32nd International Workshop on Unification (UNIF 2018), Informal Proceedings
Oxford, 7th July, 2018

Formalization of First-Order Syntactic Unification

Kasper Fabæch Brandt Anders Schlichtkrull Jørgen Villadsen
DTU Compute, AlgoLoG, Technical University of Denmark, 2800 Kongens Lyngby, Denmark
{andschl,jovi}@dtu.dk

Abstract

We present a new formalization in the Isabelle proof assistant of first-order syntactic unification, including a proof of termination. Our formalization follows, almost down to the letter, the ML-code from Baader and Nipkow's book "Term Rewriting and All That" (1998). Correctness is implied by the formalization's similarity to Baader and Nipkow's ML-code, but we have yet to formalize the correctness of the unification algorithm.

1 Introduction

We present a new and concise formalization in the Isabelle proof assistant [1] of a quite simple first-order syntactic unification algorithm. In particular we provide a formal proof of termination. The Isabelle2017 formalization is available here:

https://github.com/logic-tools/unification

We follow the succinct presentation of the unification algorithm in Baader and Nipkow's book "Term Rewriting and All That" [2], Section 4.7 Unification and term rewriting in ML, with Isabelle/HOL types as follows.

```
type_synonym vname = "string × int"

datatype "term" = V vname | T "string × term list"

type_synonym subst = "(vname × term) list"
```

A variable name consists of a name and an index (as mentioned in the book the index is not used in the current situation but it can simplify renaming). A term is either a variable or a compound term with a list of subterms. A substitution is a list of pairs, each with a variable name and a term, hence a so-called association list.

2 The Unification Algorithm

We first define a few auxiliary functions as in Baader and Nipkow's book [2].

```
definition indom :: "vname ⇒ subst ⇒ bool" where
  "indom x s = list_ex (λ(y, _). x = y) s"

fun app :: "subst ⇒ vname ⇒ term" where
  "app ((y,t)#s) x = (if x = y then t else app s x)"
| "app [] _ = undefined"

fun lift :: "subst ⇒ term ⇒ term" where
  "lift s (V x) = (if indom x s then app s x else V x)"
| "lift s (T(f,ts)) = T(f, map (lift s) ts)"

fun occurs :: "vname ⇒ term ⇒ bool" where
  "occurs x (V y) = (x = y)"
| "occurs x (T(_,ts)) = list_ex (occurs x) ts"
```

We then define the unification algorithm using the option type rather than ML exceptions.

```
function (sequential) solve :: "(term × term) list × subst ⇒ subst option"
  and elim :: "vname × term × (term × term) list × subst ⇒ subst option"
where
  "solve([], s) = Some s"
| "solve((V x, t) # S, s) = (
    if V x = t then solve(S,s) else elim(x,t,S,s))"
| "solve((t, V x) # S, s) = elim(x,t,S,s)"
| "solve((T(f,ts),T(g,us)) # S, s) = (
    if f = g then solve((zip ts us) @ S, s) else None)"

| "elim(x,t,S,s) = (
    if occurs x t then None
    else let xt = lift [(x,t)]
      in solve(map (λ (t1,t2). (xt t1, xt t2)) S,
        (x,t) # (map (λ (y,u). (y, xt u)) s)))"
```

The unification algorithm is called as `solve([(t1,t2)],[])` for terms `t1` and `t2` to unify and has exponential time and space complexity but performs well on practical examples, as pointed out in Section 4.7 of Baader and Nipkow's book [2].

3 The Termination Proof

The termination proof is derived from the one given on the "Unification (computer science)" Wikipedia page [3]. The idea is to define a measure on the arguments of the function that gets smaller and smaller for each recursive call with respect to the lexicographic ordering.

The measure is $<n_{var}, n_{fun}, |S|>$ with

- n_{var}: number of unsolved variables in S,
- n_{fun}: number of constants and functions in S, and
- $|S|$: number of equations in S.

This differs from the one given on the Wikipedia page because that algorithm defines the set of solved equations as the subset of equations that have a variable on their left-hand side while Baader and Nipkow have an explicit set S representing these. For the same reason, Wikipedia's algorithm reorients equations with a lone variable on the right hand side, while Baader and Nipkow remove them from the set of unsolved equations and add them to the set of variable assignments that makes up the return value.

The majority of the proof consists of showing basic theory about the three quantities, more specifically how they are reduced for each of the operations performed.

Let us look at the proof and its formalization in more detail. We want to define the relation on arguments defined by lexicographical ordering on $<n_{var}, n_{fun}, |S|>$. To do this we define functions which given the arguments to the `solve` function calculates n_{var}, n_{fun} and $|S|$. Let us consider n_{var} first. The function is as follows:

```
(λXX. case XX of Inl(l,_) ⇒ n_var l | Inr(x,t,S,_) ⇒ n_var ((V x, t)#S))
```

Here, `XX` represents the tuple consisting of the arguments given to the function call. We split on the pair-case `Inl(l,_)` representing a call to the `solve` function and the quadruple-case `Inr(x,t,S,_)` representing a call to the `elim` function. In both cases we use the function `n_var` to count the number of variables occurring in the set of unsolved equations. In the case for a call to `elim` we consider `x = t` an unsolved equation and therefore call `n_var` on `((V x, t)#S)`. The other components are defined in a similar way. We now compose the three measures to the desired relation. It is done using Isabelle's `<*mlex*>` operator. The operator is defined such that m `<*mlex*>` R takes a measure m and a relation R, and then turns the measure m into a relation and composes it lexicographically with the relation R. The measure is turned into a relation by comparing the size of the measure on given elements and letting this determine which element the relation considers larger.

The relation we get is as follows:

```
(λXX. case XX of Inl(l,_) ⇒ n_var l | Inr(x,t,S,_) ⇒ n_var ((V x, t)#S)) <*mlex*>
(λXX. case XX of Inl(l,_) ⇒ n_fun l | Inr(x,t,S,_) ⇒ n_fun ((V x, t)#S)) <*mlex*>
(λXX. case XX of Inl(l,_) ⇒ size l | Inr(x,t,S,_) ⇒ size ((V x, t)#S)) <*mlex*>
(λXX. case XX of Inl(l,_) ⇒ 1 | Inr(x,t,S,_) ⇒ 0) <*mlex*> {}
```

The last two components are `(λXX. case XX of Inl(l,_) ⇒ 1 | Inr(x,t,S,_) ⇒ 0)` and `{}`. The former is the measure which gives calls to `solve` size `1` and calls to `elim` size `0`, and the latter is the empty relation.

Our formal termination proof does an application of Isabelle's “relation” proof method which leaves us to prove subgoals expressing that the considered relation is wellfounded and that calls are related as desired by the relation.

Let R represent the above relation. For the readers with the courage to read Isabelle syntax we state the 6 subgoals.

1.
```
wf R
```
2.
```
V x = t ⟹ (Inl (S, s), Inl ((V x, t) # S, s)) ∈ R
```
3.
```
V x ≠ t ⟹ (Inr (x, t, S, s), Inl ((V x, t) # S, s)) ∈ R
```
4.
```
(Inr (x, T v, S, s), Inl ((T v, V x) # S, s)) ∈ R
```
5.
```
f = g ⟹
          (Inl (zip ts us @ S, s),
           Inl ((T (f, ts), T (g, us)) # S, s)) ∈ R
```
6.
```
 ¬ occurs x t ⟹ xa = lift [(x, t)] ⟹
          (Inl (map (λ(t1, t2). (xa t1, xa t2)) S, (x, t) #
                map (λ(y, u). (y, xa u)) s),
           Inr (x, t, S, s)) ∈ R
```

Comparing subgoals 2-6 with the definition of `solve` and `elim` it is not hard to see where the goals come from. For instance in subgoal 2 we have that `V x = t` and the two `Inl` expressions indicate that we are looking at calls to `solve`. In one call with arguments `((V x, t) # S, s))` and in the other with `(S, s)`. All this corresponds exactly to what is happening in the if-case of the second defining equation of `solve`.

We use Isabelle's automation to prove subgoal 1 (wellfoundedness) and to simplify subgoals 2-6. We then prove the subgoals using appropriate lemmas that we have also formalized.

4 Code Generation

We illustrate Isabelle's code generation with a few minimal examples. It is possible to generate and load `SML` (Standard ML) code directly in the Isabelle formalization (called code reflection).

```
code_reflect Unification
  datatypes
    "term" = V | T
    and
    char = zero_char_inst.zero_char | Char
  functions
    solve
```

For the mininal examples we only need the `solve` function in addition to the `term` and `char` datatypes.

This gives the following output displayed in Isabelle (`inta` is a generated datatype for integers used instead of the usual `int` datatype in `SML` and `num` is a generated datatype for natural numbers).

```
structure Unification:
  sig
    datatype char = Char of num | Zero_char
    type inta
    type num
    val solve:
       (term * term) list * ((char list * inta) * term) list ->
         ((char list * inta) * term) list option
    datatype term = T of char list * term list | V of char list * inta
  end
```

Alternatively one can export the generated `SML` code and for example save it in a separate file.

```
export_code solve in SML
```

Below we show a fragment of the generated `SML` code (besides `SML` it is possible to export to the functional programming languages `Haskell`, `Ocaml` and `Scala`).

```
fun elim (x, (t, (sa, s))) =
  (if occurs x t then NONE
    else let
           val xt = lift [(x, t)];
         in
           solve (List.map (fn (t1, t2) => (xt t1, xt t2)) sa,
                   (x, t) :: List.map (fn (y, u) => (y, xt u)) s)
         end)
and solve ([], s) = SOME s
  | solve ((V x, t) :: sa, s) =
    (if equal_terma (V x) t then solve (sa, s) else elim (x, (t, (sa, s))))
  | solve ((T v, V x) :: sa, s) = elim (x, (T v, (sa, s)))
  | solve ((T (f, ts), T (g, us)) :: sa, s) =
    (if List.equal_lista String.equal_char f g
      then solve (List.zip ts us @ sa, s) else NONE);
```

Back to the code reflection, we open the `Unification` structure in Isabelle (note the use of so-called cartouches ‹ … › around the `SML` code in the Isabelle formalization).

```
ML ‹ open Unification ›
```

First we consider unification of identical constants. We formally prove the expected result.

```
lemma "solve([(T([],[]),T([],[]))],[]) = Some []"
  by code_simp
```

We consider the same example in Isabelle using the generated code.

```
ML_val ‹ solve([(T([],[]),T([],[]))],[]) ›
```

Output displayed in Isabelle: `val it = SOME []: ((char list * inta) * term) list option`

We then consider unification of different constants. We again formally prove the expected result.

```
lemma "solve([(T([zero_char_inst.zero_char],[]),T([],[]))],[]) = None"
  by code_simp
```

We also consider the same example in Isabelle using the generated code.

```
ML_val ‹ solve([(T([Zero_char],[]),T([],[]))],[]) ›
```

Output displayed in Isabelle: `val it = NONE: ((char list * inta) * term) list option`

For both examples the output is as expected. More advanced examples are of course possible.

5 Related Work

There are several other formalizations of unification algorithms. An early result is Paulson's formalization [4] in LCF of an algorithm by Manna and Waldinger [5]. This formalization was used as the basis of a formalization by Coen, Slind and Krauss [6] of the same in Isabelle. In this formalization they represent terms as binary trees.

Urban, Pitts and Gabbay [7] also formalize unification in Isabelle, McBride [8] formalizes unification in LEGO, Kumar and Norrish [9] formalize unification in HOL4 and recently Avelar, Galdino, de Moura and Ayala-Rincón [10] formalize unification in PVS.

Most similar to our formalization is probably the formalization in Isabelle by Sternagel and Thiemann [11] since they formalize the same algorithm from Baader and Nipkow's book. A difference is that they have merged the functions solve and elim into one and thus our formalization is strictly more faithful to the original code. Another difference is that they do a different termination proof.

The above formalizations are of worst-case exponential running time algorithms, however, Ruiz-Reina, Martín-Mateos, Alonso and Hidalgo [12] formalize in ACL2 a worst-case quadratic running time algorithm which also stems from Baader and Nipkow's book. Brandt [13] has done preliminary work on formalizing the same algorithm in Isabelle's Imperative HOL. Except for this preliminary work all mentioned formalizations prove termination and correctness.

6 Conclusion

We have formalized first-order syntactic unification. In contrast to other formalizations from the literature we follow the ML-code from Baader and Nipkow's book undeviatingly. We formalize a proof of termination and run the unification algorithm on a number of examples using Isabelle's code-generation.

It is our hope that the stand-alone formalization can be useful for teaching unification algorithms. We did not find any mistakes in the presentation of the unification algorithm in Baader and Nipkow's book. The Isabelle proof assistant allows for quite readable formal proofs of the theorems and the 400 lines are checked in a few seconds.

Future work includes proving correctness but our proof of termination for the rather direct formalization of a succinct unification algorithm should be of interest in itself.

References

[1] Tobias Nipkow, Lawrence C. Paulson and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic.* Springer 2002.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press 1998.

[3] Wikipedia page. *Unification (computer science).* Retrieved from Wikipedia on 25 May 2018: `https://en.wikipedia.org/wiki/Unification_(computer_science)`

[4] Lawrence C. Paulson. *Verifying the unification algorithm in LCF.* Science of Computer Programming 5(2) 143–169 1985.

[5] Zohar Manna and Richard Waldinger. *Deductive synthesis of the unification algorithm.* Science of Computer Programming 1(1–2) 5–48 1981.

[6] Martin Coen, Konrad Slind and Alexander Krauss. *Theory Unification.* Isabelle2017. `https://isabelle.in.tum.de/library/HOL/HOL-ex/Unification.html`

[7] Christian Urban, Andrew Pitts and Murdoch Gabbay. *Nominal Unification.* Theoretical Computer Science 323(1–3) 473–497 2004.

[8] Conor McBride. *First-order unification by structural recursion*. Journal of Functional Programming 13(6) 1061–1075 2003.

[9] Ramana Kumar and Michael Norrish. *(Nominal) Unification by Recursive Descent with Triangular Substitutions.* In Matt Kaufmann and Lawrence C. Paulson, editors: Interactive Theorem Proving (ITP 2010), Lecture Notes in Computer Science 6172, Springer 2010.

[10] Andréia Borges Avelar, André Luiz Galdino, Flávio Leonardo Cavalcanti de Moura and Mauricio Ayala-Rincón. *First-order unification in the PVS proof assistant.* Logic Journal of the IGPL, 22(5) 758–789 2014.

[11] Christian Sternagel and René Thiemann. *First-Order Terms.* Archive of Formal Proofs 2018. `https://www.isa-afp.org/entries/First_Order_Terms.html`

[12] José-Luis Ruiz-Reina, Francisco-Jesús Martín-Mateos, José-Antonio Alonso and María-José Hidalgo. *Formal Correctness of a Quadratic Unification Algorithm*. Journal of Automated Reasoning 37(1–2) 67–92 2006.

[13] Kasper Fabæch Brandt. *Formalization of a near-linear time algorithm for solving the unification problem.* Master's thesis, DTU Compute, AlgoLoG, Technical University of Denmark, 2018. `http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=7091`