File: C:/bsd/rigel/sort/Unification/LinUnify2.docx
Date: 2020 September – 2021 September

Faster Linear Unification Algorithm

Dennis de Champeaux
ddc2 AT ontooo DOT com
2021

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 inputs. This paper describes yet another (semi) linear version that is faster and challenges also the Robinson version on small inputs. All versions have been implemented and compared against each other on different types 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 around1976-1978 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. However, that paper introduced a typo that was observed and corrected in 1991 by [Jacobson] and again in 2020 by [Motroi&Ciobaca]. The finally fixed version is available at [GitHub].

The earlier formalisms for describing the algorithms/ procedures are out of date. The 1986 description was close to the 1978 version when programming languages were not as expressive as they are these days. Procedure calls were actually functions. Our descriptions uses Java's OO-ness to clarify matters.

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 (BS) in [BaaderSnyder] 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 employed data structure. Both of them are available at [GitHub].

A novel, third algorithm 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 captures findings about the variable's matchings.

We report how these algorithms exploit their data structures effectively to tame the 'occur-check' that causes the exponential behavior of the Robinson algorithm. 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 naïve control structure. The two input arguments are directed acyclic graphs with at least unique nodes for the variables. A recursive descend 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 expression and both are committed earlier to functional expressions. This causes proceeding recursively 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.

These tactics are incorporated in the versions we will discuss in different ways. Helpful is the notion that a variable committed to an expression (constant or functional term) is the 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(x)*, *x* is the 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. We use an example. A generator produces for size 2 an example, which creates the ordered substitution:

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

The generation of a regular substitution requires doing something about the occurrence of *?x1* in *h(?x1 ?x1)* and about the occurrence of *?x2* in *h(?x2 ?x2).*

The substitution pursued in the Robinson version causes the *?x3* substitution to grow to: *?x3 -> h(h(?y1 ?y1) h(?y1 ?y1))*, while the *?y3* substitution grows to *?y3 -> h(h(?y1 ?y1) h(?y1 ?y1)).* This type of doubling gets worse for larger sizes.

An example trace

This pair is not unifiable:

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

Substitutions are assembled, but terminate with:

var: z -> f(x) & *subs: variables [x -> g(y) y -> h(z)]*

Combining the *x* and *y* substitutions gives: *x -> g(h(z))*; subsequently the *z -> f(x)* substitution yields the occurrence of *x* in *g(h(f(x))).*

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 on an I3-2310M /8GB machine the generator 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.

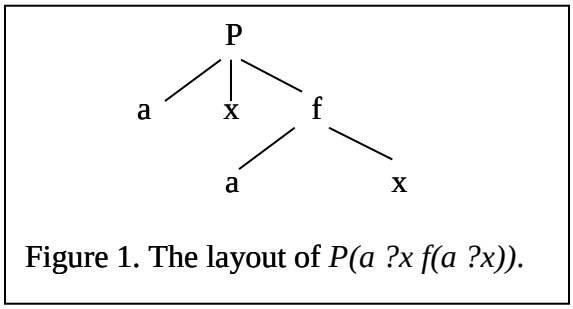

Figure 1. The layout of *P(a ?x f(a ?x))*.

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 terminals *a* and *x* are here *not* shared.

The Patterson-Wegman (PW) version

The preprocessor

This unification algorithm requires a richer infrastructure than the tree that is produced by the parser. Instead, a dag must be created for each argument where each node contains what is in the corresponding node of the input tree with additional fields. Multiple occurrences of a constant or a variable terminal *are* recognized and they are merged. Thus in the dag representation of *P(a x f(a x))* we have only the two terminals *a* and *x* is as in figure 2.

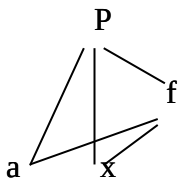

Figure 2. The dag representation of *P(a x f(a x))*

The generator for the dag uses the class hierarchy for the different type of nodes in a dag as in Figure 3.

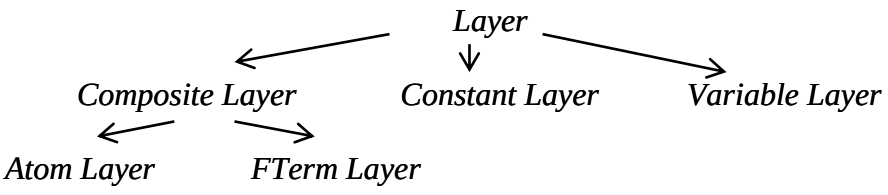

Figure 3. The class hierarchy of the Layer nodes.

The terminal classes in the *Layer* hierarchy inherit from *Layer*, among others, a *parents* attribute, which provides upward links in the dag representation. These upward links are crucial for the recognition of non-unifiable input pairs. The processing of two input arguments starts with creating a horizontal link between. Horozontal links are also created at run time to capture that encountered components of two functional expressions must match up. The following example shows how a cycle is found by following horizontal and vertical links.

This pair of inputs is not unifiable::

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

The summary of the trace shows how the next element is obtained from the preceding one by either following a 'horizontal' link, or following a parent reference; the numbers on the next line refer to the recursion depth of the `finish` function that does the work:

h(z) - link - *y* - parent - *g(y)* - link - *x* - parent - *f(x)* - link - *z* - parent - *h(z)*

1 1 2 2 3 3

A cycle is encountered between the first and last element in this sequence.

Reversing the order of the two input arguments produces the following sequence:

g(y) - link - *x* - parent - *f(x)* - link - *z* - parent - *h(z)* - link - *y* - parent - *g(y)*

1 1 2 2 3 3

Again a cycle is encountered between the first and last element in this sequence.

The code of the PW version is available at [GitHub].

Performance

We applied the PW version on the examples processed with the Robinson algorithm shown above and obtained the following timings in milliseconds on an I3 machine:

Size	Robinson Timing	PW Timing
15	81	0.45
16	103	0.5
17	483	0.5
18	1775	0.55
Table 1 Timings of the Robinson and PW versions.		

Correctness & linearity
Generic correctness proofs for unification algorithms are described in [Martelli&Montanari] and [BaaderSnyder].The key idea is to reinterpret the unification task as solving equations. The first horizontal link between the two input arguments creates the first equation. The algorithm resolves an equation when it encounters a link between, say, *a* and *a* and creates new equations following a link connecting two function expression with the same function names and their corresponding arguments, as well as a link between a variable and an expression. Correctness and termination is obtained. The transformations applied to a set of equations track the operations inside the `finish` function.

The linearity of the PW algorithm follows from the observation that the number of nodes visited is bounded by (the number of nodes) * (the max of the heights of the two input dags.

The Baader-Snyder (BS) version
The preprocessor
This unification algorithm requires similarly a richer infrastructure than the tree that is produced by the parser. Similarly, a dag must be created for each argument where each node contains what is in the corresponding node of the input tree with additional but different fields. Multiple occurrences of a constant or a variable terminal *are* recognized and again they are merged. The generator for the dag uses the class hierarchy for the different type of nodes in a dag in the figure 5.

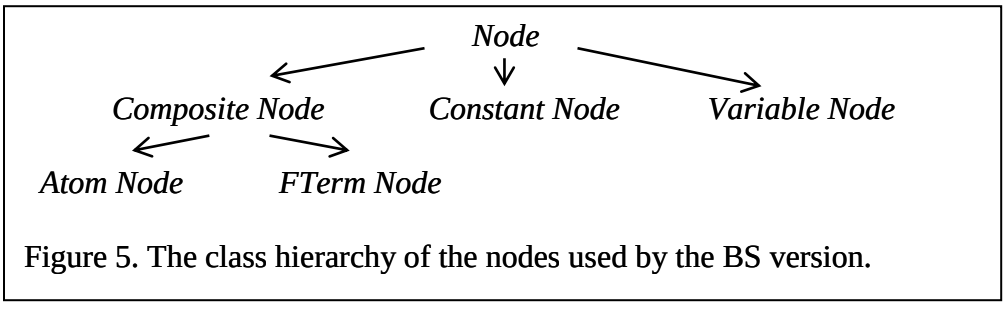

Figure 5. The class hierarchy of the nodes used by the BS version.

Again most of the attributes in the nodes needed for the unification algorithm are defined in the abstract class Node. Two attributes play a key role in the occur check:

- *myClass*
  This attribute is initialized to the node itself and denotes preliminary its equivalence class. When two nodes are compared in the union procedure the smaller one (using a *size* comparison) will change its attribute and refer to the other one.
- *schema:*
  This attribute is also initialized to the node itself; it does not change for non-variable nodes. If the size-wise selected node in the union procedure is a variable its schema reference will shift to the other node. Hence its schema value can become its substitution.

Occur-check example
We use again the not unifiable pair:
P(x h(z) f(x)) P(g(y) y z)
The `findSolution` function does a depth first, recursive descend:

P(x h(z) f(x)) is marked as visited and proceeds with the variable:
x which has the schema *g(y)* marked as visited and proceeds with the variable:
y which has the schema *h(z)* marked as visited and proceeds with the variable:
z which has the schema *f(x)* marked as visited and proceeds with the variable:
x which has the schema *g(y),* which is found as marked and hence a cycle is found.
Reversing the arguments gives the chain:
`g(y)- y - h(z) - z - f(x) - x - g(y)`

Correctness & linearity
We can refer again to the generic correctness proofs for unification algorithms are described in [Martelli&Montanari] and [BaaderSnyder].

The (semi) linearity of the BS algorithm is discussed in [BaaderSnyder] with: each function can be called at most *n* times for terms with *n* symbols, and each call performs a constant amount of work. The exception to linearity is the `union-find` combination, which 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].

Note
We noticed a glitch in our implementation of the BS algorithm, which we suspect is actually a defect in what we found in [BaaderSnyder]. We had to modify the code of their Union module. If one of the arguments is a variable is must be processed as the 1st argument in both sections of the Union module in order to update the class and schema attributes correctly; see the code in [GitHub].

The Unify DC version
No preprocessor
This unification 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 DC version exploits that object-oriented-ness of the input arguments. The Variable class has been extended with additional attributes to capture its different states:

- not (yet) matched with anything
- being a 'root', matched with a non-variable term – a potential substitution term
- matched with a variable, while still being a 'vroot'
- matched with a variable and referring to a variable that is either root or vroot

These states of a variable are updated when it is matched against another term; see below for the details.

The unification functions
Similar to the PW and BS versions unification happens inside a context. This context contains an instance of a `Hashtable` to keep track of 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 input pairs are processed. The `unify` function takes care of pre- and post-processing and invokes the function `dunify`.

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

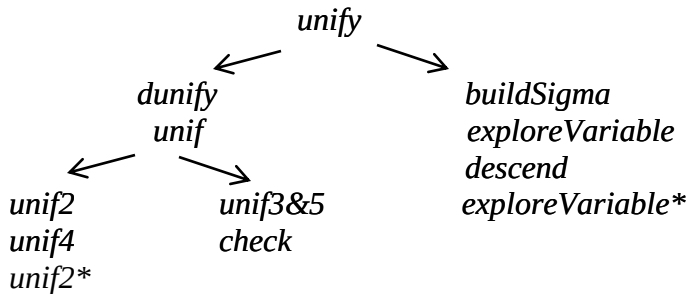

Figure 7. The control flow of the DC modules.

The right branch under `unify` takes care of post processing when unification succeeds. 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` has the reject tactics. The right branch `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. This focus obviates the need to create first the data structures for the PW and BS versions. Omitting details the following tables describe the functionality of `unif2`. The module `unif4` matches two arguments when both are non-variables. Table 2 gives the actions taken by checking whether arguments are variables or not.

Check arguments *s* and *t* whether both are variables			
	s	*t*	Action
1	Not a variable	Not a variable	Delegate to `unif4`
2	Is a variable	Not a variable	See table one variable
3	Not a variable	Is a variable	Like 2 with exchange *s* & *t*
4	Is a variable	Is a variable	See table with two variables
Table 2 Actions taken whether or not arguments are variables.			

A variable is an object with attributes. A variable has its *isRoot* set iff it has matched earlier with a non-variable term; this causes such a term to be set on the *first* attribute of the variable. Table 3 describes the consequences when *s* is a variable and t *is* not.

Argument *s* is a variable, *t* is not a variable		
	s	Action
1	s is not root	Fail if *s* occurs in *t*; otherwise *s.first* = *t*
2	s is root	Delegate *s.first* and *t* to `unif4`
Table 3 Checking whether *s* is root or not		

Matching two variables with the different combinations in which they are committed or not to a term is described in the next table 4. The attribute `isRoot` is true when the *first* attribute has been set. The table 4 shows that some occur checks can be done early as done in PW and unlike in BS. This functionality is here available because an *FTerm* 'knows' about its variables with a

`Hashset`. These occur checks are actually optional and have been disabled to not slow down handling unifiable pairs.

The arguments *s* and *t* are both variables and are different			
	s	*t*	Action
1	`isRoot`	`isRoot`	Change the *myVroot* of *s* or *t* and refer to *t* or *s*. Fail if *s* occurs in *t.first* or if *t* occurs in *s.first*; otherwise Delegate *s.first* and *t.first* to `unif4`
2	`isRoot`	not `isRoot`	Fail if *t* occurs in *s.first*; otherwise *t.myVroot* = *s*
3	not `isRoot`	`isRoot`	Like row 2
4	not `isRoot`	not `isRoot`	See the table with both variables not root
Table 4 Consequences when both arguments a variables. The occurs test are optional and have been disabled.			

When both variables have their *first* attribute not set there are still four possibilities whether or not they were matched earlier against another variable, which is captured in the table 5:

The arguments *s* and *t* are both variables and not root			
	s	*t*	Action
1	`isVroot`	`isVroot`	Change the *myVroot* of *s* or *t* and refer to *t* or *s*.
2	`isVroot`	not `isVroot`	Obtain the *Vroot* of *t* and refer it to *s*
3	not `isVroot`	`isVroot`	Like row 2
4	not `isVroot`	not `isVroot`	Obtain the *Vroot*s of both and one will refer to the other
Table 5 Consequences when both arguments are variables and both are not root.			

The member `unif4` in the left branch under `unif` does the usual structure reject tactics. The main occur check operation is done in `unif5` and `check`. Calling the `message` function is done when a cycle is encountered.

The `unif5` function feeds iteratively each found variable to the check function.

```
boolean unif5() {
   Enumeration enumx = htv.keys(); // get the variables
   while ( enumx.hasMoreElements() ) {
       Variable v = (Variable)enumx.nextElement();
       if ( !check(v) ) return message("unif5 ", v, v);
   }
   return true;
}
```

The `check` function does a depth first descend using the variable's associated attributes. Key is that there is no need to look inside the substitution term (if any) of the variable. A substitution term has an associated container (a `HashSet`) with its variables, if any.

```
private boolean check(Variable v) {
   // ZTest0.cnt++;
   if ( v.checked ) {
       return true;
   }
   if ( v.checking ) return message("check checking cycle", v, v);
   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("unif5 ", v, vnVroot);
        v.checking = false;
        v.checked = true;
        return true;
    } // v is root
    v.checking = true;
    HashSet variables = v.variables;
    if ( null == variables ) {
        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);
    }
    v.checking = false;
    v.checked = true;
    return true;
} // end check
```

Occur-check example
We use again 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)
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 & Linearity
Formal correctness we have to leave to those who have the proper tools. Our DC version uses the rejection tactics from the other versions. Each version handles encountered variables differently. PW creates a substitution for them with the current equivalent class (which can be anything) as its value. BS (in our fixed version of the Union module) will also make a reference to the equivalence class (called schema in their terminology). Our DC version makes similar adjustments as in BS with as difference that adjustments are registered in the variables object. An additional similarity is that a variable matching an unmatched 2nd variable can cause the 2nd

variable to refer to the first one, while a similar union-find-function applied to such 'lone' variables can shorten the path to the variable that is the equivalent class.

The (semi) linearity of the DC version is caused by the use of the union-find-function as done in the BS version.

The occur-checks are implemented distinctly differently. The PW version while recursing down in the input argument, also recurses upward along the parent links to check for a cycle. The BS function revisits an input argument, recurses downward, while marking FTerm nodes as visited; a variable encountered later which refers to the FTerm recognizes a cycle. The DC version, as described above, does a depth first search on the list of variables and recognizes cycles by marking a variable before descending in the variable and unmarking when returning. The check function is called as often as there are variables and their total effort is a function of their number. We consider this occur check the most transparent.

Post-processing

The PW, BS & DC version produce ordered substitutions, which need post-processing. A post-processor 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 versions differ because the different versions use different data structure. 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 binding of *?x1*, which can be large, can be used as-is yielding structure sharing.

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 milliseconds for executing the average time 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. Thus DC is also competitive on those small sized inputs.

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 and the use of the different data structures in DC made a positive difference.

Discussion, future work and conclusion
The unification algorithms described 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 checking 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. 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 capture 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 its class, which was crucial to reduce overheads. Object-Orientation has promoted conceptual architectures and designs. Newer data structures may still be hidden that will provide even better (unification) algorithms.

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 of a pair. To get an impression for 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)))