(Nominal) Unification by Recursive Descent with Triangular Substitutions

Ramana Kumar[1] and Michael Norrish[2]

[1] The Australian National University u4305025@anu.edu.au
[2] National ICT Australia Michael.Norrish@nicta.com.au

Abstract. 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.
This style of algorithm has performance benefits and has not been mechanised previously. The algorithms use nested recursion so the termination proofs are non-trivial; the termination relation is also slightly different from usual.

1 Introduction

The fastest known unification algorithms are time and space linear (or almost linear) in the size of the input terms [1–3]. In the case of nominal unification, a polynomial algorithm is known to exist [4]. 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 perform a single occurs check at the end of unification, or even to do the occurs check implicitly, in an algorithm that doesn't use recursive descent.

However, naïve algorithms are used in real systems for a number of reasons: worst case inputs do not come up often in practice, systems that perform unification frequently must pay to encode the input and decode the output of a fast algorithm (unless the same representations can be used throughout the system), and naïve algorithms are simpler to implement and teach.

Some evidence for the first two assertions can be found in Hoder & Voronkov [5] where an imperative version of the algorithm in this paper (there labelled "Robinson's") benchmarks better than the worst-case linear algorithms. 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.

One important feature of the algorithms considered by Hoder & Voronkov is that they all use *triangular* substitutions. Such substitutions are important in systems that do backtracking and need to "unapply" substitutions from terms.

The unapply operation can be made implicit (and thus, efficient) by updating a shared substitution (making it "persistent"): backtracking computations can apply the appropriate subset of the shared substitution whenever terms in context are required.

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 simply update some shared global, which is what happens in the implementation of Robinson's algorithm in Hoder & Voronkov.) So, for example, the unification algorithm in miniKanren [7, 8] takes two terms, t_1 and t_2, and an input 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 pair (z, z) then an idempotent substitution would contain three copies of the term (z, z), whereas a triangular substitution would contain just one.

Baader & 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 doing so 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; contrast the opaque indices of de Bruijn encodings.

Nominal systems (*e.g.*, αProlog [9], alphaKanren [10]) 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 [10] describes unification with idempotent substitutions; our mechanisation is of a later, more efficient implementation using triangular substitutions.)

Outline In Section 2, we describe first-order terms, triangular substitutions, and the important well-formedness condition on the latter. In Section 3, we describe the application of substitutions to terms, as generalised for non-idempotent substitutions. In Section 4, we describe the definition and termination proof for first-order unification. In Section 5, we prove unification meets its specification, with statements of correctness suitable for its accumulator-passing style. In Sec-

tion 6, we describe the extra work necessary to repeat the above for nominal unification.

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

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 $f(f(\ldots f(x)))$.

The `do` notation is used for writing in monadic style. We only use it to express bind in the option monad: the term `do` $y \leftarrow f\ x$`;` $g\ y$ `od` means `NONE` if $f\ x$ returns `NONE` otherwise `SOME (`$g\ v$`)` if $f\ x$ returns `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 $f \circ fm$.

R^+ denotes the transitive closure of a relation.

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 the act of replacing some variables in a term with terms—substitution of t_1 for x and t_2 for y in t—or it can refer to a collection of variable bindings that may be applied to a term—the substitution that binds x to t_1 and y to t_2. When a substitution in the latter sense is represented by a function that performs the variable replacement (usually a function from variables to variables that is lifted to terms), the two senses almost coincide: applying a substitution versus the function that performs the application.

But sometimes substitutions are represented by data structures admitting just variable lookup, and a separate application function, which takes a substitution and a term, substitutes and returns the new term. Different data structures may represent substitutions that are equivalent under application. This paper distinguishes substitutions as data structures from substitution application in order to investigate a representation of substitutions, 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); constants can be represented by any type.

Definition 1. *Terms*

```
datatype term
  = Var of num | Pair of term ⇒ term | Const of const
```

It should be straightforward to transfer all the results in this paper to a setting in which terms are generated from a signature. We use `Pair` and `Const` term constructors to avoid littering our proofs with assertions that all terms are from the same signature. This is also the model used by miniKanren.

We represent a substitution (the HOL type `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 ' (Var v) = case FLOOKUP s v of NONE → Var v ‖ SOME t → t
s ' (Pair t1 t2) = Pair (s ' t1) (s ' t2)
s ' (Const c) = 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 compose application functions, which gives the same effect.

Lemma 1. $\vdash \forall t.$ `selfapp` s `'` t `=` s `' (`s `'` t`)`

2.1 Well-formed Substitutions

For each substitution s we define a relation `vR` 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*

```
vR s y x ⟺
  case FLOOKUP s x of NONE → F ‖ SOME t → y ∈ vars t
```

A substitution is well-formed (`wfs`) if `vR` s is well-founded. There are three informative statements equivalent to the well-formedness of a substitution. We omit the proofs from this paper since they are not our main focus.

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

$\vdash$ `wfs` $s \iff \forall v.\ \neg$`(vR` s`)`$^+$ v v

Corollary 1. $\vdash$ `wfs` $s \Rightarrow$ `noids` s

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

$\vdash$ `wfs` $s \iff$ `wfs (selfapp` s`)`

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

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

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

We also have two interesting results about idempotent substitutions that were used in proving the above.

Lemma 5. *Only idempotent substitutions have domain and range disjoint*

$\vdash$ `idempotent` $s \wedge$ `noids` $s \iff$ `DISJOINT (FDOM` s`) (rangevars` s`)`

Lemma 6. *Idempotent substitutions are well-formed*

$\vdash$ `idempotent` $s \wedge$ `noids` $s \Rightarrow$ `wfs` s

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 easily achieved via recursion in the application function `walk*` (we write $s \triangleright t$ 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. When defined, we derive the following characterisation:

Lemma 7. *Characterisation of* `walk*`

$\vdash$ `wfs` $s \Rightarrow$
`  ` $s \triangleright$ `Var` v `=`
`    (case FLOOKUP` s v `of NONE` $\rightarrow$ `Var` v $\parallel$ `SOME` $t \rightarrow s \triangleright t$`)` $\wedge$
`  ` $s \triangleright$ `Pair` t_1 t_2 `= Pair (`$s \triangleright t_1$`) (`$s \triangleright t_2$`)` $\wedge$
`  ` $s \triangleright$ `Const` c `= Const` c

Lemma 8. `walk*` *reduces to application on idempotent substitutions*

$\vdash$ `wfs` $s \Rightarrow$ `(idempotent` $s \iff$ `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 using the current substitution. This reveals just enough of a term-in-context's structure for the current level of recursion.

The same idiom is also 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 $(\texttt{vR}\ s)^+$ over a term's variables, and the term's size.

4 Unification: Definition

Our unification algorithm, `unify`, has type

```
subst → term → term → 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 as follows; the definition of `unify` will be the `then` branch of the `if`.

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 t1a t1d,Pair t2a t2d) →
        do sx ← tunify s t1a t2a; tunify sx t1d t2d od
   ‖  (Const c1,Const c2) → if c1 = c2 then SOME s else NONE
   ‖  _ → NONE
 else
   ARB
```

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

1. `WF R`
2. $\forall$ `t2 t1 s t1a t1d t2a t2d .`
   `wfs s ∧ walk s t1 = Pair t1a t1d ∧ walk s t2 = Pair t2a t2d ⇒`
   `R (s,t1a,t2a) (s,t1,t2)`
3. $\forall$ `t2 t1 s t1a t1d t2a t2d sx .`
   `wfs s ∧`
   `(walk s t1 = Pair t1a t1d ∧ walk s t2 = Pair t2a t2d) ∧`
   `tunify_tupled_aux R (s,t1a,t2a) = SOME sx ⇒`
   `R (sx,t1d,t2d) (s,t1,t2)`

A call to `tunify_tupled_aux` appears in condition 3 because the argument *sx* in the second recursive call `tunify sx t1d t2d` 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 `uR`, defined below, makes this statement in the final conjunct. The other conjuncts are also satisfied by the algorithm and are required to ensure that `uR` is well-founded.

Definition 7. *Termination relation* (`uR`)

```
uR (sx,c1,c2) (s,t1,t2) ⟺
  wfs sx ∧ s ⊑ sx ∧ sysvars sx c1 c2 ⊆ sysvars s t1 t2 ∧
  measure (term_depth ∘ walk* sx) c1 t1
```

Theorem 1. `uR` *is well-founded*

```
⊢ WF uR
```

Proof. By contradiction. If there is an infinite `uR`-chain, then the set of variables in the arguments (`sysvars`) must reach a fixpoint because each successive set is

a subset of its predecessor, and the sets are finite. As the set of system variables is getting smaller, the substitutions are allowed to get larger (the $\sqsubseteq$ relation). However, once the set of system variables reaches its fixpoint, the substitutions will be drawing on a fixed source for new variable bindings, and must also reach a fixpoint. Once this happens, the measure conjunct of the relation has a fixed first argument (the *sx* argument is fixed) and this ensures that the supposedly infinite chain cannot exist.

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

Lemma 9. *Termination condition 2*

```
uR (s,t1a,t2a) (s,t1,t2)
```

Proof. For the conjunct involving `sysvars`: either *t1* = *Pair t1a t1d* or the pair is in the range of the substitution, and similarly for *t*2. The other `uR` conjuncts are simple.

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

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

```
uP sx s t1 t2 ⟺
  wfs sx ∧ s ⊑ sx ∧ substvars sx ⊆ sysvars s t1 t2
```

Lemma 10. `uP` *implies* `uR` *on subterms*

```
⊢ wfs s ∧ walk s t1 = Pair t1a t1d ∧
  walk s t2 = Pair t2a t2d ∧
  (uP sx s t1a t2a ∨ uP sx s t1d t2d) ⇒
  uR (sx,t1d,t2d) (s,t1,t2)
```

Lemma 11. `unify` *implies* `uP`

```
⊢ ∀s t1 t2 sx.
    wfs s ∧ tunify_tupled_aux uR (s,t1,t2) = SOME sx ⇒
    uP sx s t1 t2
```

Proof. By well-founded induction (knowing that `uR` is well-founded).

Lemma 12. *Termination condition 3*

```
⊢ wfs s ∧ walk s t1 = Pair t1a t1d ∧
  walk s t2 = Pair t2a t2d ∧
  tunify_tupled_aux uR (s,t1a,t2a) = SOME sx ⇒
  uR (sx,t1d,t2d) (s,t1,t2)
```

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.
- If there exists a unifier of $s \triangleright t_1$ and $s \triangleright t_2$, then `unify` $s\ t_1\ t_2$ succeeds.

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.

A substitution s is a *unifier* of terms t_1 and t_2 if $s \triangleright t_1 = s \triangleright t_2$.

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

$$\begin{array}{l}\vdash \forall s\ t_1\ t_2\ sx.\\ \quad \texttt{wfs}\ s \wedge \texttt{unify}\ s\ t_1\ t_2 = \texttt{SOME}\ sx \Rightarrow\\ \quad \texttt{wfs}\ sx \wedge s \sqsubseteq sx \wedge sx \triangleright t_1 = sx \triangleright t_2\end{array}$$

As sx is an extension of the input s, we can equally regard `unify` as calculating a unifier for the terms-in-context $s \triangleright t_1$ and $s \triangleright t_2$.

Proof. The first two conjuncts, that s is a sub-map of sx and sx is well-formed, are corollaries of Lemma 11. 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 13 (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 13. `walk*` *over a sub-map*

$$\vdash s \sqsubseteq sx \wedge \texttt{wfs}\ sx \Rightarrow sx \triangleright t = sx \triangleright (s \triangleright t)$$

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

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

$$\begin{array}{l}\vdash \forall s\ t_1\ t_2\ sx\ s_2.\\ \quad \texttt{wfs}\ s \wedge \texttt{unify}\ s\ t_1\ t_2 = \texttt{SOME}\ sx \wedge \texttt{wfs}\ s_2 \wedge\\ \quad s_2 \triangleright (s \triangleright t_1) = s_2 \triangleright (s \triangleright t_2) \Rightarrow\\ \quad \forall t.\ s_2 \triangleright (sx \triangleright t) = s_2 \triangleright (s \triangleright t)\end{array}$$

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

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

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

We can give a fuller account of non-empty contexts by defining a notion of "compatibility". A substitution s is *compatible* with s_0 (written $s \ni s_0$) if applying s_0 before applying s is the same as just applying s. In other words, s_0 is more general than s: the information in s_0 is present in s, but s may include more. It would be simpler to say that substitutions are compatible if the second is a sub-map of the first, but we use the weaker notion of compatibility to allow pairs of substitutions that bind variables in the opposite directions (v to u rather than u to v, say), which could not be sub-maps, but which may be compatible.

Definition 9. *Compatibility of substitutions*

$$s \ni s_0 \iff \texttt{wfs}\ s\ \wedge\ \texttt{wfs}\ s_0\ \wedge\ \forall t.\ s \triangleright (s_0 \triangleright t) = s \triangleright t$$

Property 1. Compatible means more specific

$$\vdash s \ni s_0 \iff$$
$$\texttt{wfs}\ s\ \wedge\ \texttt{wfs}\ s_0\ \wedge$$
$$\forall t_1\ t_2.\ s_0 \triangleright t_1 = s_0 \triangleright t_2 \Rightarrow s \triangleright t_1 = s \triangleright t_2$$

Property 2. Extensions are compatible

$$\vdash \texttt{wfs}\ sx\ \wedge\ s \sqsubseteq sx \Rightarrow sx \ni s$$

Lemma 15. *The kind of extensions made by* `unify` *preserve compatibility*

$$\vdash sx \ni s\ \wedge\ \texttt{wfs}\ (s\ \texttt{|+}\ (vx, tx))\ \wedge\ vx \notin \texttt{FDOM}\ s\ \wedge$$
$$sx \triangleright \texttt{Var}\ vx = sx \triangleright tx \Rightarrow$$
$$sx \ni s\ \texttt{|+}\ (vx, tx)$$

Lemma 16. *A variable occurring in a pair will* `walk*` *to a term smaller than the pair in a compatible substitution*

$$\vdash \texttt{oc}\ s\ (\texttt{Pair}\ t_1\ t_2)\ v\ \wedge\ sx \ni s \Rightarrow$$
$$\texttt{measure (term\_depth} \circ \texttt{walk*}\ sx)\ (\texttt{Var}\ v)\ (\texttt{Pair}\ t_1\ t_2)$$

Theorem 4. *The result of* `unify` *is compatible under any other unifier*

$$\vdash \forall s\ t_1\ t_2\ sx.$$
$$sx \ni s\ \wedge\ sx \triangleright t_1 = sx \triangleright t_2 \Rightarrow$$
$$\exists si.\ \texttt{unify}\ s\ t_1\ t_2 = \texttt{SOME}\ si\ \wedge\ sx \ni si$$

Since compatibility is reflexive, we get the third part of our specification as a special case of this theorem.

Proof. By recursion induction on `unify`, using Lemma 15, and using Lemma 16 in the pair 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 10. *Concrete nominal terms*

```
datatype Cterm
  = CNom of string
  | CSus of (string, string) alist ⇒ num
  | CTie of string ⇒ Cterm
  | CPair of Cterm ⇒ Cterm
  | CConst of const
```

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 11. *Walking a suspension*

```
wfs s ⇒
vwalk s π v =
  case FLOOKUP s v of
     SOME (Sus p u) → vwalk s (π ++ p) u
  || SOME t → π • t
  || NONE → Sus π 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 `vwalk` is the same as in the first-order case; the permutation doesn't play a part in the recursion. Nominal `walk` calls `vwalk s p v` for a suspension `Sus p v`, otherwise returns its argument. Nominal `walk*` uses `walk` as before, this time recursing on ties as well as pairs.

Nominal unification was first defined by Urban, Pitts, and Gabbay [11] to work in two phases. In the first phase, a substitution is constructed along with a set of *freshness constraints* (alternatively, a *freshness environment*). In [11], the substitution is idempotent; ours will be triangular. A freshness constraint is

a pair of a name and a variable, expressing the constraint that the variable is never bound to a term where 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 checking succeeds, the substitution and the new freshness environment, which together form a nominal unifier, are returned. Our definition of nominal unification differs from this in a number of respects:

- it is written in accumulator-passing style, so takes a substitution and a freshness environment as input;
- the triangular substitution returned from the first phase must be referred to as the freshness constraints are checked in the second phase; and
- the implementations of the two phases are written in a functional style, making them directly implementable. This is in contrast with the rule-based style of [11].

The final definition in HOL is thus:

Definition 12. *Nominal unification in two phases*

```
nunify (s,fe) t1 t2 =
  do
    (sx,feu) ← unify (s,fe) t1 t2;
    fex ← verify_fcs feu sx;
    SOME (sx,fex)
  od
```

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, `unify` is defined *via* a total function `tunify`. 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 ▷ 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.

Definition 13. *Phase 1 (total version)*

```
add_bdg π v t0 (s,fe) =
  (let t = π⁻¹ • t0 in
```

```
      if oc s t v then NONE else SOME (s |+ (v,t),fe))
⊢ tunify (s,fe) t1 t2 =
    if wfs s then
      case (walk s t1,walk 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
             tunify (s,fe) t1 t2
           else
             do
               fcs ← term_fcs a1 (s ▷ t2);
               tunify (s,fe ∪ fcs) t1 ([(a1,a2)]• t2)
             od
       ‖ (Pair t1a t1d,Pair t2a t2d) →
           do
             (sx,fex) ← tunify (s,fe) t1a t2a;
             tunify (sx,fex) t1d t2d
           od
       ‖ (Const c1,Const c2) →
           if c1 = c2 then SOME (s,fe) else NONE
       ‖ _ → NONE
    else
      ARB
```

Phase 2 is implemented by `verify_fcs`, which calls

```
term_fcs a (s ▷ 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 `nunify`, 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. For example, $(\lambda a.X)$ and $(\lambda b.Y)$ unify with X bound to $(a\,b)\cdot Y$ (the substitution), but only if $a\#Y$ (the freshness environment). In the absence of the latter, one might instantiate Y with a, and therefore X with b, producing non-equivalent terms.

Thus, our first correctness result depends on the definition of α-equivalence with respect to a freshness environment (`equiv`).

Lemma 17. *The freshness environment computed by* `unify_eq_vars` *makes the suspensions equivalent*

$$\begin{array}{l}\vdash \texttt{wfs}\ s\ \wedge \\ \quad \texttt{unify\_eq\_vars (dis\_set}\ \pi_1\ \pi_2\texttt{)}\ v\ \texttt{(}s\texttt{,}fe\texttt{)} = \texttt{SOME (}s\texttt{,}fcs\texttt{)} \Rightarrow \\ \quad \texttt{equiv}\ fcs\ \texttt{(}s \triangleright \texttt{Sus}\ \pi_1\ v\texttt{)}\ \texttt{(}s \triangleright \texttt{Sus}\ \pi_2\ v\texttt{)}\end{array}$$

Lemma 18. `verify_fcs` *in an extended substitution preserves equivalence*

$$\begin{array}{l}\vdash \texttt{equiv}\ fe\ \texttt{(}s \triangleright t_1\texttt{)}\ \texttt{(}s \triangleright t_2\texttt{)} \wedge \texttt{wfs}\ sx \wedge s \sqsubseteq sx \wedge \texttt{FINITE}\ fe \wedge \\ \quad \texttt{verify\_fcs}\ fe\ sx = \texttt{SOME}\ fex \Rightarrow \\ \quad \texttt{equiv}\ fex\ \texttt{(}sx \triangleright t_1\texttt{)}\ \texttt{(}sx \triangleright t_2\texttt{)}\end{array}$$

Lemma 19. *The result of* `verify_fcs` *in a sub-map can be verified in the extension*

$$\begin{array}{l}\vdash \texttt{verify\_fcs}\ fe\ s = \texttt{SOME}\ ve_0 \wedge \texttt{verify\_fcs}\ fe\ sx = \texttt{SOME}\ ve \wedge \\ \quad s \sqsubseteq sx \wedge \texttt{wfs}\ sx \wedge \texttt{FINITE}\ fe \Rightarrow \\ \quad \texttt{verify\_fcs}\ ve_0\ sx = \texttt{SOME}\ ve\end{array}$$

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

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

$$\begin{array}{l}\vdash \forall s\ fe\ t_1\ t_2\ sx\ fex\,. \\ \quad \texttt{wfs}\ s \wedge \texttt{FINITE}\ fe \wedge \texttt{nunify (}s\texttt{,}fe\texttt{)}\ t_1\ t_2 = \texttt{SOME (}sx\texttt{,}fex\texttt{)} \Rightarrow \\ \quad \texttt{FINITE}\ fex \wedge \texttt{wfs}\ sx \wedge s \sqsubseteq sx \wedge \\ \quad \texttt{equiv}\ fex\ \texttt{(}sx \triangleright t_1\texttt{)}\ \texttt{(}sx \triangleright t_2\texttt{)}\end{array}$$

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

The other correctness results have not yet been proved.

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, introduced in [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 *et al.* [11] formalise 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 [15] and later proved that the problem admits a polynomial time solution [4] 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 Persistence is one of the benefits of using triangular substitutions. There are imperative unification algorithms (such as those in the style of Huet [2]) with much better time complexity than Robinson's that use ephemeral data structures. Conchon and Filliâtre [16] 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 [11]. The improved nominal unification algorithms by Calvès and Fernández should also be formalised.

References

1. Martelli, A., Montanari, U.: An efficient unification algorithm. ACM Transactions on Programming Languages and Systems **4**(2) (1982) 258–282
2. Huet, G.: Résolution d'équations dans des langages d'ordre 1, 2, ... ω. Thèse d'état, Université de Paris VII, Paris, France (1976)
3. Paterson, M.S., Wegman, M.N.: Linear unification. Journal of Computer and System Sciences **16** (1978) 158–167
4. Calvès, C., Fernández, M.: A polynomial nominal unification algorithm. Theoretical Computer Science **403**(2-3) (2008) 285–306
5. Hoder, K., Voronkov, A.: Comparing unification algorithms in first-order theorem proving. In Mertsching, B., Hund, M., Aziz, M.Z., eds.: KI 2009: Advances in Artificial Intelligence, 32nd Annual German Conference on AI, Paderborn, Germany, September 15-18, 2009. Proceedings. Volume 5803 of Lecture Notes in Computer Science., Springer (2009) 435–443
6. Baader, F., Snyder, W.: Unification theory. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Volume I. Elsevier Science (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. Cheney, J., Urban, C.: Alpha-prolog: A logic programming language with names, binding, and alpha-equivalence. In Demoen, B., Lifschitz, V., eds.: Logic Programming, 20th International Conference. Volume 3132 of LNCS., Springer (2004) 269–283
10. Byrd, W.E., Friedman, D.P.: alphaKanren: A fresh name in nominal logic programming languages. Scheme and Functional Programming (2007)
11. Urban, C., Pitts, A.M., Gabbay, M.J.: Nominal unification. Theoretical Computer Science **323**(1-3) (2004) 473–497
12. Paulson, L.C.: Verifying the unification algorithm in LCF. Science of Computer Programming **5**(2) (June 1985) 143–169
13. McBride, C.: First-order unification by structural recursion. Journal of Functional Programming **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. Journal of Automated Reasoning **37**(1) (August 2006) 67–92
15. Calvès, C., Fernández, M.: Implementing nominal unification. Electronic Notes in Theoretical Computer Science **176**(1) (2007) 25–37
16. Conchon, S., Filliâtre, J.C.: A persistent union-find data structure. In Russo, C., Dreyer, D., eds.: Proceedings of the ACM Workshop on ML, 2007, Freiburg, Germany, October 5, 2007, ACM (2007) 37–46