed. Depending on the form of the first equation of `S`, one of the rules of $\Rightarrow_u$ is applied. But the main point here is that `S` and `U` *contain pointers* to the terms stored in `dag`. In particular, `S` is a list of pairs of indices, and `U` is a list of dotted pairs of the form `(`x `.` n`)` where x is variable symbol and n is an index pointing to the node for which the variable

[1] To print this array, we are using a function that collects the array cells in a list.

```
(defun dag-transform-mm (S U terms-dag)
  (declare (xargs :stobjs (terms-dag)
                  :mode :program))
  (let* ((ecu (car S)) (R (cdr S))
         (t1 (dag-deref (car ecu) terms-dag))
         (t2 (dag-deref (cdr ecu) terms-dag))
         (p1 (dagi t1 terms-dag)) (p2 (dagi t2 terms-dag)))
    (cond
     ((= t1 t2) (mv R U t terms-dag))                    ;;; DELETE
     ((dag-variable-p p1)
      (if (occur-check t t1 t2 terms-dag)                ;;; CHECK
          (mv nil nil nil terms-dag)
        (let ((terms-dag (update-dagi t1 t2 terms-dag))) ;;; ELIMINATE
          (mv R (cons (cons (dag-symbol p1) t2) U) t terms-dag))))
     ((dag-variable-p p2)
      (mv (cons (cons t2 t1) R) U t terms-dag))          ;;; ORIENT
     ((not (eq (dag-symbol p1) (dag-symbol p2)))         ;;; CLASH
      (mv nil nil nil terms-dag))
     (t (mv-let (pair-args bool)
                (pair-args (dag-args p1) (dag-args p2))
                (if bool                                 ;;; DECOMPOSE
                    (mv (append pair-args R) U t terms-dag)
                  (mv nil nil nil terms-dag)))))))       ;;; CLASH
```

Fig. 3. Definition of proofs and equivalence

is bound (i.e. a substitution). The function returns a multivalue with the following components, obtained as a result of the application of one step of $\Rightarrow_u$: the resulting system of equations to be solved, the resulting substitution, a boolean value (if $\perp$ is obtained, this value is `nil`) and the stobj `terms-dag`. See the definitions of the auxiliary functions in the file `dag-unification.lisp` of the supporting materials. Only when **Eliminate** is applied, this stobj is updated, causing the corresponding variable to point to the corresponding term.

Once we have defined the function that applies one step of the transformation rules, we can define a function that computes the most general unifier of two terms, whenever it exists. In short, this function stores both terms as directed acyclic graphs in the stobj (previously resizing the `dag` array), and iteratively applies the function `dag-transform-mm` until either a failure is detected or there are no more equations to be solved. In this case, the returned substitution is obtained from the dag, following the pointers of the instantiated variables. You can see the definition in the supporting materials (function `dag-mgu` in the file `dag-unification.lisp`). It is interesting to point out that the syntactic restrictions needed to define an ACL2 function that uses stobjs are naturally ensured in this case. In the file `dag-unification-examples.lisp` we include some execution examples.

It should be pointed out that this algorithm still has exponential time complexity, but with some technical modifications it is easy to implement a quadratic algorithm. We preferred to use this simpler implementation to describe the main points of this work. Moreover, this unification algorithm is probably the most popular, since the worst exponential case is very uncommon in practice.

3 Comments on the formal verification

Although at the time of this writing the above unification algorithm has not been verified, we think it is interesting to point out some issues encountered during the work completed up to this point.

3.1 Directed acyclic graphs

The first problem we face when dealing with formal verification of this algorithm is to get ACL2 to admit the function definitions. The functions described in the previous section are all in `:program` mode, because some of the auxiliary functions used by the algorithm are not total: if the graph stored in `terms-dag` contains cycles, a function traversing this graph may not terminate. For example, the auxiliary function `occur-check` checks if a variable node is in the term pointed to by an index, traversing the graph and searching for a variable occurrence. This function is not terminating in general[2].

One possible solution would be to traverse the graph taking into account the nodes already visited, thus avoiding cyclic paths. But checking cycles during computation would affect the efficiency of the process[3]. Since one of our main concerns is efficiency, we adopt a different approach: we will reason about our functions as if they were partial, only defined on a specified domain. In our case, this domain will be the set of directed acyclic graphs.

We have defined a function `dag-p`, checking if a list `g` representing a graph (using the conventions explained above when we described the `dag` field of `terms-dag`) is cycle-free. We omit its definition here (see the supporting materials, book `dags.lisp`), but the following are the main properties we proved about it, verifying that `(dag-p g)` is `nil` if and only if there is a cyclic path in `g`:

```
(defthm dag-p-soundeness
  (implies (not (dag-p g)) (cycle-p (one-cyclic-path g) g)))

(defthm dag-p-completeness
  (implies (cycle-p p g) (not (dag-p g))))
```

The definition of `dag-p` and the proof of the above theorems are inspired by [4]. The main difference with that work is that our representation of graphs is different and, more important, that we are ensuring that there are no cyclic path whatever the starting node.

If a graph `g` verifies `dag-p` it is possible to define a well-founded measure justifying that traversing that graph, depth-first and from left to right, starting from a node (or a list of nodes) is a terminating process. This measure, called `measure-rec-dag`, essentially counts the nodes that can be reached from the starting nodes.

Having defined this measure, we can get ACL2 to admit functions with recursive definitions on the structure of terms dag. For example, we can define the following function in the ACL2 logic:

```
(defpun occur-check-l (flg x h g)
  (declare (xargs :domain (dag-p g)
                  :measure (measure-rec-dag flg h g)))
  (if flg
      (let ((p (nth h g)))
        (cond ((dag-bound-p p) (occur-check-l flg x p g))
              ((dag-variable-p p) (= x h))
              (t (occur-check-l nil x (dag-args p) g))))
    (if (endp h)
```

[2] Note also that even if `occur-check` is never called, the unification algorithm can loop. For example, the **Decompose** rule could be applied and infinite number of times.

[3] Nevertheless, this check could be done in a less inefficient way (see subsection 3.3).

```
      nil
    (or (occur-check-l t x (car h) g)
        (occur-check-l nil x (cdr h) g)))))
```

This function has exactly the same body as `occur-check`, the function in `:program` mode used in the unification algorithm described above. Nevertheless, this function is defined in `:logic` mode, so we can reason about it. Note that we used the `defpun` macro (see [3]) for introducing "partial functions". In this case, this macro introduces a theorem equating the term `(occur-check-l flg x h g)` with its body, under the hypothesis `(dag-p g)`, and this theorem is stored as a `:definition` rule. Note also that the argument `g` is not a stobj; since we only need this function from a logical point of view, this is irrelevant and simplifies the reasoning.

This kind of definition is a typical recursive definition on the structure of terms. The function is defined at the same time for terms and for list of terms, by mutual recursion. The argument `flg` is a flag indicating if `h` has to be considered as one single term (i.e., one single pointer) or as a list of terms (i.e., a list of pointers). We have defined (in the logic) all the auxiliary functions that does recursion on the structure of terms in this way, using `defpun` with domain `(dag-p g)`.

3.2 Formal verification and compositional reasoning

Once we can describe the unification algorithm in the ACL2 logic, our next step will be to prove that the unification algorithm computes a unifier of two terms if and only if they are unifiable, and in that case the computed unifier is a most general unifier. We state these properties using the list representation of terms, since the formal theory about first-order terms we developed uses terms represented as lists. But we want to prove these properties about an algorithm working with the dag representation of terms.

As we said earlier, we had formally verified a unification algorithm based on $\Rightarrow_u$, using the list representation of terms. The main (and standard) idea is to use compositional reasoning (as in [7]) to translate the proved properties of this algorithm to the algorithm based on term dags. This means that we have the following proof obligations:

– If the conditions needed to apply one of the rules of transformation are met by a unification problem represented as dags, then the same conditions are met by the corresponding unification problem represented as lists.
– In such case, the unification problem obtained applying the rule of transformation to the dag representation is the same as the unification problem obtained applying the same rule to the list representation.
– The above properties has to be proved assuming the `dag-p` property, among other needed conditions. Therefore, we will need to prove that these properties are preserved in every step of transformation (in particular, when the transformation updates the stobj).

It is worth pointing out that the kind of structural recursion described above for `occur-check-l` is similar to the recursion we used in the definition of an analogous function acting on terms represented as lists. In general, there is a close relationship between definitions on the structure of a term represented as a list and the analogous definition for term dags. Therefore, we hope that the technique of compositional reasoning will be effective here.

3.3 Execution

Although with the assistance of the `defpun` macro we can reason about partial functions only defined on well-formed term dags, the functions we verify are not efficient functions operating on stobjs, because of the expensive `dag-p` check. For the moment execution of partial functions is not supported by `defpun`. We think there are two alternatives to face with this situation, in order to be confident about the function finally used for execution:

- A simple approach would be to define total versions of the functions defined with `defpun`, introducing a counter that is decremented in every recursive call, and failing if this counter reaches 0. Thus, in this case the expensive `dag-p` check is replaced by a simple integer test and this total versions can be used for execution, whenever a suitably large counter is provided. Previously we would have to prove that the `defpun` version and the counter version coincide when their arguments are well-formed term dags and the counter is not exhausted.
  In some cases, a suitable value for the counter could be computed, and we could prove that with that value of the counter the function never fails when acting on term dags. For example, a suitable value for every function that traverses dags (like the one implementing the occur check) could be the length of the array used to store the dag.
- A different approach, similar to [8], would be to "take matters into our hands" and use for execution functions in `:program` mode with the same body as the functions defined with `defpun`. Unlike [8], our `dag-p` check is not irrelevant, so we have to be careful before giving this "dangerous" step.
  In order to explore this possibility, we can use the *guarded domain* option `:gdomain` of `defpun` (see [3]). Essentially, this means that the domain is included as the `:guard` of the witness function used to justify the introduction of the partial function defined by `defpun`. In our case, the guarded domain is given by well-formed term dags. If these guards can be verified, the recursion is closed in the guarded domain. We think that this could be a good reason to be confident about this alternative.

4 Conclusions

We have presented the status of our work in the development of a verified unification algorithm representing terms as directed acyclic graphs. Our idea is to explore the use of single threaded objects to implement efficient data structures for representing first-order terms, allowing fast execution and formal verification. We have seen that it is possible to implement the algorithm (even with the syntactic restrictions enforced by the system on the use of stobjs). We have also described how we can introduce into the ACL2 logic the definitions of functions defined on the structure of a term dag. We are currently in the process of formal verification. If this attempt is successful, we may consider extending this study to other data structures and indexing techniques commonly used in automated reasoning.

References

1. Baader, F. and Snyder, W. Unification theory. *Handbook of Automated Reasoning*, Elvesier Science Publishers, 2001.
2. Boyer R.S. and Moore J S. Single-threaded objects in ACL2. In URL: `www.cs.utexas.edu/-users/moore/publications/acl2-papers.html#Foundations`.
3. Manolios, P. and Moore J S. Partial function is ACL2. In *Second ACL2 Workshop*, Technical Report TR-00-29, Computer Science Department, University of Texas, 2000.
4. Moore, J S. An exercise in graph theory. In *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 5. Kluwer Academic Publishers, 2000.
5. Ruiz-Reina, J., Alonso, J., Hidalgo, M., and Martín, F. Mechanical verification of a rule based unification algorithm in the Boyer-Moore theorem prover. In *AGP'99 Joint Conference on Declarative Programming*, pp. 289–304, 1999.
6. Ruiz-Reina, J., Alonso, J., Hidalgo, M., and Martín, F. Formalizing rewriting in the ACL2 theorem prover. In *AISC'2000 (Fifth International Conference Artificial Intelligence and Symbolic Computation)*, LNCS 1930, pp. 92–103. Springer-Verlag, 2001.
7. Sumners, R. Correctness Proof of a BDD Manager in the Context of Satisfiability Checking. In *Second ACL2 Workshop*, Technical Report TR-00-29, Computer Science Department, University of Texas, 2000.
8. Wilding, M. Using a Single-Threaded Object to Speed a Verified Graph Pathfinder. In *Second ACL2 Workshop*, Technical Report TR-00-29, Computer Science Department, University of Texas, 2000.