A brief Introduction to First-Order Logic Unification at the example of Corbin and Bidoit's variation of Robinson's Unification Algorithm

J. Sprinz

16.09.2021

Abstract

Automated reasoning and deduction have interested computer scientists and logicians since the earliest days of modern information technology. One of the fundamental processes of this field is unification, meaning the automated solving of equations containing symbolic expressions. This paper revisits the 1965 Robinson Unification algorithm, which is now recognized as being the first machine-oriented unification algorithm for first-order logic. The primary criticism to be brought against this algorithm is its exponential space complexity. Corbin and Bidoit proposed a revised version of the algorithm that aims to address this problem in 1983, which is discussed here as well.

Ludwig-Maximilians-Universität München
Master Seminar "Unification"
lead by Prof. Dr. François Bry and Dipl.-Ing. Thomas Prokosch
Summer term 2021

Motivation and Structure

Motivation

Unification algorithms enable automated solving of equations containing symbolic expressions. In computer science, the unification of first-order logic terms is one of the fundamental processes of automated reasoning and deduction. Practical applications include logic programming and programming language type systems.

Herbrand (1930) first established the theoretical groundwork for unification. However, the first known machine-oriented unification algorithm can be attributed to Robinson (1965). This prompted researchers to explore different approaches for more efficient unification algorithms and eliminate flaws in existing algorithms. This paper explores Corbin and Bidoit (1983)'s variation of the Robinson (1965) unification algorithm, demonstrating how the choice of data structure can significantly influence an algorithms' performance.

Structure

The subsequent sections are structured as follows. First, a brief introduction to the unification of first-order logic expressions is presented. Next, Robinson (1965)'s original Unification Algorithm is described. Particular attention will be paid to its exponential space complexity. Building on that, the enhancements proposed by Corbin and Bidoit (1983) are discussed, which use a different data structure to achieve better behavior. Finally, the paper closes by summarizing the observations.

Introduction

First-Order Logic, Terms

First-order logic expands propositional logic, such that the terms are predicates asserting relationships among elements of a set. The universal quantifier ($\forall$) and existential quantifier ($\exists$) operators describe whether a predicate should hold for all or just some elements of that set.

The definition of first-order logic **terms** used here is taken from Corbin and Bidoit (1983). Let F be a countable set of constants and function symbols with non-negative arity and V be a countable set of variables. The set T of valid first-order logic terms is then recursively defined by:

$$t \in T \text{ iff } t \in V \vee \exists f \in F, \exists t_1, ..., t_n \in T \text{ so that } t = f(t_1, ..., t_n)$$

In plain English: A term is either a variable or a function call, where every argument is itself a term (cf. Corbin and Bidoit 1983, 909).

Substitutions and Replacements

A **substitution** σ is a mapping $V \to T$. It can be described as a finite set of pairs of variables and terms called **replacements**. A substitution is written as $\sigma = \{[t_i \setminus x_i], ...\}$, which means that when applying the substitution σ to a term

t, the resulting term $\sigma(t)$ has all its instances of the term t_i replaced with the variable x_i (cf. Corbin and Bidoit 1983, 909).

Example

Let u, v, x, y be variables, f a binary function, $\sigma = \{[f(x,x) \setminus u], [y \setminus v]\}$ a substitution, and $t = f(y, f(x,x))$ a term. The substitution σ can then be applied to the term t as follows: $\sigma(t) = f(v, u)$.

Cascading substitutions

To emulate the behavior of applying multiple substitutions in sequence, they can be **cascaded** to a single one. The goal is to find for two substitutions σ_1, σ_2 a cascaded substitution $\sigma = \sigma_1 + \sigma_2$, so that $\forall t \in T : \sigma(t) = \sigma_2(\sigma_1(t))$. Cascaded substitutions are defined in set notations as follows:

$$\sigma_1 + \sigma_2 = \sigma = \{[s \setminus \sigma_2(r)] | [s \setminus r] \in \sigma_1\} \cup \{[s \setminus r] | [s \setminus r] \in \sigma_2 \wedge \forall [s_1 \setminus r_1] \in \sigma_1 : s \neq s_1\}$$

In plain English: The cascaded substitution of two substitutions contains all replacements from the first substitution with the second substitution applied to each replacement term, as well as all replacements from the second substitution that replace terms not replaced by the first substitution.

Example

Let u, v, w, x, y, z be variables, f an unary function, g a ternary function, and $\sigma_1 = \{[u \setminus x], [v \setminus y]\}$, $\sigma_2 = \{[y \setminus z], [f(w) \setminus x], [u \setminus z]\}$ be substitutions. Applying σ_2 to every replacement term in σ_1 results in the set $\{[u \setminus x], [v \setminus z]\} \subseteq \sigma$. Since u is already replaced, $\sigma_2 \ni [u \setminus z] \notin \sigma$. The full cascaded substitution σ is consequently $\sigma = \{[u \setminus x], [v \setminus z], [y \setminus z], [f(w) \setminus x]\}$.

Now let $t = g(u, v, g(f(w), y, f(u)))$ be a term. Applying the cascaded substitution to t results in the same term as applying both substitutions in sequence:

$\sigma_1(t) = g(x, y, g(f(w), y, f(x)))$

$\sigma_2(\sigma_1(t)) = \sigma_2(g(x, y, g(f(w), y, f(x)))) = g(x, z, g(x, z, f(x)))$

$\sigma(t) = g(x, z, g(x, z, f(x))) = \sigma_2(\sigma_1(t))$

Unification, Unifier, MGU

Unification is the process of determining whether or not two expressions can be unified, that is made identical by applying appropriate substitutions to their terms (cf. Robinson 1965, 31).

A substitution σ is called a **unifier** of two expressions φ_1 and φ_2 iff $\sigma(\varphi_1) = \sigma(\varphi_2)$ (cf. Corbin and Bidoit 1983, 909). Two expressions for which a unifier exists are called unifiable. A unifier σ_1 is as general as or more general than a unifier σ_2 iff $\exists \sigma_3 : \sigma_2 = \sigma_3(\sigma_1)$. If two terms are unifiable, there must exist a most general unifier (**MGU**) which is as general as or more general than every other unifier (cf. Corbin and Bidoit 1983, 909). The goal of unification algorithms is to find this MGU, if it exists.

Robinson's Unification Algorithm

The following is a recursive version of the original Robinson (1965) unification algorithm used by Corbin and Bidoit (1983).

Algorithm `ROBINSON_TREE` : `unify(t1, t2)`

Input	Output
Two first-order logic terms, `t1` and `t2`	A tuple (`unifiable`, `unifier`), where `unifiable` is a boolean value indicating whether or not the two terms can be unified. If `unifiable` is `true`, `unifier` is an MGU of `t1` and `t2`.

```
1  FUNCTION unify(t1, t2)
2  IF one of the terms, x, is a variable, let t be the other
3  THEN
4    IF x=t
5    THEN unifiable := true; unifier := {};
6    ELSE
7      IF the variable x occurs in the term t
8      THEN unifiable := false;
9      ELSE unifiable := true; unifier := {[t\x]};
10     FI
11   FI
12 ELSE let t1=f(x1, ... ,xn), t2=g(y1, ... ,ym);
13   IF f != g
14   THEN unifiable := false;
15   ELSE k:= 0; unifiable := true; unifier := {};
16     WHILE k < m AND unifiable
17     DO k := k+1;
18       (unifiable, subunifier) := unify(unifier(x[k], y[k]));
19       IF unifiable
20       THEN
21         unifier := subunifier + unifier;
22       FI
23     OD
24   FI
25 FI
26 RETURN (unifiable, unifier)
```

The `ROBINSON_TREE` algorithm consists of two main branches, the first of which - lines 2 through 11 - handles cases where one of the terms is a variable. If both terms are variables and identical, the terms are already unified, and an empty unifier is returned (line 5). Otherwise, the so-called *occurs-check* is performed, determining whether the variable occurs in the term (line 7). Only if the occurs-check returns false are the two input terms unifiable, and a substitution replacing

the term with the variable is added to the unifier (line 9).

The second main branch - lines 12 through 25 - addresses the case where both input terms are function calls. Calls to different functions are never unifiable and terminate the algorithm (line 13). Otherwise, a while-loop (line 16) iterates over each term's arguments. A recursive call to `unify()` is executed for every pair of arguments (line 18). The resulting substitution is cascaded with the unifier (line 21), as long as the arguments themselves were unifiable (line 19).

Tree- and Graph Representation

The data structure used by `ROBINSON_TREE` to represent the terms is a tree. For example, the two terms $t_1 = f(x, g(x, y))$ and $t_2 = f(g(y, z), g(g(h(u), y), h(u)))$ would be represented as shown in Figures 1 and 2.

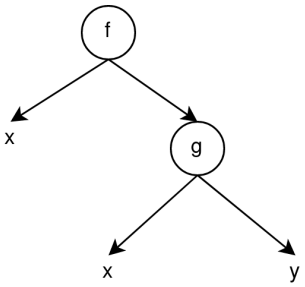

Figure 1: Tree representation of $t_1 = f(x, g(x, y))$ (cf. Corbin and Bidoit 1983, 910). Circled nodes represent function symbols, leaves represent variables.

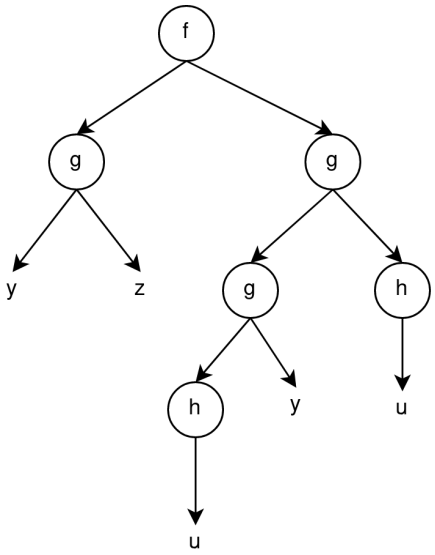

Figure 2: Tree representation of $t_2 = f(g(y, z), g(g(h(u), y), h(u)))$ (cf. Corbin and Bidoit 1983, 910). Circled nodes represent function symbols, leaves represent variables.

The problem with this approach is that it does not allow structure sharing across recursion steps (cf. Corbin and Bidoit 1983, 913; cf. Boyer and Moore 1972). Consequentially, new graphs have to be constructed and stored for every recursive call to `unify()` in line 18 of the `ROBINSON_TREE` algorithm. This is the primary reason for the observed exponential space complexity of the `ROBINSON_TREE` algorithm. Corbin and Bidoit (1983) address this problem by representing the terms not as two separate trees but as a single "finite directed acyclic graph" (Corbin and Bidoit 1983, 910). Figure 3 shows how the previously defined terms t_1 and t_2 can be represented in a single graph, illustrating the savings potential of this approach.

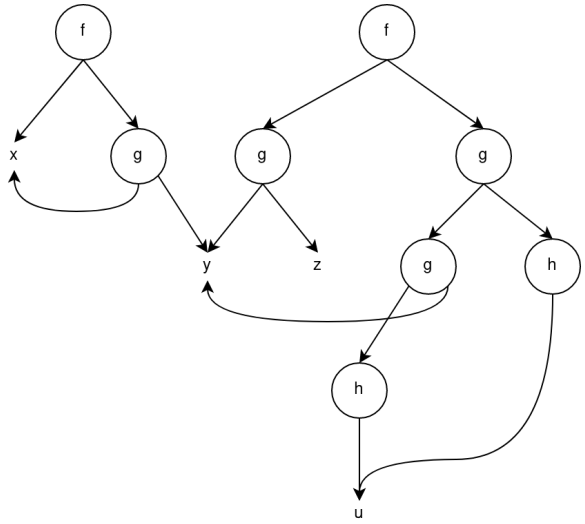

Figure 3: Graph representation of the previously defined terms t_1 and t_2 (cf. Corbin and Bidoit 1983, 912). Circled nodes represent function symbols, leaves represent variables. Observe how every variable node is stored only once.

Algorithm `ROBINSON_GRAPH` : `unify(v1, v2)`

The following is the final revised version of the original Robinson (1965) unification algorithm proposed by Corbin and Bidoit (1983) on page 912. The below pseudocode makes use of the following helper functions to aid readability:

- `replace(v1, v2)`: redirect all edges pointing to `v1` to `v2`, thus replacing `v1` with `v2` in the graph
- `term(v)`: return corresponding the term for a given vertex `v`
- `label(v)`: returns the function label for a given vertex `v`
- `successors(v)`: return the number of successors for a given vertex `v`
- `succ(v, k)`: return the `k`th successor node for a given vertex `v`

Input	Output
Two nodes `v1` and `v2` in the graph representing the two first-order logic terms to unify	A tuple (`unifiable`, `unifier`), where `unifiable` is a boolean value indicating whether or not the two terms can be unified. If `unifiable` is `true`, `unifier` is an MGU of `term(v1)` and `term(v2)`.

```
1  FUNCTION unify(v1, v2)
2  IF one of the vertices, v, is a variable node, let w be the other
3  THEN
4    IF the variable v occurs in the term w
5    THEN unifiable := false;
6    ELSE
7      unifiable := true;
8      unifier := {[term(w)\term(x)]};
9      replace(v,w);
10   FI
11 ELSE
12   IF label(v1) != label(v2)
13   THEN unifiable := false;
14   ELSE n := successors(v1); k := 0; unifiable := true; unifier := {};
15     WHILE k < n AND unifiable
16     DO k := k+1;
17       IF succ(v1, k) != succ(v2, k)
18       THEN (unifiable, subunifier) := unify(succ(v1, k), succ(v2, k));
19         IF unifiable THEN unifier := subunifier + unifier; FI
20       FI
21     OD
22     IF unifiable THEN replace(v1, v2); FI
23   FI
24 FI
25 RETURN (unifiable, unifier)
```

The `ROBINSON_GRAPH` algorithm is structurally very similar to `ROBINSON_TREE`. The main difference is that substitutions are also applied to the shared data structure using the `replace()` helper function (lines 9 and 22). Since variable nodes in the graph are always unique (see Figure 3), the equality of the variables does not have to be checked for in the first main branch. The second branch optimizes the number of recursion steps by skipping identical nodes (line 17). Due to the shared data structure, identical nodes imply that a substitution has already occurred in a previous recursion step.

Discussion

The advantage in space complexity gained through structure sharing in `ROBINSON_GRAPH` over redundant data copies in `ROBINSON_TREE` seems obvious. However, it should be noted that the real-world impact on computation time will depend heavily on how this data structure is implemented, as the

helper functions used to handle replacements on the graph might harbor significant inherent complexity. This behavior is influenced by the high-level implementation itself and the design of the chosen programming language and compiler. The platform the program will run on will further impact the result, as different operating systems will impose different constraints on memory management. Last but not least, modern processor features in representing complex data structures might further influence performance.

Corbin and Bidoit (1983) measured computation times of LISP implementations of different algorithms using the same input. The algorithms compared were the original Robinson (1965) algorithm, Corbin and Bidoit (1983)'s two proposed optimizations thereof, as well as the Martelli and Montanari (1982) algorithm, which represented a contemporary approach at the time. In this experiment, Corbin and Bidoit (1983)'s final proposed revision managed to outperform the other algorithms - hence their claim to "rehabilitate Robinson's algorithm" (Corbin and Bidoit 1983, 941) with their work. Repeating such an experiment with the technology available today might provide some insight into how almost four decades of advancements in fields like programming language theory and processor architecture impact these algorithms' real-world performance. Particular attention should be paid to how a modern low-level language like Rust[1] compares to a high-level language like Python[2]. This comparison might yield remarkable results, as both languages have seen significant development over the last decade but represent fundamentally different paradigms in modern programming language theory.

Conclusion

The goal of this paper was to provide a brief introduction to algorithmic first-order logic unification. First, the formal concepts of first-order logic have been introduced, and the unification process has been described. This served to prepare the reader to revisit the earliest machine-oriented Unification algorithm, published initially by Robinson (1965). The origins of it's exponential complexity have been explored by looking at optimizations proposed by Corbin and Bidoit (1983). Particular attention has been paid to their use of structure sharing, which was made possible by representing the data as a graph rather than a forest of trees. Finally, an experimental comparison of the discussed algorithms has been proposed to investigate the performance impact of recent advancements in different fields of computer science.

Even though no significant original work has been conducted in the context of this paper, it is provided in hopes that it helps readers new to first-order logic unification approach the topic.

[1] https://www.rust-lang.org/
[2] https://www.python.org/

Bibliography

Boyer, R. S., and J. S. Moore. 1972. "The Sharing of Structure in Theorem-Proving Programs." *Machine Intelligence* 7: 101–16.

Corbin, Jacques, and Michel Bidoit. 1983. "A REHABILITATION OF ROBINSON'S UNIFICATION ALGORITHM." In *Proceedings of the IFIP 9th World Computer Congress*, 909–14. Paris, France.

Herbrand, Jacques. 1930. "Recherches sur la théorie de la démonstration." PhD thesis, Paris, France: FACULTÉ DES SCIENCES DE PARIS.

Martelli, Alberto, and Ugo Montanari. 1982. "An Efficient Unification Algorithm." *ACM Transactions on Programming Languages and Systems* 4 (2): 258–82. https://doi.org/10.1145/357162.357169.

Robinson, J. A. 1965. "A Machine-Oriented Logic Based on the Resolution Principle." *Journal of the ACM* 12 (1): 23–41. https://doi.org/10.1145/321250.321253.