Formal Correctness of a Quadratic Unification Algorithm

José–Luis Ruiz–Reina, Francisco–Jesús Martín–Mateos, José–Antonio Alonso and María–José Hidalgo
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
E-mails: {jruiz,fjesus,jalonso,mjoseh}@us.es

Abstract. We present a case study using ACL2 [5] to verify a non-trivial algorithm that uses efficient data structures. The algorithm receives as input two first-order terms and it returns a most general unifier of these terms if they are unifiable, failure otherwise. The verified implementation stores terms as directed acyclic graphs by means of a pointer structure. Its time complexity is $O(n^2)$ and its space complexity is $O(n)$, and it can be executed in ACL2 at a speed comparable to a similar C implementation. We report the main issues encountered to achieve this formally verified implementation.

1. Introduction

It is generally accepted that there is a trade-off between the efficiency of an implementation and the simplicity of its formal correctness proof: having more sophisticated control and data structures increases the effort needed to prove its correctness. That is the reason why most of the proofs about well-known algorithms that have been carried out using theorem provers are done reasoning about non-efficient naive implementations.

Nevertheless, the ACL2 system [5] has already been demonstrated capable of efficient implementations of microarchitectural level processor models (see [4], for example) that can be executed at C–like performance. In this way, in addition to having a high-speed simulation model one has the additional benefit of being able to prove formal properties of that model. The core of the implementation is a "next state" function that receives as input a data structure representing the current state of the machine and returns an updated machine state. A single-threaded object (*stobj* in the following) is usually employed to represent the machine state. These data structures in ACL2 allow constant time access and destructive updates, while maintaining an applicative semantics for reasoning about it.

In light of this, and given our previous experience in the development of formal theories related to symbolic computation systems [8], we decided to apply ACL2 to obtain a formally verified and efficient

implementation of some non-trivial algorithm in this area. Our goal was twofold: compare the execution efficiency obtained in ACL2 with other implementations done in other languages, and explore the main issues encountered during the verification effort of the correctness of that implementation.

For this case study, we have chosen the implementation of a syntactic unification algorithm. The algorithm receives as input two first-order terms and it returns a most general unifier of these terms if they are unifiable, failure otherwise. Unification algorithms are both theoretically interesting and practically important, since they are at the heart of many symbolic computation systems [2]. The verified implementation stores terms as directed acyclic graphs (*dags* in the following) by means of a pointer structure stored in an array field of a stobj. In this way, we obtain a time complexity of $O(n^2)$ and a space complexity of $O(n)$. We followed quite closely a Pascal implementation of the algorithm described in Section 4.8 of [1], which in turn is based on the exposition by Corbin and Bidoit [3]. It should be noted that we do not prove the complexity of our implementation in ACL2; a hand-proof of this complexity can be found in [1].

The main feature of our formal proof of the correctness of the algorithm is a clear separation between the logic of the process of unification, the data structures used, the specific execution control of the algorithm and the details related to its execution in ACL2. To cope with the complexity of the whole formal proof, we introduce each of these aspects in successive refinement steps. The description presented here is guided and motivated by these steps.

This paper is a revised version of [10], presented at the ACL2 Workshop 2004. We do not present here details of the proofs, and some of the function definitions will be omitted. We urge the interested reader to consult [11], where the complete source code of the development (with detailed comments) is available.

2. An ACL2 Overview

We now give a brief overview to the ACL2 system. ACL2 stands for "A Computational Logic for an Applicative Common Lisp." Roughly speaking, ACL2 is a programming language, a logic and a theorem prover. Its programming language is an extension of an applicative subset of Common Lisp [12] (we will assume the reader familiar with this language). The ACL2 logic describes the programming language, with a formal syntax, axioms and rules of inference: the applicative subset of Common Lisp is a model of the ACL2 logic. Finally, the

theorem prover provides support for mechanized reasoning in the logic. Thus, the system constitutes an environment in which programs can be defined and executed, and their properties can be formally specified and proved with the assistance of a theorem prover.

The logic is a first-order logic with equality. The syntax of its terms is that of Common Lisp and therefore uses prefix notation. Formulas are quantifier-free and their variables are considered to be universally quantified. For example, the following formula may be read as "for all natural numbers n and x, with x even and $n > 0$, x^n is even":

```
(defthm evenp-expt
    (implies (and  (natp n) (> n 0) (natp x) (evenp x))
             (evenp (expt x n)))
```

The logic includes axioms for propositional logic and for a number of primitive Common Lisp functions and data types. Rules of inference include those for propositional calculus, equality, instantiation and a principle of proof by induction.

By the *principle of definition*, new function definitions (using `defun`) are admitted as axioms only if there exists an ordinal measure in which the arguments of each recursive call (if any) decrease, thus proving its termination. This ensures that no inconsistencies are introduced by new definitions.

The ACL2 theorem prover is an integrated system of ad hoc proof techniques, including simplification and induction among them. Simplification is a process combining term rewriting with some decision procedures (linear arithmetic, type set reasoner, etc.) Sophisticated heuristics for discovering an (often suitable) induction scheme is one of the key features in ACL2. The command `defthm` starts a proof attempt, and, if it succeeds, the theorem is stored as a rule (in most cases, a conditional rewriting rule). For example the above theorem `evenp-expt`, once proved, would allow the prover to rewrite an instance of the term `(evenp (expt x n))` to the boolean constant `t` (true), provided that the corresponding instantiated conditions of the rule can be established.

The theorem prover is automatic in the sense that, once `defthm` is submitted, the user can no longer interact with the system. However, in some sense, it is interactive. Often, non-trivial results can not be proved on a first attempt, and then the role of the user is important: she has to guide the prover by a suitable collection of definitions and lemmas, used in subsequent proofs as rewriting rules. These lemmas are suggested by a preconceived hand proof (at a higher level) or by inspection of failed proofs (at a lower level). This kind of interaction

is called "The Method" by the authors of the system [5]. We followed "The Method" to obtain the results presented in this paper.

A relevant feature of ACL2 is executability: since its axioms and rules of inference describe a subset of Common Lisp, most ground expressions in the logic are directly executable in the host Lisp (as opposed to deducing their values via the axioms). Nevertheless, this simple relationship is complicated by the fact that not all Common Lisp functions are defined on all inputs: the Common Lisp standard introduces the notion of "intended domain" of a primitive function. Outside this intended domain the behavior of a function is not specified. In contrast, in the ACL2 logic functions are total: that is, every application of a function defined has a completely specified result.

ACL2 formalizes the notion of intended domain by means of *guards*. The guard of a function (primitive or defined) is a formula describing its intended domain. *Guard verification* is the process of proving that if a function is called on an input satisfying its guard, then the evaluation of this call will proceed without any guard violation. Roughly speaking, the proof obligations generated by the guard verification process state that the guard of a function implies the guards of its definition body. Guards have no effect from the logical point of view, but they provide a means of (formally supported) direct execution in the host Common Lisp,

For more information on ACL2, the best reference is [5]. For a detailed and updated description of all the system details, we also recommend visiting the ACL2 home page [6] and the user's manual in it.

3. Syntactic Unification

Let us recall in this section some basic concepts and results about syntactic unification of first-order terms, our target example. A complete description of the theory of unification can be found in [2].

An *equation* is an ordered 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 the equation $t_1 \approx t_2$ if $\sigma(t_1) = \sigma(t_2)$. We say that a substitution is a solution of a system of equations S if it is a solution of every equation in S. We say that the system is *solvable* if it has a solution. Usually, a solvable system has more than one solution, but we will be interested in most general solutions. 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* (*mgs* in the following) if it is more general than any other solution of S.

As a particular case, we say that 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. Finally, a (syntactic) *unification algorithm* is an algorithm that decides whether two given terms are unifiable, and in that case it returns a most general unifier.

In the literature, it is quite common to describe syntactic unification algorithms by means of the relation $\Rightarrow_u$ given by the transformation rules presented in Figure 1. This set of rules is known as the *Martelli-Montanari transformation system*. The rules act on pairs of systems of equations of the form $S;U$ (the symbol $\bot$ represents unification failure). Intuitively, the system S can be seen as a set of equations to be solved, and the system U as a (partially) computed unifier. We call the pair $S;U$ a *unification problem*. Note that we are identifying 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 an mgs of itself.

The intuitive idea is that, in order to find a most general solution of a system of equations S, we can iteratively apply (in a "don't care" nondeterministic manner) the rules of $\Rightarrow_u$, starting with the unification problem $S;\emptyset$, until either a unification problem of the form $\emptyset;U$ or $\bot$ is obtained. It can be proved that this process must terminate and that S is solvable if and only if $\bot$ is not derived; in that case U is a most general solution of S.

Note that the transformation relation $\Rightarrow_u$ does not describe any concrete unification algorithm. Roughly speaking, a unification algorithm can be designed by using a data structure to represent first-order terms and substitutions, and choosing a strategy to apply the rules, starting with the pair of systems $\{t_1 \approx t_2\};\emptyset$ (where t_1 and t_2 are the two given input terms). This transformation based specification of the unification process allows us to concentrate on its logical properties without the burden of data structures or control issues.

4. Formalization of the Unification Transformation Relation

The first step is to formalize in ACL2 the transformation relation $\Rightarrow_u$ and prove its main properties. It turns out that these properties are more easily proved if we consider a "natural" representation of first-order terms and substitutions, even though this representation may

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 $x \in X$, $t \notin X$

Figure 1. Martelli–Montanari transformation system

not be the most efficient. In particular, in this first stage 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. A unification problem is a list with two elements: a system and a substitution. The failure $\bot$ is represented as `nil`. In the sequel, this representation of terms and substitutions in prefix form, using lists, will be referred to as *prefix representation* or *prefix notation*.

Let us now briefly describe how we have formalized in ACL2 the relation $\Rightarrow_u$. Note that one step of transformation of $\Rightarrow_u$ is determined by the rule applied and the equation where that rule is applied. 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. Not every operator can be applied to every unification problem, since rules have some conditions that have to be met. For example, the operator `(eliminate . 5)` can be applied to a unification problem only if it has at least five equations to be solved and its fifth equation is of the form $x \approx t$, x being a variable and not occurring in t. These considerations lead us to formalize in ACL2 the relation $\Rightarrow_u$ by means of two functions:

- `(unif-legal-p upl op)`, checking the conditions required to apply a given operator `op` to a unification problem `upl` (in prefix notation).
- `(unif-reduce-one-step-p 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 $\Rightarrow_u$:

1. The set of solutions of a unification problem is preserved in each transformation step.
2. If the second system of a unification problem is in solved form, then the transformed unification problem has its second system in solved form.
3. The transformation relation is terminating.

For example, these are the ACL2 theorems establishing property 1 above:

```
(defthm mm-preserves-solutions-1
  (implies (and (unif-legal-p upl op)
                (solution sigma (both-systems upl)))
    (solution sigma (both-systems (unif-reduce-one-step-p upl op)))))

(defthm mm-preserves-solutions-2
  (implies
    (and (unif-legal-p upl op)
         (unif-reduce-one-step-p upl op)
         (solution sigma
                   (both-systems (unif-reduce-one-step-p upl op))))
    (solution sigma (both-systems upl))))

(defthm mm-preserves-solutions-3
  (implies (and (unif-legal-p upl op)
                (not (unif-reduce-one-step-p upl op)))
           (not (solution sigma (both-systems upl)))))
```

Having proved the main properties of one-step transformations, we can easily extend these properties to finite sequences of transformations[1]. In particular we prove that given two terms t_1 and t_2 and a substitution σ, if $\{t_1 \approx t_2\}; \emptyset \stackrel{*}{\Rightarrow}_u \emptyset; \sigma$, then σ is an mgu of t_1 and t_2, and if $\{t_1 \approx t_2\}; \emptyset \stackrel{*}{\Rightarrow}_u \bot$, then t_1 and t_2 are not unifiable. This result is the key to prove the correctness of a given unification algorithm: it suffices to show that the results computed by the algorithm can be described by the iterative application of a sequence of operators (although the algorithm does not necessarily have to deal explictly with operators).

Most of the results about the relation $\Rightarrow_u$ have been reused from a previous formalization of the main properties of the lattice of first-order

[1] Note that in our formalization, a sequence of transformations can be identified with a list of operators. Each of these operators has to be applicable to the result obtained by the previous one.

terms with respect to subsumption [9]. As part of that work, we had defined and verified a unification algorithm based on the transformation system $\Rightarrow_u$ acting on terms in prefix notation. For a detailed description of the proofs and a precise statement of the properties mentioned above, we refer the reader to the supporting materials.

5. Representing Terms as Directed Acyclic Graphs

Using the prefix representation, a unification algorithm may have exponential complexity in some situations, both in time and space. Consider, for example, the following standard parameterized unification problem, which we will call U_n:

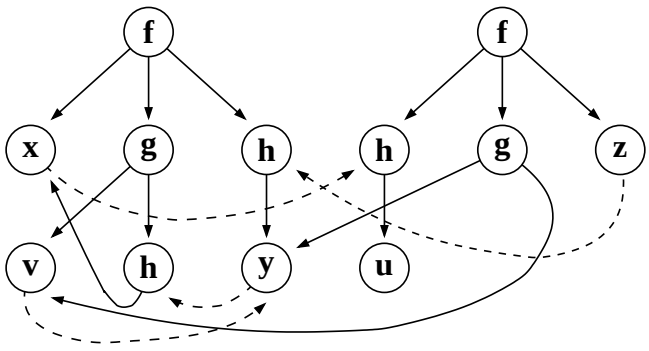
$$p(x_n, \ldots, x_2, x_1) \approx p(f(x_{n-1}, x_{n-1}), \ldots, f(x_1, x_1), f(x_0, x_0))$$

An mgu of this problem is

$$\{x_1 \mapsto f(x_0, x_0), x_2 \mapsto f(f(x_0, x_0), f(x_0, x_0)), \ldots\}$$

mapping each variable x_i to a complete binary tree of height i. This mgu is obtained by repeatedly applying the **Eliminate** rule of $\Rightarrow_u$. Using the prefix representation of terms, it would be necessary to reconstruct the instantiated systems of equations, for each application of the rule.

The standard approach to deal with this problem is to use *term dags*, a kind of pointer structures representing terms where variables can be shared. For example, the graph below represents the equation $f(x, g(v, h(x)), h(y)) \approx f(h(u), g(y, v), z)$. 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:



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 introducing 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 $\{x \mapsto h(u), y \mapsto h(h(u)), z \mapsto h(h(h(u))), v \mapsto h(h(u))\}$, which is an mgu of $f(x, g(v, h(x)), h(y))$ and $f(h(u), g(y, v), z)$.

In ACL2, we have represented a term dag as a list of nodes. In particular, if `g` is a list representing a term dag, each of its elements represents a node in the graph, uniquely identified by its position index. The nodes store information about its label and its successors, in the following way:

- If node i represents an unbound variable x, then `(nth` i `g)` (that is, the i-th element of `g`) is a dotted pair of the form `(`x `. t)`[2].
- If node i represents a bound variable, then `(nth` i `g)` is 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 `(nth` i `g)` is a dotted pair of the form `(`f `.` l`)`, where l is the list of the indices corresponding to the root nodes of $t_1, \ldots, t_n$.

For example, we can represent the term

$$equ(f(x, g(v, h(x)), h(y)), f(h(u), g(y, v), z))$$

by a list with the following elements (for a better understanding, we marked each element with its position index):

0	1	2	3	4	5	6	7
(EQU . (1 9))	(F . (2 3 7))	(X . T)	(G . (4 5))	(V . T)	(H . (6))	2	(H . (8))
(Y . T)	(F . (10 12 15))	(H . (11))	(U . T)	(G . (13 14))	8	4	(Z . T)
8	9	10	11	12	13	14	15

Although with the above conventions one can represent every first-order term as a list of nodes, the converse is not true. Thus, we need

[2] We could have used any non-list value as the second element in this dotted pair to distinghish it from the representation of non-variable terms.

to define predicates checking that a given list of nodes has the properties required to represent a term. The main property is acyclicity: we must check that no cycles can be formed following the pointers of the structure.

That is the reason why we needed to develop 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 (represented following the conventions described above) does not contain cycles. It is implemented as a standard depth-first search algorithm, looking for cycles in the graph, in a similar way to the path finder described in [7]. The theorems below establish that a graph `g` verifies the `dag-p` condition if and only if does not contain cycles. Here `(cycle-p p g)` checks if a given list of nodes `p` is a cyclic path in the graph `g`, and the function `one-cyclic-path` returns a witness cyclic path (whenever it exists) in a given graph.

```
(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))))
```

An important reason why these well-formedness conditions are needed is due to the the fact that they have to be explicitly included in the definition of every function that performs a traversal of the graph, in order to ensure its termination, and thus be accepted as a definitional axiom in the logic. For example, consider the following:

```
(defun dag-as-term (flg h g)
   (declare (xargs :measure (measure-rec-dag flg h g)))
   (if (dag-p g)
      (if flg
          (let ((p (nth h g)))
            (if (integerp p)
                (dag-as-term flg p g)
              (let ((args (cdr p)) (symb (car p)))
                (if (equal args t)
                    symb
                  (cons symb (dag-as-term nil args g))))))
        (if (endp h)
            h
          (cons (dag-as-term t (car h) g)
                (dag-as-term nil (cdr h) g))))
    'undef))
```

This function `dag-as-term` returns a term (or list of terms) in prefix form, built from the graph `g` and starting at the index (or list of indices) `h`[3]. Note that the condition `(dag-p g)` is needed for termination. Moreover, even with that condition, its termination proof is not trivial: we have to explicitly provide a measure of the arguments (given by the function `measure-rec-dag`) and show that it decreases in every recursive call; roughly speaking, this measure counts the number nodes reachable from `h`.

Another function that needs the `dag-p` condition is the one that *dereferences* indices (that is, follows a chain of instantiations until it reaches an unbound variable or a non-variable node). The following function `dag-deref` implements this operation:

```
(defun dag-deref (h g)
  (declare (xargs :measure (measure-rec-dag t h g)))
  (if (dag-p g)
      (let ((p (nth h g)))
        (if (integerp p) (dag-deref p g) h))
    'undef))
```

Similarly, the "occur check" operation (searching for an occurrence of a variable in a term), used in the **Eliminate** and **Occur-check** rules, need this explicit `dag-p` condition in its logical definition:

```
(defun occur-check-d (flg x h g)
  (declare (xargs :measure (measure-rec-dag flg h g)))
  (if (dag-p g)
      (if flg
          (let ((p (nth h g)))
            (if (integerp p)
                (occur-check-d flg x p g)
              (let ((args (cdr p)))
                (if (equal args t)
                    (equal x h)
                  (occur-check-d nil x args g)))))
        (if (endp h)
            nil
          (or (occur-check-d t x (car h) g)
              (occur-check-d nil x (cdr h) g))))
    'undef))
```

We will comment in Section 10 how to get rid of this expensive `dag-p` conditions in the executable implementation of the unification

[3] The inclusion of the boolean flag `flg` is a standard trick for considering terms and lists of terms in a mutually recursive way. If `flg` is `nil` then the input is considered as a list; otherwise, it is considered as a single element. We could use the `mutual-recursion` facility provided by ACL2 but we are more confortable with the "flag" definition.

algorithm. It should also be noted that although we are using lists of nodes to represent term dags, we will see in Section 10 that in the executable implementation we will use an array field of a stobj, which is much more efficient. Nevertheless, there is no difference from the logical point of view, and for the moment we are only interested in the logical properties of this dag representation.

6. The Unification Transformation Relation on Term Dags

The next step in our development is to define in ACL2 the relation $\Rightarrow_u$ acting on unification problems represented as term dags. Given a term dag `g`, we can naturally identify a position index with the term whose root node is in the corresponding position of `g` (this correspondence is precisely defined by the function `dag-as-term` above).

Taking this into account, and given a term dag, we can represent a system of equations as a list of dotted pair of indices (what we call an *indices system*). We can also represent a substitution by an *indices substitution*: a list of pairs of the form `(`x `.` n`)` where x is a variable symbol and n is the index of the node to which the variable is bound. A *dag unification problem* is a list with three elements: an indices system, an indices substitution and a term dag.

We say that a dag unification problem is *well-formed* if its term dag is acyclic, its repeated variable nodes are shared and all the indices appearing in its indices system, its indices substitution and in the contents of the nodes are natural numbers less than the length of the graph. This well-formedness property is implemented by the function `well-formed-upl`, whose definition we omit here. Under these well-formedness conditions, every dag unification problem represents a unification problem in prefix form. Therefore, it makes sense to define the relation $\Rightarrow_u$ acting on well-formed dag unification problems. As one can expect, we use again an operator-based representation, as in Section 4 for the prefix representation. That is, the relation is defined by means of 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`.
- `(unif-reduce-one-step-d dag-upl op)`, returning the transformed dag unification problem obtained after applying the operator `op` to `dag-upl`.

Operators are represented in exactly the same way as described in Section 4. But the key point here is that in a dag unification problem,

the indices system *only contains indices* pointing to the terms stored in the term dag. The transformations defined by the above two functions reflect, at the dag level, the corresponding transformations that one would perform at the prefix level.

To illustrate this, the following are the applicability condition and the reduction step corresponding to the **Eliminate** rule (for the rest of the rules, we refer the reader to the supporting materials):

```
(defun unif-legal-d-eliminate (t1 t2 g)
  (and (term-dag-variable-p t1 g)
       (not (occur-check-d t t1 t2 g))))

(defun unif-reduce-one-step-d-eliminate (t1 t2 R sol g)
  (list R
        (cons (cons (term-dag-symbol t1 g) t2) sol)
        (update-nth t1 t2 g)))
```

These are auxiliary functions used by `unif-legal-d` and `unif-reduce-one-step-d`, respectively; they are called when the operator is of the form `(eliminate . n)`, after selecting the n-th equation of the indices system and dereferencing its two indices, obtaining `t1` and `t2`. In this case, `R` is the indices system with the remaining equations, `sol` is the indices substitution and `g` is the term dag of the input dag unification problem. The applicability condition checks that `t1` is a variable node (function `term-dag-variable-p`) and that this variable does not occur in the term pointed by `t2`. The reduction step returns the dag unification problem obtained adding a new binding to the indices substitution and creating a link in the graph from node `t1` to node `t2` (updating the `t1`-th element of the list `g` with `t2`, using the primitive ACL2 function `update-nth`).

Let us now describe how we proved the main properties of this dag based transformation relation. Instead of proving them reasoning directly with the definitions `unif-legal-d` and `unif-reduce-one-step-d` (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, we prove that the following diagram commutes:

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

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 theorems (the function `upl-as-pair-of-systems` plays the role of the function *dp* in the diagram):

```
(defthm unif-reduce-one-step-d-preserves-well-formed-upl
  (implies (and (well-formed-upl upl)
                (unif-legal-d upl op))
           (well-formed-upl (unif-reduce-one-step-d upl op))))

(defthm unif-legal-d-implies-unif-legal-p
  (implies (and (well-formed-upl upl)
                (unif-legal-d upl op))
           (unif-legal-p (upl-as-pair-of-systems upl) op)))

(defthm unif-reduce-one-step-d-equal-unif-reduce-one-step-p
  (implies
    (and (well-formed-upl upl)
         (unif-legal-d upl op))
    (equal (upl-as-pair-of-systems (unif-reduce-one-step-d upl op))
           (unif-reduce-one-step-p (upl-as-pair-of-systems upl) op))))
```

These theorems respectively establish that:

- The well-formedness property of dag unification problems is preserved by the transformation rules. This result is not trivial: in particular, this means that the updating performed by a legal application of the **Eliminate** rule preserves acyclicity.
- 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 Section 4 to this more efficient data structure. In particular, it can be proved that we can obtain a most general unifier of two terms (or failure when they are not unifiable) by exhaustively applying the

rules of transformation on its dag representation, and that this process must terminate.

7. The Extended Transformation Relation

Recall that we have not yet defined any particular unification algorithm, and that we still stay at a rule-based specification level. What we have until now can be used to prove that every algorithm whose computation can be described as the iterative application of the transformation rules of $\Rightarrow_u$ on a dag representation of unification problems is a correct unification algorithm.

Unfortunately, every algorithm described by the above transformation rules would still have exponential time complexity, even using the term dag representation (although linear in space). Consider the following unification problem, which we will call Q_n:

$$p(x_n, \ldots, x_1, y_n, \ldots, y_1, x_n) \approx \\ p(f(x_{n-1}, x_{n-1}), \ldots, f(x_0, x_0), f(y_{n-1}, y_{n-1}), \ldots, f(y_0, y_0), y_n)$$

Note that before attempting to unify the respective last arguments of this equation, x_n and y_n are both bound to terms of exponential size. These terms are complete binary trees of heigth n, where the two subtrees of every interior node have the same complex structure. Then, the unification of x_n and y_n leads to an exponential number of attempts to unify the same subtrees again and again.

The solution is to share not only variables but also non-variable terms. The intuitive idea is that after two subterms in the term dag have been unified, we can replace the contents of the root node of one of them by a pointer to the root node of the other. And that is sound, because if they have been unified, they represent the same term.

But before translating this idea to the definition of a concrete quadratic unification algorithm, *let us still stay at the rule-based specification level*. With our current definition of $\Rightarrow_u$, we cannot perform the operation of identifying two terms by updating the corresponding node contents.

Therefore, we extend our definition of the transformation relation. As usual, this definition will be operator-based. We simply consider a new kind of operators, called *identification* operators: these are of the form `(identify` i j`)`, where i and j are indices. This operator has the effect of creating a pointer from node i to node j. The operator will be applicable only when i and j are two different non-variable root nodes of equal terms. The following functions define the applicability and the reduction step for identification operators:

```
(defun unif-legal-q-identify (i j g)
  (and (natp i) (< i (len g)) (term-dag-non-variable-p i g)
       (natp j) (< j (len g)) (term-dag-non-variable-p j g)
       (not (equal i j))
       (equal (dag-as-term t i g) (dag-as-term t j g))))

(defun unif-reduce-one-step-q-identify (i j S sol g)
  (list S sol (update-nth i j g)))
```

The functions `unif-legal-q` and `unif-reduce-one-step-q` define the extended transformation relation. Note that this extended relation is defined on a term dag representation and includes all the transformation rules of $\Rightarrow_u$ as well as identifications:

```
(defun unif-legal-q (upl op)
  (if (equal (first op) 'identify)
      (unif-legal-q-identify (second op) (third op) (third upl))
    (unif-legal-d upl op)))

(defun unif-reduce-one-step-q (upl op)
  (if (equal (first op) 'identify)
      (unif-reduce-one-step-q-identify
        (second op) (third op) (first upl) (second upl) (third upl))
    (unif-reduce-one-step-d upl op)))
```

The following theorems establish the main properties of this extended transformation relation:

```
(defthm unif-reduce-one-step-q-preserves-well-formed-upl
  (implies (and (well-formed-upl upl)
                (unif-legal-q upl op))
           (well-formed-upl (unif-reduce-one-step-q upl op))))

(defthm unif-reduce-one-step-q-for-identifications
  (implies
    (and (well-formed-upl upl)
         (unif-legal-q upl op)
         (equal (first op) 'identify))
    (equal (upl-as-pair-of-systems (unif-reduce-one-step-q upl op))
           (upl-as-pair-of-systems upl))))
```

That is:

- Well-formedness of the dag unification problem is preserved. Note again that this result is not trivial: it means that updating a node by a legal identification do not create cycles in the graph.
- An identification does not change the unification problem in prefix form represented by the dag unification problem. That is, no "harm" is done by identifications, from the point of view of the unification problem.

From these theorems and the results of the previous section, it is not difficult to prove that for every sequence of these transformation steps (including identifications) performed at the dag level, there exists a sequence of transformation steps of $\Rightarrow_u$ performed at the corresponding prefix representation. Therefore, every algorithm whose computation can be described as the iterative application of these rules on dag unification problems is a correct unification algorithm.

8. An Improved Occur Check

Before defining the quadratic unification algorithm in the next section, we must fix another technical detail that could cause exponential behavior. Assume that at some point of the unification process, a variable is bound to a term of exponential size, but this term is stored in the term dag in linear size because its subterms are shared. If we have to check the occurrence of a variable in this term, we should avoid visiting these shared subgraphs repeatedly.

This exponential behavior may appear with the naive implementation of occur check defined by the function `occur-check-d` given in Section 5: we do not take care of repeated visits to the same subgraph.

To optimize this implementation, we follow the idea given in [1]. We will use a `stamp` list of integers: the number in position i of this list represents the last time node i of the term dag was visited for occur check. We also use a `time` counter that will be incremented every time the unification procedure calls to the occur check function. Before visiting a subgraph to check the occurrence of a variable, we check if its `stamp` information is equal to `time`. If that is the case we simply return `nil`, without traversing the subgraph; otherwise we traverse the subgraph, updating the `stamp` information if the variable does not occur in the subgraph. The definition below implements in ACL2 this improved occur check. Note that it returns a list of two elements: the first is a boolean indicating occurrence and the second is the (possibly modified) stamp list.

```
(defun occur-check-q (flg x h g stamp time)
  (if (dag-p g)
      (if flg
          (let ((p (nth h g)))
            (if (integerp p)
                (occur-check-q flg x p g stamp time)
              (let ((args (cdr p)))
                (cond ((equal args t) (list (equal x h) stamp))
                      ((equal (nth h stamp) time) (list nil stamp))
                      (t (let* ((bool-stamp
```

```
                              (occur-check-q nil x args g stamp time))
                             (bool (first bool-stamp))
                             (stamp (second bool-stamp)))
                        (if bool
                            bool-stamp
                          (list nil (update-nth h time stamp)))))))))
      (if (endp h)
          (list nil stamp)
        (let* ((bool-stamp
                (occur-check-q t x (car h) g stamp time))
               (bool (first bool-stamp))
               (stamp (second bool-stamp)))
          (if bool
              bool-stamp
            (occur-check-q nil x (cdr h) g stamp time)))))
  (list 'undef stamp)))
```

The following theorem establishes that the result computed by the improved function `occur-check-q` is consistent with the result computed by the function `occur-check-d`.

```
(defthm occur-check-d-occur-check-q
  (implies (occur-check-invariant x h g stamp time)
           (equal (first (occur-check-q t x h g stamp time))
                  (occur-check-d t x h g))))
```

The function `occur-check-invariant` in this theorem describes an invariant condition that we will prove that is met in every step of our implemented unification algorithm. Roughly speaking, all the numbers in the `stamp` list have to be strictly smaller than the `time` counter.

9. A Quadratic Unification Algorithm

It is time to define our implementation of a quadratic unification algorithm. That is, having proved the main properties of the rule-based specification of the unification process on term dags, we deal with control issues. Not surprisingly, we simply choose a certain strategy to apply the rules of the extended transformation relation: in our case, we always select the first equation to be solved. To avoid exponential complexity, we need some technical details in order to do identifications properly and also we use the improved occur check defined in the previous section.

The function `dag-transform-mm-q` defines the individual steps of transformation performed by the algorithm. This is the main component of the algorithm. Roughly speaking, the implemented algorithm will apply this function until there are no equations to be solved or failure is detected.

```
(defun dag-transform-mm-q (ext-upl)
  (let* ((ext-S (first ext-upl)) (equ (first ext-S)) (R (rest ext-S))
         (U (second ext-upl)) (g (third ext-upl))
         (stamp (fourth ext-upl)) (time (fifth ext-upl)))
    (if (equal (first equ) 'id)
        (let ((g (update-nth (second equ) (third equ) g))) ;;; IDENTIFY
          (list R U g stamp time))
      (let ((t1 (dag-deref (car equ) g))
            (t2 (dag-deref (cdr equ) g)))
        (if (equal t1 t2)
            (list R U g stamp time)                          ;;; DELETE
          (let ((p1 (nth t1 g)) (p2 (nth t2 g)))
            (cond
             ((dag-variable-p p1)
              (let* ((bool-stamp (occur-check-q t t1 t2 g stamp time))
                     (bool (first bool-stamp))
                     (stamp (second bool-stamp)))
                (if bool
                    nil                                 ;;; OCCUR-CHECK
                    (let ((g (update-nth t1 t2 g)))
                      (list R (cons (cons (dag-symbol p1) t2) U) g
                            stamp (1+ time))))))          ;;; ELIMINATE
             ((dag-variable-p p2)
              (list (cons (cons t2 t1) R) U g stamp time))   ;;; ORIENT
             ((not (equal (dag-symbol p1) (dag-symbol p2)))
              nil)                                          ;;; CLASH1
             (t (let* ((pairs-bool
                        (pair-args (dag-args p1) (dag-args p2)))
                       (pairs (first pairs-bool))
                       (bool (second pairs-bool)))
                  (if bool
                      (list (append pairs
                                    (cons (list 'id t1 t2) R))
                            U g stamp time)               ;;; DECOMPOSE
                      nil)))))))))                           ;;; CLASH2
```

This function receives as input what we call an extended unification problem. An *extended unification problem* is a list with five elements: an extended indices system, an indices substitution, a term dag, a stamp list and a time counter. An *extended indices system* is an indices system that could include also some *identification marks* of the form `(id` i j`)`.

In this function, the transformation step to apply is determined by the first element of the extended indices system. If this first element is an ordinary equation between indices, then the corresponding rule of $\Rightarrow_u$ is applied. If it is an identification mark of the form `(id` i j`)`, then an identification of the nodes i and j is applied. In order to guarantee that identifications are always done with root nodes of already unified subterms, identification marks are included at every application of the

Decompose rule, just after the equations pairing[4] the arguments of the nodes to be unified. In this way, extended indices systems can be seen as a stack: when an identification mark is at the top of the stack, we are sure that the nodes to be identified have successfully been unified.

The function `dag-transform-mm-q` has to be iteratively applied until the system of equations to be solved is empty or until `nil` (unsolvability) is obtained. The following function `solve-upl-q` does this job:

```
(defun normal-form-syst (ext-upl)
  (not (and  (consp ext-upl) (consp (first ext-upl)))))

(defun solve-upl-q (ext-upl)
  (declare (xargs :measure (unification-measure-q ext-upl)))
  (if (unification-invariant-q ext-upl)
      (if (normal-form-syst ext-upl)
          ext-upl
        (solve-upl-q (dag-transform-mm-q ext-upl)))
    'undef))
```

The condition `(unification-invariant-q ext-upl)` in the above definition is needed for termination. Among many other properties, it includes the `dag-p` condition. Termination of `solve-upl-q` is not trivial at all, and a lexicographic measure has to be supplied to instruct the prover in the termination proof. This measure (given by the function `unification-measure-q`, omitted here) is mainly based on the measure that justifies the termination of $\Rightarrow_u$.

In addition, the function `unification-invariant-q` defines the properties needed to ensure that the function `dag-transform-mm-q` is applying a legal transformation step of the extended transformation relation[5]. Note that this is trivial for the case of non-identification transformations, because the applicability conditions are explicitly checked. Nevertheless, that is not the case for identifications. Recall that an identification can be applied only when the terms pointed by the identified nodes are equal. But this applicability condition is not checked (and that is essential for the efficiency of the algorithm).

The key point is that, due to the way the successive transformation steps are carried out, it is guaranteed that every time an identification step is performed, this identification is legal. In other words, there is some "well-formedness" conditions on the extended unification problem that can be seen as an invariant of the unification process, and this invariant condition implies that every transformation step performed

[4] Given two lists $(l_1 \ \ldots \ l_n)$ and $(m_1 \ \ldots \ m_k)$ the auxiliary function `pair-args` returns the list $(((l_1 \ . \ m_1) \ \ldots \ (l_n \ . \ m_k))$ `t`$)$ if $n = k$, `(nil nil)` otherwise.

[5] And also that we can safely use the improved occur check function.

by `dag-transform-mm-q` is a legal transformation step with respect to the extended transformation relation defined in Section 7. The following theorems establish this fact[6]:

```
(defthm unification-invariant-q-preserved
   (implies (and (not (normal-form-syst ext-upl))
                 (unification-invariant-q ext-upl))
            (unification-invariant-q (dag-transform-mm-q ext-upl))))

(defthm transform-mm-q-applies-a-legal-operator
  (implies (and (not (normal-form-syst ext-upl))
                (unification-invariant-q ext-upl))
           (unif-legal-q (ext-upl-to-upl ext-upl)
                         (dag-transform-mm-q-op ext-upl))))

(defthm transform-mm-q-applies-an-operator
  (implies
    (unification-invariant-q ext-upl)
    (equal (ext-upl-to-upl (dag-transform-mm-q ext-upl))
           (unif-reduce-one-step-q (ext-upl-to-upl ext-upl)
                                   (dag-transform-mm-q-op ext-upl)))))
```

We save the reader from the definition of the function `unification-invariant-q`. It is a *very long* definition (more than 300 lines of code) including well-formedness properties such as acyclicity of the term dag, the occur-check invariant and the correct placement of the identification marks in the extended indices system stack. Due to this, the above theorem `unification-invariant-q-preserved` turns out to be the most difficult to prove of all the verification effort.

In the above theorems, the function `dag-transform-mm-q-op` returns the corresponding "witness" operator justifying that `dag-transform-mm-q` is applying a rule of the extended transformation relation. This means that the exhaustive iteration of `dag-transform-mm-q`, as implemented by `solve-upl-q`, is a correct unification procedure. Thus, we are almost done. But before we need to deal with some technical issues related to the execution of the algorithm in ACL2.

10. Execution of the Algorithm in ACL2

The function `solve-upl-q` in the previous section can be executed in ACL2. But from the practical point of view, this execution is completely unfeasible, mainly for two reasons:

[6] The function `ext-upl-to-upl` removes the identification marks, the `stamp` list and the `time` counter of an extended dag unification problem.

– The term dag is stored in a list. This means that accessing (with `nth`) and updating (with `update-nth`) the information of the nodes are not done in constant time. Moreover, updates are not destructive and need copying.

– As we have seen, some of the recursive functions implemented have expensive well-formedness conditions (like `dag-p` or `unification-invariant-q`) in their bodies, needed for termination. And these conditions would be evaluated in *every recursive call*.

Fortunately, we can fix these two problems. To deal with the first, we will use a *single-threaded object*. In ACL2, it is possible to declare some objects in the language as single-threaded (*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 functional semantics of ACL2. Using stobjs we can combine efficient imperative implementations with the semantic of functional languages to reason about them.

The following creates a stobj called `terms-dag` with two resizable array fields to store the term dag and the stamp:

```
(defstobj terms-dag
  (dag :type (array t (0)) :resizable t)
  (stamp :type (array integer (0)) :initially -1 :resizable t))
```

The effect of this ACL2 form is to introduce the stobj `terms-dag` and its associated recognizers, creator, accessors, updaters, and length and resize functions for the array fields. In particular, given an index i, the expressions `(dagi` i `terms-dag)` and `(update-dagi` i v `terms-dag)` respectively access and update (with value v) the i-th cell of the `dag` array. Similarly, functions `stampi` and `update-stampi` are introduced. These operations are executed in constant time and the update is destructive (at the price of syntactic restrictions on the use of `terms-dag`).

Now, we redo all the definitions of the implemented algorithm, taking into account that the term dag is stored in this stobj. 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. The function `dag-transform-mm-st` below is the stobj counterpart of `dag-transform-mm-q`. The key point is that from the logical point of view, the `dag` and `stamp` arrays of the stobj are lists. Thus it is straightforward to translate the already proved properties about the list version of the algorithm to the stobj version.

```
(defun dag-transform-mm-st (S U terms-dag time)
  (declare (xargs :stobjs terms-dag) ...)
  (let* ((equ (car S)) (R (cdr S)))
    (if (equal (car equ) 'id)                                ;;; IDENTIFY
        (let ((terms-dag (update-dagi (second equ) (third equ)
                                      terms-dag)))
          (mv R U t terms-dag time))
      (let* ((t1 (dag-deref-st (car equ) terms-dag))
             (t2 (dag-deref-st (cdr equ) terms-dag))
             (p1 (dagi t1 terms-dag))
             (p2 (dagi t2 terms-dag)))
        (cond
         ((= t1 t2) (mv R U t terms-dag time))           ;;; DELETE
         ((dag-variable-p p1)
          (mv-let (oc terms-dag)
            (occur-check-st t t1 t2 terms-dag time)
            (if oc                                      ;;; OCCUR-CHECK
                (mv nil nil nil terms-dag nil)
              (let ((terms-dag (update-dagi t1 t2 terms-dag)))
                (mv R (cons (cons (dag-symbol p1) t2) U) t
                    terms-dag (1+ time))))))             ;;; ELIMINATE
         ((dag-variable-p p2)
          (mv (cons (cons t2 t1) R) U t terms-dag time))     ;;; ORIENT
         ((not (eql (dag-symbol p1) (dag-symbol p2)))
          (mv nil nil nil terms-dag nil))                   ;;; CLASH1
         (t (mv-let (pairs bool)
              (pair-args-mv (dag-args p1) (dag-args p2))
              (if bool
                  (mv (append pairs (cons (list 'id t1 t2) R))
                      U t terms-dag time)                ;;; DECOMPOSE
                (mv nil nil nil terms-dag nil)))))))     ;;; CLASH2
```

Another optimization for execution that is worth pointing out is the use of multivalues in functions that returned several values in a list, such as `occur-check-q` or `pair-args`. In the stobj version of the algorithm, we used `mv` and `mv-let` to handle this (see [6] for details on multivalues). Again, there is no difference from the logical point of view, since according to the logic, `mv` returns a list. Nevertheless, a list is never created for storing multiple return values during execution, making it more efficient.

Let us now deal with the second problem, or how to get rid of the expensive well-formedness conditions in the bodies of some of the recursive functions of our implementation. These conditions are only needed for the logical definitions: they can be safely removed in execution because they are preserved in each recursive call. For that purpose, we use `defexec` and `mbe`: this ACL2 feature allows us to associate 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.

Let us explain this in more detail. In the logic, the expression `(mbe :logic` *logic_body* `:exec` *exec_body*`)` is equal to *logic_body*; the value of *exec_body* is ignored. Nevertheless, for execution in the host Lisp this form macroexpands simply to *exec_body*. The guard verification mechanism plays a key role here. Roughly speaking, the guard proof obligations generated by the above call of `mbe` are `(equal` *logic_body exec_body*`)` along with those generated by the executable body. Therefore, whenever a function defined using `mbe` is called on an input satisfying its guard, then *exec_body* may be safely used in the host Common Lisp to obtain a result, since it is provably equal in the ACL2 logic to *logic_body*. In addition, `defexec` generates a proof obligation ensuring that the executable body terminates on its intended domain.

For example, the following is the complete definition of the function `solve-upl-st`, the stobj counterpart of the function `solve-upl-q` defined in the previous section[7]. Note that the expensive `unification--invariant-q` condition is removed in the executable body.

```
(defexec solve-upl-st (S U terms-dag time)
  (declare
    (xargs :stobjs terms-dag
           :guard (and (true-listp S)
                       (unification-invariant-q
                          (list S U (dag-component-st terms-dag)
                                (stamp-component-st terms-dag) time)))
           ...))
  (mbe
   :logic
   (if (unification-invariant-q
        (list S U (dag-component-st terms-dag)
              (stamp-component-st terms-dag) time))
       (if (endp S)
           (mv S U t terms-dag time)
           (mv-let (S1 U1 bool terms-dag time1)
                   (dag-transform-mm-st S U terms-dag time)
                   (if bool
                       (solve-upl-st S1 U1 terms-dag time1)
                       (mv S U nil terms-dag time))))
       (mv S U nil terms-dag time))
   :exec
   (if (endp S)
       (mv S U t terms-dag time)
       (mv-let (S1 U1 bool terms-dag time1)
               (dag-transform-mm-st S U terms-dag time)
```

[7] The functions `dag-component-st` and `stamp-component-st` collects in a list the contents of the `dag` and `stamp` arrays of the stobj.

```
(if bool
    (solve-upl-st S1 U1 terms-dag time1)
    (mv S U nil terms-dag time))))))
```

The guard verification of this function is not trivial. We have to prove that the property `unification-invariant-q` is preserved in every recursive call. But essentially, that is the theorem `unification--invariant-q-preserved` discussed in the previous section.

In general, we used `defexec` and `mbe` in the definition of all the recursive functions that need well-formedness conditions in their logical bodies whenever these conditions can be safely removed for execution. In particular, in dereferencing, in occur checking and in the function that builds a term in prefix form from the contents of a term dag.

Finally, the top level function of our implemented algorithm is called `dag-mgu`. This function receives as input two terms `t1` and `t2` *in prefix notation* and computes its most general unifier (or failure) in the following way (see the supporting materials for the definitions):

1. It creates `terms-dag` as a local stobj, resizing the `dag` and `stamp` arrays according to the sizes of `t1` and `t2`.

2. It stores both terms in the `dag` array, as directed acyclic graphs, building an initial dag unification problem.

3. Applies the function `solve-upl-st` to the initial unification problem.

4. If failure is detected, it returns `(mv nil nil)`; otherwise, it returns `(mv t` σ`)`, where σ is the most general unifier (in *prefix notation*) obtained from the final indices substitution computed by `solve-upl-st`.

It is worth pointing out that the input and output of this top level function are in prefix notation, although the main process of the algorithm is performed with the dag representation. The guard of the function `dag-mgu` is quite simple, and only checks that the two input terms are in prefix form. In contrast, the guards of the intermediate functions are quite complicated and expensive, including the well-formedness conditions and invariants described in the preceding sections. But since guards are verified these intermediate guards are never evaluated.

The following three theorems establish the correctness of the implemented unification algorithm, showing that it computes a most general unifier of two given terms, whenever they are unifiable, and failure otherwise:

```
(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
  (let* ((dag-mgu (dag-mgu t1 t2))
         (unifiable (first dag-mgu))
         (sol (second dag-mgu)))
    (implies (and (term-p t1) (term-p t2)
                  unifiable)
             (equal (instance t1 sol) (instance t2 sol)))))

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

Note again that although the algorithm is mainly based on a dag representation of terms, the theory used to establish its properties is based on a prefix representation of terms. In particular, the predicate `term-p` checks that a given ACL2 object represents a term in prefix form, the function `instance` computes the application of a substitution to a term in prefix form, and `subs-subst` defines the subsumption relation between substitutions in prefix form.

11. Some Quantitative Information

To test the execution efficency of our implemented unification procedure, we have applied it to the problems U_n and Q_n of Sections 5 and 7, respectively[8]. We compare the times obtained with three other implementations:

- An ACL2 naive implementation, presented in [9], based on a prefix representation of terms.
- An ACL2 implementation acting on terms dag, but without the technical improvements that make it quadratic.

[8] These tests have been carried out in a Intel© Pentium© Centrino 1600GHz, with 1024Mb RAM. 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 0.001.

Table I. Execution times of the algorithms

U_n	Prefix	Exponential	Quadratic	C Quadratic
15	0.077	ϵ	ϵ	ϵ
20	5.643	ϵ	ϵ	ϵ
25	–	ϵ	ϵ	ϵ
30	–	ϵ	ϵ	ϵ
100	–	0.001	0.001	0.001
500	–	0.006	0.007	0.007
1000	–	0.023	0.025	0.020
5000	–	0.581	0.582	0.380
10000	–	2.274	2.285	1.330

Q_n	Prefix	Exponential	Quadratic	C Quadratic
15	0.615	0.033	ϵ	ϵ
20	–	1.041	ϵ	ϵ
25	–	26.135	ϵ	ϵ
30	–	728.316	ϵ	ϵ
100	–	–	0.001	0.001
500	–	–	0.032	0.016
1000	–	–	0.093	0.055
5000	–	–	2.324	1.382
10000	–	–	9.200	5.809

– A C implementation of the described quadratic unification algorithm. In this implementation terms dags are stored using records with pointers.

The results obtained are presented in Table I. As it can be seen, the ACL2 quadratic algorithm is executed at about 60% of the speed of the corresponding C implementation.

As for the proof effort, Table II shows 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 [9]).

These numbers may give an idea of the complexity of the formalized theories and the degree of automation of the proofs obtained. We

Table II. Quantitative information about the proof

Phase	Definitions	Theorems
Properties of $\Rightarrow_u$ (prefix representation)	24	81
Acyclic graphs	39	101
Diagram commutativity	39	76
Storing the initial terms in the graph	29	206
Extended transformation relation	10	25
Quadratic improvements and invariant	47	184
The stobj implementation and guards	26	102
Total	214	775

should say that most of the lemmas needed during the first phase were already proved in [9]. 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.

12. Conclusions

We have presented a formal verification of a non-trivial implementation of a syntactic unification algorithm. This implementation represents first-order terms as directed acyclic graphs, stored as pointer structures. Our implementation follows closely a Pascal implementation given in Section 4.8 of [1]. To store the term dags in an efficient way, we used a single-threaded object.

The main features of the formal verification carried out are:

- Most of the verification is done reasoning about a rule-based specification of the algorithm, avoiding reasoning about the final implementation until the last stages of the verification.
- There is a clear separation between the logic of the unification process, the data structures used for our particular implementation, the specific execution control and optimizations of the algorithm, and the details related with its execution in ACL2.
- And specifically related to ACL2, the use of single-threaded objects and the newly introduced `defexec`/`mbe` feature.

Regarding this last point, we should say that this work has been greatly benefited with the introduction of `mbe` in ACL2. Prior to version 2.8, we had to execute the algorithm using functions in `:program` mode[9], similar to the `:logic` body of the function, but removing the conditions required by the termination proof. With the introduction of `mbe` and `defexec` in ACL2 2.8, this correspondence is formally supported by the system, via the guard verification mechanism. We think that this new feature greatly improves the system.

As for execution, the data in Table I is quite satisfactory: it shows how our ACL2 implementation obtains a performance comparable to a C implementation of the same algorithm. Thus, we think that this case study is an example of how it is possible to obtain efficient and verified implementations. Note: we do not have a formal proof of the complexity of the algorithm, but it is possible to reason informally (and also corroborated by the data) that our implementation has quadratic time complexity and a linear space complexity (see [1]).

This work should be seen as a case study. We are not claiming that this implementation could be used as a verified component of a real symbolic computation system or theorem prover. In such systems a typical problem is, for example, to find if a given query term unifies with some element of a large set of terms. This is usually done by indexing techniques. In our algorithm, the overhead caused by storing the initial input terms as term dags would make it impractical in that context.

The pessimistic conclusions come from the data presented in Table II. The intuitive idea that algorithms employing more complex data structures or more sophisticated control structures require more verification effort is supported by that data. These contrast with the effort needed in the verification of a naive implementation of unification using a prefix representation of terms [9]. In that work, we needed 19 definitions and 129 theorems, and in this case we needed 214 definitions and 775 theorems.

Maybe the main reason for this extra verification effort is that we focused on obtaining a final version of the algorithm which had to be efficiently executable. Thus, we first implemented the algorithm in `:program` mode, trying to get the best execution efficiency that we could obtain from the resources provided by the ACL2 programming language. And we did this thinking exclusively as programmers, without taking into account the subsequent verification task. Once we checked that the program was efficient (at least for solving the problems

[9] A function in `:program` mode can be executed, but its definition is not introduced as an axiom in the logic.

U_n and Q_n above) we started the verification effort. And the final verified implementation is exactly the same as originally developed.

For example, if execution were not our main concern, we could have defined the algorithms that traverse the term dag (occur check, for example) taking the recursion depth as an extra parameter, and checking that the paths followed have that depth at most. This would do termination proofs much easier. But, as programmers, we would never define those functions in that way, because it is less efficient and we know that the algorithm always deal with acyclic graphs. In contrast, as verifiers, we needed to build a library with results about directed acyclic graphs. Another source of complexity of the proof came from technical details that are not essential to the logic of the process, but are needed in the final executable version. For example, as we pointed out in the previous section, it was somewhat surprising the proof effort needed in the verification of a function that translates terms in prefix form to terms in dag form, with shared variables.

Hopefully, this additional verification effort has resulted in the development of a number of ACL2 theories that could be used in other formalizations. For example, the theory about directed acyclic graphs could be used in other formal verification projects of properties of pointer programs.

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. Corbin, J. and Bidoit, M. A Rehabilitation of Robinson's Unification Algorithm. *Information Processing 83*, pp. 909–914, North-Holland, 1983.
4. Greve, D. and Wilding, M. High-Speed, Analyzable Simulators. In *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 8. Kluwer Academic Publishers, 2000.
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 home page, 2005.
   URL: `http://www.cs.utexas.edu/users/moore/acl2`.
7. Moore, J S. An Exercise in Graph Theory. In *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 5. Kluwer Academic Publishers, 2000.
8. Ruiz–Reina, J.L., Alonso, J.A., Hidalgo, M.J. and Martín, F.J. Formal Proofs About Rewriting Using ACL2. *Annals of Mathematics and Artificial Intelligence 36*, pp. 239–262, Kluwer Academic Publishers, 2002.
9. 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.
   URL: `http://www.cs.utexas.edu/users/moore/acl2/workshops.html`.

10. Ruiz–Reina, J.L., Alonso, J.A., Hidalgo, M.J. and Martín, F.J. A Formally Verified Quadratic Unification Algorithm. In *Fourth ACL2 Workshop*, Austin, 2004.
URL: `http://www.cs.utexas.edu/users/moore/acl2/workshops.html`.
11. Ruiz-Reina, J.L., Alonso, J.A., Hidalgo, M.J., y Martín, F.J. A verified quadratic dag unification algorithm in ACL2, 2005.
URL: `www.cs.us.es/~jruiz/q-dag-unification`.
12. Steele, Jr., G. L. *Common Lisp The Language, second edition*. Digital Press, 1990.