Mechanical verification of a rule-based unification algorithm in the Boyer-Moore theorem prover

J.L. Ruiz-Reina, J.A. Alonso, M.J. Hidalgo and F.J. Martín
{jruiz,jalonso,mjoseh,fjesus}@cica.es
Departamento de Ciencias de la Computación e Inteligencia Artificial.
Facultad de Informática y Estadística, Universidad de Sevilla
Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain
Phone: 954552791 Fax: 954557970

Abstract

We describe here the formalization and mechanical proofs of the main properties of a unification algorithm. The verified unification algorithm is defined as a set of transformation rules acting on sets of equations, following Martelli and Montanari. These rules are non-deterministically applied until a solution (or the lack of solutions) is detected. This algorithm and its properties are formalized using the Boyer-Moore logic and carried out mechanically using the Boyer-Moore theorem prover. The language used is very similar to pure Lisp. We have formalized and mechanically proved a number of concepts and properties about first-order terms, substitutions, equations and transformation rules. We have also formalized the non-deterministic behaviour of the verified algorithm using selection functions.

Keywords: Unification, Mechanical Verification, Boyer-Moore Theorem Prover, First Order Terms.

1 Introduction

Unification is a central process in logic programming. Through its use of resolution, Prolog inherited unification as a fundamental operation. It is also important in a number of fields, including automated deduction, natural language processing and machine learning. The use of a theorem prover (the Boyer-Moore theorem prover in this case) to mechanically prove the correctness of a unification algorithm is interesting for some reasons:

- Formal methods are applied to an algorithm widely used in a number of systems. It is not rare to see some bugs in unification algorithms given in the literature (see, for example [8])[1].

This work has been supported by DGES/MEC: Projects PB96-0098-C04-04 and PB96-1345.

[1] The unification algorithm given in page 53 of [8] should compose the substitutions partially computed instead of using `append`.

- This is a non-trivial example of how a theorem prover can be used to verify an algorithm defined in a language similar to pure Lisp. It is shown how automated deduction can be used to examine and understand its properties with much greater detail, rigor, and clarity. In the following, when we talk about "prove", we mean "mechanically prove".
- Unification can be seen as a process that applies transformations to a set of equations until a solution (or the lack of solutions) is detected. This rule-based specification turns out to be suitable for mechanical verification. These transformations can be applied non-deterministically (i.e., every strategy for applying transformations leads to a most general solution, whenever it exists). We show how this non-determinism can be formalized and verified.

There is some related work done in mechanical verification of properties of unification algorithms. Paulson ([10]) describes the verification of a unification algorithm using the theorem prover LCF. Rouyer ([11]) does the same using Coq. Using the Boyer-Moore theorem prover, we get a higher degree of automation. Using a rule-based approach, we get more abstractness, verifying a family of algorithms, instead of a particular algorithm. For some related work in the Boyer-Moore theorem prover, see [5], where Kaufmann presents a proof of a generalization algorithm used in PC-Nqthm.

Due to the lack of space, we do not present details of the proofs here. The complete events files are available on the web in `http://www-cs.us.es/~jruiz/terms/`.

1.1 First order terms and substitutions

Given a set Σ of function symbols (called *signature*) and a denumerable set X of variable symbols, the set of *(first-order) terms* $T(\Sigma, X)$ is the smallest set containing X such that $f(t_1, \ldots, t_n) \in T(\Sigma, X)$ whenever each $t_i \in T(\Sigma, X)$ (when $n = 0$, we say that the term is a *constant*). Note that this definition allows function symbols with *variable arity*. The set of variables of a term t is denoted as $\mathcal{V}(t)$. A function $\sigma : X \to T(\Sigma, X)$ is a *substitution* if only finitely many variables $x_1, \ldots, x_n$ are not mapped to themselves. This is denoted as $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where $\sigma(x_i) = t_i$. A substitution σ can be extended to a function from terms to terms in such a way that $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$. A term t *matches* a term s if $\sigma(s) = t$ for some substitution σ. In that case, we write $s \leq t$, we say that t is an *instance* of s or that s *subsumes* t and we say that σ is a *matching substitution for s and t*. Also, a subsumption relation on substitutions can be defined by: $\sigma \leq \delta$ if there exists a substitution γ such that $\delta = \gamma \circ \sigma$, where $\circ$ stands for functional composition.

We say that a substitution σ *unifies* (or is a *unifier* of) two terms s and t if $\sigma(s) = \sigma(t)$. In that case we say that s and t are *unifiable*. Not every pair of terms is unifiable. We say that a unifier σ of s and t is a *most general unifier (mgu)* if for every unifier δ of s and t, $\sigma \leq \delta$. A *unification algorithm* is an algorithm that finds, whenever it exists, a mgu of two given terms. See [7] for a survey on unification.

1.2 The Boyer-Moore logic and theorem prover

We will briefly describe here the Boyer-Moore theorem prover (also known as Nqthm, precursor of ACL2) and its logic. For a complete description, see [1]. For a descrip-

tion of ACL2, see [6].

The Boyer-Moore logic is a quantifier-free first-order logic with equality. The language used is very similar to pure Lisp. There are two logical constants, abbreviated as `T` and `F`. The propositional connectives have functional analogues: `AND`, `IMPLIES`, `OR`, `IFF` etc. These connectives are all defined in terms of the primitive 3-place connective `IF`. The functional analogue of the equality predicate is `EQUAL`. The theory includes axioms and rules of inference for propositional logic, equality, and instantiation.

The *shell principle* allows the extension of the logic by addition of axioms defining a new data type. The initial theory includes axioms for natural numbers, ordered pairs, literal atoms, and negative integers. Natural numbers are defined by a constructor function `ADD1`, a destructor function `SUB1` and a base object, `(ZERO)`. Numbers are abbreviated by the numerals, `0`, `1`, `2`, etc. Ordered pairs are defined by a shell recognizer (`LISTP`), two destructor functions, (`CAR` and `CDR`), and one constructor function (`CONS`). The function `NLISTP` is the negation of `LISTP`. The functions `CAR` and `CDR`, when applied to a non-ordered pair, return `0`. As in Lisp, ordered pairs are used to construct finite sequences or lists. The literal atom `NIL` is conventionally used to represent the empty list (but `NIL` is not a `LISTP` object). We will call an object a *proper list* if it is `NIL`, or it is a `LISTP` and its `CDR` is proper (in other words, a proper list is `NIL` or a `LISTP` object with final tail `NIL`).

By the *principle of definition*, new function definitions are admitted in the theory only if there exists and ordinal measure in which the arguments of each recursive call decrease. This ensures that no inconsistencies are introduced by new definitions. The theory has a constructive definition of the ordinals up to ε_0, in terms of lists and natural numbers. One important rule of inference is the *principle of induction*, that permits proofs by induction on ε_0.

The *constraint principle* allows one to define functions partially by introducing some properties without complete definitions. To preserve consistency, a *witness* function must be given, having the same properties.

The Boyer-Moore theorem prover is a mechanization of their logic, including commands for adding new data types (shells), defining new functions, and proving theorems. The command `DEFN` allows one to define new functions, whenever they can be admitted by the principle of definition, and can include hints about the ordinal measure to be used. The shell principle is implemented by the command `ADD-SHELL`. The command `(PROVE-LEMMA name rule-classes statement hints)` starts the attempt of a proof. Usually, `rule-classes` is an empty list or `(REWRITE)`, in which case the lemma is stored as a rewrite rule if the proof attempt succeeds. If `ELIM` is one of the rule classes, the system stores the rule as an elimination rule. This means that the rule will be used to replace destructor terms in favour of equivalent constructor terms. The `CONSTRAIN` command mechanizes the constraint principle.

The main proof techniques used by the prover are simplification and induction. Simplification is a combination of decision procedures, mainly term rewriting, using the rewrite rules previously proved by the user. For a successful application of its principle of induction, the system has a number of heuristics to find inductive arguments suitable for a conjecture. Each recursive function in the conjecture "suggests" an induction scheme.

The theorem prover is automatic in one sense: once `PROVE-LEMMA` is invoked, the user can no longer interact with the system. However, the user can interact with the

prover by previously proving lemmas and definitions, used in the proofs as rewrite rules. Additionally, the user can give "hints" to the prover when `PROVE-LEMMA` is invoked. The `USE` hint forces the use of specific instances of previously proved lemmas. We mainly use the `USE` hint (instead of the rewriting mechanism) if free variables appear in the lemma to be used (see [1] for details). For readability, we will not print the hints of a `PROVE-LEMMA`, but we will comment it if needed.

2 Formalizing first-order terms and substitutions

In this section, we explain how we define terms and substitutions as objects in our logic, to reason about them.

We will not use any specific predicate for defining terms. Every object in the logic represents a first order term, with the following conventions. Every non-`LISTP` object will be considered as a variable symbol:

```
(defn variablep (x) (nlistp x))
```

Every `LISTP` object can be seen as the term with its `CAR` as the top function symbol and its `CDR` as the list of its arguments. In this way, every first order term for a given signature can be represented. For example, the term $k(x, a, g(y), h(x))$ is represented as `'(k x (a) (g y) (h x))`.

Using this wider representation, some objects represent *non-proper* terms, because the list of its arguments can be non-proper. For example, the term `'(k x (a) (g y) (h x) . 3)` has the same structure than the above term, but it is a different term. Since our lemmas don't use a predicate for defining terms, our results are also valid for these non-proper terms (this means that we are formalizing a theory that strictly contains first-order terms). Note that terms and lists of terms are defined by mutual recursion. Most of our results are proved also for lists of terms, with some minor modifications.

We represent substitutions as lists of ordered pairs and we will not use any specific function to define them: every object can be seen as the representation of a substitution. We use the following convention: if several pairs with the same first element appear in a substitution, only the first one is considered. The function `(VALUE x sigma)` returns the value assigned by the substitution `sigma` to `x`. Unlike terms, the same substitution can be represented by different association lists. Thus, we cannot use the predicate `EQUAL` when we talk about equality of substitutions. Instead, we will use some kind of functional equality in finite domains.

We define the application of a substitution to a term (or list of terms) by mutual recursion, using a standard trick in the Boyer-Moore logic. If `flg` is not `F`, `(APPLY flg sigma term)` is the term obtained by applying the substitution `sigma` to the term `term`. `(APPLY F sigma term)` is the list of terms obtained by applying `sigma` to the list `term` of terms:

```
(defn apply (flg sigma term)
  (if flg
      (if (variablep term) (value term sigma)
        (cons (car term) (apply F sigma (cdr term))))
    (if (nlistp term) term
      (cons (apply T sigma (car term))
            (apply F sigma (cdr term))))))
```

This style for defining functions on terms is typical in our formalization. Functions are defined for terms and for lists of terms, and properties are stated for terms and for lists of terms. This is specially well-suited for the heuristics of the system used to find an induction scheme: definitions by mutual recursion suggest an induction scheme analogue to induction on the term structure used in most of hand proofs in the literature. Let's see an example.

An elementary property says that if $\sigma(t) = \delta(t)$, then $\sigma(x) = \delta(x)$ for every $x \in \mathcal{V}(t)$. If we previously define `(EQUAL-IN-LIST sigma delta l)` as the function that tests if the substitutions `sigma` and `delta` take the same values on the variables of the list `l`, and `(VARIABLES flg term)` as the function that returns the list of variables of the term (or list of terms) `term`, then the property can be stated as follows (valid for terms and for list of terms):

```
(prove-lemma equal-in-term-implies-equal-in-variables (rewrite)
  (implies (equal (apply flg sigma term)
                  (apply flg delta term))
           (equal-in-list sigma delta (variables flg term))))
```

This command leads the prover to a proof attempt with the induction scheme in Figure 1, which is, in essence, an induction on the structure of `term` (here, `(p SIGMA DELTA FLG TERM)` is an abbreviation of the lemma to be proved).

```
(AND (IMPLIES (AND FLG (VARIABLEP TERM))                ; *1*
              (p SIGMA DELTA FLG TERM))
     (IMPLIES (AND FLG (NOT (VARIABLEP TERM)))          ; *2*
                       (p SIGMA DELTA F (CDR TERM)))
              (p SIGMA DELTA FLG TERM))
     (IMPLIES (AND (NOT FLG) (NLISTP TERM))             ; *3*
              (p SIGMA DELTA FLG TERM))
     (IMPLIES (AND (NOT FLG) (NOT (NLISTP TERM)))       ; *4*
                   (p SIGMA DELTA T (CAR TERM))
                   (p SIGMA DELTA F (CDR TERM)))
              (p SIGMA DELTA FLG TERM))).
```

Figure 1: An example of structural induction

Cases `*1*` and `*3*` are respectively the base cases for terms (variable) and lists of terms (empty list of terms). The inductive case `*2*` corresponds to non-variable terms and the induction hypothesis assumes the lemma true for the list of its arguments. The inductive case `*4*` corresponds to a non-empty list of terms. The induction hypothesis assumes the lemma true for the first term and for the list of the rest of the terms. The lemma is easily proved by the prover using this induction scheme, without help from the user.

Several functions on terms and substitutions are defined. Here is a brief description of some of them:

- `(SIZE flg term)`, the number of function symbols in a term or list of terms.
- `(DOMAIN sigma)`, the list of `CAR`'s of each element of `sigma`.
- `(CO-DOMAIN sigma)`, the list of `CDR`'s of each element of `sigma`.

- `(EXTENSION sigma1 sigma2)`, a predicate for testing if `sigma1` associates the same terms as `sigma2` for the variables of the domain of `sigma2`.
- `(RESTRICTION sigma l)`, the functional restriction of `sigma` to the variables in `l`.

3 Subsumption between terms and substitutions

3.1 A subsumption algorithm

The definition of subsumption between terms previously given is not constructive. Due to the restrictions of our logic, we need to define the subsumption relation between terms in a constructive way. That is, we will define a subsumption algorithm, that given two terms, finds, if it exists, a substitution that, when applied to the first term, gives the second. This function, given in Figure 2, is `(SUBSUMPTION flg t1 t2 sigma)` and receives as arguments one flag `flg`, two terms (or list of terms, depending on the value of `flg`) `t1` and `t2`, and a substitution `sigma`. We have to prove that it returns a substitution that extends `sigma` and that applied to `t1` gives `t2`, if it exists. If not, it returns `F`.

```
(defn subsumption (flg t1 t2 sigma)
  (if flg
      (if (variablep t1)
         (if (member t1 (domain sigma))
             (if (equal (value t1 sigma) t2) sigma F)
           (cons (cons t1 t2) sigma))
       (if (variablep t2) F
         (if (equal (car t1) (car t2))
             (subsumption F (cdr t1) (cdr t2) sigma) F)))
     (if (nlistp t1)
         (if (equal t1 t2) sigma F)
       (if (nlistp t2) F
         (let ((subs-first (subsumption T (car t1) (car t2) sigma)))
           (if subs-first
               (subsumption F (cdr t1) (cdr t2) subs-first)
             F))))))
```

Figure 2: A subsumption algorithm

We define subsumption between terms or lists of terms as the result of this subsumption algorithm starting with an empty set of bindings as the initial substitution. We call this function `subs*`:

```
(defn subs* (flg t1 t2) (subsumption flg t1 t2 nil))
```

We prove the two following lemmas stating that the algorithm `subs*` has the intended behaviour:

```
(prove-lemma subs*-soundness ()
    (implies (subs* flg t1 t2)
             (equal (apply flg (subs* flg t1 t2) t1) t2)))
```

```
(prove-lemma subs*-completeness ()
  (implies (equal (apply flg sigma t1) t2)
           (subs* flg t1 t2)))
```

3.2 The subsumption relation between terms

The two previous lemmas characterize exactly what we intended when we defined the subsumption relation. Thus, we can define the subsumption relation as a function, called `subs`, with these two key properties, using the `CONSTRAIN` command (Figure 3). To preserve consistency, we have to exhibit a witness function, `subs*` in our case, having the same properties.

The use of `CONSTRAIN` assures that we will not use any other particular property of our subsumption algorithm: for example, although the substitution returned by our algorithm only binds variables in `t1`, we cannot use this. That means that our proofs will be valid also if we use another subsumption algorithm with the same two fundamental properties.

```
(constrain subsumption-definition ()
  (and (implies (subs flg t1 t2)
                (equal (apply flg (subs flg t1 t2) t1) t2))
       (implies (equal (apply flg sigma t1) t2)
                (subs flg t1 t2)))
  ((subs subs*))
  ((use (subs*-soundness) (subs*-completeness))))

(prove-lemma subsumption-soundness (rewrite elim)
  (implies (subs flg t1 t2)
           (equal (apply flg (subs flg t1 t2) t1) t2)))

(prove-lemma subsumption-completeness ()
  (implies (equal (apply flg sigma t1) t2)
           (subs flg t1 t2)))
```

Figure 3: Subsumption: definition and rules.

In Figure 3 we also give two lemmas for `subs`, one for each fundamental property. Note that `subsumption-soundness` is stored both as rewrite rule and elimination rule. The use of the elimination rule is especially fruitful here: in a proof attempt, if `(subs flg t1 t2)` is among the assumptions, then the prover will substitute `t2` by `(apply flg x t1)`, for some substitution `x`. The use of the lemma `subsumption-completeness` is not so automatic, because of the free variable `sigma`. The standard way to prove a property of the form `(subs flg t1 t2)` is to find a witness matching substitution and use completeness making it explicit with an `USE` hint. For example, transitivity of subsumption can be easily proved using `subsumption-completeness` as follows:

```
(prove-lemma subsumption-transitive ()
  (implies (and (subs flg t1 t2) (subs flg t2 t3))
           (subs flg t1 t3))
  ((use (subsumption-completeness
```

```
(sigma (composition (subs flg t2 t3) (subs flg t1 t2)))
(t2 t3)))))
```

Thus, we first state and prove properties at the level of instances and then we reformulate the lemmas using the subsumption relation.

3.3 Subsumption between substitutions

It is evident from the given definition of most general unifier that if we want to express the formal properties of our unification algorithm, we have to define the notion of subsumption between substitutions. The definition commonly given in the literature is $\sigma \leq \delta \iff \exists\gamma(\gamma \circ \sigma = \delta)$. This definition is not suitable for our logic, due to two reasons. First, we have to find a "witness" substitution to eliminate the existential quantification. Second, we cannot state functional equality between substitutions because this needs the use of universal quantification.

Instead, we will use an equivalent[2] definition:

$$\sigma \leq \delta \iff \forall t(\sigma(t) \leq \delta(t))$$

We can remove the universal quantifier by paying attention only to the variables of the domains of σ and δ and the variables of the range of σ (returned by the function `IMPORTANT-VARIABLES`). Thus, the following is our definition of subsumption between substitutions:

```
(defn subs-subst (sigma delta)
  (let ((V (important-variables sigma delta)))
    (subs F (apply F sigma V) (apply F delta V))))
```

We prove that this definition of subsumption between substitutions is equivalent to the intended definition. The following lemma state its main property[3]:

```
(prove-lemma subs-subst-main-property (rewrite)
  (implies (subs-subst sigma delta)
      (subs flg (apply flg sigma term) (apply flg delta term))))
```

This lemma is proved using the same matching substitution for all terms. In other words, if `(subs-subst sigma delta)` we find in a constructive way, a substitution such that composed with `sigma` is functionally equal to `delta`. This matching substitution is:

```
(defn subs-sust-restriction (sigma delta)
  (restriction (subs-subst sigma delta)
               (important-variables sigma delta)))
```

It is worth pointing that we cannot assure that `subs-subst` is the witness matching substitution, although with our subsumption algorithm, `subs-subst` and `subs-sust-restriction` are the same (when they succeed). But this is a particular property of our subsumption algorithm and cannot be proved using only the two characteristic properties (soundness and completeness), a limitation we imposed ourselves to build a more general theory.

[2] If at least we have a binary function symbol, which is our case.

[3] Note that we don't need to prove the reverse implication because this is a trivial consequence of our definition.

4 Transforming system of equations

Unification can be seen as an algorithm to solve term equations or, more generally, systems of equations. We will define an algorithm for solving systems of equations using the *transformation method*, which transforms systems of equations until the solution is obvious. This was already anticipated in Herbrand's thesis ([3]) and was used for the first time in the context of unification by Martelli and Montanari ([9]). In this moment, it is a standard formalism for discussing unification algorithms ([4], [2]).

4.1 Systems of equations

An equation is a pair of terms, denoted as $t_1 =^? t_2$. A substitution σ is a solution of $t_1 =^? t_2$ 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 member of S. The system obtained by applying the substitution σ to the system S is denoted as σS. A system with no solution is called *unsolvable*, and *solvable* otherwise. A solution of S is a *most general solution (mgs)* if it subsumes every other solution of S. Note that a substitution σ is a mgu of t_1 and t_2 if, and only if, it is a mgs of the system $\{t_1 =^? t_2\}$, so if we have an algorithm for finding an mgs of a system of equations, in particular we have a unification algorithm. Our unification algorithm, called `mgs` and defined later, finds, whenever it exists, a most general solution of a given system of equations, following Martelli and Montanari's.

In our formalization, we will represent equations as ordered pairs of terms and systems of equations as lists of equations. We also consider an special system, `F`, representing unsolvability. Note that every substitution can be seen as a system. As usual, we will not use any specific predicates for defining equations or systems of equations.

In our definition, the transformation rules are defined on pairs of systems. For the sake of readability, we will define a new data type, `PAIRP`, to construct pairs of systems:

```
(add-shell pair nil pairp ((first (none-of) zero)
                           (second  (none-of) zero)))
```

We now briefly describe some useful functions over systems of equations:

- `(SOLUTION sigma S)`, tests if `sigma` is a solution of the system `S`.
- `(SYSTEM-VARS S)`, the list of variables of the terms in the equations of `S`.
- `(RANGE-VARS S)`, the list of variables of the terms in the right-hand side of equations of `S`.
- `(APPLY-SYSTEM sigma S)`, the system obtained by applying the substitution `sigma` to every term in the equations of `S`.
- `(APPLY-RANGE sigma S)`, the system obtained by applying the substitution `sigma` to every term in the right-hand side of equations of `S`.
- `(PAIR-ARGS l m)`, the system obtained pairing the respective elements of the lists of terms `l` and `m` if they have the same length and final tail, `F` otherwise.

4.2 Idempotent substitutions and solved systems

We say that a substitution σ is *idempotent* if $\sigma = \sigma \circ \sigma$. Again, this definition is not suitable for the Boyer-Moore logic. Instead, we define the function `(IDEMPOTENT S)`, that tests if `S` is a system whose `domain` is a set of variables, disjoint from `(range-vars S)`. Note that we are exploiting here that a system of equations can be seen as a substitution. Idempotent substitutions are also called systems in *solved form*, depending on the context.

The following is the main lemma for expressing the relationship between subsumption and solution of systems, and exploits that substitutions can be seen as systems:

```
(prove-lemma main-property-mgs (rewrite)
  (implies (solution sigma delta)
           (equal (apply flg sigma (apply flg delta term))
                  (apply flg sigma term))))
```

In other words, if σ is a solution of δ, then $\sigma = \sigma \circ \delta$, and, consequently, $\delta \leq \sigma$. This means that if δ is a solution of itself, it is the least such solution with respect to the subsumption ordering. Idempotent substitutions (or systems in solved forms) are solutions, (and therefore most general solutions) of themselves.

```
(prove-lemma idempotent (rewrite)
  (implies (idempotent S) (solution S S)))
```

Note that the above two lemmas confirm that our definition of idempotency is the intended: if σ is idempotent, then σ is a solution of σ and, using the main property, $\sigma = \sigma \circ \sigma$.

4.3 Transformation rules and selection function

The transformation rules given by Martelli and Montanari ([9]) appears in Figure 4. This set of rules will suffice to solve every system of equations, as we will prove. Note that the rules act on pairs of systems of equations, denoted as $S;T$. The first system contains the equations to be solved and the second one the solved equations.

To solve a system of equations S, we begin with the pair of systems $S;\emptyset$ and apply the transformation rules until unsolvability (i.e., `F`) is detected or a pair of systems in the form $\emptyset;T$ appears. To apply a rule, an equation is *selected* in the system of non-solved equations, and the form of this equation determines the rule to apply. This kind of non-determinism can be formalized in the Boyer-Moore logic using a constrained definition of a selection function. The only property that we will assume is that the selection function chooses an element of non-empty systems. The following `CONSTRAIN` command defines the selection function `sel`. The witness function we use in this case is `car`.

```
(constrain selection-function (rewrite)
  (implies (listp l) (member (sel l) l))
  ((sel car)))
```

The function `TRANSFORM` in Figure 5 implements the rules of transformation. It applies, in a non-deterministic way, one of the rules, to perform one step of transformation.

Delete:	$\{t =^? t\} \cup R; T$	$\Rightarrow R; T$
Check:	$\{x =^? t\} \cup R; T$	$\Rightarrow F$ if $x \in \mathcal{V}(t)$, $x \neq t$
Eliminate:	$\{x =^? t\} \cup R; T$	$\Rightarrow \{x \mapsto t\}R; \{x =^? t\} \cup \{x \mapsto t\}T$ if $x \in X$ y $x \notin \mathcal{V}(t)$
Decompose:	$\{f(s_1, \ldots, s_n) =^? f(t_1, \ldots, t_n)\} \cup R; T$	$\Rightarrow \{s_1 =^? t_1, \ldots, s_n =^? t_n\} \cup R; T$
Conflict:	$\{f(s_1, \ldots, s_n) =^? g(t_1, \ldots, t_n)\} \cup R; T$	$\Rightarrow F$ if $f \neq g$
Not-pair:	$\{f(s_1, \ldots, s_n) =^? f(t_1, \ldots, t_m)\} \cup R; T$	$\Rightarrow F$ if $n \neq m$
Orient:	$\{t =^? x\} \cup R; T$	$\Rightarrow \{x =^? t\} \cup R; T$ if $x \in X$, $t \notin X$

Figure 4: Transformation rules

5 The unification algorithm

5.1 Applying transformations non-deterministically

We define a function `solve` to apply transformations to pairs of systems of equations until a *normal form* is reached. A pair of systems is in normal form if it is `F` or if the first system is empty:

```
(defn normal-form-syst (S-sol)
  (or (nlistp (first S-sol)) (not S-sol)))
```

The definition of `solve` is very simple but its admission is not trivial. To prove its termination, we have to define a measure function, `unification-measure`, on pair of systems:

```
(defn unification-measure (S-sol)
  (cons (cons (add1 (n-system-var (first S-sol)))
              (size-system (first S-sol)))
        (n-variables-right-hand-side (first S-sol))))
```

The function `unification-measure` is a lexicographic combination of:

1. The number of distinct variables in the first system, `n-system-var`.
2. The number of function symbols in the first system, `size-system`.
3. The number of equations in the first system with a variable in its right-hand side, `n-variables-right-hand-side`.

Lemmas have been proved for each of the rules of transformation and each of these three quantities, some to prove that the quantity remains the same, some to prove that it decreases, in each step of transformation. With these lemmas, and a hint about the measure, the following definition is admitted:

```
(defn transform (S-sol)
  (let ((S (first S-sol)) (sol (second S-sol)))
    (let ((equ (sel S)))
      (let ((t1 (car equ)) (t2 (cdr equ)) (R (delete equ S)))
        (cond
          ((equal t1 t2) (pair R sol))              ;;; *DELETE*
          ((variablep t1)
             (if (member t1 (variables t t2))
                 F                                  ;;; *CHECK*
                 (pair                              ;;; *ELIMINATE*
                   (apply-system (list equ) R)
                   (cons equ (apply-range (list equ) sol)))))
          ((variablep t2)
             (pair (cons (cons t2 t1) R) sol))      ;;; *ORIENT*
          ((not (equal (car t1) (car t2))) F)       ;;; *CONFLICT*
          (t (let ((pairing (pair-args (cdr t1) (cdr t2))))
               (if pairing
                   (pair (append pairing R) sol)    ;;; *DESCOMPOSE*
                 F))))))))                          ;;; *NOT-PAIR*
```

Figure 5: Transformation rules

```
(defn solve (S-sol)
 (if (normal-form-syst S-sol) S-sol (solve (transform S-sol)))
 ((ord-lessp (unification-measure S-sol))))
```

Note that this function is not completely specified because the selection function used in `transform` is only partially defined. For every particular selection function there exists an "instance" of `solve`, that takes a pair of systems and applies the rules of transformation, with that specific selection criterion, until a normal form is reached. Conversely, for every sequence of transformations starting from a pair of systems and ending in a normal form, there exists a particular selection function for which the corresponding "instance" of `solve` returns this normal form when applied to the initial pair of systems, performing the given sequence of transformations. In this sense, we are formalizing *non-determinism*, and at the same time verifying the formal properties of a number of unification algorithms, that apply the rules with some specific selection criterion that can be seen as a control strategy.

5.2 Invariants of the transformations

The two key properties of the given rules of transformation can be stated in terms of invariants. There are two invariants in any sequence of transformations:

- The set of solutions of both systems of the pair.
- The idempotency of the second system (if unsolvability is not detected).

The following lemmas state that the transformations preserve the set of solutions (here, `(union-systems S-sol)` is the union of the systems that form the pair `S-sol`):

```
(prove-lemma transform-equivalent (rewrite)
  (implies (and (pairp S-sol) (listp (first S-sol))
                (transform S-sol))
           (iff (solution sigma (union-systems (transform S-sol)))
                (solution sigma (union-systems S-sol)))))

(prove-lemma transform-unsolvable (rewrite)
  (implies (and (pairp S-sol) (listp (first S-sol))
                (not (transform S-sol)))
             (not (solution sigma (union-systems S-sol)))))
```

Idempotency of the second system is preserved if we also have an additional invariant: the variables of the domain of the second system does not appear in the first system (i.e., the variables are *solved*). The following lemma states this:

```
(prove-lemma transform-preserves-idempotency (rewrite)
  (let ((transformed (transform (pair S sol))))
    (let ((St (first transformed)) (solt (second transformed)))
      (implies (and (listp S)
                    transformed
                    (idempotent sol)
                    (disjoint (system-vars S) (domain sol)))
          (and (idempotent solt)
               (disjoint (system-vars St) (domain solt)))))))
```

To prove the lemmas stated in this subsection, we have to prove previously analogous lemmas for each of the transformation rules that `transform` may apply.

5.3 A unification algorithm

The function `solve` is used to define an algorithm that finds a most general solution of a system of equations, whenever it exists. We define `mgs`, a function acting on systems of equations in the following way:

```
(defn mgs (S)
  (let ((solved (solve (pair S nil))))
    (if solved (second solved) F)))
```

This function applies the rules of transformation starting with $S; \emptyset$, until a normal form is reached. If unsolvability is detected, it returns `F`, otherwise returns the second system of the pair. The previous properties are used to verify that `mgs` implements a correct unification algorithm. The fundamental properties of the function `mgs`, appear in Figure 6. These properties are a formalization of the following theorem, and are the desired properties of a *correct* unification algorithm:

Theorem: The function `mgs` has the following properties:

1. If `S` is a solvable system of equations, then `mgs` returns a non-`F` value (i.e., the algorithm succeeds). This is stated in the lemma `completeness-mgs`.

2. If `mgs` succeeds when applied to a system `S`, then:

```
(prove-lemma completeness-mgs ()
  (implies (solution sigma S) (mgs S)))

(prove-lemma soundness-mgs ()
  (implies (mgs S) (solution (mgs S) S)))

(prove-lemma most-general-solution-mgs ()
  (implies (solution sigma S) (subs-subst (mgs S) sigma)))

(prove-lemma idempotent-mgs ()
  (idempotent (mgs S)))
```

Figure 6: Properties of the `mgs` algorithm

(a) It returns a solution of `S`. This is the lemma `soundness-mgs`.

(b) It returns a substitution that subsumes every solution of `S`. This the lemma called `most-general-solution-mgs`.

(c) It returns an idempotent substitution. The lemma `idempotent-mgs` state this property.

Note that because of the definition of `solve` and the use of a selection function, we have also proved that *any* sequence of transformations starting with $S;\emptyset$ ends in $\emptyset;T$, with T a most general solution, if S is solvable, or in `F` if S is unsolvable. This property is called *non-deterministic completeness*.

An interesting remark can be made here. Our defined algorithm can be seen as a *pattern* of different unification algorithms. For every particular strategy in applying transformation rules, we have a concrete unification algorithm. This particular strategy corresponds to an instantiation of the partially defined selection function `sel`. The correctness of all those concrete unification algorithms are a trivial consequence of the above theorem, and are easily proved by the system using the command `FUNCTIONALLY-INSTANTIATE`.

6 Conclusions

We have presented here a successful application of the Boyer-Moore theorem prover to the proof of correctness of a unification algorithm. This algorithm is described in terms of transformation rules acting non-deterministically on system of equations. To perform the mechanical proofs, we have formalized a theory of terms and substitutions in the Boyer-Moore logic.

The use of rules as a formalism to represent algorithms, unification algorithms in this case, turns out to be well-suited for mechanical verification. The use of selection functions, partially defined, formalizes the non-deterministic behaviour of this kind of algorithms, getting a clearer separation of logic and control.

This is a first step to provide a number of basic results needed to apply formal methods to the design of reasoning systems. This mechanical proof is part of the mechanical proof of lattice-theoretic properties of first-order terms ([12]). As we mentioned earlier, terms and their properties are used in most of fields in Artificial Intelligence and Declarative Programming.

References

[1] Boyer, R., and Moore, J. *A Computational Logic Handbook*, 2nd ed. Academic Press, 1998.

[2] Gallier, J., and Snyder, W. Designing unification procedures using transformations: A survey. *Bulletin of the EATCS*, 40 (1990), 273–326.

[3] Herbrand, J. *Recherches sur la Théorie de la Démonstration.* PhD thesis, Sorbonne, Paris, 1930.

[4] Jouannaud, J.-P., and Kirchner, C. Solving equations in abstract algebras: A rule based survey of unification. In *Computational Logic: Essays in honour of Alan Robison* (1992), J.-L. Lassez and G. Plotkin, Eds., MIT Press.

[5] Kaufmann, M. Generalization in the presence of free variables: A mechanically checked proof for one algorithm. *Journal of Automated Reasoning 7*, 1 (1991), 109–158.

[6] Kaufmann, M., and Moore, J. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering 23*, 4 (1997), 203–213.

[7] Lassez, J.-L., Maher, M., and Marriott, K. Unification revisited. In *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, 1988, pp. 587–625.

[8] Lucas, P., and Van der Gaag, L. *Principles of Expert Systems.* Addison Wesley, 1991.

[9] Martelli, A., and Montanari, U. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems 4*, 2 (1982), 258–282.

[10] Paulson, L. Verifying the unification algorithm in LCF. *Science of Computer Programming*, 5 (1985).

[11] Rouyer, J. Développement de l'algorithme d'unification dans le calcul des constructions avec types inductifs. Tech. Rep. 1795, INRIA Lorraine, 1992.

[12] Ruiz-Reina, J., Alonso, J., Hidalgo, M., and Martín, F. Formalizing properties of first-order terms in the Boyer-Moore logic. Tech. rep., CCIA, University of Sevilla, 1999. `http://www-cs.us.es/~jruiz/terms/`.