Formal Reasoning About Efficient Data Structures: A Case Study in ACL2 ⋆

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

Computational Logic Group
Dept. of Computer Science and Artificial Intelligence, University of Seville
E.T.S.I. Informática, Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain

Abstract. We describe in this paper the formal verification, using the ACL2 system, of a syntactic unification algorithm where terms are represented as directed acyclic graphs (*dags*) and these graphs are stored in a single-threaded object (*stobj*). The use of stobjs allows destructive operations on data (thus improving the performance of the algorithm), while maintaining the applicative semantics of ACL2. We intend to show how ACL2 provides an environment where execution of algorithms with efficient data structures and formal reasoning about them can be carried out.

1 Introduction

The ACL2 system includes a programming language, a logic for formal reasoning about the properties of the functions defined in the language, and a theorem prover supporting mechanized reasoning in the logic. The ACL2 programming language is an extension of an applicative subset of Common Lisp and the logic is a first-order logic with equality, without quantifiers (all the formulas are implicitly universally quantified).

Since the programming language is applicative, logical arguments about the correctness and termination of algorithms are made as they are in ordinary mathematics, without the complications incured by consideration of state. Notwithstanding, it is possible to declare some objects in the language as single-threaded objects (in the sequel, *stobjs*) and perform destructive updates on them. When an object is declared to be single-threaded, ACL2 enforces certain syntactic restrictions on its use, ensuring that in every moment only one copy of the object is needed. With these restrictions, the destructive updates are consistent with the applicative semantics of ACL2. Using stobjs we can combine efficient imperative implementations with the semantic of functional languages to reason about these implementations.

In this paper we present a case study where we use ACL2 to implement and verify a unification algorithm. A standard approach in the implementation of

⋆ This work has been supported by project TIC2000-1368-C03-02 (Ministry of Science and Technology, Spain) and FEDER funds.

unification is to represent terms as directed acyclic graphs (*dags* in the following), allowing some amount of structure sharing; in this way, it is not needed to build new terms during the unification process, but merely update (destructively) the graph, thus improving the performance of the algorithm. In our implementation, the dags will be stored using a stobj.

To achieve the formal proof, we follow the well-known methodology of *compositional reasoning*. As a first step, we reason about unification at a very abstract level, without entering in details related to the control of the algorithm or the data structures used. By stepwise-refinement, we finally obtain the proof of the desired properties of our concrete unification algorithm.

Another interesting point in this case study is the use of a new feature in ACL2 (the `mbe` feature) that associates an "executable body" with a (possibly different) "logical body". This association will be allowed by the system after proving that on the intended domain of the function, the executable body and the logical body are equal. We describe this new feature of ACL2, and explain how it can be used to improve the execution efficiency of the verified unification algorithm.

Although we will not give an introduction to ACL2, we will comment the relevant questions in passing, when needed. An excellent introduction to ACL2 is [5]. A detailed description of the system can be found in the manual, available in [6]. We will assume the reader familiar with Common Lisp. Due to the lack of space, we will not give here details about the proofs obtained and some function definitions will be omitted. We urge the interested reader to consult [11], where the complete development (with a detailed description) is available.

2 Dag unification

We briefly review some basic concepts about (syntactic) unification, a fundamental process upon which many methods of automated deduction are based. A complete description of the theory of unification can be found in [2].

An *equation* is a pair of first-order terms, denoted as $t_1 \approx t_2$, and a *system of equations* is a finite set of equations. A substitution σ is a *solution* of $t_1 \approx t_2$ if $\sigma(t_1) = \sigma(t_2)$ and it is a solution of a system of equations S if it is a solution of every equation in S. Given two substitutions σ and δ, we say that σ is more general than δ if there exists a substitution γ such that $\delta = \gamma \circ \sigma$, where $\circ$ denotes functional composition. We say that a solution of S is a *most general solution* if it is more general than any other solution of S. Two terms t_1 and t_2 are *unifiable* if there exists a solution (called *unifier*) of the system $\{t_1 \approx t_2\}$. A *most general unifier* (*mgu* in the sequel) of t_1 and t_2 is a most general solution of that system. A *unification algorithm* is an algorithm that decides whether two given terms are unifiable, and in that case it returns a most general unifier.

Essentially, the unification algorithm we have implemented is based on the relation $\Rightarrow_u$ given by the set of transformation rules presented in Figure 1 (known as the *Martelli-Montanari transformation system*). This system acts on pairs of systems of equations of the form $S; U$. Intuitively, the system S can be seen as a

Delete:	$\{t \approx t\} \cup R; U \Rightarrow_u R; U$
Occur-check:	$\{x \approx t\} \cup R; U \Rightarrow_u \bot$ 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:	$\{f(s_1, ..., s_n) \approx f(t_1, ..., t_n)\} \cup R; U \Rightarrow_u \{s_1 \approx t_1, ..., s_n \approx t_n\} \cup R; U$
Clash:	$\{f(s_1, ..., s_n) \approx g(t_1, ..., t_m)\} \cup R; U \Rightarrow_u \bot$ 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. Martelli–Montanari transformation system

set of pairs of terms to be unified, and the system U as a (partially) computed unifier[1] (we say that the pair $S; U$ is a *unification problem*). The symbol $\bot$ represents unification failure. Starting with the pair of systems $S; \emptyset$, these rules can be (non-deterministically) applied iteratively, until either a pair of systems of the form $\emptyset; U$ or $\bot$ is obtained. It can be proved that this process must terminate and that S has a solution if and only if $\bot$ is not derived; in that case U is a most general solution of S. Thus, a unification algorithm can be designed choosing an strategy to apply the rules, starting with the pair of systems $\{t_1 \approx t_2\}; \emptyset$, where t_1 and t_2 are two given input terms.

In [10] we had defined and verified a unification algorithm based on this set of transformation rules, as part of an ACL2 library with formal proofs of the lattice-theoretic properties of first-order terms. In that library, terms are represented in prefix notation, using lists (except variables, which are represented by atomic objects). For example, the term $f(x, g(y), h(x))$ is represented by the list `(f x (g y) (h x))`. Substitutions are represented as association lists, and systems of equations as lists of dotted pairs of terms. In the sequel, this representation of terms and substitutions in prefix form, using lists, will be referred to as *prefix representation* or *prefix notation*.

Using the prefix representation, a unification algorithm may be inefficient in some situations. Consider, for example, the following standard parameterized unification problem, which we will call U_n:

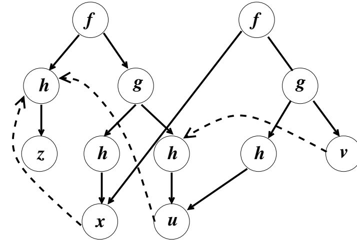
$$p(x_n, \ldots, x_2, x_1) \approx p(f(x_{n-1}, x_{n-1}), \ldots, f(x_1, x_1), f(x_0, x_0))$$

A mgu of this problem is $\{x_1 \mapsto f(x_0, x_0), x_2 \mapsto f(f(x_0, x_0), f(x_0, x_0)), \ldots\}$, which maps each variable x_i to a complete binary tree of height i. This mgu can be obtained by repeatedly applying the **Eliminate** rule of $\Rightarrow_u$. If we use the prefix representation of terms, it will be necessary to reconstruct the instantiated system of equations, each time the rule is applied.

The standard approach to deal with this problem is to use *term dags* where variables are shared. For example, the following graph represents the equation

[1] We will identify a system of equations of the form $\{x_1 \approx t_1, \ldots, x_n \approx t_n\}$, where the x_i are variables, with the substitution $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. If none of the x_i appear in any of the t_j, we say that the system is in *solved form*. Note that every system in solved form is a mgu of itself.

$f(h(z), g(h(x), h(u))) \approx f(x, g(h(u), v))$. 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 term dag with the whole term. Note also that there is a certain amount of *structure sharing*, at least for the repeated variables[2]:



To implement a unification algorithm with this term representation, the main idea is never to 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 the graph above, 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 $f(x, g(h(u), v))$.

3 An ACL2 implementation

The implementation described here is based on the Pascal implementation given in section 4.8 of [1]. The main difference is that instead of a record with pointers, we use a single-threaded object. This stobj is a structure called `terms-dag` with only one field: an array called `dag` (whose size can be modified dynamically). This array is used to store the unification problem in dag form:

```
(defstobj terms-dag
  (dag :type (array t (0)) :resizable t))
```

The effect of this ACL2 event is to introduce the stobj `terms-dag` and its associated recognizers, creator, accessors, updaters, and length and resize functions of the array field. In particular, given an index i (a natural number) corresponding to a cell of the `dag` array, the expressions `(dagi` i `terms-dag)` and `(update-dagi` i v `terms-dag)` access and update (with value v) respectively the i-th cell of the `dag` array. These operations are done in constant time and the update is destructive. Nevertheless, from the logical point of view, the array can be thought as a list, with an applicative semantic (that is, as if in every update a

[2] It should be remarked that this is simply one possible representation in which only variables are shared; this is not the most compact representation, but the one that serves as the basis of the verified unification algorithm.

new object were created) . This is possible due to the fact that in ACL2, the use of stobjs is syntactically restricted, ensuring that in every moment only one copy of the object is needed. Roughly speaking, these syntactic restrictions enforce that the only references to the stobj are done via its name (`terms-dag`, in this case). See [4, 6] for further information about *stobjs* in ACL2 and the restrictions on its use.

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 successors of one 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 root node of the term to 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 the root nodes of $t_1, \ldots, t_n$.

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

0	1	2	3	4	5	6
(EQU . (1 9))	(F . (2 4))	(H . (3))	(Z . T)	(G . (5 7))	(H . (6))	(X . T)

7	8	9	10	11	12	13	14
(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 term whose root node is stored in the corresponding array cell. Taking advantage of this idea, we can define a function (called `dag-transform-mm-st`, figure 2) that applies one step of the transformation relation $\Rightarrow_u$ to a unification problem stored in `terms-dag`.

Let us precise about the behavior of `dag-transform-mm-st`. In addition to the stobj, this function receives as input a (non-empty) system of equations `S` to be unified and a partially computed substitution `U`. The key point here is that `S` and `U` *only contain indices* pointing to the terms stored in `terms-dag`. In particular, `S` is a list of pairs of indices, and `U` is a list of pairs of the form `(`x `.` n`)` where x is a variable symbol and n is the index of the node for which the variable is bound (we say that `S` is an *indices system* and `U` an *indices substitution*). Depending on the pair of terms pointed to by the first equation of `S`[3], one of the rules of $\Rightarrow_u$ is applied. The function returns a multivalue with the following components, obtained as a result of the application of one step of transformation: the resulting indices system of equations to be solved, the resulting indices substitution, a boolean (if $\perp$ is obtained, this value is `nil`) and

[3] Note that the indices of the selected equation are *dereferenced* using the function `dag-deref-st`, which follows a chain of instantiations until it reaches an unbound variable or non-variable node.

```
(defun dag-transform-mm-st (S U terms-dag)
  (declare (xargs :stobjs terms-dag))
  (let* ((ecu (car S))
         (t1 (dag-deref-st (car ecu) terms-dag))
         (t2 (dag-deref-st (cdr ecu) terms-dag))
         (R (cdr S))
         (p1 (dagi t1 terms-dag))
         (p2 (dagi t2 terms-dag)))
    (cond
     ((= t1 t2) (mv R U t terms-dag))
     ((dag-variable-p p1)
      (if (occur-check-st t t1 t2 terms-dag)
          (mv nil nil nil terms-dag)
       (let ((terms-dag (update-dagi t1 t2 terms-dag)))
         (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))
     ((not (eql (dag-symbol p1)
                (dag-symbol p2)))
      (mv nil nil nil terms-dag))
     (t (mv-let (pair-args bool)
                (pair-args (dag-args p1) (dag-args p2))
                (if bool
                    (mv (append pair-args R) U t terms-dag)
                  (mv nil nil nil terms-dag)))))))
```

Fig. 2. One step of transformation

the stobj terms-dag. Note that only when **Eliminate** is applied, the stobj is updated, causing the corresponding variable to point to the corresponding term.

With dag-transform-mm-st as its main component, we can define the unification algorithm. In short, this function, called dag-mgu, receives as input two terms in prefix form; after storing these terms as directed acyclic graphs in the stobj (previously resizing the dag array properly), it iteratively applies the function dag-transform-mm-st until either non-unifiability is detected or there are no more equations to be solved. In this last case, the returned substitution (in prefix form) is built from the final contents of dag, following the pointers of the instantiated variables. The following are two examples obtained with dag-mgu. Note that the function returns two values: the first one is a boolean indicating whether the terms are unifiable or not, and, in case of unifiability, the second is the mgu.

```
ACL2 !>(dag-mgu '(f (h z) (g (h x) (h u))) '(f x (g (h u) v)))
(T ((V . (H (H Z))) (U . (H Z)) (X . (H Z))))
ACL2 !>(dag-mgu '(f y x) '(f (k x) y))
(NIL NIL)
```

It is worth pointing out that the syntactic requirements needed to ensure the single-threadedness of the ACL2 functions that use stobjs are naturally met in this algorithm. See [11] for the definitions of all the auxiliary functions used. Since the ACL2 language is a subset of Common Lisp (and we have verified guards[4]), the defined algorithm can be compiled and executed in every compliant Common Lisp, with the appropriate ACL2 files loaded.

4 The formal properties of the unification algorithm

Once defined the function `dag-mgu`, we use the ACL2 logic and its theorem prover to formally establish that it computes the most general unifier of two terms if and only if the terms are unifiable:

```
(defthm dag-mgu-completeness
  (implies (and (term-p t1) (term-p t2)
                (equal (instance t1 sigma) (instance t2 sigma)))
           (first (dag-mgu t1 t2))))

(defthm dag-mgu-soundness
  (implies (and (term-p t1) (term-p t2)
                (first (dag-mgu t1 t2)))
           (equal (instance t1 (second (dag-mgu t1 t2)))
                  (instance t2 (second (dag-mgu t1 t2))))))

(defthm dag-mgu-most-general-solution
  (implies (and (term-p t1) (term-p t2)
                (equal (instance t1 sigma) (instance t2 sigma)))
           (subs-subst (second (dag-mgu t1 t2)) sigma)))
```

The function `instance` defines the application of a substitution to a term, and the predicate `subs-subst` defines the relation "more general than" between substitutions. The predicate `term-p` recognizes those ACL2 objects that represent first-order terms in prefix notation. Note that the basic theory used to state the properties is built on the terms represented in prefix notation. For a detailed description of this theory, see [10]. Also the input and the output of the function `dag-mgu` are terms and substitutions in prefix notation. But it has to be emphasized that internally, the main process is carried out on term dags.

[4] The notion of *guard* of a function will be explained in section 6.

The first theorem, `dag-mgu-completeness`, establishes that the algorithm returns `t` (as its first value) if the input terms are unifiable[5]. The theorem `dag--mgu-soundness` establishes that in that case it returns (as its second value) a unifier of both terms. Finally, the theorem `dag-mgu-most-general-solution` establishes that the returned substitution is more general than any other unifier of both terms. These three proved theorems constitute a formal proof of the correctness of the algorithm.

5 Comments about the proof

In this section, we give an overview of the proof process. To emphasize the "compositional reasoning" methodology followed, we have structured it in subsections. First we begin with the subsections describing properties of the algorithm at a more abstract level. These abstract properties can be gradually concretized to finally obtain the theorems shown in the previous section.

5.1 Reasoning about the reduction $\Rightarrow_u$

One step of transformation of $\Rightarrow_u$ is determined by the rule applied and the equation selected. To formalize this intuitive idea in ACL2, we define $\Rightarrow_u$ by means of *operators*. In this context, an operator is a dotted pair of the form (*name* . *i*) where *name* is one of the rule names in figure 1 and i is a natural number, corresponding to the i-th equation of the system. Thus, the transformation $\Rightarrow_u$ can be seen as applying one operator to a unification problem. This operator can be applied whenever the conditions of the particular rule applied are met. For example, the operator `(eliminate . 3)` can be applied to a unification problem if its third equation is of the form $x \approx t$ and x does not occur in t. The following two functions formalize this idea in ACL2:

- `(unif-legal-pr upl op)`, checking the conditions needed to apply a given operator `op` to a unification problem `upl` (in prefix notation).
- `(unif-reduce-one-step-pr upl op)`, returning the transformed unification problem (in prefix notation) after applying `op` to `upl`.

With this operator-based representation we proved in ACL2 the main properties of $\Rightarrow_u$. That is: a) the set of solutions of a unification problem is preserved in each step, b) if the second system of a unification problem is in solved form, then the transformed unification problem has its second system in solved form, and c) the transformation relation is terminating. These properties are more naturally proved with terms represented in prefix form, and this allows us to reuse part of the theory developed in [10] for the verification of the applicative unification algorithm.

[5] Note that the variable `sigma`, although implicitly universally quantified, can be seen as existentially quantified, since it only appears in the hypothesis of the theorem.

Having proved the main properties of one-step transformations, we can easily extend these properties to finite sequences of transformations. In particular we prove that if $\{t_1 \approx t_2\}; \emptyset \overset{*}{\Rightarrow}_u \emptyset; \sigma$, then σ is a *mgu* of t_1 and t_2, and if $\{t_1 \approx t_2\}; \emptyset \overset{*}{\Rightarrow}_u \bot$, then t_1 and t_2 are not unifiable. Note that in our formalization, a sequence of transformation can be identified with a list of (legal) operators.

It is remarkable that these results do not deal with control or data structures issues: to prove the correctness of a concrete unification algorithm, it suffices to show that the actions of the algorithm can be simulated by a finite sequence of transformations w.r.t. $\Rightarrow_u$. That is the main advantage of rule-based specifications: they allow to prove the essential properties of the procedure without the burden of technical implementation issues.

5.2 Dags and well-formedness conditions

In order to translate the main properties of $\Rightarrow_u$ to our implemented algorithm, we have to relate the information stored in the `terms-dag` stobj with the terms in prefix notation it may represent. In general, not every possible contents of the `dag` array represent first-order terms. The main reason is that the graph could contain cycles, and in that case, no first-order term is represented by the cells of the array.

This means that we have to define predicates to recognize the properties needed to ensure that the array contents represent a first-order term; the main of those properties is acyclicness, ensuring that the graph stored in the `dag` array is actually a dag. Some other well-formedness properties are also needed (for instance the sharing of variables).

Another important reason why these well-formedness conditions are needed has to do with the restrictions imposed by the ACL2 logic in its principle of definition: new function definitions are admitted as axioms in the logic only if there exists a measure in which the arguments of each recursive call decrease with respect to a well-founded relation, ensuring in this way that the function terminates on all inputs (and consequently no inconsistencies are introduced by new function definitions). For example, a function implementing "occur-check" (looking for the occurrence of a given variable in a term) may not terminate if the graph stored in the array contains cycles. The same happens with dereferencing or even with the function that iteratively applies `dag-transform-mm-st`. Thus, these functions require an explicit check to verify that the stobj does indeed represent an acyclic graph, ensuring their termination. We will comment more about this point in section 6.

For these reasons, we have developed a library of results about directed acyclic graphs. For example, this library contains the definition of the function `dag-p`; this function checks that a given graph (stored following the conventions described in section 3) does not contain cycles. It is implemented as a standard depth-first search algorithm, looking for cycles in the graph. The following theorems establish that a graph `g` verifies the `dag-p` condition if and only if does not contain cycles:

```
(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))))
```

Some other general definitions and results about dags are part of this library. See [11] for details. Having `dag-p` as its main auxiliary function, we can define a function checking the well-formedness conditions of a unification problem given in dag form: `(well-formed-upl dag-upl)` is true if and only if `dag-upl` is a three-element list such that its first element is an indices system, the second is an indices substitution and the third is an acyclic term graph with shared variables. In the following, by *well-formed dag unification problem* we mean an ACL2 object that satisfies `well-formed-upl`. Every well-formed dag unification problem has a unification problem in prefix notation associated.

One technical issue is worth pointing out. The main advantage in the reasoning about stobjs is that from the logical point of view, an array field of a stobj is like a list whose elements are the contents of the array[6]. For this, we can reason about dags as if the graph were a list, instead of an array field of a stobj. For example, the function `dag-p` above is defined on lists. In addition to simplifying the formulation of the theorems, this allows to define some properties about graphs following the usual "`car-cdr`" recursion style. This style would not be allowed if the definition were on the stobj, due to the syntactic restrictions imposed by ACL2 on stobjs. Of course, those functions that are going to be executed have to be defined on the stobj; but we define a "list version" for reasoning and then translate the main properties proved to the "stobj version" (for example, we define a function `dag-p-st` on the `terms-dag` stobj, logically equivalent to `dag-p`).

5.3 Compositional reasoning

Since our implemented algorithm acts on terms represented as dags, we must now define in ACL2 the behavior of the relation $\Rightarrow_u$ acting on well-formed dag unification problems. As in subsection 5.1, we adopt an operator based approach. That is, we define the following two functions:

- `(unif-legal-d dag-upl op)`, checking the conditions needed to apply a given operator `op` to a dag unification problem `dag-upl`. These conditions are similar, for each rule, to the conditions checked by the function `dag-transform-mm-st` (figure 2) before applying a transformation.

[6] For example, the accessors and updaters are logically equivalent to the list functions `nth` and `update-nth` which, respectively, access and update the contents of a list.

- `(unif-reduce-one-step-d dag-upl op)`, returning the transformed dag unification problem obtained after applying `op` to `dag-upl`. These transformations are similar, for each rule, to the transformations performed by the function `dag-transform-mm-st`.

Instead of proving the properties of these transformations reasoning directly with the definitions of the above functions (which can be difficult due to the more sophisticated data structures used), we can translate the properties proved for the transformations on the prefix representation, using compositional reasoning. More precisely, denoting as UPL_p the set of unification problems represented in prefix form, and as UPL_d the set of well-formed dag unification problems, the key point is to prove that the following diagram commutes:

$$\begin{array}{ccc} UPL_p & \xrightarrow{\Rightarrow_{u,p}} & UPL_p \\ dp \uparrow & & dp \uparrow \\ UPL_d & \xrightarrow{\Rightarrow_{u,d}} & UPL_d \end{array}$$

Here dp is a function such that given a well-formed dag unification problem, it returns the corresponding unification problem in prefix form; $\Rightarrow_{u,p}$ and $\Rightarrow_{u,d}$ denote, respectively, the relation $\Rightarrow_u$ defined on the prefix representation and on the dag representation. The commutativity of the above diagram is formally established in ACL2 by the following theorem (the function `upl-as-pair-of-systems` plays the role of the function dp in the diagram):

```
(defthm conmutativity-of-diagram-prefix-dag
  (implies (and (well-formed-upl dag-upl)
                (unif-legal-d dag-upl op))
     (and (well-formed-upl (unif-reduce-one-step-d dag-upl op))
          (unif-legal-pr (upl-as-pair-of-systems dag-upl) op)
          (equal (upl-as-pair-of-systems
                       (unif-reduce-one-step-d dag-upl op))
                 (unif-reduce-one-step-pr
                       (upl-as-pair-of-systems dag-upl) op)))))
```

This theorem establishes that:

- The well-formedness property of dag unification problems is preserved by the transformation rules.
- If the conditions needed to apply a rule to a well-formed dag unification problem are met, then the conditions required to apply the same rule to the corresponding unification problem in prefix form are also met.
- In that case, the transformed unification problem obtained applying the rule to the prefix representation is the same as the unification problem in prefix form corresponding to the dag unification problem obtained applying the same rule to the dag representation.

These properties allow us to easily translate the main properties described in subsection 5.1 to this more efficient data structure. In particular, it can be proved that we can obtain a most general unifier of two terms by exhaustively applying the rules of transformation on its dag representation.

5.4 Final steps

Now that we have all the main pieces needed for the verification of the algorithm, we proceed as follows:

- First, we define a function that, given two terms in prefix form, returns the corresponding dag unification problem. We must prove that this dag unification problem is well-formed. This result turned out to be one of the hardest part of all the verification process.
- Second, we show that the transformations performed by our unification algorithm can be simulated by a sequence of transformations of $\Rightarrow_{u,d}$. That is, we deal with the specific control (or selection strategy) of the algorithm. In our case, we always select the first equation, but any other strategy could work. In terms of operators, this means that we have to explicitly give a sequence of operators that, iteratively applied to the initial dag unification problem, obtains the same final dag unification problem as the implemented algorithm. Note that even though operators are used for defining the transformation relation, these are an intermediate concept used for reasoning, but not used by the unification algorithm.
- Finally, since the above properties are established for the "dag-list version" of the algorithm, we translate the properties to the executable "dag-stobj version", finally proving the formal properties presented in section 4.

5.5 Quantifying the proof effort

The ACL2 theorem prover supports mechanized reasoning in the ACL2 logic, being particularly well-suited for obtaining automated proofs based on induction and simplification. The prover is automatic in the sense that once `defthm` is invoked, the user can no longer interact with the system. However, in a deeper sense, the system is interactive: usually, when proving non-trivial theorems, the user has to guide the prover by adding lemmas and definitions (used in subsequent proofs as rewrite rules during the simplification process), or giving some hints to the `defthm` command, such as the scheme for a induction proof.

A typical ACL2 proof effort consists of formalizing the problem in the logic and helping the prover to find a preconceived proof by means of a suitable set of rewrite rules. These rules can be found by inspecting the failed proofs: when the proof attempt deviates from the expected proof, usually a lemma is needed to deal with that part of the proof by simplification. This methodology produces a collection of lemmas (and definitions) leading the prover to the proof of the main result. Some of these lemmas are interesting by themselves and can be reused later in other parts of the development. This way of interacting with the system

is called "The Method" by the authors of the system and it is explained in detail in [5]. We followed "The Method" in this case study.

The table below shows some quantitative information about the proof effort, the number of definitions and theorems needed during the different stages of the verification process (we have not included in the table data of the basic theory about first-order terms):

Phase	Definitions	Theorems
Properties on the prefix representation	24	81
Acyclic graphs	37	95
Diagram commutativity	39	66
Storing the initial terms in the graph	34	208
Properties of the implemented algorithm	43	76
Total	177	703

These numbers may give an idea of the complexity of the formalized theories and the degree of automation of the proofs obtained. We should say that most of the lemmas needed during the first phase were already proved in [10]. It is also remarkable (and somewhat surprising) the number of theorems needed to prove the properties of a function that stores the initial terms as directed acyclic graphs.

6 Execution of the algorithm

As we have already said, ACL2 is a logic of total functions. That is, a proof of the termination of the function on all possible inputs is required for the definition to be accepted by the prover. In some cases, this means that a definition must include in its body an explicit check on their arguments, ensuring its termination. This check may seriously affect the execution performance of the function. Until the current ACL2 version 2.7, this was a weakness of the system that appeared when dealing with functions that only terminate on their intended domain, but not for every possible input. In the next ACL2 release, the new `mbe` feature (which stands for "must be equal") overcomes that weakness: it allows to assign to a function an alternate "executable body" to that provided for the logic.

We use `mbe` in this work, avoiding (for execution) the expensive well-formedness checks that are needed in the logical definitions of some of the functions of the dag unification algorithm. We explain this with an example. The following is the definition of the function `dag-solve-upl-st`, which iteratively applies steps of transformation to a given unification problem, until either there are no equations to be solved or unsolvability is detected:

```
(defun dag-solve-upl-st (S U bool terms-dag)
  (declare
   (xargs :stobjs terms-dag
          :guard (well-formed-upl-st S U terms-dag)
```

```
        ...)
 (MBE
  :logic
  (if (well-formed-upl-st S U terms-dag)
      (if (or (not bool) (endp S))
          (mv S U bool terms-dag)
        (mv-let (S1 U1 bool1 terms-dag)
                (dag-transform-mm-st S U terms-dag)
                (dag-solve-upl-st S1 U1 bool1 terms-dag)))
    (mv nil nil nil terms-dag))
  :exec
  (if (or (not bool) (endp S))
      (mv S U bool terms-dag)
    (mv-let (S1 U1 bool1 terms-dag)
            (dag-transform-mm-st S U terms-dag)
            (dag-solve-upl-st S1 U1 bool1 terms-dag)))))
```

This `defun` defines the function in the logic using the body given by the `:logic` key argument, but when the function is evaluated on arguments of its intended domain (this intended domain is given by the `:guard` key) then the body given by the `:exec` key argument is used. The logical body needs an explicit check performed by the function `well-formed-upl-st`[7], in order to ensure its termination (which it is not trivial to prove). This condition is expensive, since acyclicness is checked; moreover, if we use this logic body for execution, this expensive check would be evaluated *in every recursive call*, making execution of the function impractical.

From the logical point of view (`mbe :logic` *logicbody* `:exec` *execbody*) is equal to *logicbody*, so *execbody* is ignored for reasoning. But when the function is evaluated on its intended domain, the underlying Common Lisp uses the (hopefully) more efficient *execbody*. The "intended domain" is specified in ACL2 by its *guard*, and the proof obligations generated by the guard verification mechanism ensure the soundness of using the executable body instead of the logic body.

Guards in ACL2 are used to specify the intended domain of a function. Although this specification is actually ignored by the logic, the guard verification mechanism allows to evaluate the function directly in Common Lisp. If the guards of a function are verified, then it is ensured that when the function is evaluated on arguments satisfying its guard, then all subsequent function calls during that evaluation will be on arguments satisfying the guard of the called function. The proof obligations generated by the guard verification mechanism ensure this property. Since the primitive Common Lisp functions of ACL2 has guards consistent with the Common Lisp specification, an ACL2 function with its guards verified is Common Lisp compliant and can be evaluated, on arguments satisfying its guard, directly in the underlying Common Lisp.

[7] This function is the "dag-stobj version" of the function `well-formed-upl` described in subsection 5.2.

The guard of the function `mbe` specifies that its two arguments are equal. Thus, when a function that uses `mbe` has its guard verified, then it is sound to use the executable body for execution, whenever the input arguments are in the intended domain specified by the guard.

In addition to the above function, some of the auxiliary functions used by the implemented algorithm `dag-mgu` are defined in a similar way, using `mbe` (for example, occur checking or dereferencing). Since we have verified the guards of `dag-mgu` (and therefore the guards of all the functions used by the algorithm), all the expensive well-formedness checks are ignored when calling the function `dag-mgu` on two ACL2 objects representing terms in prefix form (the guard of `dag-mgu`). Note that the guard of the main top level function is very simple, and since guards are verified, it is not needed to evaluate the more expensive guards of its auxiliary functions in subsequent calls.

We have tested the verified unification algorithm using the parameterized unification problem U_n (presented in section 2). We compare its performance with the applicative algorithm defined in [10]. The problem U_n is particularly well suited for the dag unification algorithm, since it is already in *dag solved form*. For that reason we also test the algorithm on the problem U_n^{-1}, where the equations of U_n are processed in reverse order. The following table summarizes the results obtained[8]:

	Pref. U_n		Dag U_n		Pref. U_n^{-1}		Dag U_n^{-1}		Quadratic U_n^{-1}	
n	Time	Space	Time	Space	Time	Space	Time	Space	Time	Space
20	19	376839	ϵ	3	7	49160	4	10	ϵ	20
22	78	1638409	ϵ	4	21	196654	18	11	ϵ	22
24	–	–	ϵ	5	90	786444	72	12	ϵ	26
1000	–	–	0.2	151	–	–	–	–	15	195
5000	–	–	2	781	–	–	–	–	61	945

It can be observed that the space complexity is much better in the dag implementation than in the applicative implementation in all cases. The time performance it is also much better with U_n, and about 25% faster with U_n^{-1}. Note that the definition of the algorithm using `mbe` is essential for obtaining this time performance, since, as we have said, the logical definition of the algorithm is impractical for execution.

The column labeled Dag U_n^{-1} reveals that the implemented algorithm has still exponential time complexity. The problem is that some operations, like the occur check, may traverse terms exponential in size. Nevertheless, the implemented algorithm is the most often used in practice, since that exponential behavior is not usual. Anyway, we have implemented a quadratic version of the dag unification algorithm, introducing a few technical modifications to the verified algorithm. We also include in the table the tests for this improved version, which it is much

[8] Tested on an AMD© 2200XP processor, with 512Mb RAM. The data are obtained with the function `time` of Clisp. The dash denotes that either an output is not obtained in reasonable time, or that a stack overflow occurs. ϵ stands for a quantity less than 0.01. Numbers bigger than 1 are rounded to the nearest integer.

faster, being able to solve U_n^{-1} for $n = 5000$. For the moment, this quadratic implementation is not formally verified.

7 Conclusions

We have presented a case study in ACL2, where we verify a unification algorithm acting on term dags, implemented using ACL2 single-threaded objects. We urge the interested reader to consult the complete development in [11]. The main features of this case study are:

- The formal verification of an executable algorithm that uses efficient data structures.
- The methodology used: from a rule-based specification of the algorithm, we prove its more abstract properties. The final properties of the algorithm can be seen as an optimization process, using compositional reasoning.
- The use of the new `mbe` feature of ACL2, that permits to associate to a function some "executable body" that can be different from its "logical body".

The intuitive idea that algorithms employing more complex data structures or more sophisticated control structures require more effort in verification is supported by the table of subsection 5.5. These data contrast with the effort needed in the verification of the same algorithm using a prefix representation of terms [10]. In that work, we needed 19 definitions and 129 theorems, and in this case we needed 177 definitions and 703 theorems. Anyway, this additional verification effort has resulted in the development of a number of ACL2 files that could be used in other formalizations (for example, the theory about directed acyclic graphs).

As for related works, unification algorithms have been the center of several formalizations. In particular, formal proofs of the correctness of a unification algorithm have been given in LCF [8], Coq [9] and ALF [3]. Although these works are related to ours, the logic used is quite different and, more important, their main concern is not efficiency or the data structures used.

Other related work is done by Mehta and Nipkow [7], who have recently developed in Isabelle/HOL a general framework for reasoning about programs that use pointers. As a non-trivial case study, they present a proof of the correctness of the Schorr–Waite graph marking algorithm. This work is more general than ours, since all the reasoning about pointers that we do is especifically devoted to the results needed by the algorithm. Moreover, the logics used are different: in [7], a Hoare logic for pointer programs is embedded in Isabelle/HOL, whereas we are using the ACL2 logic for reasoning about ACL2 functions that can be directly executed in any compliant Common Lisp. Nevertheless, some of the techniques used in [7] are similar to ours: for example, what they call abstraction (mapping low level structures in the heap to higher level concepts) is similar to what we do when we first reason about the main properties of the algorithm using the prefix representation of terms (a higher level representation) and then we translate them to the algorithm that uses dags (a lower level representation).

As for further work, we already pointed out at the end of subsection 6 that we can introduce some technical improvements in order to make the verified algorithm run in quadratic time. We also plan to verify this improved algorithm.

Finally, note that although our main concern is an efficient and formally verified algorithm, we do not prove theorems about the efficiency of the algorithm. Although reasoning about complexity of algorithms in the ACL2 logic is (in principle) possible, we think that it could be much more difficult than reasoning about the correctness of the algorithm, mainly due to the need of formalizing the "big-O notation" (and its asymptotic character) in the ACL2 logic.

Acknowledgments

Part of this work was done during a visit of the first author to the Computer Science Department of the University of Texas at Austin. We would like to thank the ACL2 group in Austin, especially to J Moore and Matt Kaufmann, for their support, and for introducing `mbe` in ACL2.

References

1. Baader, F. and Nipkow, T. *Term Rewriting and All That.* Cambridge University Press, 1998.
2. Baader, F. and Snyder, W. Unification theory. *Handbook of Automated Reasoning*, Elsevier Science Publishers, 2001.
3. Bove, A. Programming in Martin-Lf Type Theory: Unification - A non-trivial Example. *Licentiate Thesis*, Department of Computer Science, Chalmers University of Technology, 1999.
4. Boyer R.S. and Moore J S. Single-threaded objects in ACL2. In *Practical Aspects of Declarative Languages*, LNCS 2257, pages 9–27, Springer–Verlag, 2002.
5. Kaufmann, M., Manolios, P. and Moore, J S. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.
6. Kaufmann, M. and Moore, J S. ACL2 Version 2.7, 2002.
Homepage: `http://www.cs.utexas.edu/users/moore/acl2/`
7. Mehta, F. and Nipkow, T. Proving Pointer Programs in Higher-Order Logic . to be presented at *CADE-19*, 2003.
8. Paulson, L. Verifying the unification algorithm in LCF. *Science of Computer Programming*, 5, 1985.
9. Rouyer, J. Dveloppement de l'algorithme d'unification dans le calcul des constructions avec types inductifs. Tech. Rep. 1795, INRIA Lorraine, 1992 (in french).
10. Ruiz–Reina, J.L., Alonso, J.A., Hidalgo, M.J. and Martín, F.J. A theory about first–order terms in ACL2 In *Third ACL2 Workshop*, Grenoble, 2002.
11. Ruiz–Reina, J.L., Alonso, J.A., Hidalgo, M.J. and Martín, F.J. A verified dag unification algorithm in ACL2, 2002.
Available at `http://www.cs.us.es/~jruiz/unificacion-dag`