Verification of the Completeness of Unification Algorithms à la Robinson

Andréia B. Avelar[1], Flávio L. C. de Moura[2], André Luiz Galdino[3] and Mauricio Ayala-Rincón[1,2]⋆

Departamentos de [1]Matemática e [2]Ciência da Computação

Universidade de Brasília, Brasília, Brazil

[3]Departamento de Matemática, Universidade Federal de Goiás, Catalão, Brazil

{andreia@mat, **ayala@**, flaviomoura@, galdino@}**.unb.br**

Abstract. This work presents a general methodology for verification of the completeness of first-order unification algorithms à la Robinson developed in the higher-order proof assistant PVS. The methodology is based on a previously developed formalization of the theorem of existence of most general unifiers for unifiable terms over first-order signatures. Termination and soundness proofs of any unification algorithm are proved by reusing the formalization of this theorem and completeness should be proved according to the specific way in that non unifiable inputs are treated by the algorithm.

1 Introduction

In a previous development, done in the PVS proof assistant [ORS92], a formalization of the theorem of existence of most general unifiers (mgu's) for unifiable terms over first-order theories was presented. That development was given as the PVS *theory* `unification` [AdMARG10]. The formalization was based on three constructive operators: given a pair of unifiable terms as input, the first one generates the first position of conflict whenever the terms are different; the second one builds a resolution for the conflict and; the third one builds an mgu. These operators use the powerful machinery of types available in PVS in order to build a dependent type of pairs of unifiable terms as input. Thus, these operators correspond to a unification algorithm restricted to unifiable terms in the style of Robinson's original unification algorithm [Rob65]. This theorem of existence is enough for several applications, as for instance, for a formalization of the well-know Knuth-Bendix Critical Pair Theorem [KB70] presented in [GAR10]. The failure cases that appear for non unifiable terms are not treated in that formalization. But all the proof techniques applied are reusable as a general methodology useful to verify termination and soundness of unification algorithms in this style of unification. The verification of completeness of any unification algorithm depends upon proving that the specific treatment of the failure cases given by the unification algorithm is adequate.

⋆ Corresponding author

In [Rob65], a constructive proof of correctness of the unification algorithm was introduced in order to prove, by contradiction, the completeness of the resolution method for the propositional calculus. The introduced unification algorithm either gives as output an mgu for each unifiable pair of terms, or fails whenever the terms are not unifiable. The proof of correctness of this algorithm consists in proving that the algorithm always terminates, and that, when it terminates an mgu is provided if and only if the terms are unifiable. Several variants of this first-order unification algorithm appear in well-known textbooks on computational and mathematical logic, semantics of programming languages, rewriting theory, etc (e.g., [Llo87, EFT84, Bur98, BKdV03, BN98]). Since the formalization follows the classical proof schema, which is one of the main positive aspects of the current work, no analytic presentation of this proof will be given here.

In order to illustrate the general proof methodology, a specification of a unification algorithm is provided in the PVS *theory* `robinsonunification` inside the PVS library `trs` for term rewriting systems that was introduced in [GAR09] and is available at `http://ayala.mat.unb.br/publications.html`. In addition, the *theory* `unification` is available inside the library `trs`.

Section 2 presents a specification of a unification algorithm à la Robinson; Section 3 explains how the verification methodology works in order to prove termination and soundness for the case of unifiable terms. Section 4 illustrates the solution to prove completeness, that is, it show how to deal with failure in unification. Related work and conclusions are then presented. Familiarity with the notions and notations related with unification theory is assumed, but for the benefit of the refeering process two appendices are included, in which basic notions are given and it is explained how these notions were specified.

2 Specification of unification algorithms

The methodology of verification of first-order unification algorithms is based on the formalization of the existence of first-order mgu's as presented in the *theory* `unification` which consists of 57 lemmas from which 30 are *type proof obligations* or *type correctness conditions* (TCCs) that are lemmas automatically generated by the type-checker of the prover. The specification file has 272 lines and its size is 9.5 KB and the proof file has 11.424 lines and 638.4 KB. The verification of the completeness of a unification algorithm is given in the *theory* `robinsonunification` and consists of 49 lemmas from which 25 are TCCs in a specification file of 252 lines of code (9.0 KB) and a file of proofs of 12.397 lines of proofs (747.8 KB).

Basic notions on unification are specified straightforwardly in the language of PVS (see also the appendices). For instance the notion of most general substitution is given as

```
<=(theta, sigma): bool = EXISTS tau: sigma = comp(tau, theta)
```

From this definition, one proves that the relation `<=` is a pre-order, that is, it is reflexive and transitive. The notions of unifier, unifiable, the set of unifiers of two terms and a mgu of two terms are defined as

```
unifier(sigma)(s,t): bool = ext(sigma)(s) = ext(sigma)(t)

unifiable(s,t): bool = EXISTS sigma: unifier(sigma)(s,t)

U(s,t): set[Sub] = {sigma: Sub | unifier(sigma)(s,t)}

mgu(theta)(s,t): bool =
  member(theta, U(s,t)) & FORALL sigma: member(sigma, U(s,t)) => theta <= sigma
```

The key point of the proposed general methodology of proof is to reuse the proof techniques inside the theory `unification`. In this *theory*, a unification algorithm, called `unification_algorithm`, restricted to unifiable terms, is given for which the main two properties formalized are:

- the restricted algorithm **terminates** and
- it is **sound**, that is, it gives as output an mgu of the (unifiable) inputs.

Thus, reusing the proof techniques for formalizing these two properties, it is possible to complete the verification of any unification algorithm that has as input two terms that may not be unifiable. What remains in order to verify a unification algorithm is to prove the **completeness** of the specific treatment of the exception cases; i.e., to prove the completeness of the treatment of non unifiable terms according to the specific algorithmic methodology.

The unification algorithm inside `unification` receives two unifiable terms as arguments, gives a substitution as output and is specified as follows:

```
unification_algorithm(s: term, (t: term | unifiable(s,t))):
  RECURSIVE Sub =
    IF s = t THEN identity
    ELSE LET sig = sub_of_frst_diff(s, t) IN
      comp( unification_algorithm((ext(sig))(s), (ext(sig)(t))) , sig)
    ENDIF
  MEASURE Card(union(Vars(s), Vars(t)))
```

In this specification, the function `sub_of_frst_diff` provides the *linkage substitution*, that is the one that resolves the first conflict appearing from left to right between the two terms `s` and `t`. The proof of the existence of this linkage substitution, that is a link from a variable to a term without occurrences of this variable is formalized inside the *theory* `unification` and the methodology of proof is reusable for any unification algorithm in the Robinson style. In the *theory* `robinsonunification`, the type dependence on the parameters `t` and `s` is eliminated in order to obtain a constructive unification algorithm for unrestricted terms. In general, completeness of any algorithm should be proved guaranteeing that it detects all possible fail cases, that is, conflicts without resolution, whenever the terms are not unifiable. Inside robinsonunification is specified a unification algorithm as the operator robinson_unification_algorithm.

```
robinson_unification_algorithm(s, t: term): RECURSIVE Sub =
   IF s = t THEN identity
   ELSE LET sig = link_of_frst_diff(s,t) IN
      IF sig = fail THEN fail
      ELSE
      LET sigma = robinson_unification_algorithm(ext(sig)(s) , ext(sig)(t)) IN
             IF sigma = fail THEN fail ELSE comp(sigma, sig) ENDIF
      ENDIF
   ENDIF
MEASURE Card(union(Vars(s), Vars(t)))
```

This operator calls the function link_of_frst_diff, that in contrast to the function sub_of_frst_diff, used by the unification_algorithm operator, allows as parameters different unrestricted terms and gives as output either "fail" or a linkage substitution, whenever the first found conflict between the terms is solvable. The key point of any unification algorithm à la Robinson is exactly the way which unresolved conflicts are reported.

The operator link_of_frst_diff has as parameters two different terms and invokes the operator first_diff that returns the position of the first conflict between these terms.

```
link_of_frst_diff(s: term , (t: term | s /= t )): Sub =
 LET k: position = first_diff(s,t) IN
    LET sp = subtermOF(s,k) , tp = subtermOF(t,k) IN
       IF vars?(sp)
       THEN IF NOT member(sp, Vars(tp))
            THEN (LAMBDA (x: (V)): IF x = sp THEN tp ELSE x ENDIF)
            ELSE fail ENDIF
       ELSE
          IF vars?(tp)
          THEN IF NOT member(tp, Vars(sp))
               THEN (LAMBDA (x: (V)): IF x = tp THEN sp ELSE x ENDIF)
               ELSE fail ENDIF
          ELSE fail ENDIF
       ENDIF
```

The specification of the operator first_diff is presented below. The parameters of this operator are two unrestricted, but different terms.

```
first_diff(s: term, (t: term | s /= t ) ):
  RECURSIVE position =
   (CASES s OF
      vars(s): empty_seq,
      app(f, st):
      IF length(st) = 0 THEN empty_seq
      ELSE
       (CASES t OF
          vars(t): empty_seq,
          app(fp, stp):
          IF f = fp THEN
            LET k: below[length(stp)] =
               min({kk: below[length(stp)] |
                         subtermOF(s,#(kk+1)) /= subtermOF(t,#(kk+1))}) IN
               add_first(k+1,
                      first_diff(subtermOF(s,#(k+1)),subtermOF(t,#(k+1))))
          ELSE empty_seq ENDIF
        ENDCASES)
      ENDIF
    ENDCASES)
 MEASURE s BY <<
```

Inside the theory `unification` the functions `resolving_diff` and `sub_of_frst_diff` play the same role as the functions `first_diff` and `link_of_frst_diff`, respectively, but the latter can receive as argument non unifiable terms.

3 Reusing the proof technology: termination and soundness

Exactly the same proof technology applied in the *theory* `unification` is possible for formalizing the properties of the corresponding operators in `robinsonunification` for unifiable inputs. In what follows, it is explained how the properties of termination and soundness are formalized for unifiable inputs inside the former *theory*.

Termination The formalization of this property follows the usual proof methodology: to prove that after each recursive input the measure, that is given by the number of variables occurring in the terms, decrease. The measure of the operator `unification_algorithm` is the cardinality of the union of the sets of variables occurring in its parameters `s` and `t`. The PVS type-checker automatically generates the type proof obligation below that guarantees termination.

```
unification_algorithm_TCC6: OBLIGATION
  FORALL (s, (t | unifiable(s, t))):
    NOT s = t IMPLIES
     (FORALL (sig: Sub):
        sig = sub_of_frst_diff(s, t) IMPLIES
         Card(union(Vars(ext(sig)(s)), Vars(ext(sig)(t))))
          <
          Card(union(Vars(s), Vars(t))))
```

This TCC is not automatically proved and it requires the proof of the auxiliary lemma:

```
vars_ext_sub_of_frst_diff_decrease: LEMMA
    FORALL (s: term, t: term | unifiable(s, t) & s /= t):
        LET sig = sub_of_frst_diff(s, t) IN
           Card(union( Vars(ext(sig)(s)), Vars(ext(sig)(t))))
           < Card(union( Vars(s), Vars(t)))
```

The proof of this lemma requires the existence of a linkage substitution σ for the first conflicting position, which maps a variable into a term without occurrences of this variable. This guarantees that the mapped variable disappears from the instantiated terms $\hat{\sigma}$`(s)` and $\hat{\sigma}$`(t)`, and hence the decreasingness property holds.

Soundness Inside the *theory* `unification` the correctness of the restricted unification algorithm is given by the lemma:

```
unification: LEMMA  unifiable(s,t) => EXISTS theta: mgu(theta)(s,t)
```

The proof of this lemma is obtained from two auxiliary lemmas: the first one, states that the substitution given by the operator `unification_algorithm` is, in fact, a unifier and the second one that it is an mgu.

```
unification_algorithm_gives_unifier: LEMMA
    unifiable(s,t) IMPLIES member(unification_algorithm(s, t), U(s, t))


unification_algorithm_gives_mg_subs: LEMMA
    member(rho, U(s, t)) IMPLIES unification_algorithm(s, t) <= rho
```

The former lemma is proved by induction on the cardinality of the set of variables occurring in `s` and `t`, for which, three auxiliary lemmas are necessary:

- the lemma `vars_ext_sub_of_frst_diff_decrease` described in the previous subsection, which guarantees that the cardinality of the set of variables decreases;

– the lemma

```
ext_sub_of_frst_diff_unifiable: LEMMA
    FORALL (s: term, t: term | unifiable(s, t) & s /= t):
        LET sig = sub_of_frst_diff(s, t) IN
           unifiable(ext(sig)(s), (ext(sig)(t)))
```

which states that the instantiations of two different and unifiable terms $\mathtt{s}\hat{\sigma}$ and $\mathtt{t}\hat{\sigma}$ with the substitution σ that resolves the first conflict between these terms, are still unifiable; and

– the lemma `unifier_o` presented at the beginning of this section, which states that for any unifier θ of $\mathtt{s}\hat{\sigma}$ and $\mathtt{t}\hat{\sigma}$, $\theta \circ \sigma$ is a unifier of `s` and `t`.

The formalization of the lemma `unification_algorithm_gives_mg_subs` is done by induction on the same measure. For proving this lemma two auxiliary lemmas are applied: the lemma `vars_ext_sub_of_frst_diff_decrease` and the lemma presented below, which states that for each unifier ρ of `s` and `t`, two different and unifiable terms, and given σ the substitution that resolves the first difference between these terms, there exists θ such that $\theta \circ \sigma = \rho$.

```
sub_of_frst_diff_unifier_o: LEMMA
    FORALL (s: term, t: term | unifiable(s, t) & s /= t):
        member(rho, U(s, t)) IMPLIES
           LET sig = sub_of_frst_diff(s, t) IN
              EXISTS theta: rho = comp(theta, sig)
```

4 Treatment of exceptions: proof of completeness

The *theory* `robinsonunification` illustrates the application of the methodology of proof. The main operators inside this *theory* give a treatment of failing cases in such a way that whenever unsolvable conflicts between non unifiable terms are detected (by the operator `first_diff`) the substitution "`fail`" is returned. This substitution is built explicitly as the substitution with the singleton domain {`xx`} and image `ff(xx)`, where `xx` and `ff` are, respectively, a specific variable and a unary function symbol. In this way, the substitution `fail` is discriminated from any other possible unifier which is built by the function `robinson_unification_algorithm`, for all pair of terms. The formalization of the property of termination follows the same lines of the lemma `vars_ext_sub_of_frst_diff_decrease` from the *theory* `unification`.

```
termination_lemma: LEMMA
  FORALL (s: term, t: term | s /= t):
         LET sig = link_of_frst_diff(s, t) IN
           NOT sig = fail IMPLIES
```

```
            Card(union( Vars(ext(sig)(s)), Vars(ext(sig)(t))))
            < Card(union( Vars(s), Vars(t)))
```

Similarly, the formalization of the property of soundness is followed in order to verify the lemmas below.

```
robinson_unification_algorithm_gives_unifier: LEMMA
   unifiable(s,t) IFF member(robinson_unification_algorithm(s, t), U(s, t))

robinson_unification_algorithm_gives_mg_subs : LEMMA
   member(rho, U(s, t)) IMPLIES
       robinson_unification_algorithm(s, t) <= rho
```

The former states that the algorithm gives as output a unifier of the input terms, whenever they are unifiable, and the latter that the output is in fact an mgu of the input terms.

In order to obtain completeness, two additional lemmas that distinguish the selected `fail` substitution from any possible unifier are necessary. These lemmas respectively state that, for unifiable inputs, the substitution built by the operator `robinson_unification_algorithm` has as domain a subset of variables occurring in the input terms, and as range terms whose variables also range in this set and that conform a set of variables disjoint from the domain. This distinguish the substitution `fail` from any resolving substitutions.

```
 rob_uni_alg_dom_subset_union_vars: LEMMA
   unifiable(s, t) IMPLIES
      LET sigma = robinson_unification_algorithm(s, t) IN
         subset?( Dom(sigma), union(Vars(s), Vars(t)) )

rob_uni_alg_dom_ran_disjoint: LEMMA
   unifiable(s,t) IMPLIES
         LET sigma = robinson_unification_algorithm(s, t) IN
          subset?( VRan(sigma) ,
                    difference( union(Vars(s), Vars(t)), Dom(sigma) ))
```

In addition, it is necessary to formalize an auxiliary lemma that states that the algorithm gives the output `fail` exactly when the input terms are not unifiable.

```
robinson_unification_algorithm_fails_iff_non_unifiable : LEMMA
 NOT unifiable(s,t) IFF robinson_unification_algorithm(s,t) = fail
```

The completeness theorem states that, for given `s` and `t`, the operator `robinson_unification_algorithm` either returns `fail` or the mgu of these terms correctly. Its formalization follows easily from the previous lemmas on soundness and failure.

```
completeness_robinson_unification_algorithm : THEOREM
  IF unifiable(s,t) THEN  mgu(robinson_unification_algorithm(s,t))(s,t)
                    ELSE  robinson_unification_algorithm(s,t) = fail
  ENDIF
```

Notice that in the specific approach to deal with failing cases given in the *theory* `robinsonunification`, the property of idempotence is a simple corollary proved as consequence of the selection of `fail`.

5 Related work

To the best of our knowledge, the first formalization of the unification algorithm was given by Paulson [Pau85]. Paulson's formalization of Manna and Waldinger's theory of unification was done in the theorem prover LCF and subsequently this approach was followed by Konrad Slind in the theory `Unify` in the proof assistant Isabelle/HOL from which an improved version called `unification` is available now. Similarly to our approach, idempotence of the computed unifiers is unnecessary to prove neither termination nor correctness of the specified unification algorithm.

In contrast with our termination proof, which is based on the fact that the number of different variables occurring in the terms being unified decreases after each step of the unification algorithm (Section 3), the termination proof of the theory `Unify` is based on separated proofs of *non-nested* and *nested* termination conditions and the unification algorithm is specified based on a specification of terms built by a binary combinator operator.

Additional facts that make our formalization closer to the usual theory of unification (algorithms) as presented in well-known textbooks (e.g., [Llo87, BN98]), is the decision to present terms as a data type built from variables and the operator `app` that builds terms as an application of a function symbol (of a given arity) to a sequence of terms with the right size. In this way, the substitution was specified as a function from variables to terms and its homomorphic extension is straightforward.

An algorithm similar to Robinson's one was extracted from a formalization done in the Coq proof assistant [Rou92]. That formalization uses a generalized notion of terms, that uses binary constructors in the style of Manna and Waldinger, whose translation to the usual notation is not straightforward.

In [RRAH06], Ruiz-Reina *et al* presented a formalization in ACL2 of the correctness of an implementation of an $O(n^2)$ run-time unification algorithm. The specification is based on Corbin and Bidot's development [CB83] as presented in [BN98] in which terms are represented as directed acyclic graphs (DAGs). The merit of this formalization is that by taking care of an specific data structure such as DAGs for representing terms, the correctness proof results much more elaborated than the current one. But in the current paper, the focus is to have a natural mechanical proof of the completeness of any unification algorithm in the Robinson style, reusing the general methodology for the verification of termination and soundness, which come from the proof of existence of mgu's for unifiable terms. Although the representation of terms is sophisticated (via DAGs), the refereed formalization diverges from textbooks proofs of correctness of the unification algorithm in which it is first-order restricted. In fact, instead of representing second-order objects such as substitutions as functions from the domain of variables to the range of terms, they are specified as first-order association lists. In our approach, taking the decision to specify substitutions as functions allows us to apply all the theory of functions available in the higher-order proof assistant PVS, which makes our formalization very close to the ones available in textbooks.

Programming and proving are closely related in what concerns the construction of correct software. In fact, declarative programming style is much closer to formal specification than imperative programming, and this permits one to think about the extraction of executable code from a PVS specification. In [JSS07], a unification algorithm à la Robinson is specified, and functional code is generated via a translator that is in its prototype stage. This specification of the unification algorithm is proved sound and complete but it just claims that whenever the given terms are unifiable, the output substitution is the most general one. This property can be proved using the technology provided by our specification.

6 Conclusions and Future Work

The formalization of the theorem of existence of mgu's for unifiable terms, previously developed in PVS, provides general proving techniques for the treatment of the properties of termination and soundness of unification algorithms. For the treatment of non necessarily unifiable terms, this methodology can be reused taking into account how the exceptions or failing cases are specifically treated by any algorithm. The application of the general methodology of verification of completeness was illustrated by showing how verification is given for a specification of the unification algorithm in which the failing cases were (correctly) detected and distinguished by giving as output a non-idempotent substitution.

Recently, in [CM09], a certified resolution algorithm for the propositional calculus is extracted from a Coq specification. This specification uses the built in pattern matching of the Coq proof assistant that is enough to deal with resolution in the propositional calculus. An extension to first-order logic will requires first-order unification and hence an explicit treatment of unification as presented here. As future work, it is of great interest the extraction of certified unification algorithms alone, or in several contexts of its possible applications such as the ones of first-order resolution and of type inference. Notice that for doing this it is essential to give constructive specifications such as the current one. Several contributions on the extraction of executable code from PVS specifications were given in [LMG09], among others.

References

[AdMARG10] A.B. Avelar, F.L.C. de Moura, M. Ayala-Rincón, and A. Galdino. A Formalization of The Existence of Most General Unifiers. Departamentos de Matemática e Ciência da Computação, Universidade de Brasília, Available: `http://ayala.mat.unb.br/publications.html`, 2010.

[BKdV03] M. Bezem, J.W. Klop, and R. de Vrijer, editors. *Term Rewriting Systems by TeReSe*. Number 55 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and* All That. Cambridge University Press, 1998.

[Bur98] S. N. Burris. *Logic for Mathematics and Computer Science*. Prentice Hall, 1998.

[CB83] J. Corbin and M. Bidoit. A Rehabilitation of Robinson's Unification Algorithm. In *IFIP Congress*, pages 909–914, 1983.

[CM09] R. Constable and W. Moczydlowski. Extracting the resolution algorithm from a completeness proof for the propositional calculus. *Annals of Pure and Applied Logic*, 161(3):337–348, 2009.

[EFT84] H. D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Springer, 1984.

[GAR09] A. L. Galdino and M. Ayala-Rincón. A PVS *Theory* for Term Rewriting Systems. In *Proc. of the 3^{rd} Workshop on Logical and Semantic Frameworks with Applications - LSFA 2008*, volume 247 of *Electronic Notes in Theoretical Computer Science*, pages 67–83. Elsevier, 2009.

[GAR10] A. L. Galdino and M. Ayala-Rincón. A Formalization of the Knuth-Bendix(-Huet) Critical Pair Theorem. *J. of Automated Reasoning*, 2010. DOI 10.1007/s10817-010-9165-2, Springer Online First.

[JSS07] Bart Jacobs, Sjaak Smetsers, and Ronny Wichers Schreur. Code-carrying theories. *Formal Asp. Comput.*, 19(2):191–203, 2007.

[KB70] D. E. Knuth and P. B. Bendix. *Computational Problems in Abstract Algebra*, chapter Simple Words Problems in Universal Algebras, pages 263–297. J. Leech, ed. Pergamon Press, Oxford, U. K., 1970.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation – Artificial Intelligence. Springer, second edition, 1987.

[LMG09] L. Lensink, C. Muñoz, and A. Goodloe. From verified models to verifiable code. Technical Memorandum NASA/TM-2009-215943, NASA, Langley Research Center, Hampton VA 23681-2199, USA, June 2009.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11thInt. Conf. on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer Verlag.

[Pau85] Lawrence C. Paulson. Verifying the Unification Algorithm in LCF. *Science of Computer Programming*, 5(2):143–169, 1985.

[Rob65] J. A. Robinson. A Machine-oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Rou92] J. Rouyer. Développement de l'algorithme d'unification dans le calcul des constructions. Technical Report 1795, INRIA, 1992.

[RRAH06] J.-L. Ruiz-Reina, F.-J. Martín-Mateos J.-A. Alonso, and M.-J. Hidalgo. Formal Correctness of a Quadratic Unification Algorithm. *J. of Automated Reasoning*, 37(1-2):67–92, 2006.

A Basic notions on first-order unification

Consider a signature Σ in which function symbols and their associated arities are given (that is, the arity n ($n \in \mathbb{N}$) for each function symbol f in Σ is known) and a enumerable set V of variables is given. The set of well-defined terms, denoted by $T(\Sigma, V)$, over the signature Σ and the set V is recursively defined as:

- $x \in V$ is a well-defined term;
- for each n-ary function symbol $f \in \Sigma$ and well-defined terms $t_1, \ldots, t_n$, $f(t_1, \ldots, t_n)$ is a well-defined term.

Note that constants are 0-ary function symbols, and hence are well-defined terms.

In the sequel, for brevity "terms" instead of "well-defined terms" will be used.

A substitution in $T(\Sigma, V)$, by convention denoted by lowercase Greek letters, is a function from a finite set of variables to $T(\Sigma, V)$.

Definition 1 (Substitution). *A substitution σ is defined as a function from V to $T(\Sigma, V)$, such that the domain of σ, defined as the set of variables $\{x \mid x \in V, \sigma(x) \neq x\}$ and denoted by $Dom(\sigma)$, is finite.*

The homomorphic extension of a substitution from the set V to $T(\Sigma, V)$ is given as usual and denoted as $\hat{\sigma}$.

Definition 2 (Homomorphic extension of a substitution). *The homomorphic extension of a substitution σ, denoted as $\hat{\sigma}$, is inductively defined over the set $T(\Sigma, V)$ as*
- $x\hat{\sigma} := x\sigma$;
- $f(t_1, \ldots, t_n)\hat{\sigma} := f(t_1\hat{\sigma}, \ldots, t_n\hat{\sigma})$.

Given the notion of homomorphic extension, it is possible to define substitution composition.

Definition 3 (Composition of substitutions). *Consider two substitutions σ and τ, its composition is defined as the substitution $\sigma \circ \tau$ such that $Dom(\sigma \circ \tau) = Dom(\sigma) \cup Dom(\tau)$ and for each variable x in this domain, $x(\sigma \circ \tau) := (x\tau)\hat{\sigma}$.*

Two terms s and t are said to be unifiable whenever there exists a substitution σ such that $s\sigma = t\sigma$.

Definition 4 (Unifiers). *The set of unifiers of two terms s and t is defined as*

$$U(s, t) := \{\sigma \mid s\sigma = t\sigma\}$$

Definition 5 (Most generality of substitutions). *Given two substitutions σ and τ, σ is said to be most general than τ whenever, there exists a substitution γ such that $\gamma \circ \sigma = \tau$. This is denoted as $\sigma \leq \tau$.*

Definition 6 (Most General Unifier). *Given two terms s and t such that $U(s,t) \neq \emptyset$. A substitution σ such that for each $\tau \in U(s,t)$, $\sigma \leq \tau$, is said to be a most general unifier of s and t. For short it is said that σ is an mgu of s and t.*

Now, it is possible to state the theorem of existence of mgu's.

Theorem 1 (Existence of mgu's). *Let s and t be terms built over a signature $T(\Sigma, V)$. Then, $U(s,t) \neq \emptyset$ implies that there exists an mgu of s and t.*

The analytic proof of this theorem is constructive and the first introduced proof was by Robinson itself in [Rob65]. In this paper, a unification algorithm was introduced, which either gives as output a most general unifier for each unifiable pair of terms or fails when there are no unifiers. The proof of correctness of this algorithm, which consists in proving that the algorithm always terminates and that when terminates gives an mgu implies the existence theorem. Several variants of this first-order unification algorithm appear in well-known textbooks on computational and mathematical logic, semantics of programming languages, rewriting theory, etc. (e.g., [Llo87, EFT84, Bur98, BKdV03, BN98]). Since the formalization follows the classical proof schema, no analytic presentation of this proof will be given here.

B Specification of basic notions

The *sub theory* `robinsonunification`, inside the *theory* `trs`, imports *sub theories* for substitution, terms and positions, among others. The most relevant notions related with unification are inside the *sub-theories* `positions`, `subterm` and `substitution`. The PVS notions used for specifying these basic concepts are taken from the prelude *theories* for `finite_sequences` and `finite_sets`. Namely, finite sequences are used to specify well-formed terms which are built from variables and function symbols with their associated arities. This is done by application of the PVS `DATATYPE` mechanism which is used to define recursive types.

```
term[variable: TYPE+, symbol: TYPE+, arity: [symbol -> nat]] : DATATYPE
BEGIN
 vars(v:variable): vars?
 app(f:symbol, args:{args:finite_sequence[term] | args`length=arity(f)}): app?
END term
```

Notice that the fact that a term is well-formed, that is, that function symbols are applied to the right number of arguments is guaranteed by typing the arguments of each function symbol `f` as a finite sequence of length `arity(f)`.

Finite sets and sequences are also used to specify sets of subterms and sets of term positions, as is shown below.

The (finite) set of positions of a term `t` is recursively defined on the structure of the term as follows, where `only_empty_seq` is a set containing only an empty finite sequence, that is the set containing the root position only.

```
positionsOF(t: term): RECURSIVE positions =
  (CASES t OF
     vars(t): only_empty_seq,
     app(f, st): IF length(st) = 0
                 THEN only_empty_seq ELSE
                  union(only_empty_seq,
                     IUnion((LAMBDA (i: upto?(length(st))):
                             catenate(i, positionsOF(st(i-1)) ))))
                 ENDIF
     ENDCASES)
MEASURE t BY <<
```

In the *subtheory* `subterm`, the subterm of `t` at position `p` also is specified in a recursive way (now on the length of `p`), as follows:

```
subtermOF(t: term, (p: positions?(t))): RECURSIVE term =
   (IF length(p) = 0
    THEN  t  ELSE
      LET st = args(t),
           i = first(p),
           q = rest(p)  IN
            subtermOF(st(i-1), q)
    ENDIF)
MEASURE length(p)
```

where `first` and `rest` are constructors that return, respectively, the first element and the rest of a finite sequence, and `positions?(t)` is the (dependent) type of all positions in `t`, which is specified as follows:

```
positions?(t: term): TYPE = {p: position | positionsOF(t)(p)}
```

Several necessary results on terms, subterms and positions are formalized by induction on the structure of terms following the lines of these definitions. For instance, properties such as the one that states that the set of positions of a term is finite (lemma `positions_of_terms_finite` in the *subtheory* `positions`) and the one that states that the set of variables occurring in a term is finite (lemma `vars_of_term_finite` in the *subtheory* `subterm`) are proved by structural induction on terms. Also, several useful rules for computing with positions and subterms are specified. For example,

```
pos_subterm: LEMMA
  FORALL (p, q: position, t: term):
     positionsOF(t)(p o q)
       => subtermOF(t, p o q) = subtermOF(subtermOF(t, p), q)
```

is formalized in the *subtheory* `subterm`, where `p o q` means the concatenation of the sequences `p` and `q` denoted by pq in standard rewriting notation, and its proof is given by induction on the length of `p` according to the formal definitions given above.

The *subtheory* `substitution` specifies the algebra of substitutions. In this *subtheory* the type of substitutions is built as functions from variables to terms `sig : [V -> term]`, whose domain is finite: `Sub?(sig): bool = is_finite(Dom(sig))` and `Sub: TYPE = (Sub?)`. Also, the notions of domain, range, and the variable range are specified, closer to the usual theory of substitution as presented in well-known textbooks (e.g., [BN98]). These notions are specified as follows:

```
Dom(sig): set[(V)] = {x: (V) | sig(x) /= x}
Ran(sig): set[term] = {y: term | EXISTS (x: (V)): member(x, Dom(sig))  &  y = sig(x)}
VRan(sig): set[(V)] = IUnion(LAMBDA (x | Dom(sig)(x)): Vars(sig(x)))
```

where the operator `IUnion` can be found in the PVS prelude *theory*, `(V)` denote the type of all terms that are variables and `Vars(t)` denotes the set of all variables occurring in a term `t`.

Also, in the *subtheory* `substitution` the homomorphic extension `ext(sig)` of a substitution `sig` is specified inductively over the structure of terms, and the composition of two substitutions, denoted by `comp`, is specified as

```
comp(sigma, tau)(x: (V)): term = ext(sigma)(tau(x))
```

In standard rewriting notation, the homomorphic extension of a substitution σ from its domain of variables to the domain of terms is denoted by $\hat{\sigma}$, but to simplify notation, usually textbooks do not distinguish between a substitution σ and its extension $\hat{\sigma}$. In the formalization this distinction should be maintained carefully.

Several important results, that are useful for the development of *subtheory* `unification` were formalized in the *subtheory* `substitution`, as for instance, the property that states that the application of an homomorphic extension of a substitution preserves of the original set of positions of the term. This property is specified as,

```
ext_preserv_pos: LEMMA
   FORALL (p: position, s: term, sigma: Sub):
           positionsOF(s)(p) => positionsOF(ext(sigma)(s))(p)
```