EPiC Series in Computing

Volume 40, 2016, Pages 37–51

IWIL-2015. 11th International Workshop on the Implementation of Logics

A Method to Simplify Expressions: Intuition and Preliminary Experimental Results

Baudouin Le Charlier and Mêton Mêton Atindehou

Université catholique de Louvain, ICTEAM
B–1348, Louvain-la-Neuve, Belgium
baudouin.lecharlier@uclouvain.be
meton.atindehou@uclouvain.be

Abstract

We present a method to simplify expressions in the context of a formal, axiomatically defined, theory. In this paper, equality axioms are typically used but the method is more generally applicable. The key idea of the method is to represent large, even infinite, sets of expressions[1] by means of a special data structure that allows us to apply axioms to the sets as a whole, not to single individual expressions. We then propose a bottom-up algorithm to finitely compute theories with a finite number of equivalence classes of equal terms. In that case, expressions can be simplified (i.e., minimized) in linear time by "folding" them on the computed representation of the theory. We demonstrate the method for boolean expressions with a small number of variables. Finally, we propose a "goal oriented" algorithm that computes only small parts of the underlying theory, in order to simplify a given particular expression. We show that the algorithm is able to simplify boolean expressions with many more variables but optimality cannot be certified anymore.

1 Introduction

Algorithms to simplify expressions often start by simplifying sub-expressions. Then they attempt to apply a number of simplification rules to the whole already partly simplified expression. Very often the simplification rules are restricted to rules that are guaranteed to produce a simpler (shorter) expression. This ensures that the simplification process is fast. However, in many situations it is necessary to first compute a more complicated expression in order to get a satisfactorily simplified one. For example, let us consider the boolean expression $a + ba$. Its sub-expressions are already simplified. To complete the simplification using basic axioms of the boolean calculus, we must write a sequence of equalities such as: $a + ba = 1a + ba = (1 + b)a = 1a = a$. At least the expression $1a + ba$ is more complicated than the initial one. And it is the key for the simplification.

In this paper, we propose an approach to simplification where we basically compute all expressions that are equivalent (i.e., equal with respect to a given theory) to an expression to be simplified. And we pick the simplest one (or possibly all simplest ones) at the end. The key idea for making that possible is to apply rules (i.e., axioms) to (large) sets of expressions instead

[1] We use the words "term" and "expression" as synonymous.

of single ones. To do so we introduce a data structure that allows us to compactly represent such sets of terms and we show how it can be used to compute (representations of) some theories in such a way that a given expression can be mapped to its equivalence class in linear time. So simplifiying the expression amounts to pick a simplest expression in the equivalence class. The approach is not applicable to all theories but it can be adapted to theories that are not finitely representable to derive a simplification algorithm that gives good results in some interesting situations.

The rest of this paper is organised as follows. In Section 2, we introduce the data structure that is used to represent sets of equal terms. This data structure can be related to the congruence closure method used in [8, 1] but it allows us to represent sets of terms in a much more compact way. In Section 3, we describe a bottom-up algorithm that computes (a representation of) the equivalence classes of terms that can be built from a set of equality axioms and a set of initially given terms. The algorithm (theoretically) terminates if and only if the number of these sets is finite. But the sets themselves can of course be infinite. We illustrate the functioning of the algorithm on a simple theory. In Section 4, we use the bottom-up algorithm together with a set of axioms for the boolean calculus to compute a complete representation of all boolean expressions using at most three variables. Based on this representation we show that any boolean expression can be simplified in linear time. It is also possible to use this representation to write down all minimal boolean expressions using three variables, according to various size notions. In Section 5, we propose a different but related algorithm to simplify expressions in the context of larger theories. Minimization cannot be guaranteed anymore. We show that the algorithm is able to simplify boolean expressions with many variables. In Section 6, we relate our work to the literature. Finally, Section 7 provides the conclusion and a list of extensions and improvements that we plan to make in the future.

All program runs presented in this paper are executed on a MacBook Pro 2.4GHz (Intel Core i5, 4Gb RAM) using Mac OS X 10.6.8. The programs are written in Java, and compiled and executed using the basic `javac` and `java` commands without any option. Timings are measured using the method `System.nanoTime()`.

2 Structures and Sets of Structures

To represent terms and sets of terms, we use "objects" that we simply call *structures*. A structure is of the form $f(i_1, i_2) : i$ where f is a function symbol, and i_1, i_2 and i are so-called *set of structures identifiers*. It is convenient to use natural numbers as identifiers and it is done so in the following and in our implementation but, from a "theoretical" standpoint, identifiers could be chosen from any infinite set I. We call $f(i_1, i_2)$, the *key* of the structure. The identifier i, is the identifier of the *set of structures* to which the structure belongs. Thus, at a given time, we consider a finite "collection" of structures that is partitioned in a finite number of sets of structures. For convenience, we only use binary keys and we "simulate" constant and unary function symbols by binary ones that are applied to the special identifier i_{null} which is the identifier of a conventional "dummy" set of structures. This can be related to the Currying and flattening method of [9] and to the transformation to directed graph of out-degree 2 of [4]. When we display structures, however, we use a simplified notation with constant and unary symbols.

The meaning of structures and sets of structures is defined as follows. Given n sets of structures $E_1, \ldots, E_n$, those sets denote together the smallest sets of terms $T_1, \ldots, T_n$ such that a term $f(t_1, t_2)$ belongs to T_i whenever E_i contains the structure $f(i_1, i_2) : i$ and t_1, t_2 belong to T_{i_1} and T_{i_2}, respectively.

As a simple example, let us consider the case of three structures partitioned into two sets of structures:

$$E_1 = \{f(1,2) : 1, a : 1\} \qquad E_2 = \{b : 2\}$$

These two sets of structures denote the sets of terms:

$$T_1 = \{a, f(a,b), f(f(a,b),b), \ldots, f(\ldots f(a,b)\ldots,b), \ldots\} \qquad T_2 = \{b\}$$

We observe that the set T_1 is infinite. The sets of structures E_1 and E_2 constitue what is computed by our method when it is given the equality $a = f(a,b)$ or, maybe more intriguingly, the two equalities $f(f(f(a,b),b),b) = a$ and $f(f(f(f(f(a,b),b),b),b),b) = a$.

2.1 Operations over Sets of Structures

There are two main operations over sets of structures: *toSet* and *unify*.

The operation *toSet* takes as input a term $f(t_1, t_2)$ and returns the identifier i of a set of structures to which the term belongs.[2] (More exactly the term belongs to the set of terms T_i denoted by E_i.) The operation first recursively computes the identifiers i_1 and i_2 corresponding to t_1 and t_2. Then there are two cases. Either a structure $f(i_1, i_2) : i$ already exists for some i. Then the identifier i is returned. Otherwise, a new set identifier i is chosen and a new set $E_i = \{f(i_1, i_2) : i\}$ is created. Finally, the identifier i is returned.

The operation *unify* puts two sets of structures together, assuming that the terms denoted by the two sets are all equal, and taking into account the fact that two terms that have equal corresponding subterms are equal as well (function congruence [8, 1]). It uses two sub-operations: *substitute* and *normalize*.

- The operation *substitute* takes as input two set of structures identifiers i and j. It removes, from all sets of structures, all structures that involve j (i.e., structures of one of the three forms $f(i_1, i_2) : j$ or $f(j, i_2) : i'$ or $f(i_1, j) : i'$, for some i_1, i_2, i') and it substitutes to them possibly new structures obtained by replacing j by i in the removed ones. The set E_j is then discarded.
- The operation *normalize* takes into account the fact that different structures with the same key can result from the operation *substitute*. Since these structures denote the same non empty set of terms, the sets of structures to which they belong must be recursively unified. Thus, the operation *normalize* collects all pairs of distinct structures with identical keys and apply the operation *unify* to the identifiers of the sets of structures to which they belong.
- The operation *unify* simply consists of executing *substitute* followed by *normalize*.

It is important to say that both operation *toSet* and *unify* can be implemented efficiently by means of adequate data structures: mainly, a hash table for keys and three doubly linked lists for the identifiers i_1, i_2, and i used by the structures. (In fact, there are three such lists for each set identifier.) However, we do not give the details of the implementation here.

Another important fact to note is that the operation *unify* always reduces the number of structures and the number of sets of structures that are "currently living", while, at the same time, it increases the set of all terms that are represented by the sets of structures. The more the sets of structures are reduced the more the sets of represented terms are increased. This is a key observation to understand the power of our method.

[2] This way of representing terms can be related to the term banks of [10].

3 Bottom-up Algorithm

We now describe, mainly by showing how it works on examples, a bottom-up algorithm that aims at computing a set of sets of structures that describes exactly the equivalence classes of terms that can be built from a set of equality axioms and a set of initially given terms. Thus, the algorithm, at the same time, builds a representation of all terms that can be constructed from the initially given terms and the function symbols used by the axioms, and classifies them into the equivalence classes determined by the axioms. The algorithm terminates if and only if there are only finitely many such equivalence classes. (Of course, in practice, it may fail to terminate because of lack of memory or it may take too much time.)

To fix ideas, we present first an example of an execution of the algorithm. Consider the following simple set of axioms:

```
x.x = x ;          x.y = y.x ;          (x.y).z = x.(y.z) ;
```

This set of axioms states that the binary operation `.` is idempotent, commutative and associative. The letters `x`, `y`, `z` are thus universally quantified *variables*. We simply use the last letters of the alphabet as variables. Others characters such as `a`, `b`, `f`, `0`, `!`, `+`, ... stand for constant and function symbols. Assuming that these axioms are put in a file called `Simple.txt`, let us consider the following run of the algorithm.

```
java BottomUpV3 Simple '(ab)'
================================================
Number of sets of structures : 3
Total number of structures : 11
Intermediate time              : 9.32E-4 sec
Number of created sets of structures : 12
================================================
Total time                      : 0.001659 sec
Number of created sets of structures : 12
================================================
?t
----------------------------------------------------------------
Set of structures no 1 [id = 1] [size = 1]  [tcS = 2]
----------------------------------------------------------------
Minimal term : a
----------------------------------------------------------------
.(1, 1):1 [size = 3] [key = 433592]
a:1 [size = 1] [key = 65]
Number of structures : 2
----------------------------------------------------------------
Set of structures no 2 [id = 2] [size = 1]  [tcS = 3]
----------------------------------------------------------------
Minimal term : b
----------------------------------------------------------------
.(2, 2):2 [size = 3] [key = 867170]
b:2 [size = 1] [key = 66]
Number of structures : 2
----------------------------------------------------------------
Set of structures no 3 [id = 3] [size = 3]  [tcS = 4]
----------------------------------------------------------------
Minimal term : ba
----------------------------------------------------------------
.(3, 2):3 [size = 5] [key = 300630]
```

```
.(2, 3):3 [size = 5] [key = 867297]
.(1, 3):3 [size = 5] [key = 433846]
.(3, 3):3 [size = 7] [key = 300757]
.(3, 1):3 [size = 5] [key = 300503]
.(2, 1):3 [size = 3] [key = 867043]
.(1, 2):3 [size = 3] [key = 433719]
Number of structures : 7
================================================
Number of sets of structures : 3
Total number of structures : 11
?s
>aaaaaaaaababababbbbbbbbbababababbbbbbbbbbbbbbbbbab
Simplified expression : ab
Simplification time : 1.01E-4 sec
================================================
```

- The first line launches the algorithm with the file of axioms `Simple.txt` and the initial term $a.b$. The program stops after 0.001659 seconds and displays that 12 sets of structures have been created during the whole execution.
- Then, the command `t` allows us to display the sets of structures that are obtained as the final result of the bottom-up algorithm. We see that there are only three different sets of structures representing the terms that are respectively equals to a, b, and $a.b$. The first set contains the two structures $.(1,1):1$ and $a:1$ because a term t is equal to a if and only if it is just a itself or if it is of the form $t_1.t_2$ where both t_1 and t_2 are equal to a. The third set contains seven different structures describing all possible forms of the terms that are equals to $a.b$. For instance, the structure $.(2,3):3$ stands for all terms $t_1.t_2$ such that t_1 is equal to b and t_2 is equal to $a.b$.
- Finally, the command `s` allows us to enter expressions to be simplified. We see that a rather long expression quickly simplifies to $a.b$. Note that the operation `.` is denoted by simple juxtaposition here. Moreover, we do not use parentheses since we assume that the operation associates to the left. The simplification is obtained by first applying the operation *toSet* to the introduced expression and by afterwards displaying a minimal term chosen among all those that are represented by the set of structures (whose identifier is) returned by *toSet*.

Let us now explain how the bottom-up algorithm works. Basically, it applies all axioms to all sets of structures by replacing variables by sets identifiers and unifying the sets corresponding to the left-hand and right-hand sides of the axioms. Axiom application uses a variant (generalization) of the *toSet* operation that takes as an additional argument a mapping from axiom variables to set of structures identifiers. To apply all axioms to all sets of structures, we simply generate all sequences of identifiers whose length is at most the maximum number of variables in an axiom, and, for each such sequence, we apply all axioms using the corresponding number of variables. Let us show how it works on the previous example. We display all generated sequences as well as all axiom applications.

```
java BottomUpV3 Simple '(a.b)' lx
1
[x<1>x<1>: 4  =  x<1>: 1] ==>
    aa = a  [BU: 1[a]]
1 1
[x<1>y<1>: 1  =  y<1>x<1>: 1] ==>
    aa = aa  [BU: 1[a]]
```

```
1 1 1
[x<1>y<1>z<1>: 1  =  x<1>(y<1>z<1>): 1] ==>
    aaa = a(aa)  [BU: 1[a]]
2
[x<2>x<2>: 4  =  x<2>: 2] ==>
    bb = b  [BU: 2[b]]
2 1
[x<2>y<1>: 4  =  y<1>x<2>: 3] ==>
    ba = ab  [BU: 3[ba]]
2 1 1
[x<2>y<1>z<1>: 4  =  x<2>(y<1>z<1>): 3] ==>
    baa = b(aa)  [BU: 3[ba]]
2 1 2
[x<2>y<1>z<2>: 4  =  x<2>(y<1>z<2>): 5] ==>
    bab = b(ab)  [BU: 4[bab]]
2 2
[x<2>y<2>: 2  =  y<2>x<2>: 2] ==>
    bb = bb  [BU: 2[b]]
2 2 2
[x<2>y<2>z<2>: 2  =  x<2>(y<2>z<2>): 2] ==>
    bbb = b(bb)  [BU: 2[b]]
3
[x<3>x<3>: 5  =  x<3>: 3] ==>
    (ba)(ba) = (ba)  [BU: 3[ba]]
3 1
[x<3>y<1>: 3  =  y<1>x<3>: 5] ==>
    (ba)a = a(ba)  [BU: 3[ba]]
3 1 1
[x<3>y<1>z<1>: 3  =  x<3>(y<1>z<1>): 3] ==>
    (ba)aa = (ba)(aa)  [BU: 3[ba]]
3 1 2
[x<3>y<1>z<2>: 4  =  x<3>(y<1>z<2>): 3] ==>
    (ba)ab = (ba)(ab)  [BU: 3[ba]]
3 1 3
[x<3>y<1>z<3>: 3  =  x<3>(y<1>z<3>): 3] ==>
    (ba)a(ba) = (ba)(a(ba))  [BU: 3[ba]]
3 2
[x<3>y<2>: 3  =  y<2>x<3>: 3] ==>
    (ba)b = b(ba)  [BU: 3[ba]]
3 2 2
[x<3>y<2>z<2>: 3  =  x<3>(y<2>z<2>): 3] ==>
    (ba)bb = (ba)(bb)  [BU: 3[ba]]
3 2 3
[x<3>y<2>z<3>: 3  =  x<3>(y<2>z<3>): 3] ==>
    (ba)b(ba) = (ba)(b(ba))  [BU: 3[ba]]
3 3
[x<3>y<3>: 3  =  y<3>x<3>: 3] ==>
    (ba)(ba) = (ba)(ba)  [BU: 3[ba]]
3 3 3
[x<3>y<3>z<3>: 3  =  x<3>(y<3>z<3>): 3] ==>
    (ba)(ba)(ba) = (ba)((ba)(ba))  [BU: 3[ba]]
================================================
Number of sets of structures : 3
```

```
Total number of structures : 11
Total time                      : 0.010333 sec
Number of created sets of structures : 12
================================================
```

We see that the first generated sequence simply is `1`. The axiom `x.x = x` is then applied to it. The axiom is depicted as `[x<1>x<1>:  4 = x<1>:  1]` to indicate that the set of structures identifier `1` is substituted to `x`: the operation *toSet* is applied to the term `x.x` where `x` is replaced by `1`. It creates a new set of structures $E_4 = \{.(1,1) : 4\}$, which is then unified with $E_1 = \{a : 1\}$. After unification the set E_4 is discarded and we get $E_1 = \{.(1,1) : 1, a : 1\}$. The next line `aa = a [BU: 1[a]]` provides additional information on the effect of the axiom: the terms `a.a` and `a` are now considered equal and they belong to the set T_1 of all terms represented by E_1. A minimal term of T_1 is `a`.

The following lines show that the commutativity and associativity axioms are now trivially satisfied by the terms in T_1: no modification is made to the current sets of structure. The next three lines are similar but afterwards various axioms involving both `a` and `b` are applied. Their application progressively fills the set E_3 with the six new structures depicted in the previous execution trace of the program. We encourage the reader to find out at which axiom application each final structure is exactly created. Finally, we observe that not all sequences of numbers involving `1`, `2`, and `3` have been generated. This is because they are not all necessary due to the commutativity and associativity of the `.` operation.

This first example is particularly simple because no new set of structures created by application of the axioms remains after considering the three initial sets of structures. The program stops because the current sets of structures definitely verify the axioms. Applying them again would not create any new structure. Of course this is not true in general. Here is another run of the program in the slightly more complicated case of three constants `a`, `b`, `c`. We display a different kind of information and we show only a small part of the trace.

```
java BottomUpV3 Simple '(a.b.c)' fi
================================================
Number of sets of structures : 15
Total number of structures : 50
Number of created sets of structures : 51
================================================
Normalize .(5, 4): [5 |--> 15]  [size = 7] : we have
    bac = bacc
  because
    bac = (b(ac))c [in 5]
  and
    bacc = (b(ac))c [in 15]
Normalize .(5, 5): [5 |--> 16]  [size = 11] : we have
    bac = bacbac
  because
    bac = (b(ac))(b(ac)) [in 5]
  and
    bacbac = (b(ac))(b(ac)) [in 16]
This sequence is no longer valid : 11 2 5
================================================
Number of sets of structures : 9
Total number of structures : 61
Number of created sets of structures : 80
```

```
================================================
This sequence is no longer valid : 12 2 4
================================================
Number of sets of structures : 7
Total number of structures : 52
Total time                      : 0.009743 sec
Number of created sets of structures : 83
================================================
```

Looking at this trace we can make the following observations.

- This time, the algorithm does not stop after applying the axioms to all initial sets of structures. It iterates three times. The first iteration creates 10 new sets of structures to which –roughly speaking– the axioms are applied at the second iteration. The second iteration reduces the number of sets but it increases the number of structures and even creates a few new sets. The last iteration finishes the work.
- The trace shows two cases where the operation *normalize* is needed: at some point of the execution two structures with the same key exist, namely $.(5,4) : 5$ and $.(5,4) : 15$. Therefore, the sets E_5 and E_{15} are unified. An example of a term represented by the structure is added as a "comment": `(b(ac)).c`. Since this term is equal to `(ba).c` in T_5 and to `((ba)c).c` in T_{15}, the unification of the two sets proves –in particular– that `(ba).c` = `((ba)c).c`.
- The fact that sets of structures are removed by the operation *unify* (and thus also by *normalize*) complicates the generation of all sequences of set of structures identifiers: an identifier may "disappear" while we are generating sequences using it. Two examples of this situation are shown in the trace. In such a case, we must be careful to continue with the appropriate "next" sequence. But we omit the details here.

A last important remark is the following. When we unify two sets of structures E_i and E_j, we must choose to keep one identifier and to remove the other one. Experiments show that it is of utmost importance to keep the older one. Consider the previous example. It takes 0.01 seconds to be executed and it creates 83 sets of structures (of which most are discarded afterwards by the operation *unify*, of course). If we choose to "put the older set into the more recent one", the execution takes 0.25 seconds and it creates 627 sets of structure. (Of course, the final sets of structures are equivalent.) In less simple situations such as those of the next section, the wrong choice is simply not usable and the program runs out of memory. We explain the difference as follows: when a recent set of structures is unified with an old one, it is often the case that the old set has already been involved in many –if not all– axiom applications using its identifier; thus, removing the recent set of structures makes it disappear without using it in any axiom application. With the opposite policy however there is a high probability that all axioms will be applied later on to the more recent set of structures, which is useless since this has been implicitly already done by unifying it with the older one.

4 Simplifying Boolean Expressions

In this section, we show how the bottom-up algorithm of the previous section behaves for simplifying boolean expressions. We use the following set of axioms for the "boolean calculus".

```
  x 0 = 0 ;        !!x = x ;              x + y = y + x ;
x + 0 = x ;         xy = yx ;        (x + y) + z = x + (y + z) ;
x + 1 = 1 ;        xyz = x(yz) ;          x + yz = (x + y)(x + z) ;
 x !x = 0 ;                               !(xy) = !x + !y ;
```

Assuming that the file `Boole.txt` contains exactly this set of axioms, let us consider the following run of our program.

```
java BottomUpV3 Boole '(abc)'
================================================
Number of sets of structures : 144
Total number of structures : 307
Intermediate time              : 0.00691 sec
Number of created sets of structures : 308
================================================
Number of sets of structures : 859
Total number of structures : 4642
Intermediate time              : 0.207848 sec
Number of created sets of structures : 6804
================================================
Number of sets of structures : 783
Total number of structures : 110052
Intermediate time              : 1.776162 sec
Number of created sets of structures : 216105
================================================
Number of sets of structures : 281
Total number of structures : 134053
Intermediate time              : 3.732329 sec
Number of created sets of structures : 338710
================================================
Number of sets of structures : 256
Total number of structures : 131333
Total time                      : 3.73564 sec
Number of created sets of structures : 338728
================================================
?s
>abc + ab!c + a!bc + a!b!c + !abc + !ab!c + !a!bc + !a!b!c
Simplified expression : 1
Simplification time : 5.8E-5 sec
================================================
>abc  + a!b!c + !abc + !a!bc + !a!b!c + !a!b!c
Simplified expression : !(ac + b) + bc
Simplification time : 5.1E-5 sec
================================================
>!((b+!b)!b!(c!(cc!a!a+!b))+(!b+a)(!c+c+bca)+b+c+!((cc+!b)!(a+!b)(a+!!b)+!b+a+a))+!ccb+
(a!c(c+a!!b+c)a+(b+!a)a!(cb!bb)((b+c)!!b(c+a)+c!b+a)(ca+c!c))!b(a+c)+(a!c+c+!b+!acbc)c+
(ba+cc+!c+!a+!(c+c))(!!(c+b+b+a)+c+!b)!(a(a+!a)!(!a+!!(aa)))(c+(a+b+bc+a+!b+ca+c)!a(!c+
!c+a)(!a+a)+!!((!c+ba(!c+b))!c)+!b+b+!c)(b+b)(b+c+c+c+c(b+c)+b+!((a+a)a+!b)+!c+!b+!(c+a
+!a)+(!(ba+!a!c)+ca+!c+(ac+c)!(a!a)+!bc(a+!c))!(!c+!!b)(a+!a+b+c+b+!c))+!(!(bb(b+a)(ac+
a!ba)b(!c+!(!(cb)bb))!c(a+!!(bc)))+!(!((!a!c+!((b+!c)b)(bb+bb+c))b!(b!c)(!a+c+ba)c(b+a+
(b+c)cb)b+(b+b+b+b+c+c+a+c+!c+b+(a+a)!c+b)(b+aba(baa+a!a!c((c+a)!b+c+b)+(!c+ca(b+c))cb)
)+!(!(a!b((!b+a!c)a+a))(!!b+b!(b(!(babb)+(ca+c+!b+!(cc))cc)(!c+a+a)(c!a+!c)(b+!a!c))+!(
a!ab)+b+!c)+(!(b!a+!!a)+a+c)bab(!c+!a)(a+bb+!b!b)!a(c+c!b)+c+!(!a+ac+!b)))+(bc(a+b)(b+c
)!ab(!b+aa)c+!!b+b)!a!((!c+c!a+a+c!a)b)(ca(b+!b)+b)ac(a!c+a)))
Simplified expression : b + c
Simplification time : 4.77E-4 sec
================================================
```

We observe the following facts.

- The program stops after less than 4 seconds. Only five "iterations" are needed to get the final sets of structures.
- We get 256 sets of structures and 131333 structures in the end. This is what should be expected. Indeed, there are exactly $2^{2^3} = 256$ boolean functions with three arguments. Hence, the number of equivalence classes of boolean expressions with three letters a, b, c is 256. Moreover, any boolean expression of the form $t_1 + t_2$ must be represented by a structure $+(i_1, i_2) : i$. Since there are 256 possible values for i_1 and i_2, there must exist $256 \times 256 = 65536$ such structures. Similarly, there must exist 65536 structures of the form $.(i_1, i_2) : i$ and 256 structures of the form $!(i_1) : i$. Finally, there must be 5 structures corresponding to the five constants $a, b, c, 0, 1$. It gives us a total of $65536+65536+256+5 = 133133$ structures.
- Having computed a complete representation of all boolean expressions not containing other letters than a, b, and c, we can use it to simplify any such expression. Three examples are shown in the trace. The simplified expressions are chosen by minimizing the size of a tree representation of the expression (or, equivalently, the number of characters of the expression written in polish notation). The timings show that the simplification is fast. They are also consistent with our claim that it is done in linear time.

5 Goal Oriented Algorithm

The bottom-up algorithm described in Section 3 is not applicable to "large theories" with many equivalent classes of equal terms. We now propose a different algorithm, which is not optimal anymore but which allows us to simplify expressions with more symbols. This algorithm is "goal oriented" because it is "driven" by an initial expression to be simplified and, afterwards, by the intermediate simplifications of this initial one. Let us show a first run of this algorithm (for simplifying a boolean expression).

```
java GoalOriented
Enter an expression to be simplified : :
(b(e+f)+ca+!b+b+b+!b)(!a+d+!a)(dd+c)c!df
----------------------------------------------------------------
Current reduced term : (b(e + f) + ac + !b + b + b + !b)(!a + d + !a)(c + d)c!df
----------------------------------------------------------------
Current reduced term : 1(!a + d + !a)(c + d)c!df [size = 20]
----------------------------------------------------------------
Current reduced term : (!ac + d)c!df [size = 13]
----------------------------------------------------------------
Current reduced term : !d(!ac + d)f [size = 11]
----------------------------------------------------------------
Current reduced term : f!(d + a)(c + d) [size = 10]
----------------------------------------------------------------
Current reduced term : !(d + a)cf [size = 8]
================================================================
Number of sets of structures : 10
Total number of structures : 156
Number of created sets of structures : 69808
Intermediate time : 2.298233 sec
================================================================
```

The algorithm simplifies an expression of size 40 to an expression of size 8 in 2.3 seconds. The initial expression uses 6 different letters but the simplified one uses only four of them. Since it uses each letter and the operator ! only once, we can guess that it is minimal.

The algorithm works as follows. It basically proceeds like the bottom-up algorithm by generating sequences of set of structures identifiers but it performs an axiom application only when the left hand-side of the axiom returns an identifier of a sub-expression of the current expression to be simplified. So it only keeps sets of structures that are "relevant" for simplifying the expression "at hand". It also maintains the minimal size of all expressions represented by the set of structures "containing" the current expression to be simplified. As soon as this minimal size decreases, all sets containing a structure corresponding to an expression of minimal size are marked. Then the execution of the algorithm resumes only considering identifiers of the marked sets. Using this strategy, the algorithm really concentrates on the current smallest expressions. The less interesting (non minimal) sets of structures are not immediately removed. They can become minimal later. But, of course, it may quickly happen that too much memory is used. In that case, the structures of the non minimal sets are freed first.

Here is another, more difficult example, with an expression of size 800 using 15 letters. We only show a small part of the trace.

```
java GoalOriented
Enter an expression to be simplified : :
(ei!b!!(a+!m)+lf+!!ialm+l)(!!(!((!k(l+c)!d+!a)(e+m!h)(!g+!j)(l+b)!hj)+k!i+!(fo)j(j+o+l)
(o+!i)+!(h+e!d)+!((!e(k+jj)+e+!i+!e+d(j+l))nh)e+((!g+h)o+(e+o!n+f+!e)!l((b+k)(!!g+!n)+
!h)(o+!d!a)+m+(l+!b)k+!l)g!n!km!!((a+b)(eg+f))+(!l!j+!(!jk+!h+!e))!(!bb(m+!b))(!b(!k+m
+b+!f)+!a))+!(!i!((f+eg)!!di+!d))!g(a!j+n)(!b+a+!d)!(!(!o+(!d+!f)(!!fk!a+h))+o)o(e!nbj
+f+fl)(!(ch+!f+g+!k!i)+!!n+b)(e+i)a!e(!(((!oi(!!h+k)+!!o)(hd+o+l+f)+n!l)!(!d+m+o)(!((a
h+(!l+!h)(g+n!f))dk)+f+!jd+!(k+a)+(h+!f)fe+i+!b))+(!i!b(n+i)+(d+i+ad)(g+(a+j)(c+c)))!(
(!m+!c+a+n!h(!c(j+o+h)+do(i+l)!!g!j!djj(j+e)))je(!(hbd)+!a!k+og)))+!ol!!nj(!(!c+ei(a!j
!k+!i+j))(!!!d+!l)i+!((b+hd)j))(!f+!!o(m+!l+!f)+a(o+g+o)+jc!m)+cl!d(!i+he+!(ja))(e+k)+
!((!(ejnk)+m+f)i!f!lgm(f+a+m)(!oeme+!k)!l(f(m+d)o+lh))+!(!a(f(!d+c!l)+(!b+ch)(f+m))+((
n+!f)!b+!!l+e+!c)d!k+n!j+a+!(!l+c)+!!e+(h+l)e!f)+k!f(!k+j+(h+!m)!l))
----------------------------------------------------------------------
Current reduced term : (ei!b!!(a + !m) + lf + !!ialm + l)(!!(!((!k(l + c)!d + !a)(e + m
!h)(!g + !j)(l + b)!hj) + k!i + !(fo)j(j + o + l)(o + !i) + !(h + e!d) + !((!e(k + jj)
+ e + !i + !e + d(j + l))nh)e + ((!g + h)o + (e + o!n + f + !e)!l((b + k)(!!g + !n) + !
h)(o + !d!a) + m + (l + !b)k + !l)g!n!km!!((b + a)(eg + f)) + (!l!j + !(!jk + !h + !e))
!(0(m + !b))(!b(!k + m + b + !f) + !a)) + !(!i!((f + eg)!!di + !d))!g(a!j + n)(!b + a +
 !d)!(!(!o + (!d + !f)(!!fk!a + h)) + o)o(e!nbj + f + fl)(!(ch + !f + g + !k!i) + !!n +
 b)(e + i)a!e(!(((!oi(!!h + k) + !!o)(hd + o + l + f) + n!l)!(!d + m + o)(!((ah + (!l +
 !h)(g + n!f))dk) + f + !jd + !(k + a) + (h + !f)fe + i + !b)) + (!i!b(n + i) + (d + i
+ ad)(g + (a + j)(c + c)))!((!m + !c + a + n!h(!c(j + o + h) + do(i + l)!!g!j!djj(j + e
)))je(!(hbd) + !a!k + og))) + !ol!!nj(!(!c + ei(a!j!k + !i + j))(!!!d + !l)i + !((b + h
d)j))(!f + !!o(m + !l + !f) + a(o + g + o) + jc!m) + cl!d(!i + he + !(ja))(e + k) + !((
!(ejnk) + m + f)i!f!lgm(f + a + m)(!oeme + !k)!l(f(m + d)o + lh)) + !(!a(f(!d + c!l) +
(!b + ch)(f + m)) + ((n + !f)!b + !!l + e + !c)d!k + n!j + a + !(!l + c) + !!e + (h + l
)e!f) + k!f(!k + j + (h + !m)!l)) [size = 797]
...
?st no
?run
...
size = 633 idList.size() = 318
size = 629 idList.size() = 316
======================================================================
```

```
Number of sets of structures : 315
Total number of structures : 6387
Number of created sets of structures : 2970747
Intermediate time : 37.291304 sec
================================================================
size = 605 idList.size() = 305
size = 604 idList.size() = 305
...
size = 346 idList.size() = 185
size = 21 idList.size() = 794
size = 13 idList.size() = 165
================================================================
Number of sets of structures : 260
Total number of structures : 6731
Number of created sets of structures : 10792179
Intermediate time : 85.492769 sec
================================================================
Iteration no : 2
?st nice
Current reduced term : l + (a + !m)i!be [size = 13]
?q
```

We see that the execution is not "fully automatic". The user may enter commands to guide it. The command `st no` tells the program not to display each intermediate simplified expression. The command `run` tells the program to apply the axioms without asking the user until all marked sets have been considered. At this time, a new iteration starts because some axioms that were not applicable at the first iteration may then become applicable. The algorithm may iterate many times and it may not terminate because, due to the finite amount of memory, some sets of structures can be removed and afterwards replaced by different ones so that some axioms remain applicable indefinitely. Hence, the more practical choice is to let the user decide whether the expression is simplified or not. In this example, the expression looks simple enough. Thus, we have stopped the program with the command `q`. Some other commands are available, notably to monitor memory usage, but we do not give more details here. Notice finally that the size of the expression has suddenly "jumped" from 346 to 13. Probably because a large sub-expression was found equal to 0 or to 1.

6 Related Work

The work presented in this paper can be related to the method of G .Nelson and D. C. Oppen proposed in [8] (see also [1], Chapter 9). They build upon the well-kown algorithm of M. J. Fischer and B. A. Galler [6] (see also [7], pages 353, 360–361, and [12]) to compute congruence classes of terms, as a tool to determine the satisfiability of a conjunction of literals. Our method, which has been designed independently, allows us to represent sets of terms in a much compact way because their method represents sets of equivalent terms by directed acyclic graphs (DAGs) while our sets of structures can be viewed as cyclic graphs. In particular, their method only permits them to handle finite sets of terms. Therefore, it is not possible, for example, to use their method to implement the bottom-up algorithm that we have presented in Section 3. On the contrary, our method can be used to solve the decision problem described in [8], and, in fact, more efficiently than they do. To support our claim, we have implemented an algorithm that "solves" an arbitrary number of equations between uninterpreted ground terms. The algorithm amounts to apply the operation *unify* to the list of pairs $i_1 = j_1, \ldots, i_n = j_n$ where the i_k

and j_k are identifiers of sets of structures representing the terms in the equations. Since the operation *unify* does not maintain the initially given terms, we have to use an additional data structure to map the initial terms to their corresponding sets of structures. This can be done with a simple array as in [6]. The resulting algorithm is similar to the algorithm presented in [4] and probably even more efficient since it works on more compact structures. It is easy to show that in the worst case its execution time is bounded by $O(m \log m)$ where m is the number of structures needed to represent the initially given terms. However, it seems to behave better in practice as shown by some experiments on which we report below.

First, we have "randomly" chosen a large term of 100000 symbols (i.e., 100000 symbols if the term is written in polish notation).Then, we have represented this term as a collection of structures, using the operation *toSet* (see Subsection 2.1). This representation uses 36901 structures, each of them belonging to a different set of structures. Using the identifiers of these sets we have generated a sequence of 36901 pairs of identifiers to which the operation `unify` has been applied, one by one. In the first experiment, each identifier is used exactly twice in the equations and exactly once in the first 18450 equations (except one). The results of this experiment are depicted in the first table below. Column i provides the number of equations already "solved" at a given line. Column t provides the amount of time needed to solve these equations, in seconds. Columns $\sharp Set$ and $\sharp Struct$ respectively indicate the number of sets of structures and the number of structures existing at that stage. Columns U and N contain the numbers of set of structures unification already done, and resulting directly from applying *unify* to a pair of sets of structures identifiers (U), or indirectly from using *normalize* (N). (See again Subsection 2.1.) Column T is just $U + N$. Column t/i gives the ratio of the time (measured in microseconds) by the number of equations already solved. Column dt/di is a kind of "derivative" of the time with respect to the number of already solved equations: we divide the difference between the current time and the previous one, by the number of equations executed between the current and the previous stage. Column T/t gives the number of set reductions by millisecond and dT/dt is the "derivative" of T with respect to t, computed similarly to dt/di. Finally, dS/dt is the "derivative" of the number of removed structures ($36901 - \sharp Struct$) with respect to the time.

i	t	$\sharp Set$	$\sharp Struct$	U	N	T	t/i	dt/di	T/t	dT/dt	dS/dt
0	0.0	36901	36901	0	0	0			0	0	0
2500	0.042	34336	36836	2500	65	2565	16.8	16.8	60	60	1
5000	0.065	31758	36758	5000	143	5143	13.1	9.3	78	109	3
7500	0.079	29166	36666	7500	235	7735	10.6	5.5	97	186	6
10000	0.096	26565	36565	10000	336	10336	9.6	6.7	107	153	5
12500	0.117	23925	36425	12500	476	12976	9.4	8.4	110	125	6
15000	0.13	21301	36301	15000	600	15600	8.6	5.1	119	202	9
17500	0.148	18615	36115	17500	786	18286	8.4	7.2	123	148	10
20000	0.162	15803	35802	19999	1099	21098	8.1	5.3	130	210	23
22500	0.482	5	48	21919	14977	36896	21.4	128.1	76	49	111
25000	0.482	2	30	21921	14978	36899	19.3	0.2	76	5	34
36901	0.483	2	30	21921	14978	36899	13.1	0.0	76	0	0

We can make the following observations: until $i = 20000$ (i.e. shortly after that set of structures identifiers have all been used in one call to *unify*), the algorithm behaves uniformly by reducing the number of sets while the number of structures remain almost stable. Also, the time complexity is better than linear. Then, suddenly, between $i = 20000$ and $i = 22500$, a lot of set reductions are made, mainly due to the operation *normalize*. This happens because most

sets of structures now contain more structures, increasing the probability of having different structures with the same key. Unifying the sets containing those structures creates new structures with the same key, which has a snowball effect. By looking to the third last column, we can see that the number of set unifications by unit of time is continuously increasing until the critical interval between $i = 20000$ and $i = 22500$ is met. Then, it decreases significantly but it is because most structures are discarded, (i.e., merged) at this stage. When $i > 22500$ almost nothing is left to do, so it is not useful to comment on the two last lines. Hence, our algorithm is very efficient to solve the satisfiability problem of [8]. We conclude by showing the results of a second experiment in which the list of equations $i_1 = j_1, \ldots, i_n = j_n$ is generated completely randomly, allowing every identifier to appear in the list an arbitrary number of times and in any position. We see that the results are similar but the "critical section" of the algorithm takes place earlier (which could have been anticipated). Also most set reductions are due to the operation *normalize* contrary to the first experiment. More equations are now trivially verified because the two identifiers involved in them at the beginning are mapped on the same one.

i	t	$\sharp Set$	$\sharp Struct$	U	N	T	t/i	dt/di	T/t	dT/dt	dS/dt
0	0.0	36901	36901	0	0	0			0	0	0
2500	0.032	34339	36839	2500	62	2562	13.0	13.0	78	78	1
5000	0.055	31740	36740	5000	161	5161	11.0	9.0	93	115	4
7500	0.074	29122	36622	7500	279	7779	9.9	7.9	103	132	5
10000	0.096	26463	36463	10000	438	10438	9.6	8.6	108	122	7
12500	0.116	23613	36113	12500	788	13288	9.3	7.8	114	144	17
15000	0.314	860	2128	14296	21745	36041	20.9	79.1	114	114	171
17500	0.316	422	1295	14386	22093	36479	18.0	0.8	115	205	391
20000	0.317	296	1041	14432	22173	36605	15.8	0.5	115	97	196
22500	0.319	163	678	14471	22267	36738	14.1	0.5	115	97	265
25000	0.32	73	384	14490	22338	36828	12.8	0.5	114	71	233
36901	0.323	12	140	14510	22379	36889	8.7	0.3	113	16	66

Finally, it can be stressed that the timings reported here are consistent with the timings reported for our bottom-up and goal oriented algorithms, which create much more structures and solve many more equations.

A lot of work has been devoted to the problem of simplifying boolean expressions, but most of the work has been done to simplify expressions written in disjunctive or in conjunctive normal form (see, e.g., [2]). The algorithms presented in this paper are not intended to compete with those methods but they are more general since they are applicable to many kinds of simplification problems. Basically, we used the boolean expression simplification problem mainly as a (significant) example of application. Very often, boolean expression simplification is used to better understand facts represented by the boolean expressions. The use of OBDDs to simplify boolean expressions is often advocated in that case (see, e.g. [3]). We have applied our method to analyze the so-called guards of a medical process model constructed by the authors of [3], with good results; but we have not compared our results with the use of OBDDs, yet.

Finally, parts of our work can be related to other areas such as rewriting systems, constraint programming or SAT-solving, to name only three, but we have not investigated those relations in great details yet. We will surely do so in the future, mainly to improve the efficiency and the applicability of our goal oriented algorithm.

7 Conclusion and Future Work

We have presented a method to represent large sets of equivalent terms compactly, and we have shown how this representation can be used to solve interesting simplification problems. We have put the focus on boolean expression minimization and simplification but our method obviously is much more general. Therefore, we plan to use it to investigate other simplification problems such as the (very difficult) problem of simplifying regular expressions (see, e.g., [11]).

The algorithms that we have presented in Sections 3 and 5 can certainly be significantly improved, especially the goal oriented algorithm. We plan to improve them by using incremental techniques related to the Rete algorithm [5]. Several other avenues of research can be considered. Let us consider two of them. We can extend and improve our current syntax for writing axioms. Our simple language can be augmented with "meta predicates" to write more specific axioms, and to allow the writing of implications. A second (difficult) topic should be to specialize our main data structure (sets of structures) to take into account common properties of operations such as associativity and commutativity.

Acknowledgments

The authors wish to thank Charles Pecheur for useful discussions and for providing them with pointers to important related work. Comments from the reviewers are also greatly acknowledged.

References

[1] Aaron R. Bradley and Zohar Manna. *The calculus of computation - decision procedures with applications to verification*. Springer, 2007.

[2] Olivier Coudert. Two-level logic minimization: an overview. *Integration*, 17(2):97–140, 1994.

[3] Christophe Damas, Bernard Lambeau, and Axel van Lamsweerde. Generating process models in multi-view environments. In *Dependable Software Systems Engineering*, pages 105–127. 2015.

[4] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, October 1980.

[5] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[6] Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, 1964.

[7] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[8] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[9] Robert Nieuwenhuis and Albert Oliveras. Proof-Producing Congruence Closure. In Giesl J, editor, *Proc. 16th International Conference on Rewriting Techniques and Applications (RTA-2004), Nara, Japan*, number 3467 in LNCS. Springer, 2005.

[10] Stephan Schulz. System Description: E 1.8. In *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

[11] Alley Stoughton. *Formal Language Theory: Integrating Experimentation and Proof*. Cambridge University Press, 2016.

[12] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.