(Nominal) Unification by Recursive Descent with Triangular Substitutions

Ramana Kumar[1] and Michael Norrish[2,1]

[1] The Australian National University u4305025@anu.edu.au
[2] Canberra Research Lab., NICTA Michael.Norrish@nicta.com.au

Abstract. Using HOL4, we mechanise termination and correctness for two unification algorithms, written in a recursive descent style. One computes unifiers for first order terms, the other for nominal terms (terms including α-equivalent binding structure). Both algorithms work with triangular substitutions in accumulator-passing style: taking a substitution as input, and returning an extension of that substitution on success.

1 Introduction

The fastest known first-order unification algorithms are time and space linear (or almost linear) in the size of the input terms [1, 2]. In the case of nominal unification, polynomial [3], including quadratic [4], algorithms exist. By comparison, the algorithms in this paper are naïve in two ways: they perform recursive descent of the terms being unified, applying new bindings along the way; and they perform the occurs check with every new binding. Recursive descent interleaved with application can require time exponential in the size of the original terms. Also, it is possible to do the occurs check only once, or even implicitly, in an algorithm that doesn't recursively descend terms.

However, naïve algorithms are used in real systems for a number of reasons: worst case inputs do not arise often in practice, encoding the input and decoding the output of a fast algorithm can be costly, and naïve algorithms are simpler to implement and teach. Some evidence for the first two assertions can be found in Hoder and Voronkov [5] where an imperative version of the algorithm in this paper (there labelled "Robinson's") benchmarks better than the worst-case linear algorithms.[3]

One important feature of the algorithms considered by Hoder and Voronkov is that they all use *triangular* substitutions. This representation is useful in systems that do backtracking and need to "unapply" substitutions from terms, because it enables the use of persistent data structures. The unapply operation becomes implicit (and thus, efficient) when updates are made to a persistent substitution: backtracking computations simply apply the appropriate subset of the shared substitution whenever terms in context are required.

[3] These benchmarks were made in the context of automated theorem provers with term indexing; we don't consider the maintenance of a term index in this paper.

A triangular substitution [6] is a set of singleton maps (each binding a different variable). When this set is implemented as a list, update is constant time and sharing is maximised. When using triangular substitutions, and writing in a functional language, it is natural to write unification in an accumulator-passing style. (The analogue in an imperative setting is to update a global variable, which is what happens in the implementation of Robinson's algorithm in Hoder and Voronkov.) So, for example, the unification algorithm in miniKanren [7, 8] takes two terms, t_1 and t_2, and an accumulator substitution, s. It returns an extension of s with any new bindings necessary to make t_1 and t_2 unify (or fails if that's impossible).

Triangular substitutions are generally not *idempotent*. For example, a binding from y to z may be added to a substitution already binding x to y. Applying the extended substitution once to x yields y, but applying it twice yields z. But a triangular substitution can represent the same information as an idempotent substitution using exponentially less space. For example if x is bound to the pair (y, y) and y is bound to the ground term $(1, 2)$ then an idempotent substitution would contain three copies of $(1, 2)$, whereas a triangular substitution would contain just one.

Baader and Snyder [6] mention using triangular substitutions in a recursive descent algorithm as a good idea, but do not pursue it because of the exponential time complexity. Our own experiments agree that using triangular substitutions gives better speed and memory usage than computing idempotent substitutions.

Nominal Unification Classical unification works over first-order terms. Recently, there has been interest in the theory and implementation of logical systems using *nominal* terms, which include names and binders. Such terms provide natural representations of syntaxes occurring in logic and computer science.

Nominal unification was first defined by Urban, Pitts, and Gabbay [9]. Nominal systems (*e.g.*, αProlog [10], alphaKanren [11]) need to be able to unify nominal terms. The mechanisations in this paper are of algorithms inspired by the implementations in miniKanren (first-order) and alphaKanren (nominal). (The alphaKanren paper [11] describes unification with idempotent substitutions; our mechanisation is of a later, more efficient implementation using triangular substitutions.)

Contributions

- We provide mechanised definitions of accumulator-passing style first-order (`unify`) and nominal (`nomunify`) unification algorithms. Both definitions require the provision of a novel termination argument (Section 4).
- Since the unification algorithms may diverge if the accumulator contains loops, we define and characterise a well-formedness condition on triangular substitutions that forbids loops (Section 2), and show that the algorithms preserve it (Sections 5 and 6).
- We mechanise algorithms for applying triangular substitutions, providing the requisite termination arguments in Section 3.

– We provide statements of correctness (soundness, completeness, and generality) for unification algorithms written in accumulator-passing style, and prove them of `unify` in Section 5, and of `nomunify` in Section 6.

The mechanised theories containing all the results in this paper are available online at `https://bitbucket.org/michaeln/formal_mk/src/tip/hol/`. Since the results have been machine-checked, we will omit the proofs of some lemmas.

Notation In general, higher order logic syntax for Boolean terms uses standard connectives and quantifiers ($\wedge$, $\forall$ *etc.*). Iterated application of a function is written $f^n\ x$, meaning $f(f(\ldots f(x)))$. R^+ denotes the transitive closure of a relation. The relation `measure` f, where f is of type $\alpha \rightarrow$ `num`, relates x and y if $f(x) < f(y)$.

The `do` notation is used for writing in monadic style. We only use it to express bind in the option monad: the term `do` y `<-` $f\ x$`;` $g\ y$ `od` means `NONE` if $f\ x$ returns `NONE`. Otherwise, if $f\ x$ returns `SOME` y, then the term is the result of applying g to y (giving a value of option type, either `NONE` or `SOME` v).

`FLOOKUP` *fm* k applies a finite map, returning `SOME` v when the key is in the domain, otherwise `NONE`. The domain of a finite map is written `FDOM` *fm*. The sub-map relation is written $fm_1 \sqsubseteq fm_2$. The empty finite map is written `FEMPTY`. The update of a finite map with a new key-value pair is written *fm* `|+ (`k`,`v`)`. Composition of a function after a finite map is written $f \circ fm$.

Tuples and inductive data types can be deconstructed by case analysis. The notation is `case` t_1 `of` $p_1 \rightarrow e_1 \parallel p_2 \rightarrow e_2$. Patterns may include underscores as wildcards.

For each type, the constant `ARB` denotes an arbitrary object of that type.

2 Terms and Substitutions

The word "substitution" can refer to an action—substitution of t_1 for x and t_2 for y in t—or it can refer to an object, a collection of variable bindings—the substitution that binds x to t_1 and y to t_2. When viewing substitutions as data structures containing bindings, a separate function is used to apply a substitution to a term to produce a new term. Equivalent substitutions, under application, may be different as data structures. We distinguish substitutions from substitution application in order to investigate a representation, triangular form, suited to the functional programming idiom of implicitly shared data.

We define first-order terms inductively as follows. We represent variables by natural numbers (strings would be equally good). Terms are parameterised by the type α representing constant values (*e.g.*, function symbols).

Definition 1. *Terms*

term `= Var of num | Pair of` α `term =>` α `term | Const of` α

We represent a substitution (HOL type α `subst`) as a finite map from numbers to terms, thereby abstracting over any particular data structure (an association list or something more sophisticated) without losing the distinction between a substitution and its application. Application to a term is defined as follows. We will define a different notion of substitution application more suited to triangular substitutions in Section 3.

Definition 2. *Substitution application*

$$s\ \texttt{'}\ (\texttt{Var}\ v) = \texttt{case FLOOKUP}\ s\ v\ \texttt{of NONE} \to \texttt{Var}\ v \parallel \texttt{SOME}\ t \to t$$
$$s\ \texttt{'}\ (\texttt{Pair}\ t_1\ t_2) = \texttt{Pair}\ (s\ \texttt{'}\ t_1)\ (s\ \texttt{'}\ t_2)$$
$$s\ \texttt{'}\ (\texttt{Const}\ c) = \texttt{Const}\ c$$

A substitution is `idempotent` if repeated application is the same as a single application. Applying a substitution to a variable outside its domain yields that variable. But our representation permits a substitution explicitly binding a variable to itself. We will exclude such substitutions with the condition `noids` s.

The application of a substitution s to itself is obtained by replacing every term t in the range of s by its image under s. This is the closest we will get to substitution composition (`selfapp` s is s composed with itself); instead we will compose application functions.

Lemma 1. $\vdash \texttt{selfapp}\ s\ \texttt{'}\ t = s\ \texttt{'}\ (s\ \texttt{'}\ t)$

Well-formed Substitutions

For each substitution s we define a relation $\texttt{tri}_\texttt{R}\ s$ that holds between a variable in the domain and a variable in the corresponding term.

Definition 3. *Relating a variable to those in the term to which it's bound*

$$\texttt{tri}_\texttt{R}\ s\ y\ x \iff$$
$$\texttt{case FLOOKUP}\ s\ x\ \texttt{of NONE} \to \texttt{F} \parallel \texttt{SOME}\ t \to y \in \texttt{vars}\ t$$

A substitution is well-formed (`wfs`) if $\texttt{tri}_\texttt{R}\ s$ is well-founded. There are three informative statements equivalent to the well-formedness of a substitution.

Lemma 2. *Only well-formed substitutions have no cycles*

$$\vdash \texttt{wfs}\ s \iff \forall v.\ \neg(\texttt{tri}_\texttt{R}\ s)^+\ v\ v$$

Corollary 1. $\vdash \texttt{wfs}\ s \Rightarrow \texttt{noids}\ s$

Lemma 3. *Only well-formed substitutions are well-formed after self-application*

$$\vdash \texttt{wfs}\ s \iff \texttt{wfs}\ (\texttt{selfapp}\ s)$$

Lemma 4. *Only well-formed substitutions have fixpoints*

$$\vdash \texttt{wfs}\ s \iff \exists n.\ \texttt{idempotent}\ (\texttt{selfapp}^n\ s) \land \texttt{noids}\ (\texttt{selfapp}^n\ s)$$

Proof. From right to left, the result follows by induction on n. From left to right, the `noids` condition follows from Lemma 3 and Corollary 1. Idempotence follows by contradiction. If a substitution is not idempotent there will be a variable that maps to a term including a variable in the substitution's domain. If this occurs within a substitution iterated n times, there must be a chain of length n within the original substitution with the same property. But an arbitrarily long chain cannot exist without a loop, contradicting our well-formedness assumption.

Lemma 4 (with Lemma 1) shows that well-formedness is necessary and sufficient for being able to recover an equivalent idempotent substitution.

3 Substitution Application

Since we are interested in maintaining triangular substitutions, we want to be able to apply a non-idempotent substitution as if we had collapsed it down to an idempotent one by repeated self-application without actually doing so. This is achieved by recursion in the application function `walk*` (we write $s \triangleleft t$ for the application of `walk*` to substitution s and term t): if we encounter a variable in the domain of the substitution, we look it up and recur on the result. Defining this function presents the first of a number of interesting termination problems.

The clearest expression of `walk*`'s behaviour is the following characterisation:

Lemma 5. *Characterisation of* `walk*`

```
⊢ wfs s ⇒
  s ◁ Var v =
    (case FLOOKUP s v of NONE → Var v ‖ SOME t → s ◁ t) ∧
  s ◁ Pair t1 t2 = Pair (s ◁ t1) (s ◁ t2) ∧
  s ◁ Const c = Const c
```

Lemma 6. `walk*` *reduces to application on idempotent substitutions*

```
⊢ wfs s ⇒ (idempotent s ⟺ walk* s = (') s)
```

The `walk*` function can be viewed as performing a tree traversal ("walk") of its eventual output term. Other algorithms, including `unify`, need to perform some of this tree walk, but may not need to immediately traverse a term to its leaves. We isolate the part of `walk*` that finds the ultimate binding of a variable, calling this `vwalk`:

Definition 4. *Walking a variable*

```
wfs s ⇒
vwalk s v =
  case FLOOKUP s v of
    SOME (Var u) → vwalk s u
  ‖ SOME t → t
  ‖ NONE → Var v
```

Proving termination for `vwalk` under the assumption `wfs` s follows easily from the definitions.

Following the miniKanren code, we define a function `walk`, which either calls `vwalk` if its argument is a variable, or returns its argument. It is a common miniKanren idiom (used in `unify`, among other places) to begin functions by walking term arguments in the current substitution. This reveals just enough of a term-in-context's structure for the current level of recursion. This idiom is used in the definition of walk*, which can be stated thus:

Definition 5. *Substitution application, walking version*

```
wfs s ⇒
s ◁ t =
  case walk s t of
    Pair t1 t2 → Pair (s ◁ t1) (s ◁ t2)
  ‖ t′ → t′
```

The termination relation for walk* is the lexicographic combination of the multi-set ordering with respect to $(\mathtt{tri}_{\mathtt{R}}\ s)^{+}$ over a term's variables, and the term's size.

The "walk first" idiom is also used to define the occurs-check. We omit the definition but provide the following characterisation.

Lemma 7. *The occurs-check finds variables in the term after application*

$\vdash \mathtt{wfs}\ s \Rightarrow (\mathtt{oc}\ s\ t\ v \iff v \in \mathtt{vars}\ (s \lhd t))$

4 Unification: Definition

Our unification algorithm, unify, has type

α subst → α term → α term → α subst option

The option type in the result is used to signal whether or not the input terms are unifiable. We accept that unify will have an undefined value when given a malformed substitution as input. Our strategy for defining unify is to define a total version, tunify; to extract and prove the termination conditions; and to then show that unify exists and equals tunify for well-formed substitutions. The definition of tunify is given in Figure 1.

Three termination conditions are generated by HOL4, corresponding to the need for a well-founded relation and the two recursive calls:

1. WF R
2. $\forall t_2\ t_1\ s\ t_{11}\ t_{12}\ t_{21}\ t_{22}.$
   wfs s ∧ walk s t_1 = Pair t_{11} t_{12} ∧ walk s t_2 = Pair t_{21} t_{22} ⇒
   R (s,t_{11},t_{21}) (s,t_1,t_2)
3. $\forall t_2\ t_1\ s\ t_{11}\ t_{12}\ t_{21}\ t_{22}\ sx.$
   wfs s ∧ (walk s t_1 = Pair t_{11} t_{12} ∧ walk s t_2 = Pair t_{21} t_{22}) ∧
   tunify_tupled_aux R (s,t_{11},t_{21}) = SOME sx ⇒
   R (sx,t_{12},t_{22}) (s,t_1,t_2)

A call tunify_tupled_aux R *args* is a guarded call to the only-partially defined tunify: any recursive calls must be on arguments that are R-smaller than *args*. The call appears in Condition 3 because the argument sx in the

Definition 6. *Unification with triangular substitutions (total version)*

```
tunify s t1 t2 =
  if wfs s then
    case (walk s t1,walk s t2) of
      (Var v1,Var v2) →
        SOME (if v1 = v2 then s else s |+ (v1,Var v2))
    ‖ (Var v1,t2) →
        if oc s t2 v1 then NONE else SOME (s |+ (v1,t2))
    ‖ (t1,Var v2) →
        if oc s t1 v2 then NONE else SOME (s |+ (v2,t1))
    ‖ (Pair t11 t12,Pair t21 t22) →
        do sx <- tunify s t11 t21; tunify sx t12 t22 od
    ‖ (Const c1,Const c2) → if c1 = c2 then SOME s else NONE
    ‖ _ → NONE
  else
    ARB
```

Fig. 1. First-Order Unification: the `unify` function is the `then` branch of the `if`.

second recursive call `tunify` sx t_{12} t_{22} is the result of the first recursive call. This is thus an instance of nested recursion.

The `unify` function walks the subterms being considered in the current substitution before case analysis. The key to the termination argument is that size of the subterms, considered in the context of the updated substitution, goes down on every recursive call. The termination relation $\texttt{unify}_{\texttt{R}}$, defined below, makes this statement in the final conjunct. The other conjuncts are also satisfied by the algorithm and are required to ensure that $\texttt{unify}_{\texttt{R}}$ is well-founded.

Definition 7. *Termination relation for* `unify`

$$\texttt{unify}_{\texttt{R}}\ (sx,c_1,c_2)\ (s,t_1,t_2) \iff \texttt{wfs}\ sx \land s \sqsubseteq sx \land \texttt{allvars}\ sx\ c_1\ c_2 \subseteq \texttt{allvars}\ s\ t_1\ t_2 \land \texttt{measure}\ (\texttt{term\_depth} \circ \texttt{walk}^*\ sx)\ c_1\ t_1$$

Theorem 1. $\texttt{unify}_{\texttt{R}}$ *is well-founded*

$\vdash \texttt{WF}\ \texttt{unify}_{\texttt{R}}$

Proof. By contradiction. If there is an infinite $\texttt{unify}_{\texttt{R}}$-chain, then the set of variables in the arguments (`allvars`) must reach a fixpoint because each successive set is a subset of its predecessor, and the sets are finite. As the set of variables is getting smaller, the substitutions are allowed to get larger (the $\sqsubseteq$ relation). However, once the set of variables reaches its fixpoint, the substitutions will be drawing on a fixed source for new variable bindings, so they must also reach a fixpoint. Once the substitution (sx) is fixed, the first argument of the `measure` conjunct becomes fixed. Hence the supposedly infinite chain would have to stop (when $sx \triangleleft c_1$ has zero depth): contradiction.

We thereby satisfy Termination Condition 1. Condition 2 is easy because the substitution doesn't change.

Lemma 8. *Termination Condition 2*

$\vdash$ `wfs` s $\wedge$ `walk` s t_1 $=$ `Pair` t_{11} t_{12} $\wedge$ `walk` s t_2 $=$ `Pair` t_{21} t_{22} $\Rightarrow$
`unify`$_{\mathtt{R}}$ (s, t_{11}, t_{21}) (s, t_1, t_2)

Proof. For the conjunct involving `allvars`: either t_1 $=$ `Pair` t_{11} t_{12} or the pair is in the range of the substitution, and similarly for t_2. The other `unify`$_{\mathtt{R}}$ conjuncts are simple.

Condition 3, however, requires some work. We define another relation, `subst`$_{\mathtt{R}}$, weaker than `unify`$_{\mathtt{R}}$, which asserts that the variables of the result substitution all come from the arguments. The `subst`$_{\mathtt{R}}$ relation serves as a bridge: weak enough that we can prove it is satisfied by `tunify` by induction and strong enough that it implies `unify`$_{\mathtt{R}}$. We use a relation that restricts the substitution only since at this point we can't say much about recursive calls without proving `unify`$_{\mathtt{R}}$ for each call.

Definition 8. *Relation between the output substitution and input arguments*

`subst`$_{\mathtt{R}}$ sx s t_1 t_2 $\iff$
`wfs` sx $\wedge$ $s \sqsubseteq sx$ $\wedge$ `substvars` $sx \subseteq$ `allvars` s t_1 t_2

Lemma 9. `subst`$_{\mathtt{R}}$ *implies* `unify`$_{\mathtt{R}}$ *on subterms*

$\vdash$ `wfs` s $\wedge$ `walk` s t_1 $=$ `Pair` t_{11} t_{12} $\wedge$ `walk` s t_2 $=$ `Pair` t_{21} t_{22} $\wedge$
(`subst`$_{\mathtt{R}}$ sx s t_{11} t_{21} $\vee$ `subst`$_{\mathtt{R}}$ sx s t_{12} t_{22}) $\Rightarrow$
`unify`$_{\mathtt{R}}$ (sx, t_{12}, t_{22}) (s, t_1, t_2)

Lemma 10. `unify` *implies* `subst`$_{\mathtt{R}}$

$\vdash$ `wfs` s $\wedge$ `tunify_tupled_aux unify`$_{\mathtt{R}}$ (s, t_1, t_2) $=$ `SOME` sx $\Rightarrow$
`subst`$_{\mathtt{R}}$ sx s t_1 t_2

Proof. By well-founded induction (knowing that `unify`$_{\mathtt{R}}$ is well-founded).

Lemma 11. *Termination Condition 3*

$\vdash$ `wfs` s $\wedge$ `walk` s t_1 $=$ `Pair` t_{11} t_{12} $\wedge$ `walk` s t_2 $=$ `Pair` t_{21} t_{22} $\wedge$
`tunify_tupled_aux unify`$_{\mathtt{R}}$ (s, t_{11}, t_{21}) $=$ `SOME` sx $\Rightarrow$
`unify`$_{\mathtt{R}}$ (sx, t_{12}, t_{22}) (s, t_1, t_2)

Proof. From the lemmas above.

5 Unification: Correctness

There are three parts to the correctness statement: if `unify` succeeds then its result is a unifier; if `unify` succeeds then its result is most general; and if there exists a unifier of $s \lhd t_1$ and $s \lhd t_2$, then `unify` $s\ t_1\ t_2$ succeeds. A substitution s is a *unifier* of terms t_1 and t_2 if $s \lhd t_1 = s \lhd t_2$.

It is not generally true that the result of `unify` is idempotent. But `unify` preserves well-formedness, which (as per Lemma 4) ensures the well-formed result can be collapsed into an idempotent substitution.

Theorem 2. *The result of* `unify` *is a unifier and a well-formed extension*

$$\vdash \texttt{wfs}\ s \land \texttt{unify}\ s\ t_1\ t_2 = \texttt{SOME}\ sx \Rightarrow$$
$$\texttt{wfs}\ sx \land s \sqsubseteq sx \land sx \lhd t_1 = sx \lhd t_2$$

Proof. The first two conjuncts, that s is a sub-map of sx and sx is well-formed, are corollaries of Lemma 10. Essentially, `unify` only updates the substitution, and then only with variables that aren't already in the domain.

The rest follows by recursion induction on `unify`, using Lemma 12 (below), which states that applying a sub-map of a substitution, and then the larger substitution, is the same as simply applying the larger substitution on its own.

Lemma 12. `walk*` *over a sub-map*

$$\vdash s \sqsubseteq sx \land \texttt{wfs}\ sx \Rightarrow sx \lhd t = sx \lhd (s \lhd t)$$

Corollary 2. `walk*` *with a fixed substitution is idempotent*

Given Lemma 12 and Theorem 2, we can equally regard `unify` $s\ t_1\ t_2$ as calculating a unifier for t_1 and t_2 or for the terms-in-context $s \lhd t_1$ and $s \lhd t_2$.

The context provided by the input substitution is relevant to our notion of a most general unifier, which differs from the usual context-free notion. A unifier of terms in context is *most general* if it can be composed with another substitution to equal any other unifier *in the same context.* In the empty context, however, the notions of most general unifier coincide.

Lemma 13. *The kinds of extensions made by unify are innocuous*

$$\vdash \texttt{wfs}\ s_1 \land \texttt{wfs}\ (s\ \texttt{|+}\ (vx,tx)) \land vx \notin \texttt{FDOM}\ s \land$$
$$s_1 \lhd \texttt{Var}\ vx = s_1 \lhd (s \lhd tx) \Rightarrow$$
$$\forall t.\ s_1 \lhd (s\ \texttt{|+}\ (vx,tx) \lhd t) = s_1 \lhd (s \lhd t)$$

Proof. By recursion induction on `walk*`.

Lemma 14. *The result of* `unify` *is most general (in context)*

$$\vdash \texttt{wfs}\ s \land \texttt{unify}\ s\ t_1\ t_2 = \texttt{SOME}\ sx \land \texttt{wfs}\ s_2 \land$$
$$s_2 \lhd (s \lhd t_1) = s_2 \lhd (s \lhd t_2) \Rightarrow$$
$$\forall t.\ s_2 \lhd (sx \lhd t) = s_2 \lhd (s \lhd t)$$

Proof. By recursion induction on `unify` using the lemma above.

Theorem 3. *The result of* `unify` *is most general (empty context)*

$$\vdash \texttt{unify FEMPTY } t_1 \; t_2 = \texttt{SOME } sx \Rightarrow$$
$$\forall s.\ \texttt{wfs } s \wedge s \triangleleft t_1 = s \triangleleft t_2 \Rightarrow \exists s'.\ \forall t.\ s' \triangleleft (sx \triangleleft t) = s \triangleleft t$$

Remark 1. By the lemma above we see that the witness is s itself.

We now turn to the third correctness result.

Lemma 15. *A variable and a term containing that variable remain different under application*

$$\vdash \texttt{oc } s\ t\ v \wedge (\forall w.\ t \neq \texttt{Var } w) \wedge \texttt{wfs } s \wedge \texttt{wfs } s_2 \Rightarrow$$
$$s_2 \triangleleft \texttt{Var } v \neq s_2 \triangleleft (s \triangleleft t)$$

Proof. By considering the term sizes.

Theorem 4. *If the terms are unifiable, then* `unify` *succeeds*

$$\vdash \texttt{wfs } s \wedge \texttt{wfs } s_2 \wedge s_2 \triangleleft (s \triangleleft t_1) = s_2 \triangleleft (s \triangleleft t_2) \Rightarrow$$
$$\exists sx.\ \texttt{unify } s\ t_1\ t_2 = \texttt{SOME } sx$$

Proof. By recursion induction on `unify`, using Lemma 15 for the non-trivial occurs checks, and using Lemma 14 for the recursive case.

6 Nominal Unification

Nominal terms extend first-order terms with two new constructors, one for names (also called atoms), and one for *ties*, which represent binders (terms with a bound name). We also replace the `Var` constructor with a constructor for *suspensions*, the nominal analogue of variables. A suspension is made up of a variable name and a permutation of names, and stands for the variable after application of the permutation. When (if) the variable is bound, the permutation can be applied further.

Definition 9. *Concrete nominal terms*

Cterm
```
  = CNom of string
  | CSus of (string, string) alist => num
  | CTie of string => α Cterm
  | CPair_n of α Cterm => α Cterm
  | CConst_n of α
```

We represent permutations as lists of pairs of names; such a list stands for a ordered composition of swaps, with the head of list applied last. There may be more than one list representing the same permutation. We abstract over these different lists by creating a quotient type. The nominal term data type is the quotient of the concrete type above by permutation equivalence `(==)`.

Constructors in the quotient type are the same as in the concrete type but with the `C` prefix removed.

Following the example of the first-order algorithm, we begin by defining the "walk" operation that finds a suspension's ultimate binding:

Definition 10. *Walking a suspension*

$\mathtt{wfs_n}\ s \Rightarrow$
$\mathtt{vwalk_n}\ s\ \pi\ v =$
  $\mathtt{case\ FLOOKUP}\ s\ v\ \mathtt{of}$
    $\mathtt{SOME\ (Sus}\ p\ u\mathtt{)} \rightarrow \mathtt{vwalk_n}\ s\ (\pi\ \mathtt{++}\ p)\ u$
  $\|\ \mathtt{SOME}\ t \rightarrow \pi \bullet t$
  $\|\ \mathtt{NONE} \rightarrow \mathtt{Sus}\ \pi\ v$

The π `++` p term appends π and p, producing their composition; $\pi \bullet t$ is the (homomorphic) application of a permutation to a term.

The termination argument for $\mathtt{vwalk_n}$ is the same as in the first-order case; the permutation doesn't play a part in the recursion. Substitution application is analogous to the first-order case: $\mathtt{walk_n}$ calls $\mathtt{vwalk_n}\ s\ p\ v$ for a suspension `Sus` $p\ v$, otherwise returns its argument; $\mathtt{walk_n^*}$ uses $\mathtt{walk_n}$, recurring on ties as well as pairs.

In the first phase of nominal unification (as defined in [9]), a substitution is constructed along with a set of *freshness constraints* (alternatively, a *freshness environment*). A freshness constraint is a pair of a name and a variable, expressing the constraint that the variable is never bound to a term in which the name is free.

The second phase of unification checks to see if the freshness constraints are consistent, possibly dropping irrelevant constraints along the way. If this check succeeds, the substitution and the new freshness environment, which together form a nominal unifier, are returned. The use of triangular substitutions and the accumulator-passing style means that our definition of nominal unification differs from [9] in that the substitution returned from the first phase must be referred to as the freshness constraints are checked in the second phase.

The final definition in HOL is presented in Definition 12 (Figure 2). In both phases, we use the auxiliary `term_fcs`. This function is given a name and a term, and constructs a minimal freshness environment sufficient to ensure that the name is fresh for the term. If this is impossible (*i.e.*, if the name is free in the term), `term_fcs` returns `NONE`.

Following our strategy in the first-order case, $\mathtt{unify_n}$ is defined *via* a total function $\mathtt{tunify_n}$. The pair and constant cases are unchanged, and names are treated as constants. With suspensions, there is an extra case to consider: if the variables are the same, we augment the freshness environment with a constraint $(a, s \lhd_{\mathtt{n}} \mathtt{Sus}\ [\,]\ v)$ for every name a in the disagreement set of the permutations (done by `unify_eq_vars`). In the other suspension cases, we apply the inverse (reverse) of the suspension's permutation to the term before performing the binding (done in `add_bdg`). (We invert the permutation so that applying the permutation to the term to which the variable is bound results in the term with which the suspension is supposed to unify.)

In the `Tie` case, a simple recursive descent is possible when the bound names are the same. Otherwise, we ensure that the first name is fresh for the body of the second term, and swap the two names in the second term before recursing.

Phase 2 is implemented by `verify_fcs`, which calls

`term_fcs` a ($s \lhd_{\mathrm{n}}$ `Sus [] ` v)

for each constraint (a, v) in the environment, accumulating the result.

Termination The termination argument for Phase 1 is analogous to the termination argument for `unify` in the first-order case. We use the same termination relation (this time measuring nominal term depth, and ignoring the freshness environment). The extra termination condition for recursion down a `Tie` is handled like the easier of the `Pair` conditions because the substitution doesn't change and the freshness environment is irrelevant to termination.

Termination for Phase 2 depends only on the freshness environment being finite. We assume the freshness environment is finite in all valid inputs to `nomunify`, and it's easy to show that `term_fcs` (and hence Phase 1) preserves finiteness by structural induction on the nominal term.

6.1 Correctness

In the first-order case, unified terms are syntactically equal. In the nominal case, unified terms must be α-equivalent with respect to a freshness environment, written $fe \vdash t_1 \approx t_2$. For example, $(\lambda a.X)$ and $(\lambda b.Y)$ unify with X bound to $(a\,b)\cdot Y$ (the substitution), but only if $a\#Y$ (a singleton freshness environment). In the absence of the latter, one might instantiate Y with a, and therefore X with b, producing non-equivalent terms. We write $fe \vdash a \# t$ to mean that a name is fresh for a term with respect to a freshness environment.

Lemma 16. *The freshness environment computed by* `unify_eq_vars` *makes the suspensions equivalent*

$\vdash$ `wfs`$_{\mathrm{n}}$ s $\wedge$
`unify_eq_vars (dis_set` π_1 π_2`)` v `(`s`,`fe`) = SOME (`s`,`fcs`)` $\Rightarrow$
$fcs \vdash s \lhd_{\mathrm{n}}$ `Sus` π_1 $v \approx s \lhd_{\mathrm{n}}$ `Sus` π_2 v

Lemma 17. `verify_fcs` *extends equivalence to terms under the substitution*

$\vdash fe \vdash t_1 \approx t_2$ $\wedge$ `wfs`$_{\mathrm{n}}$ s $\wedge$ `FINITE` fe $\wedge$
`verify_fcs` fe s `= SOME` fex $\Rightarrow$
$fex \vdash s \lhd_{\mathrm{n}} t_1 \approx s \lhd_{\mathrm{n}} t_2$

Lemma 18. *The result of* `verify_fcs` *in a sub-map can be verified in the extension*

$\vdash$ `verify_fcs` fe s `= SOME` ve_0 $\wedge$ `verify_fcs` fe sx `= SOME` ve $\wedge$
$s \sqsubseteq sx$ $\wedge$ `wfs`$_{\mathrm{n}}$ sx $\wedge$ `FINITE` fe $\Rightarrow$
`verify_fcs` ve_0 sx `= SOME` ve

Corollary 3. `verify_fcs` *with a fixed substitution is idempotent.*

Definition 11. *Phase 1 (total version)*

```
add_bdg π v t0 (s,fe) =
  (let t = π⁻¹ • t0 in
     if ocn s t v then NONE else SOME (s |+ (v,t),fe))

tunifyn (s,fe) t1 t2 =
  if wfsn s then
    case (walkn s t1,walkn s t2) of
      (Nom a1,Nom a2) → if a1 = a2 then SOME (s,fe) else NONE
    ‖ (Sus π1 v1,Sus π2 v2) →
        if v1 = v2 then
          unify_eq_vars (dis_set π1 π2) v1 (s,fe)
        else
          add_bdg π1 v1 (Sus π2 v2) (s,fe)
    ‖ (Sus π1 v1,t2) → add_bdg π1 v1 t2 (s,fe)
    ‖ (t1,Sus π2 v2) → add_bdg π2 v2 t1 (s,fe)
    ‖ (Tie a1 t1,Tie a2 t2) →
        if a1 = a2 then
          tunifyn (s,fe) t1 t2
        else
          do
            fcs <- term_fcs a1 (s ◁n t2);
            tunifyn (s,fe ∪ fcs) t1 ([(a1,a2)] • t2)
          od
    ‖ (Pairn t11 t12,Pairn t21 t22) →
        do
          (sx,fex) <- tunifyn (s,fe) t11 t21;
          tunifyn (sx,fex) t12 t22
        od
    ‖ (Constn c1,Constn c2) →
        if c1 = c2 then SOME (s,fe) else NONE
    ‖ _ → NONE
  else
    ARB
```

Definition 12. *Nominal unification in two phases*

```
nomunify (s,fe) t1 t2 =
  do
    (sx,feu) <- unifyn (s,fe) t1 t2;
    fex <- verify_fcs feu sx;
    SOME (sx,fex)
  od
```

Fig. 2. Nominal Unification

Theorem 5. *The result of* `nomunify` *is a unifier, the freshness environment is finite, and the substitution is a well-formed extension*

$$\vdash \texttt{wfs}_{\texttt{n}}\ s \land \texttt{FINITE}\ fe \land \texttt{nomunify}\ (s,fe)\ t_1\ t_2 = \texttt{SOME}\ (sx,fex) \Rightarrow$$
$$\texttt{FINITE}\ fex \land \texttt{wfs}_{\texttt{n}}\ sx \land s \sqsubseteq sx \land fex \vdash sx \lhd_{\texttt{n}} t_1 \approx sx \lhd_{\texttt{n}} t_2$$

Proof. By recursion induction on `unify` *using the lemmas above.*

Theorem 6. *The result of* `nomunify` *is most general*

$$\begin{aligned}
&\vdash \texttt{wfs}_{\texttt{n}}\ s \land \texttt{FINITE}\ fe \land \texttt{nomunify}\ (s,fe)\ t_1\ t_2 = \texttt{SOME}\ (sx,fex) \land \\
&\quad \texttt{wfs}_{\texttt{n}}\ s_2 \land fe_2 \vdash s_2 \lhd_{\texttt{n}} (s \lhd_{\texttt{n}} t_1) \approx s_2 \lhd_{\texttt{n}} (s \lhd_{\texttt{n}} t_2) \Rightarrow \\
&\quad (\forall a\ v. \\
&\qquad (a,v) \in fex \Rightarrow \\
&\qquad (\exists b\ w\ fcs. \\
&\qquad\quad (b,w) \in fe \land \texttt{term_fcs}\ b\ (sx \lhd_{\texttt{n}} \texttt{Sus}\ []\ w) = \texttt{SOME}\ fcs \land \\
&\qquad\quad (a,v) \in fcs) \lor fe_2 \vdash a \# s_2 \lhd_{\texttt{n}} \texttt{Sus}\ []\ v) \land \\
&\quad \forall t.\ fe_2 \vdash s_2 \lhd_{\texttt{n}} (sx \lhd_{\texttt{n}} t) \approx s_2 \lhd_{\texttt{n}} (s \lhd_{\texttt{n}} t)
\end{aligned}$$

Proof. The second part of the conclusion is analogous to Lemma 14, and the proof is similar.

The first part is *via* the following lemma, which is proved by recursion induction on `unify`: that any freshness constraint generated by `nomunify` either originates in the input environment or is a member of the minimal freshness environment required to equate $sx \lhd_{\texttt{n}} t_1$ and $sx \lhd_{\texttt{n}} t_2$. We then use the second part to show that (s_2, fe_2) must satisfy that minimal environment.

Theorem 7. *If the terms are unifiable, then* `nomunify` *succeeds*

$$\begin{aligned}
&\vdash fe_2 \vdash s_2 \lhd_{\texttt{n}} (s \lhd_{\texttt{n}} t_1) \approx s_2 \lhd_{\texttt{n}} (s \lhd_{\texttt{n}} t_2) \land \texttt{wfs}_{\texttt{n}}\ s_2 \land \texttt{wfs}_{\texttt{n}}\ s \Rightarrow \\
&\quad \exists sx. \\
&\quad\ \ \forall fe. \\
&\qquad \texttt{FINITE}\ fe \land \texttt{verify_fcs}\ fe\ sx \neq \texttt{NONE} \Rightarrow \\
&\qquad \exists fex.\ \texttt{nomunify}\ (s,fe)\ t_1\ t_2 = \texttt{SOME}\ (sx,fex)
\end{aligned}$$

We can always provide the empty set for the input freshness environment, since $\texttt{verify\_fcs}\ \emptyset\ s = \texttt{SOME}\ \emptyset$.

Proof. By recursion induction on `unify`; the proof is similar to that of Theorem 4. Since `unify` extends but otherwise ignores the input freshness environment, we assume the input freshness environment is empty for the inductive proof. We also use the following lemma in the recursive case.

Lemma 19. *The freshness environment generated by one side of a pair will verify in the substitution computed for both sides.*

$$\begin{aligned}
&\vdash fe \vdash t_1 \approx t_2 \land \texttt{term_fcs}\ a\ t_1 = \texttt{SOME}\ fcs_1 \land fcs_1 \subseteq fe \Rightarrow \\
&\quad \exists fcs_2.\ \texttt{term_fcs}\ a\ t_2 = \texttt{SOME}\ fcs_2 \land fcs_2 \subseteq fe
\end{aligned}$$

7 Related Work

Robinson's recursive descent algorithm traditionally takes two terms as input and produces an idempotent most general unifier on success. This algorithm has been mechanised elsewhere in an implementable style (*e.g.*, by Paulson [12]). McBride [13] shows that the algorithm can be structurally recursive in a dependently typed setting, and formalises it this way using LEGO. McBride also points to many other formalisations. The other main approach to the presentation and formalisation of unification algorithms is the Martelli-Montanari transformation system [1]. Ruiz-Reina *et al.* [14] formalise a quadratic unification algorithm (using term graphs, due to Corbin and Bidoit) in ACL2 in the transformation style.

Urban [15] formalised nominal unification in Isabelle/HOL in transformation style. Nominal unification admits first-order unification as a special case, so this can also be seen as a formalisation of first-order unification. Much work on implementing and improving nominal unification has been done by Calvès and Fernández. They implemented nominal unification [16] and later proved that the problem admits a polynomial time solution [3] using graph-rewriting.

8 Conclusion

This paper has demonstrated that the pragmatically important technique of the triangular substitution is amenable to formal proof. Unification algorithms using triangular substitutions occur in the implementations of logical systems, and are thus of central importance. We have shown correctness results for unification algorithms in this style, both for the traditional first-order case, and for nominal terms.

Future Work There are imperative unification algorithms (such as Paterson and Wegman's [2]) with much better time complexity than Robinson's that use ephemeral data structures. Conchon and Filliâtre [17] have shown that Tarjan's classic union-find algorithm can be transformed into one using persistent data structures. It would be interesting to see if similar ideas can be applied to an imperative unification algorithm; indeed some unification algorithms make use of union-find.

The walk-based substitution application algorithms in this paper can benefit from sophisticated representations of substitutions, as well as from optimizations to the walk algorithm itself. We have done some work on formalising the improvements to `walk*` described by Byrd [8]. Future work includes continuing this formalisation and also investigating representations of triangular substitutions other than the obvious lists.

The Martelli-Montanari transformation system has become a standard platform for presenting unification algorithms, but wasn't immediately applicable for us because it assumes idempotent substitutions are used. However it may be possible to create a transformation system based on triangular substitutions, and it would be interesting to see how it relates to the usual system.

In this paper we formalised the original, inefficient presentation of nominal unification from [9]. The improved nominal unification algorithms by Calvès and Fernández should also be formalised.

Acknowledgements NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. Martelli, A., Montanari, U.: An efficient unification algorithm. ACM Trans. Program. Lang. Syst. **4**(2) (1982) 258–282
2. Paterson, M., Wegman, M.N.: Linear unification. J. Comput. Syst. Sci. **16**(2) (1978) 158–167
3. Calvès, C., Fernández, M.: A polynomial nominal unification algorithm. Theor. Comput. Sci. **403**(2-3) (2008) 285–306
4. Levy, J., Villaret, M.: Nominal unification from a higher-order perspective. In Voronkov, A., ed.: RTA. Volume 5117 of Lecture Notes in Computer Science., Springer (2008) 246–260
5. Hoder, K., Voronkov, A.: Comparing unification algorithms in first-order theorem proving. In Mertsching, B., Hund, M., Aziz, M.Z., eds.: KI. Volume 5803 of Lecture Notes in Computer Science., Springer (2009) 435–443
6. Baader, F., Snyder, W.: Unification theory. In Robinson, J.A., Voronkov, A., eds.: Handbook of Automated Reasoning. Elsevier and MIT Press (2001) 445–532
7. Friedman, D.P., Byrd, W.E., Kiselyov, O.: The Reasoned Schemer. The MIT Press (2005)
8. Byrd, W.E.: Relational Programming in miniKanren: Techniques, Applications, and Implementations. PhD thesis, Indiana University (2009)
9. Urban, C., Pitts, A.M., Gabbay, M.: Nominal unification. Theor. Comput. Sci. **323**(1-3) (2004) 473–497
10. Cheney, J., Urban, C.: alpha-Prolog: A logic programming language with names, binding and α-equivalence. In Demoen, B., Lifschitz, V., eds.: ICLP. Volume 3132 of Lecture Notes in Computer Science., Springer (2004) 269–283
11. Byrd, W.E., Friedman, D.P.: alphaKanren: A fresh name in nominal logic programming languages. Scheme and Functional Programming (2007)
12. Paulson, L.C.: Verifying the unification algorithm in LCF. Sci. Comput. Program. **5**(2) (1985) 143–169
13. McBride, C.: First-order unification by structural recursion. J. Funct. Program. **13**(6) (2003) 1061–1075
14. Ruiz-Reina, J.L., Martín-Mateos, F.J., Alonso, J.A., Hidalgo, M.J.: Formal correctness of a quadratic unification algorithm. J. Autom. Reasoning **37**(1-2) (2006) 67–92
15. Urban, C.: Nominal unification. `http://www4.in.tum.de/~urbanc/Unification/` (2004)
16. Calvès, C., Fernández, M.: Implementing nominal unification. Electr. Notes Theor. Comput. Sci. **176**(1) (2007) 25–37
17. Conchon, S., Filliâtre, J.C.: A persistent union-find data structure. In Russo, C.V., Dreyer, D., eds.: ML, ACM (2007) 37–46