File: C:/bsd/rigel/Unification/LinUnify3.docx
Date: 2020 September – 2021 November – 2022 May

Faster Linear Unification Algorithm

Dennis de Champeaux
San Jose, CA USA
ddc2 AT ontooo DOT com
2022

Abstract
The Robinson unification algorithm has exponential worst case behavior. This triggered the development of (semi-)linear versions around 1976 by Martelli, A. & U. Montanari as well as by Paterson & Wegman. Another version emerged by Baader & Snyder around 2001. While these versions are distinctly faster on larger input pairs, the Robinson version still does better than them on small sized inputs. This paper describes yet another (semi-)linear version that is faster and challenges also the Robinson version on small sized inputs. All versions have been implemented and compared against each other on different types and sizes of input pairs.

Introduction
Robinson created the original unification algorithm [Robinson]. The significance of this algorithm was described by [Martelli&Montanari] with:

> This single, powerful rule can replace all the axioms and inference rules of the first-order predicate calculus and thus was immediately recognized as especially suited to mechanical theorem provers.

The Robinson algorithm – while used effectively in practice – has exponential behavior. Linear versions were developed around 1976-1978 by Martelli, A. & U. Montanari [Martelli&Montanari0] as well as by Paterson & Wegman [PatersonWegman] (PW). Unfortunately the core procedure in the last paper of the PW version still had an error and it could be improved [deChampeaux], 1986. That paper created another typo that was observed and corrected in 1991 by [Jacobson] and again in 2020 by [Motroi&Ciobaca]. The thoroughly corrected version is available at [GitHub].

Going from exponential to linear performance comes with a price. The PW linear algorithm depends on a custom data structure that did not get the attention that it deserved. A preprocessor is required to transform the traditional input format into that custom data structure; a postprocessor produces a proper, unordered substitution (when the two arguments are unifiable).

We found a semi-linear algorithm in [BaaderSnyder] (BS) from 2001. Our implementation of this version relies also on a custom, intricate data structure and requires similarly preprocessing and post processing. The algorithm description in the original publication can be improved as well by replacing three of four procedural modules by functions and by clarifying the data structure employed. This version is also available at [GitHub].

A novel, third algorithm, DC, is described that does not require preprocessing to wrap the input into another data representation. Instead each variable encountered during traversal of the input

(without repetitions) exploits attributes in the variable's object, which are used to capture findings about the variable's matches. It can be found also at [GitHub].

We compare those versions using generators for input pairs: four for unifiable pairs and four for non-unifiable pairs. The preprocessing is overhead that makes the Robinson version (potentially) competitive against the (semi-)linear ones on small inputs. Hence the three versions are also compared against the Robinson version on two sets of small sized inputs.

Tactics for unification

The general approach in unification algorithms is showing doggedly that the two input arguments are *not* unifiable and when running out of reject options admit that unification succeeds with a unifier that has been assembled on the way.

We assume here for illustrative purpose a simple control structure. The two input arguments are directed acyclic graphs with at least unique nodes for the variables and constants. A recursive descent is done for the two input arguments. Rejections can occur when:

- Both arguments are constants that are different, for example *a* and *b*. One argument is a constant and the other argument is a function expression, for example *a* and *f(c).* Both arguments are function expressions but the function names are different, for example *f(c)* and *g(d e).*
- One argument is a variable and the other argument is a function expression that contains the variable, for example *x* and *f(x).* (Note that our parser associates with a functional expression a container (`HashSet`) of the variables that occur in the expression.)
- Both arguments are variables while one is committed earlier to a functional expression that contains the other variable, for example *x* is committed to *f(y)* and the other variable is *y*.
- Both arguments are variable expressions and both are committed earlier to functional expressions. This causes a recursion with the two functional expressions as input arguments, which can fail.

A next tactic can be employed when all recursive traversals of the input have terminated and when it is observed that two committed variables cause an occur-cycle. Consider the two inputs *P(x x y)* and *P(f(y) y g(x)).* The variable *x* is committed to *f(y),* and *y* is committed to *g(x),* while *x* and *y* are committed to each other, which causes the cycle: *x-> f(g(x)).*

These tactics are incorporated in the versions we have encountered. It is helpful that a variable committed to an expression (constant or functional term) has a representative of an equivalence class of expressions that can be made identical. Finding an expression that *should* be in the equivalence class but does not *fit* causes a failure. We saw this happening in the last example where *x* is committed to *f(y),* a representative of an equivalence class, and the attempted addition of *g(x)* via *y* fails.

A Robinson type version

Our version of a Robinson type unification algorithm is part of a theorem prover with, among others, a Kowalski type connection graph module [Kowalski]. The full code is available as part of the class `Atom` [GitHub]. It has the tactics described above. After the tactics checks are

finished it still needs to produce a substitution (when a failure was not found). That is where the exponential behavior resides.

The Robinson version performance

We use eight different generators for generating challenging argument pairs. The `gen1` generator produces unifiable pairs with this pattern:

$P(h(x_1\ x_1)\ \ldots\ h(x_n\ x_n)\ y_2\ \ldots\ y_{n+1}\ x_{n+1})$
$P(x_2\ \ldots\ x_{n+1}\ h(y_1\ y_1)\ \ldots\ h(y_n\ y_n)\ y_{n+1})$

To show the exponential nature of the timings of the Robinson algorithm we used examples in the range 15-18:

Size	Timing
15	81
16	103
17	483
18	1775

We ran the generator on an I3-2310M /8GB machine for the range of 10-18 seven times and picked for each size the smallest time (in milliseconds).

Overview of the (semi-)linear unifier versions

As discussed above input arguments need to be preprocessed before a unification attempt can be made for the PW version and the BS version, and if an ordered unifier is produced it needs to be post processed.

This gives the following steps:

- Preprocessing:
  - Parse the two arguments using a parser that takes a linear string representation of a predicate expression and produces the typical LISP like tree representation
  - In case of PW and BS create for each argument a directed acyclic graph (dag) where a node contains an element from the tree representation with additional infrastructure components
- Create a context with infrastructure for the unification function in which an outcome is delivered:
  - Apply the unification function to the two dags
  - return the outcome `null` or the substitution
- Post-processing;
  - Terminate with `null` if the unification failed, otherwise create a non-ordered substitution

The parsing of predicate expression *P(a ?x f(a ?x))* (dropping the '?'question marks) yields the tree in figure 1.

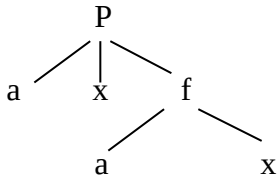

Figure 1. The layout of *P(a ?x f(a ?x))*.

The root *P* has three downward links to the constant *a*, the variable *x* and a reference to the function *f*, which has two downward links to the constant *a* and the variable *x*. Notice that the multiple occurrences of terminals *a* and *x* are *not* here shared, while they are shared in the representations of the versions.

The Unify DC version

No preprocessor

The DC version does not require the generation of a dag like structure as for the PW and the BS versions. Constants and variables have unique representations. While the BS version requires traversing an input argument twice, our version, like the PW version, traverses the input arguments only once.

The unification functions

As in the PW and BS versions unification happens inside a context. This context contains an instance of a `Hashtable` to keep track of all variables and there is also a `sigma` list that contains a substitution, if any. In addition there is a 'manual' stack for pairs of arguments to reduce system stack overflow when large (over size 100) input pairs are processed. The `unify` function takes care of pre- and post-processing and invokes the function `dunify` for deciding unifiability.

Like the BS version the DC version has multiple components. The control flow with recursion in `unif2` and `exploreVariable` is shown in figure 7.

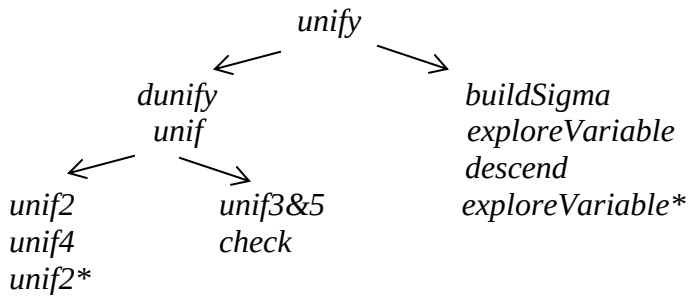

Figure 7. The control flow of the DC modules.

The left branch under `unify` relies on `dunify`, which delegates the input arguments to `unif` and assembles an ordered substitution if `unif` reports success. The left branch under `unif with unif2` and `unif4` has the reject tactics. The right branch under `unif` with `unif3&5` does the final occur-check. A key difference of this version resides in the `unif2` module. It investigates its two arguments thoroughly when at least one is a variable. The right branch under `unify` takes care of post-processing when unification succeeds.

The DC version exploits the object-orientedness of the input arguments. The Variable class has been extended with additional attributes to capture the different states that a variable can be in; among which:

- `first`, a `Term` that, if set, is a representative of an equivalence class
- `isVroot`, a `boolean` used for whether or not a variable is referring to another variable
- `myVroot`, a `Variable` which is set only for those variables whose `isVroot` attribute is false

- `variables`, a `HashSet` type attribute which is a reference to a `HashSet`, if available, in the `first` attribute that contains the variables in `first`

There is also a Boolean function `isRoot()`, which reports true if `first != null`.
The state of a variable is updated when it is matched against another term.

Pseudo code of the key functions `unif2`, `unif4` and `check`
Variables play a key role in the DC version. A variable is an object with attributes that characterize the different states it can be in:

- *Initial:*
  A variable has not yet been encountered; its attributes are set to:
  `first = null`; the type of `first` is Term, which can be a constant, or a functional expression
  `isVroot = true`; a default
  `myVroot = null`; the type of `myVroot` is Variable; a default
- *Variable contains the representative of an equivalence class:*
  The variable has matched and its attributes are set to:
  `first =` a term, which is a representative of an equivalence class
  `isVroot = true`
  `myVroot = null`
- *Variable does not contain the representative of an equivalence class:*
  The variable has matched against a variable; its attributes are set to:
  `first = null`
  `isVroot = false`
  `myVroot =` a variable which contains the representative of an equivalence class, or it will refer to a variable that has its `isVroot = true`

There is a `size` attribute with associated functions that is used to reduce the depth of the trees of variables that point to a variable with the representative of an equivalence class. The `findVroot()` function produces the current variable with `isVroot = true` and shortens the path to the `myVroot` variable. For example when there is a `myVroot` chain of variables *?p->?q->?r>?s>?t* using `findVroot(?p)` will return `?t` and as a side effect `?p`, `?q` and `?r` will directly refer to `?t`.

The `unif2` function updates the states of variables when at least one of its two arguments is a variable. We need the notion of a pre-substitution, which we define as a substitution that has not yet been tested to be cycle free. The main purpose of `unif2` is to generate a pre-substitution (a representative of an equivalence class) for an unassigned variable. When it encounters two variables both having already an assigned pre-substitution, it will recursively match them because they belong – if the match succeeds in the same equivalence class. The `message(..)` function returns `false` and optionally produces trace information in the implementation. The `unif2` function delegates its arguments to the function `unif4` if both arguments are not variables. (Hence the post condition of `unif2` relies on the post condition of `unif4` and the other way around.)
The function `unif2`:

```
boolean unif2(Term s, Term t) {
  if ( s.equals(t) ) return true;  // identical constants or variables
  if ( (s instanceof Variable) ) {
```

```
 Variable sv = (Variable) s;
 String svName = sv.getName();
 if ( null == htv.get(svName) ) { // insert sv in hash table
     htv.put(sv, svName);
}
if ( !sv.isVroot ) sv = findVroot(sv);
// sv is root or sv is Vroot
if ( (t instanceof Variable) ) {
    Variable tv = (Variable) t;
    String tvName = tv.getName();
    if ( null == htv.get(tvName) ) { // insert tv in hash table
        htv.put(tv, tvName);
    }
    if ( !tv.isVroot ) tv = findVroot(tv);
    // sv is root or sv is Vroot and tv is root or tv is Vroot
    if ( sv.equals(tv) ) return true;
    int sizes = sv.getSize(), sizet = tv.getSize();
    if ( sizet <= sizes )
        sv.setSize(sizes + sizet);
    else
        tv.setSize(sizes + sizet);
    if ( sv.isRoot() ) {
        Term sterm = sv.getFirst();
        if ( tv.isRoot() ) {
            if ( sizet <= sizes ) {
                tv.isVroot = false;
                tv.myVroot = sv;
            } else {
                sv.isVroot = false;
                sv.myVroot = tv;
            }
            Term tterm = tv.getFirst();
            if ( !unif2(sterm, tterm) ) // match them
                return message("unif2C sterm tterm", sterm, tterm);
            return true;
        }
        // sv is root and tv is not root, tv is Vroot
        tv.isVroot = false;
        tv.myVroot = sv; // tv refers to sv
        return true;
    }
    // sv is not root, is Vroot
    if ( tv.isRoot() ) {
        sv.isVroot = false;
        sv.myVroot = tv; // sv refers to tv
        return true;
    }
    // sv is not root, is Vroot & tv is not root, is Vroot
    if ( sizet <= sizes ) {
        tv.isVroot = false;
        tv.myVroot = sv;
    } else {
        sv.isVroot = false;
        sv.myVroot = tv;
    }
    return true;
}
```

```
    // s variable & t not a variable
    if ( sv.isRoot() ) {
        Term sterm = sv.getFirst();
        if ( !unif2(sterm, t) ) return message("unif2F sterm t", sterm, t);
        return true;
    }
    // s variable & s is Vroot  t not a variable
    sv.variables = t.getVariables();
         // the variables in the pre-substitution
    if ( sv.variables != null && sv.variables.contains(sv) )
        return message("unif2G t contains sv", t, sv);
    sv.setFirst(t); // create pre-substitution
    return true;
  }
  // s not a variable
  if ( (t instanceof Variable) ) {
      Variable tv = (Variable) t;
      String tvName = tv.getName();
      if ( null == htv.get(tvName) ) {
          htv.put(tv, tvName); // insert tv in hash table
      }
      if ( !tv.isVroot ) tv = findVroot(tv);
      if ( tv.isRoot() ) {
          Term tterm = tv.getFirst();
          if ( !unif2(tterm, s) ) return message("unif2H tterm s", tterm, s);
          return true;
      }
      // s not a variable & tv is not root isVroot
      tv.variables = s.getVariables();
          // the variables in the pre-substitution
      if ( tv.variables != null && tv.variables.contains(tv) )
          return message("unif2I s contains tv", s, tv);
      tv.setFirst(s); // create pre-substitution
      return true;
  }
  // s not a variable & t not a variable
  return unif4(s, t);
} // end unif2
```

The `unif4` function takes as arguments two terms, both of which are not variables. Its task is to recognize non-unifiability due to incompatible constants, function names, etc. Corresponding arguments of functional terms are delegated to the `unif2` module. However the description below does *not* channel these argument pairs into `unif2`; instead they are pushed on a stack, from which another function fetches the pairs and channels them into the `unif2` module. This modification was introduced to process large (over 100) sized input argument pairs.

```
boolean unif4(Term s, Term t) { // s and t are both not variables
  if ( s instanceof FTerm ) {
      FTerm s1 = (FTerm) s;
      if ( t instanceof FTerm ) {
          FTerm t1 = (FTerm) t;
          if ( !s1.getFunction().equals(t1.getFunction()) ) return false;
          Vector s1Args = s1.getArgs(), t1Args = t1.getArgs();
          int lng = s1Args.size();
          for ( int i = 0; i < lng; i++ ) {
```

```
                Term s1A = (Term)s1Args.elementAt(i),
                     t1A = (Term)t1Args.elementAt(i);
                // to avoid stack overflow
                stack.push(new TermsPair1(s1A, t1A));
            }
            return true;
        } // t not FTerm
        return false;
    } // s not FTerm
    if ( t instanceof FTerm ) return false;
    // s & t are symbols
    return s.equals(t);
} // end unif4
```

The function `unif5`, not shown, has access to the set of variables in the two input arguments. It contains a loop that invokes the `check` function on each variable and returns `false` if any of these calls returns `false`.

The `check` function does the occur check. If the variable has a pre-substitution term in the `first` attribute the variable has a reference to a `HashSet`, which contains the variables in that term. Hence a simple depth-first search will identify a cycle if there is one.

```
boolean check(Variable v) {
   if ( v.checked ) {
          return true;
   }
   if ( v.checking ) return message("check checking cycle", v, v); // cycle
   if ( !v.isRoot() ) { // not matched against a term
        if ( v.isVroot ) { // not matched against a variable
           v.checked = true;
           return true;
        }
        v.checking = true;
        Variable vnVroot = v.myVroot;
        if ( !check(vnVroot) ) return message("check2 ", v, vnVroot); // cycle
        v.checking = false;
        v.checked = true;
        return true;
   } // v is root
   v.checking = true;
   HashSet variables = v.variables;
   if ( null == variables ) { // nothing to check
        v.checking = false;
        v.checked = true;
        return true;
   }
   for( Iterator i = variables.iterator(); i.hasNext(); ) {
        Variable w = (Variable)i.next();
        if ( !check(w) ) return message("check3 v w ", v, w); // cycle
   }
   v.checking = false;
   v.checked = true;
   return true;
} // end check
```

Occur-check examples

We use the not unifiable pair:

P(x h(z) f(x)) P(g(y) y z)

A recursive depth-first search of the check function starting with the variable *z* and proceeds with:

z - f(x) - x - g(y) - y - h(z) - z, which yields a cycle.

Reversing the arguments gives:

y - h(z) - z - f(x) - x - g(y) - y, which yields a cycle.

The following example shows a cycle involving a *myVroot* link from *x* to *z*:

P(x y z) *P(f(y) z x)*

which produces the cycle:

x - z - f(y) - y - z

Reversing the arguments:

P(z y x) *P(x z f(y))*

produces the cycle through the *myVroot* chain:

x - z - y and the match of *x* with *f(y)* producing the *y* in *f(y)* failure.

Correctness

To prove correctness of the DC version we need to establish first what is to be shown. We will address:

- The `unif2` and `unif4` functions produce false when their inputs are not unifiable or they produce pre-substitutions that eliminate differences between the two input arguments downwards.
- The `check` function recognizes a cycle in a collection of pre-substitutions if present.
- If DC finds an ordered substitution *su* for the arguments pair *a1-a2* without a cycle then: *a1.su = a2.su*
- If there is a cycle free unifier *su* then DC will find it.
- If DC fails there is no cycle free substitution *su* for the arguments pair *a1-a2* so that *a1.su = a2.su*
- If there is a cycle free unifier *su* for the arguments pair *a1-a2* then DC will find it.
- If there is not a cycle free unifier *su* for the arguments pair *a1-a2* then DC will report failure.

Lemma A The `unif2` and `unif4` functions produce false when their inputs are not unifiable or they produce pre-substitutions that eliminate differences between the two input arguments downwards.

Proof: The functions `unif2` and `unif4` are cross recursive; hence we can obtain post conditions for one of them from what a (cross) recursive invocation produces. A base case in `unif2` is the test that the two input arguments are identical. Another base case is the recognition of non-unifiability when a variable occurs in a potential pre-substitution term; otherwise the variable's `first` attribute is set with the other argument term, which indeed eliminates a difference between the two input arguments. Invoking `unif2` on already encountered representatives of equivalent classes in its two variable arguments also eliminates differences between these representatives or identifies non unifiable arguments. A base case in `unif4` is the test for equality when both arguments are constants. Reject base cases are combinations of a constant with a function expression and two function expressions with different function names. Two function expressions with the same function names triggers recursive invocations of `unif2` on the

corresponding argument pairs and thereby contribute to the post condition of `unif4` and hence to the post condition of `unif2`.
QED

Lemma B The `check` function recognizes a cycle in a collection of pre-substitutions if present.
Proof: Reminder: a variable has a container with associated variables, a reference to the container of the variables occurring in its pre-substitution, if any. A simplified recursive definition *checkR* for checking a set of pre-substitutions to be cycle free is:

```
boolean checkR(Variable v) {
    if ( v.checked )
        return true; // v has been investigated already
    }
    if ( v.checking ) {
        return false; // found cycle
    }
    v.checking = true;
    if ( v has no associated variables ) {
        v.checking = false;
        v.checked = true;
        return true;
    }
    for each associated variable vx {
            if ( !checkR(vx) ) return false;
    }
    v.checking = false;
    v.checked = true;
    return true;
}
```

The `check` function is an implementation of this definition.
QED

Lemma C If DC finds an ordered substitution *su* for the arguments pair *a1-a2* without a cycle then: *a1.su = a2.su*
Proof: Lemma A shows that pre-substitutions eliminate differences between the two input arguments.
QED

Lemma D If there is a cycle free unifier *su* then DC will find it.
Proof:
Let *x->xs* be a substitution member of *su*.
Let *Vx* = { *v* | *v* is a variable in the equivalence class of *x* }
Let *Tx* = (*t* | *t* is a term in the equivalence class of *x* }
For each *v* in *Vx* there is a path of pairs $(v_0, v_1) \ldots (v_{n-1}, v_n)$ so that $v_0 = x$ and $v_n = v$ and (v_z, v_{z+1}) were matched.
For each t in *Tx* there is a *v* in *Vx* so that they were matched, hence *t* and *xs* were matched successfully.

DC will find these matches and thus obtains also the *x->xs* substitution member of *su*.
QED

Lemma E If DC fails there is no cycle free substitution so that *a1.su = a2.su*.
Proof: By contradiction according to lemma D.
QED

Lemma F If there is not a cycle free unifier *su* then DC will report failure.
Proof: By contradiction. If DC succeeds it obtains a cycle free unifier *su*.
QED

Theorem The DC version is correct according to the above lemmas.

Semi-Linearity
The functions `unif2` and `unif4` are executed at most as often as the number of nodes in the input arguments. The `check` function is called as often as there are variables and their total effort is a function of their number N. It is possible that after iterating over *p* variables the number of check invocations is $p+q < N$. This entails that *q* variables will be skipped when iterating further over the range (p+1 : N).
We consider this design of the occur check the most transparent we have encountered thus far.

The exception to linearity is the use, as in the BS version, of the *findVroot()* function, This function implements the *union-find* algorithm. It has complexity *O(n alpha(n))*, where *alpha* is the functional inverse of the Ackermann's function, which may be considered a small constant factor [Union-Find].

Post-processing
The PW, BS & DC versions produce ordered substitutions, which need post-processing. A postprocessor was described for the PW version in [deChampeaux]. The design of that version has been improved by replacing iteration by recursion. The implementations for the different versions differ due to using different data structures. An example should explain how it works. We can reuse the ordered substitution introduced in the Robinson section.

[?x2 -> h(?x1 ?x1) ?x3 -> h(?x2 ?x2) ?y2 -> h(?y1 ?y1) ?y3 -> h(?y2 ?y2) ?x1 -> ?y1]

We need to replace *?x1* in *?x2 -> h(?x1 ?x1)*. A depth-first search on the variables will recognize that the first occurrence of *?x1* in *?x2 -> h(?x1 ?x1)* needs replacement. This triggers an investigation whether *?y*1in *?x1 -> ?y1* needs adjustment, which is not the case; hence *?x1* can be marked with its *?x1 -> ?y1* substitution as being 'ready'. This will prevent having to investigate again the 2nd occurrence of *?x1* in *h(?x1 ?x1)* and any other binding of *?x1*, which can be large number.

Timing comparisons of the versions
We use different tests to compare the four different versions. An appendix has examples of the eight different generators for input pairs. These generators employ a size parameter for the number of arguments of the input arguments' signatures.

The timings were done on an I3 machine. After a warm-up, there is a loop for 500 iterations which measures the run time. This is repeated 10 times and the shortest timing is reported. This helps to compensate for the garbage collections that can interfere at any time. The entries represent the average execution time in milliseconds over the four generators. The best timings are shown in bold.

Small range for unifiable pairs

Size	Robinson	PW	BS	DC
1	**0.00550**	0.01100	0.00950	0.00750
2	0.01450	0.01750	0.01550	**0.01200**
3	0.02650	**0.01550**	0.01700	0.01650
4	0.08200	0.02050	0.02450	**0.01450**
5	0.29900	0.02600	0.02250	**0.01850**
6	1.19700	0.03300	0.02800	**0.02300**
Table 6 Timings for small sized unifiable pairs.				

The Robinson version is the clear winner for the size = 1 row. This supports the position that linear versions have too much overhead. The other rows challenge this verdict.

Small range for non-unifiable pairs

Size	Robinson	PW	BS	DC
1	0.00500	0.00600	0.00500	**0.00450**
2	0.00850	0.00950	0.00850	**0.00750**
3	0.01200	0.01350	0.01250	**0.01100**
4	**0.01400**	0.01650	0.01650	0.01450
5	**0.01650**	0.02100	0.02100	0.01850
6	**0.02000**	0.02600	0.02600	0.02300
Table 7 Timings for small sized non-unifiable pairs.				

Adding the timings for Robinson and DC versions for the sizes 1 &2 and for the unifiable and non-unifiable pairs we obtain:
Robinson: 3350 and DC: 3150.
For the size 3 we obtain:
Robinson: 3850 and DC: 2750.
Thus DC is also competitive on those small sized inputs.
The advantages that Robinson has on sizes 4-6 on non-unifiable pairs is lost when considering the timings of the unifiable pairs.

Large range for a combination of the unifiable and non-unifiable pairs

Size	PW	BS	DC
5	0.02475	0.02250	**0.01875**
10	0.05750	0.05400	**0.04175**
20	0.16400	0.15350	**0.12350**
40	0.56850	0.52400	**0.47875**
Table 8 Timing for the combination of large sized unifiable and non-unifiable pairs.			

The data in table 8 supports the conjecture that the removal of preprocessing, the use of the different data structures and the central role of variables in DC made a positive difference.

Discussion, future work and conclusion

Unification algorithms are similar in that they distinguish between non-unifiability based on structural differences in the two input arguments and alternatively failing an occur check. The three (semi-)linear versions have different designs for doing the occur check using different data structures for representing the input arguments. Two versions have their own ways to preprocess the arguments and wrap them in richer data structures. The third version avoids the overhead of preprocessing. While checking for structural differences it uses additional attributes in variables to register properties that accumulate during the arguments scrutinizing phase. Our timings show that our 3rd version, DC, out performs the two other (semi-)linear ones.

An earlier extensive study [HoderVoronkov] claimed that the original Robinson version is best for theorem proving settings. Our admittedly limited testing does *not* support their conclusion that small sized arguments are best handled by the Robinson version. Hence we hope that others can elaborate our findings.

The representation of a physical object could be done with nine parameters for location, velocity and spin. Reasoning about Sudoku problem solvers could rely also on predicates with nine arguments. Hence domains are conceivable where the versions discussed are a better fit than the Robinson algorithm.

Unification algorithms are used also in problem solving settings to ascertain that an operator can be applied and if so obtain the required substitutions. The PDDL language was developed to specify domains with its operators [PDDL]. It is likely that the versions discussed can be used to deal with complex contextual/ temporal preconditions in these PDDL domains.

Our experiments were done using implementations in the Java language. Using its object-oriented features has been a major advantage to represent predicate calculus expressions and to model the additional infrastructures required in the different versions. The DC version exploits the object-ness of variables so that it was easy to add additional attributes in their class, which was crucial to reduce overheads. Object-Orientation has promoted conceptual architectures and designs. Newer data structures may still be forthcoming that will provide even better (unification) algorithms.

Statements and Declarations
The author has not been funded and there are no competing interests.

References

[BaaderSnyder] Baader, F. & W.Snyder, Unification theory, Chapter 8 in HANDBOOK OF AUTOMATED REASONING, Edited by Alan Robinson and Andrei Voronkov, Elsevier Science Publishers B.V., 2001.
https://www.cs.bu.edu/fac/snyder/publications/UnifChapter.pdf

[deChampeaux] de Champeaux, D., About the Paterson-Wegman Linear Unification Algorithm, Journal of Computer and System Sciences, vol 32, no 1, pp 79-90, 1986 February.
https://doi.org/10.1016/0022-0000(86)90003-6

[Jacobson] Jacobson, E., Unification and anti-unification, Technical report, 1991.
http://erikjacobsen.com/pdf/unification.pdf

[GitHub] https://github.com/ddccc/Unification

[HoderVoronkov] Hoder K., Voronkov A. (2009) Comparing Unification Algorithms in First-Order Theorem Proving. In: Mertsching B., Hund M., Aziz Z. (eds) KI 2009: Advances in Artificial Intelligence. KI 2009. Lecture Notes in Computer Science, vol 5803. Springer, Berlin, Heidelberg.
https://doi.org/10.1007/978-3-642-04617-9_55

[Kowalski] Kowalski, A Proof Procedure Using Connection Graphs, Journal of the Association for Computing Machinery, Vol. 22, No. 4, October 1975, pp. 572-59.
https://doi.org/10.1145/321906.321919

[Martelli&Montanari0] Martelli, A. & U. Montanari, Unification in linear time and space: A structured presentation. Internal Rep. B76-16, Ist. di Elaborazione delle Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy, July 1976.

[Martelli&Montanari] Martelli, A. & U. Montanari, An Efficient Unification Algorithm, 1982.
https://doi.org/10.1.1.96.6119.pdf

[Motroi&Ciobaca] Motroi, V. & S. Ciobaco, A Typo in the Paterson-Wegman-de Champeaux algorithm, 2020.
https://dev.arxiv.org/abs/2007.00304

[PatersonWegman] Paterson, M.S. & M.N. Wegman, Linear Unification, Journal of Computer System Science 16 (2): pp 158-167, 1978.
https://doi.org/10.1016/0022-0000(78)90043-0

[PDDL] https://en.wikipedia.org/wiki/Planning_Domain_Definition_Language

[Robinson] Robinson, J.A., A machine-oriented logic based on the resolution principle, J. Assoc. Comput. Mach. 12, No 1, pp 23-41, 1965.

[Union-Find] https://en.wikipedia.org/wiki/Disjoint-set_data_structure

Appendix Eight generators

Eight generators were used for obtaining argument pairs to compare the performance of the versions discussed. These generators take as argument the size for a pair. To get an impression of the size of these problems we show for each generator the instances for size 1 and 2. The generators produce unifiable or non-unifiable pairs. The generator names that end with "f" produce non-unifiable pairs.

Generator	Size 1	Size 2
gen1	P(h(x1 x1) y2 x2) P(x2 h(y1 y1) y2)	P(h(x1 x1) h(x2 x2) y2 y3 x3) P(x2 x3 h(y1 y1) h(y2 y2) y3
gen1f	P(h(x1 x1) y2 aa) P(x2 h(y1 y1) y2)	P(h(x1 x1) h(x2 x2) y2 y3 aa) P(x2 x3 h(y1 y1) h(y2 y2) y3
gen3	P(x0 f(x1 x1) x1 f(x2 x2)) P(f(y0 y0) y0 f(y1 y1) y2)	P(x0 f(x1 x1) x1 f(x2 x2) x2 f(x3 x3)) P(f(y0 y0) y0 f(y1 y1) y1 f(y2 y2) y3)
gen3f	P(x0 f(x1 x1) x1 f(x2 x2)) P(f(y0 y0) y0 f(x0 x0) y2)	P(x0 f(x1 x1) x1 f(x2 x2) x2 f(x3 x3)) P(f(y0 y0) y0 f(y1 y1) y1 f(x0 x0) y3)
gen4	P(x1 y1) P(g(y1 y1) f(x2))	P(x1 y1 x2 y2) P(g(y1 y1) f(x2) g(y2 y2) f(x3))
gen4f	P(x1 y1) P(g(y1 y1) x1)	P(x1 y1 x2 y2) P(g(y1 y1) f(x2) g(y2 y2) x1)
gen2	P(x1) P(f(y))	P(x1 f(x2)) P(f(x2) f(f(y)))
gen2f	P(x1) P(f(x1))	P(x1 f(x2)) P(f(x2) f(f(x1)))