EPiC Series in Computing

Volume 39, 2016, Pages 81–94

SCSS 2016. 7th International Symposium on Symbolic Computation in Software Science

EPiC Computing

A Data Structure to Handle Large Sets of Equal Terms

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 data structure to represent and manipulate large sets of (equal) terms (or expressions). Our initial and main motivation for this data structure is the simplification of expressions with respect to a formal theory, typically, an equational one. However, it happens that the data structure is also efficient to compute the congruence closure of a relation over a set of terms. We provide an abstract definition of the data structure, including a precise semantics, and we explain how to implement it efficiently. We prove the correctness of the proposed algorithms, with a complexity analysis and experimental results. We compare these algorithms with previous algorithms to compute the congruence closure and we also sketch how we use the data structure to tackle the expression simplification problem.

1 Introduction

The original motivation for the work presented in this paper is to find a good general method to simplify expressions. An *a priori* naive and inefficient approach to this problem is to generate the set of all expressions that are equal to an expression to be simplified, and to pick a simplest one in this set. In this paper, we introduce a seemingly novel data structure that allows us to represent very large –often infinite– sets of expressions (or terms) that are equal according to a formal theory, typically an equational one. This data structure makes it possible to successfully apply the naive idea described above, in a number of interesting cases.

The rest of this paper is organised as follows. In Section 2, we present a few preliminary notions about terms and sets of terms. In Section 3, we introduce our data structure, called *collection of structures*, as well as its associated operations. Collections of structures represent sets of terms endowed with a binary relation. Their precise meaning as well as the meaning of the associated operations is expressed in terms of sets of terms and of operations over sets of terms. In Section 4, we describe an efficient implementation of the data structure abstractly defined in the previous section. We pay attention to justify the correctness of the algorithms. Section 5 provides both a complexity analysis and an experimental evaluation of the algorithms of Section 4. Section 6 summarizes how we use the data structure to simplify expressions. Section 7 discusses some related work while Section 8 contains the conclusion and suggests future work.

2 Terms and Sets of Terms

A *term* is either a *variable*, a *constant*, or a *compound term* consisting of a *function symbol* and a non empty list of simpler terms, called its *direct sub-terms.* A *sub-term* of a term is either itself or any sub-term of its direct sub-terms. We assume given an (implicit) set of *all terms.* A *ground term* is a term not containing variables. In this section, we only consider ground terms. Furthermore, we only use *binary terms*, of the form $f(t_1, t_2)$, and a single "dummy" constant *null.* A regular constant a is represented by the binary term $a(\mathit{null}, \mathit{null})$, and a unary term $h(t)$ by the binary term $h(t, \mathit{null})$. Terms with more than two sub-terms can be represented by using a special binary function symbol to build lists of arguments. These conventions allow us to simplify algorithms and proofs as well as the implementation. Nevertheless, we stick to standard notations, such as a and $h(f(a, b))$, for writing examples.

Let T be a set of terms and ρ a binary relation over T. We say that T is *sub-term complete* if every sub-term of a term belonging to T also belongs to T. All sets of terms considered in this paper are sub-term complete. We say that $\langle T, \rho\rangle$ is *function complete* if the following two implications hold for all terms $s_1, s_2, t_1, t_2 \in T$, and any function symbol f:

$$\left.\begin{array}{c} s_i \;\rho\; t_i \;\; (i = 1, 2) \\ f(s_1, s_2),\;\; f(t_1, t_2) \in T \end{array}\right\} \Longrightarrow f(s_1, s_2) \;\rho\; f(t_1, t_2)$$

$$\left.\begin{array}{c} s_i \;\rho\; t_i \;\; (i = 1, 2) \\ f(s_1, s_2) \in T \end{array}\right\} \Longrightarrow f(t_1, t_2) \in T$$

Moreover, we say that $\langle T, \rho\rangle$ is a *congruence* when it is function complete and ρ is an equivalence relation. In that case, we often use the symbol $\sim$ to denote the relation (instead of ρ) and we say that $\sim$ is a *congruence over T.*

Let $(T_i)_{i\in I}$ be a family of sets of terms. We say that $(T_i)_{i\in I}$ is a *presentation* of $\langle T, \rho\rangle$ if the two conditions below hold:

$$T = \bigcup_{i\in I} T_i \qquad\qquad s \;\rho\; t \iff \exists i \in I : s, t \in T_i \qquad (\forall s, t \in T)$$

A pair $\langle T, \rho\rangle$ has a presentation if and only if the relation ρ is reflexive and symmetric. We always assume it in the following. It is easy to see that the relation ρ is an equivalence relation if $(T_i)_{i\in I}$ is a *presentation* of $\langle T, \rho\rangle$ and $\{T_i \mid i \in I\}$ is a partition of T. (Since we assume that the relation ρ is reflexive, the set T is equal to the set of all terms that occur in at least one pair of ρ. So, the notation $\langle T, \rho\rangle$ is somewhat redundant, but we stick to it for clarity.)

We denote by $\cong_\rho$, the smallest equivalence relation over the set of *all terms*, such that, for all terms s, t, $f(s_1, s_2)$, $f(t_1, t_2)$, the following implications hold:

$$\begin{array}{rcc} s \;\rho\; t & \Longrightarrow & s \;\cong_\rho\; t \\ s_i \;\cong_\rho\; t_i \quad (i = 1, 2) & \Longrightarrow & f(s_1, s_2) \;\cong_\rho\; f(t_1, t_2) \end{array}$$

The relation $\cong_\rho$ is a congruence over the set of all terms and it is clear that, for any relations ρ and ρ', we have: $\rho \subseteq \rho' \implies \cong_\rho \subseteq \cong_{\rho'}$. Moreover, the following equality holds: $\cong_{\cong_\rho} = \cong_\rho$. However, if $\langle T, \sim\rangle$ is a congruence, we abbreviate $\cong_\sim$ to $\cong$ and we say that $\cong$ is the *extension* of $\sim$ to all terms. Let again $\langle T, \rho\rangle$ as specified before. We define the *congruent completion* of $\langle T, \rho\rangle$ as the pair $\langle \tilde{T}, \sim_\rho\rangle$ such that

$$\begin{array}{rcl} \tilde{T} & = & \{\; t \mid \exists s \in T : s \;\cong_\rho\; t \;\} \\ \sim_\rho & = & \{\; \langle s, t\rangle \mid s, t \in \tilde{T} \;\text{ and }\; s \;\cong_\rho\; t \;\} \end{array}$$

In words, $\tilde{T}$ is the set of all terms that are equivalent, with respect to $\cong_\rho$, to at least one term of T, and the relation $\sim_\rho$ is the restriction of the relation $\cong_\rho$ to $\tilde{T}$. The pair $\langle \tilde{T}, \sim_\rho\rangle$ is the smallest congruence that contains $\langle T, \rho\rangle$ and $\langle T, \rho\rangle$ is a congruence if and only if it is equal to $\langle \tilde{T}, \sim_\rho\rangle$.

3 Collections of Structures

3.1 Structures, Sets of Structures, and Collections 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 $\mathcal{I}$. 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 into a finite number of sets of structures. As for terms, 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 can be viewed as the identifier of the set of structures $E_{i_{null}} = \{null : i_{null}\}$. When we display structures, however, we use a simplified notation with constant and unary symbols. Let E be a collection of structures. Let I be the set of identifiers used by the structures of E. By E_i, we denote the set of structures of E that are of the form $f(i_1, i_2) : i$. Therefore, we have $E = \bigcup_{i \in I} E_i$. The meaning of the collection of structures E is then defined as follows: we say that E *denotes* the (unique) family of sets of terms $(T_i)_{i \in I}$ 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:

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

These two sets of structures denote the sets of terms:

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

We observe that the set T_1 is infinite.

Let $\langle T, \rho\rangle$ be the pair *presented* by $(T_i)_{i \in I}$. We say that $\langle T, \rho\rangle$ is the *abstract* denotation of E (because it gets rid of the particular choice of the set I), while $(T_i)_{i \in I}$ is the *concrete* denotation of E, or, simply, its denotation. By definition, T is sub-term complete.

We say that E is *well-formed* if none of the sets T_i of its denotation are empty. This is true if and only if we can write the elements of I as a sequence $i_1, i_2, \ldots, i_n$ in such a way that $i_1 = i_{null}$ and that, for every set identifier i_l such that $l > 1$, there exists a structure $f(i_j, i_k) : i_l$ such that $j, k < l$. In the following, we always assume that E is well-formed.

We say that E is *normalized* if it does not contain two different structures $f(i_1, i_2) : i$ and $f(i_1, i_2) : i'$ (with the same key but different set identifiers). It can be proved that E is normalized if and only if $\{T_i \mid i \in I\}$ is a partition of T and $T_i \neq T_j$ $(\forall i, j \in I : i \neq j)$. It is also the case that $\langle T, \rho\rangle$ is a congruence as soon as E is normalized.

3.2 Operations over Collections of Structures

There are four main operations to handle structures: *toSet*, *substitute*, *normalize* and *unify*.

We present every operation in three or four steps. First, we give an easy to understand but possibly not entirely accurate specification of the operation. Second, we explain it operationally so that it can be executed "by hand" on examples. (The actual efficient implementation is the subject of Section 4.) Third, we make the first description more (i.e., completely) accurate. (For some operations, this is not needed.) Fourth, we justify that the operational description of the operation is correct. Each operation is also illustrated by means of examples.

The operation *toSet* takes as input a term t and a normalized collection of structures E. It returns the identifier i of the set of structures E_i to which the term belongs. (More exactly, the term belongs to the set of terms T_i denoted by E_i.) If the term t is not initially represented in the collection of structure E, the collection is first minimally extended. At the algorithmic level, assuming that $t = f(t_1, t_2)$ and that E is implemented as a mutable global object, the operation *toSet* first recursively computes the identifiers i_1 and i_2 corresponding to t_1 and t_2. Then, if no structure $f(i_1, i_2) : i$ already exists for some i, such a structure is created with a novel identifier i. In any case, the identifier i (of the new or old structure $f(i_1, i_2) : i$) is returned. The precise semantic characterization of the operation is the following: the operation extends the abstract denotation $\langle T, \sim\rangle$ of E with all terms that are equivalent in the sense of $\cong$ to a sub-term of t, not already in T.[1] Clearly, it is the best (i.e., the least) that can be done. There are two extreme cases. First, if the term t already belongs to T, then $T' = T$ (nothing changes). Second, if $T = \{\}$, then T' simply is the set of all sub-terms of t. A shared representation of t has been created.[2] (Here, T' represents the final denotation of E.) The correctness of the operation is proved as follows. The final collection of structure is normalized since all structures that are added to it have unique keys. Moreover, the new structures are such that the relation $\cong$ is unchanged. So, the terms that are added to T are exactly the terms that are equivalent with respect to $\cong$ to the sub-terms of t not belonging to T. As an example, consider the collection of structures E partitioned into the two sets E_1 and E_2, below:

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

It denotes the sets of terms:

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

Let us apply the operation *toSet* to the term $f(b)$ and to the collection E. We get the new collection of structures E' defined by:

$$E'_1 = \{a : 1\} \qquad\qquad E'_2 = \{b : 2,\ f(1) : 2\} \qquad\qquad E'_3 = \{f(2) : 3\}$$

which denotes the sets of terms:

$$T'_1 = \{a\} \qquad\qquad T'_2 = \{b,\ f(a)\} \qquad\qquad T'_3 = \{f(b),\ f(f(a))\}$$

The term $f(b)$ has been added to T, but also $f(f(a))$ because $f(b) \cong f(f(a))$. Note finally that the identifier 3 is returned by the operation, in this example.

The second operation, called *substitute*, takes as input two identifiers k and l and a collection of structures E that uses k and l. It is not assumed that E is normalized. It adds an equivalence constraint between the terms represented by E_k and E_l. To define it operationally, we introduce a new notation. Let *struct* be a structure, (i.e., *struct* is of the form $f(i_1, i_2) : i$) and let k and l be two set identifiers. We denote by *struct*$[k \mapsto l]$ the possibly new structure obtained by replacing any occurrence of l in *struct* by k. For example, $f(1, 3) : 3[2 \mapsto 3]$ is equal to $f(1, 2) : 2$. If the structure is modified, we say that the operation performs a *renaming* of it. The operation *substitute* just removes from E every structure *struct* that involves l (i.e., a structure of one of the three forms $f(i_1, i_2) : l$ or $f(l, i_2) : i$ or $f(i_1, l) : i$, for some i_1, i_2, i) and it adds to E every corresponding structure *struct*$[k \mapsto l]$, if it is not already in it. The semantic characterization of the operation *substitute* is as follows. Let E' be the resulting collection of structures and let $\langle T', \rho'\rangle$ be its abstract denotation. The congruence $\cong_{\rho'}$ is the smallest of all congruences $\cong$ over all terms for which the two implications below hold:

$$\begin{array}{rcl} s\ \rho\ t & \Longrightarrow & s \cong t \\ s \in T_k,\quad t \in T_l & \Longrightarrow & s \cong t \end{array}$$

[1]Remember that $\cong$ is the extension of $\sim$ to all terms.

[2]This way of representing terms can be related to, e.g, the term banks of [8].

To prove this property we may show, on the one hand, that $\rho \subseteq \rho'$ and $s \in T_k,\ t \in T_l \implies s\ \rho'\ t$, and, on the other hand, that $\rho' \subseteq \cong$ for any congruence $\cong$ specified as above. Both parts can be reformulated as lemmas relating the sets T_i and T'_i and then proved by induction on the structure of terms. However, the detailed proofs are a bit too long to be given here. As an example, let us apply the operation *substitute* to the collection of structures obtained at the end of the previous example and to the identifiers 1 and 2. All occurrences of 2 are replaced by 1 in the structures. Thus, we get the collection E^2 below:

$$E^2_1 = \{a : 1,\ \ b : 1,\ \ f(1) : 1\} \qquad\qquad E^2_3 = \{f(1) : 3\}$$

It denotes the sets of terms:

$$T^2_1 = \{a, b, f(a), f(b), f(f(a)), f(f(b)), \ldots\} \qquad\qquad T^2_3 = \{f(a), f(b), f(f(a)), f(f(b)), \ldots\}$$

We observe that the collection of structures E^2 is not normalized since it contains two different structures with the same key $f(1)$. The next operation can be used to simplify it.

The operation *normalize* takes as input a collection of structures E and returns a collection of structures $\tilde{E}$ denoting the *congruence completion* of the denotation of E. It can be implemented as follows: if E does not contain two different structures with the same key, return E; otherwise, choose k and l such that E contains two structures $f(i_1, i_2) : k$ and $f(i_1, i_2) : l$, with the same key; apply the operation *substitute* to k, l and E, giving E'; and, finally, apply *normalize* to E', giving $\tilde{E}$. Obviously, this recursive formulation can be replaced by a simple loop. In both formulations, the algorithm terminates since the number of set identifiers is decreased by one, at each step. The correctness of the operation is proved as follows: If E is normalized, it denotes a congruence, which is its own congruence completion. Thus, E itself is a correct result. Otherwise, let us choose two structures $f(i_1, i_2) : k$ and $f(i_1, i_2) : l$ (with the same key) in E. Let $(T_i)_{i \in I}$ and $\langle T, \rho \rangle$ be the concrete and abstract denotations of E. One has $T_k \cap T_l \neq \{\}$. Therefore, by transitivity of $\cong_\rho$, we can write: $s \in T_k,\ t \in T_l \implies s \ \cong_\rho t$. Let us assume that *normalize* first applies *substitute* to k, l, and E, giving E'. Then, it follows from the specification of *substitute* given above, that $\cong_{\rho'} = \cong_\rho$ where $\langle T', \rho' \rangle$ is the abstract denotation of E'. Consequently, the congruence completions of $\langle T', \rho' \rangle$ and $\langle T, \rho \rangle$ are equal. So, by induction on the number of identifiers, applying *normalize* to E' will produce a collection of structures denoting the congruence completion of $\langle T, \rho \rangle$. It can be noticed that the final result is independent of the particular choice of the identifiers k and l, at each step, since the congruence completion of the denotation of the initial E is uniquely defined. Of course, this is true only "up to the choice" of the identifiers that are kept in the final collection of structures. However, the choice of which identifier is removed at each step is important to ensure a good time complexity of the algorithm. This issue is discussed later on. As an exemple, let us apply the operation *normalize* to the collection of structure E^2 from the previous exemple. We choose to apply *substitute* to 1, 3, and E^2 since it entails less renamings than the opposite choice (3, 1, and E^2). We get the collection of structures E^3 below, which uses only one identifier:

$$E^3_1 = \{a : 1,\ \ b : 1,\ \ f(1) : 1\}$$

It denotes the set of all terms that can be constructed from the symbols a, b, and f:

$$T^3_1 = \{a, b, f(a), f(b), f(f(a)), f(f(b)), f(f(f(a))), f(f(f(b))), \ldots\}$$

The last operation, called *unify*, takes as input two set identifiers i and j, and a collection of structures E. It first apply the operation *substitute* to i, j and E; afterwards, it applies the operation *normalize* to the resulting collection of structures. Semantically, the effect of this operation can be explained as follows. Let $\langle T, \rho \rangle$ be the abstract denotation of the (initial) collection of structures E. The relation ρ can be viewed as a set of "equality constraints" between the terms belonging to T. Applying the operation *unify* to i, j, and E first adds to this set of equality constraints, the constraints that all terms in T_i are equal to all terms in T_j. Then, it computes a representation of all terms that must be equal to the terms in T according to the updated set of constraints. As a (very general) example, assume that E is initially empty, and consider a list of equations between terms:

$$s_1 = t_1, s_2 = t_2, \ldots, s_m = t_m.$$

Applying the operation *toSet* to the terms in the equations produces a list of "equations" between set identifiers:

$$i_1 = j_1, i_2 = j_2, \ldots, i_m = j_m.$$

Applying the operation *unify* to the pairs of identifiers i_k, j_k computes a representation of the theory containing all terms that are equal to some sub-term of the terms involved in the equations. Notice that our method is "incremental" since the operation *unify* can be applied to the identifiers i_k and j_k "at any time" (provided that the operation *toSet* has already been applied to the corresponding terms). In practice, it is often desirable to apply it "as soon as possible" to reduce the size of the collection of structures E. As a concrete instance of this very general example, let us consider the following three equations:

$$b \;=\; f(a) \qquad\qquad f(b) \;=\; f(f(a)) \qquad\qquad a \;=\; b$$

Applying the operation *toSet* to the terms of the first equation, we get, for instance, the sets of structures below:

$$E_1^0 = \{a:1\} \qquad\qquad E_2^0 = \{b:2\} \qquad\qquad E_3^0 = \{f(1):3\}$$

These sets of structures represent the following sets of terms:

$$T_1^0 = \{a\} \qquad\qquad T_2^0 = \{b\} \qquad\qquad T_3^0 = \{f(a)\}$$

To "solve" the equation $b = f(a)$, we apply *unify* to 2, 3 and E^0, which gives us the collection E in the first example of page 84. Note that the operation *normalize* does nothing here. Applying the operation *toSet* to the two terms of the second equation $f(b) = f(f(a))$ creates the collection of structures E' on the same page. Since the terms $f(b)$ and $f(f(a))$ are represented by the same identifier 3, the operation *unify* does nothing. Finally, solving the last equation $a = b$ successively produces the collection E^2 (by the application of *substitute* to 1, 2 and E'), and the collection E^3 (by the application of *normalize* to E^2) previously constructed. One can check that the three equations logically implies that all terms constructed with the symbols a, b, and f are equal, and that they are the only terms that must be equal to a sub-term of the equations. Thus, E^3 is a correct result since it denotes exactly this set of terms.

A different example of the same problem is detailed in the appendix A.

4 Practical Implementation of Collections of Structures

In this section, we describe an efficient "low level" implementation of collections of structures. It is necessary to do so to justify that collections of structures are efficiently implementable and, especially, to make an accurate time complexity analysis possible. The implementation is written in Java. The choice of Java is incidental since we do not use any Java "favourite feature". Our programming style is not quite "object oriented" but mainly simply "imperative". Therefore, since the key algorithms are short, we prefer to explain the actual code rather than a pseudo code translation, which can possibly be wrongly understood by places. We concentrate on the implementation of the three operations *substitute*, *normalize*, and *unify*. This implementation uses two classes to handle lists of natural numbers. The first class is called `MultiList`. An instance of this class is parameterized by two numbers: the maximum number of lists that can be maintained by the instance and the maximal number that a list may contain. Very importantly, the lists must be disjoint. The class provides a method `add(i, j)` that adds the number j into the i-th list as well as a method `remove(i, j)` to remove it. Both operations are executed in constant time. The class also provides an "iterator" to apply a set of instructions to all elements of a list. Some elements can be added to or removed from a list during the execution of the iterator on that list. The second class is called `OneList`. It maintains a single set of natural numbers not exceeding a maximal number, and it supports similar methods as `MultiList`. Both classes can optionally maintain a "free list" of all numbers not used in the "regular" lists. The method `getId` choses a number in the free list. The method `free` removes a number from a list and returns the number to the free list. Now, we turn to the implementation of the main data structure, i.e., collections of structures. The implementation maintains a single "mutable" collection of structures, thanks to three Java classes called `Keys`, `Structs`, and `Sets`. Part of the code of the class `Keys` is given below.

```
class Keys{
static final int NBRKEYS ;
static char[] tf  = new char[NBRKEYS] ;
static int[]  ti1 = new int[NBRKEYS] ;
static int[]  ti2 = new int[NBRKEYS] ;
static MultiList sameKey = new MultiList(NBRKEYS, Structs.NBRSTRUCTS) ;
static OneList dbList = new OneList(NBRKEYS) ;

static int getKey(char f, int i1, int i2){ ... }
static void removeKey(int key){ ... }}
```

The class `Keys` implements a hash table containing the keys of the structures belonging to the current collection of structures. Every key $f(i_1, i_2)$ of an "existing" structure $f(i_1, i_2) : i$ is represented by a natural number *key* less than `NBRKEYS`, and one has `tf`$[key] = f$, `ti1`$[key] = i_1$, and `ti2`$[key] = i_2$. We call the number *key* the key identifier of $f(i_1, i_2)$. The *key*-th list of the `MultiList` object `sameKey` contains the list of all structures (in fact, of all structure *identifiers*, see below) having this key. This *key*-th list contains only one number when the current collection of structure is normalized. Finally, the `OneList` object `dbList` contains the list of the identifiers of all keys belonging to at least two structures. This list is needed to efficiently implement the operation *normalize*. We only speak of the two main methods of the class. The method `getKey` returns the key identifier of a key $f(i_1, i_2)$. If the key does not exist in the current table, it creates it. The method `removeKey` removes a key from the hash table, assuming that it is no longer used by any structure. Part of the code of the class `Structs` is now given hereunder.

```
class Structs{
static final int NBRSTRUCTS  ;
static int[] key = new int[NBRSTRUCTS] ;
static int[] id = new int[NBRSTRUCTS] ;
static MultiList sameId = new MultiList(Sets.NBRSETS, NBRSTRUCTS) ;
static MultiList sameI1 = new MultiList(Sets.NBRSETS, NBRSTRUCTS) ;
static MultiList sameI2 = new MultiList(Sets.NBRSETS, NBRSTRUCTS) ;
static MultiList keyId = new MultiList(NBRKEYS, NBRSTRUCTS) ;

static void removeStruct(int struct, int i1, int i2, int id, int key)
{
  sameId.free(id, struct) ;
  sameI1.remove(i1, struct) ;
  sameI2.remove(i2, struct) ;
  Keys.sameKey.remove(key, struct) ;
  keyId.remove(keyId(struct), struct) ;
}
static int addStruct(int key, int i1, int i2, int id)
{
  int struct = sameId.getId() ;
  Structs.key[struct] = key ;
  Structs.id[struct]  = id ;
  sameId.add(id, struct) ;
  sameI1.add(i1, struct) ;
  sameI2.add(i2, struct) ;
  Keys.sameKey.add(key, struct) ;
  keyId.add(keyId(struct), struct) ;
  return struct ;
}
static boolean existStruct(int key, int i){ ... }}
```

Every structure $f(i_1, i_2) : i$ belonging to the current collection of structures is represented by a natural number *struct* less than `NBRSTRUCTS`. We call it the identifier of the structure. The equalities `key`[*struct*] = *key* and `id`[*struct*] = i must hold (where *key* is the key identifier of $f(i_1, i_2)$). The three `Multilist` objects `sameId`, `sameI1`, and `sameI2` contain lists of identifiers of structures sharing the same set identifier at some position in the structures. For instance, the $i-$th list of the object `sameId` contains the structure identifiers of all "existing" structures of the form $f(i_1, i_2) : i$ (for some f, i_1, i_2). Similarly, the i_1-th list of the object `sameI1` contains the structure identifiers of the structures of the form $f(i_1, i_2) : i$ (for some f, i_2, i). The last `Multilist` object `keyId` implements a hash table to determine in constant time [3] if a structure already exists. It is used by the method `existStruct`. The methods `addStruct` and `removeStruct` respectively adds a "non existing" structure to the current collection of structures, and removes an "existing" one from it. Their correctness, i.e., the fact that they maintain all the invariant relations previously stated, can be checked by simple code examination. It is also clear that they execute in constant time.

We now turn to the class `Sets`, which contains the Java code of the operations *normalize*, *substitute*, and *unify* but, due to lack of space, we only concentrate on the code of the key auxiliary methods and we explain how they are used in the main methods. The code of the main methods can easily be inferred from these explanations and those of Section 3.2.

```
class Sets{
static final int NBRSETS ;
static final int iNull = newSet() ;
static int[] nbrOcc = new int[NBRSETS] ;
static OneList idList = new OneList(NBRSETS)  ;

static void killStruct(int struct, int key, int i1, int i2, int id)
{
  Structs.removeStruct(struct, i1, i2, id, key) ;
  nbrOcc[i1] -- ; nbrOcc[i2] -- ; nbrOcc[id] -- ;

  if (Keys.sameKey.isEmpty(key))
    Keys.removeKey(key) ;
  else if (Keys.sameKey.isOne(key))
          Keys.dbList.remove(key) ;
}
```

[3](on the average)

```
static void putInStruct(char f, int i1, int i2, int id)
{
  int key = Keys.getKey(f, i1, i2) ;

  if (!Structs.existStruct(key, id))
  {
    boolean keyIsOne = Keys.sameKey.isOne(key) ;
    int struct = Structs.addStruct(key, i1, i2, id)  ;
    nbrOcc[i1] ++ ; nbrOcc[i2] ++ ; nbrOcc[id] ++ ;
    if (keyIsOne)
      Keys.dbList.add(key) ;
  }
}
```

The constant `NBRSETS` defines the number of available set identifiers. The constant `iNull` provides the actual value of the "dummy" set identifier i_{null}. The array `nbrOcc` maintains the number of occurrences of every set identifier in the structures of the current collection of structures. It is used by the method `substitute` to choose which identifier to remove and which to keep, to be most efficient. The `OneList` object `idList` maintains the list of set identifiers currently in use. It allows us notably to reuse set identifiers that have been freed. The method `normalize` checks the first element of the list `dbList` of the keys occurring in at least two structures, until the list is empty. Notice that this element is not explicitely removed here: it will be removed elsewhere, by the method `killStruct`, when only one structure will be left for this key. At each iteration, the method `normalize` chooses two different structures with the same key, and it applies the method `substitute` to them. Hence, the method correctly implements the operation *normalize* (provided that `substitute` is correct). Note, however, that the first key in `dbList` can be removed from it not only because it can now be the key of only one structure, but also because it disappears due to the call to `substitute`. In that case, it can possibly be "re-added" after renaming. (This is a "subtle" point.) The method `substitute(i, j)` possibly swaps i and j in order to discard the set identifier making the least number of occurrences in the structures. Then, it retrieves all structures containing j by running through the $j-$th lists of `Structs.sameId`, `Structs.sameI1`, and `Structs.sameI2`. Finally, the set identifier j is freed from `idList`. Each time a structure is retrieved, it is removed from *all* lists to which it belongs, it is "renamed" by replacing j by i, and then possibly re-added to all relevant lists. It is important to note that the same structure may initially appears in several of the lists in `Structs.sameId`, `Structs.sameI1`, and `Structs.sameI2` (up to three) but that it is "handled" only once because it is removed from all lists the first time it is encountered. The method `killStruct` removes a structure from all the relevant lists and maintains the correct lists in `sameKey` and `dbList`. The method `putInStruct` does the opposite. It can be checked that both methods execute in constant time (assuming that the method `existStruct` does).

5 Complexity Analysis and Experimental Evaluation

We first analyse the time complexity of the method `normalize` in the previous section. It is clear from the description of the implementation that this complexity is proportional to the number of time that a structure is "renamed", since the methods `killStruct` and `putInStruct` are executed in constant time. Remember that we call *renaming* such a modification of the data structure (see bottom of page 4). Let us assume that the method `normalize` executes `substitute` m times. If it is so, it "merges" $m + 1$ sets of structures $S_{i_0}, \ldots, S_{i_m}$ into a single one. (Note that the sets S_{i_j} are different from the sets E_{i_j}. Actually, one has $E_{i_j} \subseteq S_{i_j}$ for all i_j.) Let us assume that the sets $S_{i_0}, \ldots, S_{i_m}$ respectively initially contain $n_0, \ldots, n_m$

structures and let us define $n = n_0 + \ldots + n_m$. We claim that the number of renamings that are executed is bounded by $n \log_2(n) - (n_0 \log_2(n_0) + \ldots + n_m \log_2(n_m))$. We prove it by induction on m. The claim is obviously correct if $m = 0$. Let us assume that $m \geq 1$ and consider the last execution of the method `substitute`. At that point, we are left with only two sets of structures to be "merged". Let us assume that the two sets respectively contains n'_1 and n'_2 structures. We may choose the identifiers $i_0, \ldots, i_m$ in such a way that the first remaining set of structures is obtained from $S_{i_0}, \ldots, S_{i_j}$ and the second one from $S_{i_{j+1}}, \ldots, S_{i_m}$. Let us define $n''_1 = n_0 + \ldots + n_j$ and $n''_2 = n_{j+1} + \ldots + n_m$. Obviously, we have $n'_k \leq n''_k$ $(k = 1, 2)$ since structure renamings can only reduce the number of "living" structures. Moreover, by induction hypothesis, the number of renamings performed up to now is bounded by $n''_1 \log_2(n''_1) + n''_2 \log_2(n''_2) - (n_0 \log_2(n_0) + \ldots + n_m \log_2(n_m))$. Now, assuming that $n'_1 \leq n'_2$, the last execution of `substitute` performs n'_1 additional renamings giving a total number of renamings bounded by $n'_1 + n''_1 \log_2(n''_1) + n''_2 \log_2(n''_2) - (n_0 \log_2(n_0) + \ldots + n_m \log_2(n_m))$. It remains to show that $n'_1 + n''_1 \log_2(n''_1) + n''_2 \log_2(n''_2) \leq n \log_2(n)$. Let us suppose first that $n''_1 \leq n''_2$. Then $2n''_1 \leq n$. Consequently, we have $n'_1 + n''_1 \log_2(n''_1) + n''_2 \log_2(n''_2) \leq n''_1 + n''_1 \log_2(n''_1) + n''_2 \log_2(n''_2) = n''_1 \log_2(2n''_1) + n''_2 \log_2(n''_2) \leq n''_1 \log_2(n) + n''_2 \log_2(n) = n \log_2(n)$. Now, suppose that $n''_2 \leq n''_1$. Then, $n'_1 \leq n'_2 \leq n''_2$. Thus, $n'_1 + n''_1 \log_2(n''_1) + n''_2 \log_2(n''_2) \leq n''_2 + n''_1 \log_2(n''_1) + n''_2 \log_2(n''_2) = n''_1 \log_2(n''_1) + n''_2 \log_2(2n''_2) \leq n''_1 \log_2(n) + n''_2 \log_2(n) = n \log_2(n)$.

A simpler but less tight upper bound can be obtained by observing that the minimal possible value for $n_0 \log_2(n_0) + \ldots + n_m \log_2(n_m)$ is $n \log_2(n/(m+1))$. Therefore, an upper bound to the number of renamings is $n \log_2(n) - n \log_2(n/(m+1)) = n \log_2(n) - n(\log_2(n) - \log_2(m+1))$ $= n \log_2(m+1)$. Now, the number n is bounded by $3N$ where N is the number of structures in the collection of structures at the start. So, a simpler upper bound to the number of renamings is $3N \log_2(m+1)$. However, some of these structures are not considered for renaming. Let M be the number of sets of structures –in the usual sense– in the collection of structures at the start. We may guess that $n = 3N(m+1)/M$ (on the average). A probably better upper bound to the number of renamings is thus $(3N/M)(m+1)\log_2(m+1)$.

To conclude this complexity analysis, let us assume that an arbitrary number of operations *substitute* or *unify* are executed, for example, by "solving" a large number of equations between terms. The maximum possible number of executions of the operation *substitute* is $M-1$. Therefore, an upper bound to the total number of renamings for the overall execution is $3N \log_2(M)$. Finally, suppose that we start from an empty collection of structures and that we "solve" a large number of equations between terms. Let N be the total number of sub-terms in the equations and assume that we apply the operation *toSet* to all terms in the equations before starting to solve them. Then, the initial number of structures is equal to N as well as the number of sets of structures, so that an upper bound to the number of renamings is $3N \log_2(N)$.

Since the description of the implementation given in the previous section allows us to claim that the execution time for any sequence of executions of the operations *normalize*, *substitute*, or *unify* is proportional to the number of renamings that are performed, the above discussion can be used to derive various time complexity formulas, such as, for instance, $O(N \log N)$ in the last and simplest case.

We now complement the complexity analysis with a short experimental evaluation. This evaluation has been done on a MacBook Pro 2.4GHz (Intel Core i5, 4Gb RAM) using Mac OS X 10.6.8. Timings are measured using the method `System.nanoTime()`. We have "randomly" chosen a large term of 100000 symbols (i.e., 100000 symbols if the term is written in polish notation). The term is a large (uninterpreted) boolean formula using 20 different constants (propositional variables). Then, we have represented this term as a collection of structures, using the operation *toSet*. This representation uses 36901 structures, each of them belonging to a different set of structures. Using the identifiers of these sets we have randomly generated a sequence of 36901 pairs of identifiers to which the operation `unify` has been applied, one by one. The results of the experiment are depicted in the table below. Column *eq* provides the number of equations already "solved" at a given line. Column m provides the number of applications of the operation *substitute* at that stage. Column $3m \log_2(m)$ depicts just this number. Column r provides the number of structure renamings already performed. The next column is the ratio of the two previous ones. Column t provides the amount of time needed to perform the m operations *substitute*, in seconds. Column t/r gives the ratio of the time to the number of renamings, in microseconds. Columns N and M respectively provide the number of structures and the number of sets in the current collection of structures.

eq	m	$3m \log_2(m)$	r	$(3m \log_2(m))/r$	t	t/r	N	M
0	0		0		0.000		36901	36901
2435	2500	84657	5089	16.6	0.038	7.6	36837	34401
4862	5000	184315	10401	17.7	0.069	6.7	36764	31901
7256	7500	289635	16139	17.9	0.087	5.4	36658	29401
9609	10000	398631	22376	17.8	0.108	4.8	36511	26901
11879	12500	510361	29692	17.1	0.130	4.4	36281	24401
13887	15000	624270	39858	15.6	0.149	3.8	35788	21901
13893	17500	739991	63847	11.5	0.189	3.0	32887	19401
13893	20000	857262	82840	10.3	0.215	2.6	29052	16901
13893	22500	975890	98057	9.9	0.235	2.4	24790	14401
13893	25000	1095723	108849	10.0	0.248	2.3	20653	11901
13893	27500	1216639	120459	10.1	0.262	2.2	16195	9401
13893	30000	1338540	128188	10.4	0.271	2.1	12116	6901
13893	32500	1461344	136023	10.7	0.280	2.1	7896	4401
13893	35000	1584982	143020	11.0	0.287	2.0	3584	1901
36901	36891	1679018	148140	11.3	0.298	2.0	81	10

We see that $3m \log_2(m)$ is an upper bound to the number of renamings and that it is not tight. Column t/r supports our claim that the execution time is $O(r)$. Thus, the whole experiment supports our claim that the overall complexity of solving a large number of equations involving N sub-terms is $O(N \log(N))$. Moreover, the timings in column t show that our Java implementation is fast. Finally, it is interesting to note that most operations *substitute* are executed during a single execution of *normalize* when the 13894-th equation is solved.

6 Application to Expression Simplification

We have used collections of structures to tackle various expression simplification problems. Some formal theories can be finitely represented by a collection of structures. For instance, the set of all boolean formulas using a bounded number of propositional variables can be finitely represented. We have designed a "bottom-up" algorithm to compute such a finite representation from a set of equational axioms. The algorithm starts from a small collection of structures representing a small numbers of terms and it instantiates all axiom variables with the set identifiers of those structures to create new sets of structures that are used in turn to instantiate the axioms. The operation *unify* is used to instantiate the axioms. The algorithm terminates if the computed theory determines a finite number of equivalence classes. We have applied the algorithm to the boolean calculus, to compute a finite representation of all formulas involving at most three propositional variables. The resulting collection of structures contains 256 sets of structures and 131333 structures. It is computed in less than 5 seconds and it can be used to simplify (i.e., minimize) arbitrarily large boolean formulas containing at most three propositional variables, in linear time. However, when the theory is infinite or simply too large, the bottom-up algorithm is not applicable or takes too much space and time. Therefore, we have designed another "goal oriented" algorithms, which concentrates on a particular formula to be simplified and on its sub-terms to only compute terms that are equal to them. When a shorter formula is found, the algorithm discards less interesting sets of structures, corresponding to non minimal terms (at this stage), and computes more terms that are equal to the terms currently represented by the collection of structures. This algorithm thus uses a "local search" approach. We have used it to simplify large boolean formulas involving up to 20 variables. More information about these algorithms can be found in [5].

7 Related Work

The initial and main motivation for the work presented in this paper is to get an efficient method to represent large sets of equal terms (or expressions) in order to simplify expressions. Preliminary results on this issue have been reported in [5]. It however happens that, although designed independently, our work can be strongly related to existing work on computing the congruence closure of a relation over ground terms (see, e.g., [1] (Chapter 9), [2, 4, 6, 7]). In Section 5, we have shown that collections of structures are convenient to tackle the decision problem that motivates this related work. Our method allows us to "solve" a set of ground equations with time complexity $O(N \log N)$ where N is the number of distinct sub-terms in the equations, while the algorithms presented in [1, 4, 6] are $O(N^2)$. The algorithms in [2, 7] are $O(N \log N)$ as our method. Now, we explain the key differences between our collections of structures and the data structures used in those previous proposals. All of them keep in memory a representation of all terms involved in the equations while our method progressively build a compact representation of a larger set of terms to which they belong. They also use the Union-Find data structure [3] to represent the equivalence classes of terms while we do not need it anymore since equivalence classes are represented by set of structures identifiers. So, those methods cannot be used to represent infinite sets of equal terms and therefore to simplify expressions as we want to do it. Nevertheless, the relation between our work and previous work on congruence closure is especially clear for Shostak's algorithm (see, e.g. [4, 7]). The so-called *signatures* that are used, in all variants of this algorithm, to detect that different terms are equivalent, are very similar to our structures $f(i_1, i_2) : i$. In fact, there is a strong correspondence between the two notions but our algorithms directly work on the structures (the signatures), not on the original terms (which only remain implicitely present in the collection of structures). It is argued in [4] that Shostak's algorithm can be viewed as computing a Knuth-Bendix completion of the set of ground equations to which it is applied. Our collections of structures can also be viewed as a confluent set of rewrite rules (when the collection is normalized). Our set of structures identifiers are similar to the "new constants" used in [4] (and also in [7]) and our operation *toSet* simply "applies" the rewrite rules to a term. However, our "set of rewrite rules" is not exactly the same as in [4, 7] because we do not use the "trick" of considering identifiers as new constants. We have implemented versions of the algorithms in [4] and [7] using the same low level data structures as in our collection of structures implementation and applied them to the problem described at the end of Section 5. On a few experiments, we have found that the algorithm in [7] is $1,4$ times slower than ours, on the average, while the algorithm in [4] is 41 times slower, because it reconsiders many times terms with the same signature, which are filtered by the algorithm in [7] using a hash table for the signatures, similar to the hash table that we maintain for structure keys.

Alternatively, our approach to expression simplification can be viewed as an instance of *term indexing* [9]: in our case, the index is the collection of structure E and the set of indexed term is its denotation T. The relation $R(l, t)$ between an indexed term l and a "query" t is that l is a simplification of t. This relation is very different in nature from relations such as generalization or instantiation that are usually considered in classical term indexing but some methods from this area also apply to equational theories such as AC theories. So, we need to investigate if they can be used to improve our implementation of collection of structures for specific theories.

8 Conclusion and Future Work

We have presented a data structure to compactly represent large sets of equivalent terms, we have demonstrated its efficiency for computing the congruence closure of a set of ground equations, and we have sketched how to use it to simplify expressions.

In the future, we plan to mainly work on the problem of simplifying expressions. We want to improve our current simplification algorithms and to apply them to various classes of expressions such as, for instance, regular expressions (see e.g., [10] for an existing simplification algorithm). We will also undertake an extensive experimental comparison between our work and the congruence closure algorithms from [2, 4, 6, 7].

Several other avenues of research can be considered. For instance, we could try to specialize collections of structures to take into account common properties of operations such as associativity and commutativity using e.g., term indexing techniques [8, 9].

Acknowledgments

Comments from Loïc Lecharlier, Charles Pecheur, Gauthier van den Hove, José Vander Meulen, and the reviewers of this paper and of a previous one are greatly acknowledged.

References

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

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

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

[4] Deepak Kapur. Shostak's congruence closure as completion. In *Rewriting Techniques and Applications*, pages 23–37. Springer, 1997.

[5] Baudouin Le Charlier and Mêton Mêton Atindehou. A Method to Simplify Expressions: Intuition and Preliminary Experimental Results. In Stephan Schulz, editor, *Proc. 11th International Workshop on the Implementation of Logics, Suva, Fiji*, EPiC series, 2015.

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

[7] 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.

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

[9] R. Sekar, I. V. Ramakrishnan, and Andrei Voronkov. Term indexing. In *Handbook of Automated Reasoning*, pages 1853–1964, Amsterdam, The Netherlands, 2001.

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

A Solving Equations between Terms: another Example

Let us consider the two equations between terms:

$$b = f(f(f(a))) \qquad\qquad a = f(f(a))$$

Applying the operation *toSet* to the terms of the first equation, we get, for instance, the sets of structures below:

$$E_1^0 = \{b : 1\} \;\; E_2^0 = \{a : 2\} \;\; E_3^0 = \{f(2) : 3\} \;\; E_4^0 = \{f(3) : 4\} \;\; E_5^0 = \{f(4) : 5\}$$

These sets of structures represent the following sets of terms:

$$T_1^0 = \{b\} \;\; T_2^0 = \{a\} \;\; T_3^0 = \{f(a)\} \;\; T_4^0 = \{f(f(a))\} \;\; T_5^0 = \{f(f(f(a)))\}$$

To "solve" the equation $b = f(f(f(a)))$, we apply *unify* to 1, 5 and E^0, which gives us E^1:

$$E_1^1 = \{b : 1,\; f(4) : 1\} \;\; E_2^1 = \{a : 2\} \;\; E_3^1 = \{f(2) : 3\} \;\; E_4^1 = \{f(3) : 4\}$$

Note that the operation *normalize* does nothing here. The corresponding sets of terms are:

$$T_1^1 = \{b,\; f(f(f(a)))\} \;\; T_2^1 = \{a\} \;\; T_3^1 = \{f(a)\} \;\; T_4^1 = \{f(f(a))\}$$

Then, we apply the operation *toSet* to the two terms of the second equation $a = f(f(a))$. Those two applications do not modify the current collection of structures since $a \in T_2^1$ and $f(f(a)) \in T_4^1$. Finally, to solve the second equation, we apply the operation *unify* to 2, 4 and E^1. The sub-operation *substitute* is first applied to the same arguments, giving E^2:

$$E_1^2 = \{b : 1,\; f(2) : 1\} \;\; E_2^2 = \{a : 2,\; f(3) : 2\} \;\; E_3^2 = \{f(2) : 3\}$$

Now, the new collection of structures denotes a family of three infinite sets:

$$T_1^2 = \{b\} \cup T_3^2 \;\; T_2^2 = \{a, f(f(a)), f(f(f(f(a)))), \ldots\} \;\; T_3^2 = \{f(a), f(f(f(a))), f(f(f(f(f(a))))), \ldots\}$$

We observe that the sets E_1^2 and E_3^2 contain two structures with the same key. Therefore, the operation *substitute* is applied again, to 1, 3, and E^2, giving E^3:

$$E_1^3 = \{b : 1,\; f(2) : 1\} \;\; E_2^3 = \{a : 2,\; f(1) : 2\}$$

The collection E^3 is the "final result". It represents two infinite "similar" sets:

$$T_1^3 = \{b,\; f(a),\; f(f(b)),\; f(f(f(a))),\; \ldots\} \;\; T_2^3 = \{a,\; f(b),\; f(f(a)),\; f(f(f(b))),\; \ldots\}$$

It can be checked that T_1^3 and T_2^3 contain all terms that must respectively be equal to b and a, according to the two equations to be solved. They also contain all terms that are equal to $f(a)$, $f(f(a))$, or $f(f(f(a)))$. Thus, all terms that are equal to any sub-term of the terms involved in the equations, and no other term, belong to $T^3 = T_1^3 \cup T_2^3$, as it is "predicted" by the "theory" of Section 3.