Algorithms

In this section, we will present a congruence closure algorithm that is able to produce explanations. The algorithm is a mix of the approaches of the algorithms presented in [10] and [13, 14]. The basic structure of the algorithm is inherited from [10], which itself inherits its structure from the algorithm of Nelson and Oppen [12]. The technique to store and deduce equations of non constant terms is inspired from [13, 14]. Additionally the proof forest structure described below, was proposed by [13, 14].

Preliminaries

Our congruence closure algorithm operates on curried terms. Curried terms use a single binary function symbol to represent general terms. More formally let $\mathcal{F}$ be a finite set of functions with a designated binary function symbol $f \in \mathcal{F}$ and let every other function symbol in $\mathcal{F}$ be a constant. A term w.r.t. a signature of this form is called a *curried term*.

It is possible to uniquely translate a general set of terms $\mathcal{T}^{\Sigma}$ with signature $\Sigma = \langle \mathcal{F}, arity \rangle$ into a set of curried terms $\mathcal{T'}^{\Sigma'}$. Σ' is obtained from Σ by setting $arity$ to zero for every function symbol in $\mathcal{F}$ and introducing the designated binary function symbol f to $\mathcal{F}$. The translation of a term $t \in \mathcal{T}^{\Sigma}$ is given in terms of the function $curry$.

$$curry(t) = \begin{cases} t & \text{if } t \text{ is a constant} \\ f(\dots(f(f(g, curry(t_1)), curry(t_2)))\dots, curry(t_n)) & \text{if } t = g(t_1, \dots, t_n) \end{cases}$$

The idea of currying was introduced by M. Schönfinkel [15] in 1924 and independently by Haskell B. Curry [7] in 1958, who also lends his name to the concept. Currying is not restricted to terms. The general indea is to translate functions of type $A \times B \to C$ into functions of type $A \to B \to C$. There is a close relation between currying and lambda calculus [5]. Lambda calculus uses a single binary function λ. Its arguments can either be elements of some set or again lambda terms. For an introduction to lambda calculus, including currying in terms of lambda calculus and its relation to functional programming, see [2].

The benefit of working with curried terms is an easier and cleaner congruence closure algorithm that runs in optimal time $O(n \log(n))$.

Recently so called abstract congruence closure algorithms have been proposed and shown to be more efficient than traditional approaches [1]. The idea of abstract congruence closure is to introduce new constants for non constant terms. Doing so, all of equations the algorithm has to take into account are of the form $c = d$ and $c = f(a, b)$, where a, b, c, d are constants. This replaces tedious preprocessing steps, for example transformation to a graph of outdegree 2 [9],that are necessary for other algorithms to achieve the optimal running time.

Our method is does not employ the idea of abstract congruence closure. We found that using currying is enough to obtain an algorithm with optimal running time and no tedious preprocessing steps. The reason why we did not go for abstract congruence closure is, that we do not want to have the overhead of introducing and eliminating fresh constants. In the context of proof compression, our congruence closure algorithm will be applied to relatively small instances very often. We could introduce the extra constants for the whole proof before processing, but would still have to remove them from explanations every time we produce a new subproof. It would be interesting to investigate, whether our intuition in that regard is right, or if it pays off to deal with extra constants.

Comming back to the explanation producing congruence closure algorithms that inspired ours, [13, 14] describes an abstract one using currying. [10] uses a traditional algorithm without currying and extra constants. Our algorithm is a middle ground between them.

Congruence structure

We call the underlying data structure of our congruence closure algorithm a *congruence structure*. A congruence structure for set of terms $\mathcal{T}$ is a collection of the following data structures.

- Representative $r : \mathcal{T} \to \mathcal{T}$
- Congruence class $[.] : \mathcal{T} \to 2^{\mathcal{T}}$
- Left neighbors $lN : \mathcal{T} \to 2^{\mathcal{T}}$
- Right neighbors $rN : \mathcal{T} \to 2^{\mathcal{T}}$
- Lookup table $l : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$
- Congruence graph g
- Queue $\mathcal{Q}$ of type $\mathcal{T} \times \mathcal{T}$
- Current explanations $\mathcal{M} : \mathcal{T} \times \mathcal{T} \to \mathcal{E}$

The representative is one particular term of a class of congruent terms. It is used to identify whether two terms are already in the same congruence class and the data structures used for detecting equalities derived from the congruence axiom are kept updated only for representatives. The congruence class structure represents a set of pairwise congruent terms. It is used to keep track which representatives have to be updated when merging the classes of two terms. The structures left neighbor and right neighbor for every term keep track of other terms that appear as the second argument in a compound term of the form $f(a, b)$. The lookup table is used to keep track of all compound terms in the congruence structure and to merge compound terms, which arguments are congruent. The congruence graph stores the derived equalities in a structured way, that allows to create explanations for a given pair of terms. Edges are added to the graph in a lazy way, meaning that they are buffered and only actually entered into the graph when demanded. The queue $\mathcal{Q}$ keeps track of the order in which edges should be added to the graph. The function $\mathcal{M}$ stores the explanation for a buffered edge. The main idea about buffering is to overwrite explanations, when an edge was added due to the congruence axiom, while it is entered as an input equation later on. We call the unique congruence structure for $\mathcal{T} = \emptyset$ the *empty congruence structure*. It is not by coincidence that many of the used data structures are described as functions. In fact our congruence closure algorithm can and is implemented in a functional way and the data structures can be implemented immutable.

Algorithm 0.1: addEquation

```
Input: equation s = t or null
addNode(s)
addNode(t)
merge(s, t, s = t)
```

Algorithm 0.2: addNode

```
   Input: term v
1  if r is not defined for v then
2      r(v) ← v
3      [v] ← {v}
4      lN(v) ← ∅
5      rN(v) ← ∅
6      if v is of the form f(a, b) then
7          addNode(a)
8          addNode(b)
9          if l is defined for (r(a), r(b)) and l(r(a), r(b)) ≠ f(a, b) then
10             merge(l(r(a), r(b)), f(a, b), ∅)
11         else
12             l(r(a), r(b)) ← f(a, b)
13         lN(r(b)) ← lN(r(b)) ∪ {a}
14         rN(r(a)) ← rN(r(a)) ∪ {b}
```

Congruence closure algorithms

In this section we present the pseudocode of our congruence closure algorithm, state and prove its properties. Most importantly we show that it the method is sound and complete and has optimal running time $O(n\log(n))$. Computing the congruence closure of some set of equations E is done by adding all of them to an ever growing congruence structure, which initially is empty. Most algorithm pseudocodes do not include a return statement. In fact every algorithm implicitly returns a (modified) congruence structure or simply modifies a global variable, which is the current congruence structure. Since this has to be done in some order, we will often assume that E is given as a sequence of equations rather than a set. Adding an equation to a congruence structure is done with the addEquation method. The method adds boths sides of the equation to the current set of terms $\mathcal{T}$ using the addNode method and afterwards merges the classes of the two terms. The `addNode` method enlarges the set of terms and searches for equalities that are due to the congruence axiom. The updates of $\mathcal{T}$ are not outlined explicitly, but are understood to happen implicitly. The method `merge` initializes and guides the merging of terms. The actual merging is done by the method union by modifying the data structures.

Invariant 0.0.1 (Class)**.** *For every $s \in \mathcal{T}$ and every $t \in [r(s)]$, $r(t) = r(s)$.*

Proof. Clearly the invariant is true when intializing $[s]$ in line 2 of addNode.

Algorithm 0.3: merge

Input: term s
Input: term t
Input: extended equation eq

if $r(s) \neq r(t)$ **then**
  $c \leftarrow \{s = t\}$
  $eq \leftarrow s = t$
  while $c \neq \emptyset$ **do**
    Let (u, v) be some element in c
    $c \leftarrow c \setminus \{(u, v)\} \cup union(u, v)$
    lazy_insert(u, v, eq)
    $eq \leftarrow null$

Algorithm 0.4: lazy_insert

Input: term s
Input: term t
Input: extended equation eq

if *$\mathcal{M}$ is set for (s, t) or (t, s)* **then**
  if *eq is not null* **then**
    $\mathcal{M}(s, t) \leftarrow s = t$
else
  $\mathcal{Q} \leftarrow \mathcal{Q}.enqueue(s, t)$
  $\mathcal{M}(s, t) \leftarrow eq$

Algorithm 0.5: lazy_update

while *$\mathcal{Q}$ is not empty* **do**
  $(u, v) \leftarrow \mathcal{Q}.dequeue$
  $eq \leftarrow \mathcal{M}(u, v)$
  $g.insert(u, v, eq)$

The only other point in the code that changes $[s]$ is line 36 of union. Suppose the class of u is enlarged by the class of v in union and suppose the invariant holds before the union for those terms. Before the update of $[r(u)]$ the representative of every term in $[r(v)]$ is set to $r(u)$. Therefore the invariant remains valid after the update.

□

Invariant 0.0.2 (Lookup)**.** *The lookup structure l is defined for a pair of terms (s, t) if and only if there is a term $f(a, b) \in \mathcal{T}$ such that $r(a) = r(s)$ and $r(b) = r(t)$.*

Proof. Suppose l is defined for some pair of terms (s, t). The value of $l(s, t)$ was either set in lines 32 or 18 of union or in line 40 of addNode. In the latter case, l is set to $f(a, b)$ for the tuple

$(r(a), r(b))$ and therefore the invariant holds at this point. For changes to $r(a)$ or $r(b)$ in union the one implication of the invariant remains valid in case l is defined for the new representatives, or l is set for an additional pair of terms in lines 32 or 18. In case l is set to $(new\_left, r(u))$ or $(r(u), new\_right)$ in union, there is an l-entry l_v for which the invariant held before the union. The changes in representatives of x are reflected by new_left and new_right, while the representative of v is changed to $r(u)$. The new entry for l therefore respects the implication of the invariant.

To show the other implication, let $f(a, b) \in \mathcal{T}$. The term $f(a, b)$ is entered via the addEquation and subsequently via the addNode method. For compound terms lines and assert that l is defined for $(r(a), r(b))$. All changes to $r(a)$ or $r(b)$ must happen in union and they are reflected by matching updates to the l structure.

□

Invariant 0.0.3 (Neighbours)**.** *For every $s \in \mathcal{T}$, every $t_r \in rN(r(s))$ and $t_l \in lN(r(s))$, l is defined for $(r(s), r(t_r))$ and $(r(t_l), r(s))$.*

Proof. We show the result for the structure rN. The result about lN can be obtained similarly. Since rN is initialized with the empty set in line 4 of addNode, the invariant clearly holds initially. To show that the invariant always holds, it has to be shown that all modifications of r and rN do not change the invariant. The structure l is not modified after initialization. The structure r is modified in line 35 of union. The structure rN is modified in line 13 of addNode and line 38 of union.

Line 13 of addNode adds b to $rN(r(a))$ and the four lines before that addition show that l is defined for $(r(a), r(b))$.

Union modifies rN in such a way that it adds all right neighbors of some representative $r(v)$ to $rN(r(u))$. Lines 19 to 33 make sure that l is defined for all these right neighbors.

□

A consequence of this invariant is and the fact that the statement is true after inserting, that for every term $t \in \mathcal{T}$ of the form $f(a, b)$, l is defined for $(r(a), r(b))$.

Proposition 0.0.4 (Sound- & Completeness)**.** *Let $\mathcal{C}$ be the congruence structure obtained by adding equations $E = \langle (u_1, v_1), \ldots, (u_n, v_n) \rangle$ to the empty congruence structure. For every $s, t \in \mathcal{T}$: $E \models s \approx t$ if and only if $r(s) = r(t)$.*

Proof. **Completeness**

We show that from $E \models s \approx t$ follows $r(s) = r(t)$ by induction on n.

Base case $n = 1$: $E \models s \approx t$ implies either $s = t$ or $\{u_1, v_1\} = \{s, t\}$. In the first case $r(s) = r(t)$ is trivial. In the second case, the claim follows from the fact that, when (u_1, v_1) is entered, union is called with arguments s and t. After this operation $r(s) = r(t)$.

Induction hypothesis: For every sequence of equations E_n with n elements and every $s, t \in \mathcal{T}_{E_n}$: $E_n \models s \approx t$ then $r(s) = r(t)$.

Induction step: Let $E = \langle (u_1, v_1), \ldots, (u_{n+1}, v_{n+1}) \rangle$ and $E_n = \langle (u_1, v_1), \ldots, (u_n, v_n) \rangle$. There are two cases: $E_n \models s \approx t$ and $E_n \not\models s \approx t$. In the former case, the claim follows from the induction hypothesis, the invariant class and the fact that union always changes representatives for all elements of a class. We still have to show the claim in the latter case. We write $E \models_n u \approx v$ as an abbreviation for $E_n \not\models u \approx v$ and $E \models u \approx v$. We show the claim by induction on the structure of the terms s and t.

Base case: s or t is a constant and therefore the transitivity reasoning was used to derive $E \models_n s \approx t$. In other words, there are l terms $t_1, \ldots, t_l$ such that $s = t_1$, $t = t_l$ and for all $i = 1, \ldots, l-1 : E \models_n t_i \approx t_{i+1}$. We prove by yet another induction on l that $r(t_1) = r(t_l)$. Base case $l = 2$. It has to be the case (up to swapping u_{n+1} with v_{n+1}), that $E_n \models s \approx u_{n+1}$ and $E_n \models t \approx v_{n+1}$, and the outmost induction hypothesis implies $r(s) = r(u_{n+1})$ and $r(t) = r(v_{n+1})$. Therefore it follows from Invariant Class, that after the call to union for (u_{n+1}, v_{n+1}) it is the case that $r(t_1) = r(t_2)$. Suppose that the claim holds for some $l \in \mathbb{N}$. In the induction step, going from l to $l+1$, the claim follows from a simple application of the transitivity axiom, since $t_1, \ldots, t_l$ and $t_2, \ldots, t_{l+1}$ are both sequences of length l.

For the induction step of the term-structure induction, suppose that $s = f(a,b)$ and $t = f(c,d)$. There are two cases such that $E \models_n s \approx t$ can be derived. Using a transitivity chain, the claim can be shown just like in the base case. Using the congruence axiom, it has to be the case that $E \models_n a \approx c$ and $E \models_n b \approx d$ (in fact one of those can also be the case without the n index). The terms a, b, c, d are of lower structure than s and t. Therefore it follows from the induction hypothesis that $r(a) = r(c)$ and $r(b) = r(d)$. The Invariants Neighbour and Lookup imply that either $r(s) = r(t)$ or (s,t) is added to d in line 14 or line line 28 of union. Subsequently union is called for s and t, after which $r(s) = r(t)$ holds.

Soundness

For $s = t$ the claim follows trivially. Therefore we show soundness in case $s \neq t$. We show that from $r(s) = r(t)$ follows $E \models s \approx t$ by induction on the number k of calls to union induced by adding all equations of E to the empty congruence structure for all s and t that are arguments of some call to union. The original claim then follows from invariant Class, since only union modifies the r structure and the fact that two terms are in the same class if and only if union was called for some elements in the respective classes.

Base case $k = 1$: $r(s) = r(t)$ implies $\{u_1, v_1\} = \{s, t\}$ and $E \models s \approx t$ is trivial.

Induction hypothesis: For every $l < k$, if a set of equations F induces l calls to union, then from $r(s) = r(t)$ follows $F \models s \approx t$ for all terms s, t that are arguments of some call to union.

Induction step: Suppose $E = \langle (u_1, v_1), \ldots, (u_n, v_n) \rangle$ induces k calls to union with arguments $(h_1, g_1), \ldots, (h_k, g_k)$. The subsequence $E_n = \langle (u_1, v_1), \ldots, (u_{n-1}, v_{n-1}) \rangle$ induced the first l calls to union for some $n - 1 \leq l < k$. In other words, adding (u_n, v_n) to the congruence structure induces the calls to union with arguments $(h_{k-l}, g_{k-l}), \ldots, (h_k, g_k)$. The first call to union with arguments (h_{k-l}, g_{k-l}) is either an original input equation, or a deduced equality from line 9 of `addNode`. In both cases $E \models h_{k-l} \approx g_{k-l}$, which is trivial in the former case and an application of the induction hypothesis in the latter case. Union induces additional union calls in such a way that the arguments of the additional call are on parent terms of the respective original arguments. Therefore, using induction on the structure of terms, the original induction hypothesis, Invariants Lookup and Neighbour and lines 5 to 33 of `union`, it can be shown that for all pairs (h_m, g_m) and all $m = k - l + 1, \ldots, k$ it is the case that $E \models h_m \approx g_m$.

□

Proposition 0.0.5 (Runtime)**.** *Let E be a set of equations that uses n terms. Computing the* *congruence closure with our congruence closure algorithm takes worst-case time* $O(n \log(n))$.

Proof. There are three loops in the method `union`, which are nested within the loop of `merge`. These loops are clearly the dominating factor for runtime.

Lines reverse1 and reverse2 of `union` make sure that everytime the representative of a term is changed, the size of its congruence class is doubled. The maximum size of a congruence class is n. Therefore the representative of a single term is changed maximally $\log(n)$ times overall and line 35 of `union` is not executed more than $n \log(n)$ times.

□

Algorithm 0.6: union

Input: term s
Input: term t
Output: a set of deduced equations

if $[r(s)] \geq [r(t)]$ **then**
  reverse1 $(u, v) \leftarrow (s, t)$
else
  reverse2 $(u, v) \leftarrow (t, s)$
$d \leftarrow \emptyset$
for *every* $x \in lN(r(v))$ **do**
  $l_v \leftarrow l(r(x), r(v))$
  if $r(x) = r(v)$ **then**
    $new_left \leftarrow r(u)$
  else
    $new_left \leftarrow r(x)$
  if l *is defined for* $(new\_left, r(u))$ **then**
    $l_u \leftarrow l(new\_left, r(u))$
    if $r(l_u) \neq r(l_v)$ **then**
      $d \leftarrow d \cup \{(l_u, l_v)\}$
    else
      $lN(r(v)) \leftarrow lN(r(v)) \setminus \{x\}$
  else
    $l(new\_left, r(u)) \leftarrow l_v$
for *every* $x \in rN(r(v))$ **do**
  $l_v \leftarrow l(r(v), r(x))$
  if $r(x) = r(v)$ **then**
    $new_right \leftarrow r(u)$
  else
    $new_right \leftarrow r(x)$
  if l *is defined for* $(r(u), new\_right)$ **then**
    $l_u \leftarrow l(r(u), new\_right)$
    if $r(l_u) \neq r(l_v)$ **then**
      $d \leftarrow d \cup \{(l_u, l_v)\}$
    else
      $rN(r(v)) \leftarrow rN(r(v)) \setminus \{x\}$
  else
    $l(r(u), new\_right) \leftarrow l_v$
$[r(u)] \leftarrow [r(u)] \cup [r(v)]$
for *every* $x \in [r(v)]$ **do**
  $r(x) \leftarrow r(u)$
$[r(u)] \leftarrow [r(u)] \cup [r(v)]$
$lN(r(u)) \leftarrow lN(r(u)) \cup lN(r(v))$
$rN(r(u)) \leftarrow rN(r(u)) \cup rN(r(v))$
return d

Congruence graph

For our purpose, the input equations together with deduced equalities have to be stored in a data structure that supports the production of explanations. We support two different such data structures. Both structures store equations in a labeled graph, that we call a congruence graph. The nodes represent terms and an edge between two nodes denotes that the represented terms are congruent w.r.t. the set of input equations. A path in a congruence graph is a sequence of undirected, unweighted, labeled edges in the underlying graph. The set of labels for both types of graphs is the set of extended equations $\mathcal{E}$.

Invariant 0.0.6 (Paths)**.** *For terms s, t such that $s \neq t$ and a congruence structure with representative function r holds if $r(s) = r(t)$ then there is a path in the congruence graph of the structure between s and t*

Proof. For terms s, t with equal representatives, there is a sequence of calls to union. Every union is called from `merge`, which calls `lazy_insert` right after union with the same argument. The medthod `lazy_insert` eventually adds an edge between the respective terms. Therefore the sequence of unions corresponds to the path between s and t.

□

Invariant 0.0.7 (Deduced edges)**.** *For every edge in a congruence structure between vertices u, v with label null, there are $a, b, c, d \in \mathcal{T}$ such that $u = f(a, b)$, $v = f(c, d)$ and there are paths in the underlying graphs between a and c aswell as b and d.*

Proof. Edges with label *null* are added, when `merge` is called from `addNode`, or `union` induces an additional merge. In both cases there are subterms with respective equal representatives. The claim follows by using the invariant Paths.

□

The method `explain` returns a path between its two arguments, if one exists. Depending on the actual type of graph used, this path can be unique or not. The method `inputEqs` for a path in the congruence graph returns the input equations that were used to derive the equality between the first and the last node of the path. Therefore the statement `inputEqs(explain(s,t,g),g)` returns an explanation for $E \models s \approx t$.

Equation Graph

A equation graph stores input and deduced equalities in a labeled weighted undirected graph (V, E) with $V \subseteq \mathcal{T}$, $E \subseteq V \times \mathcal{E} \times V \times \mathbb{N}$. The weight for an edge is the number of input equalities used to derive the equality between its two nodes. This number is one for input equalities and the size of the explanation for deduced equalities. Edges are added to the graph, regardless whether the nodes are already connected in the graph. Therefore there is a choice which path the explain method returns. To produce short explanations, the shortest path w.r.t. the edge weights is returned.

Finding the shortest path between two nodes in a weighted graph is not trivial. The single source shortest path problem (SSSP) is a classical graph problem in computer science. The task

Algorithm 0.7: inputEqs

Input: path p in g
Input: congruence graph g
Output: set of input equations used in p

```
Let p be (u_1, l_1, v_1), ..., (u_n, l_n, v_n)
eqs ← ∅ for i ← 1ton do
    if l_i = null then
        f(a, b) ← u_i
        f(c, d) ← v_i
        p1 ← explain(a, c, g)
        p2 ← explain(b, d, g)
        eqs ← eqs ∪ inputEqs(p1, g) ∪ inputEqs(p2, g)
    else
        eqs ← eqs ∪ {l_i}
return eqs
```

is to find the shortest path in a graph between one designated node, the source, and all other nodes in the graph. To the best knowledge of the authors, there is no algorithm to find the shortest path between two nodes which has better asymptotic runtime than one to solve SSSP. There is a whole variety of algorithms that solve SSSP. Classical algorithms for SSSP are those of Dijkstra [8] and Bellman-Ford [3, 11]. The algorithms work on different kinds of graphs. Our setting is an undirected graph with positive integer weights. We chose to use Dijkstra's algorithm, even though the algorithm does not have optimal asymptotic runtime. It's worst-case runtime is $O(n \log(n))$ [6], if the priority queue is implemented as a Fibonacci Heap, which is the case in our implementation. [16] reports of an linear time algorithm for the undirected single source shortest path with positive integer weights problem. However, the algorithm has a big overhead and needs several precomputations. [4] is an extensive study of several shortest path algorithms which shows that Dijkstra's algorithm performs well in practice.

Dijkstra's algorithm finds shortest paths to an increasing set of nodes, until every node has been discovered. It does so by keeping track of the the shortest paths and the distances, being the combined weights of edges on the path, of nodes to the source. Initially, the only discovered node is the source itself and the distance to every other node is infinite. The algorithm discovers new nodes by selecting the lowest weight outgoing edge of all nodes that have been discovered so far and updates shortest paths and distances while doing so. It is a greedy algorithm in the sense that it always locally chooses lowest weight edges and never discards previously made decisions.

The algorithm has been slightly modified to take into account decisions that are edges for deduced equalities. These edges represent explanations, which are a sets of input equations. Previously included input equations do not increase the size of the global explanation when including them again. Therefore the modified Dijkstra algorithm adds an edge with weight 0 for every input equation in the explanation of a deduced equality edge. This is done to reduce the size of explanations. Since previous decisions are not discarded, it is not guaranteed that the modified algorithm returns the shortest path in the final graph, including the extra edges. Example 0.0.1 demonstrates that the modified shortest path algorithm does not always produce the shortest explanation, but can produce shorter explanations than the unmodified version in some situations. The shortest path algorithm's inability to return shortest explanations is not surprising, since it runs in $O(n \log(n))$ and in Section **??** it was shown that finding the shortest explanation is NP-complete.

Example 0.0.1. Consider the congruence graph shown in Figure 1, where solid edges are input equation and the dashed edge marks an application of the congruence axiom. The equality of $f(c_1, e)$ and $f(c_4, e)$ was deduced using the equations $(c_1, c_2), (c_2, c_3), (c_3, c_4)$, which is the shortest path in the graph between c_1 and c_4, obtained from a previous call to the shortest path algorithm.

Suppose we want to compute an explanation for $a \approx b$. Clearly the input equalities $(a, f(c_1, e))$, $(f(c_4, e), c_1)$ and the explanation for $f(c_1, e) \approx f(c_4, e)$ have to be included in the explanation. Additionally $c_1 \approx b$ has to be explained. For this equality the set $(c_1, d_1), (d_1, d_2), (d_2, b)$ is the shortest explanation in the original graph. This sub explanation adds three new equations to the explanation for $a \approx b$. Therefore when the shortest path algorithm iterates over the edge $(f(c_1, e), f(c_4, e))$, it can add add zero weight edges $(c_1, c_2), (c_2, c_3), (c_3, c_4)$ to the graph. By doing so the shortest explanation for $c_1 \approx b$ becomes $(c_1, c_2), (c_2, c_3), (c_3, c_4), (c_4, b)$, which only adds one extra equation to the global explanation.

This method is successful in finding the shortest explanation in this example if the search begins in the node a. Should the search begin in the node b, the edges including d_1, d_2 are added to the shortest path before the edge $(f(c_1, e), f(c_4, e))$ is touched. Therefore the undesired long explanation would be returned.

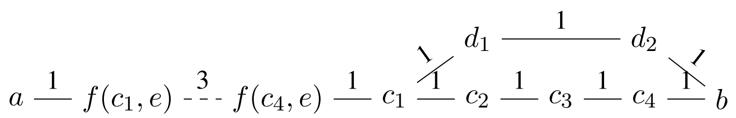

Figure 1: Short explanation example

Algorithm 0.8: insert (equation graph)

Input: term s
Input: term t
Input: equation $eq \in \mathcal{E}$
if $eq! = null$ **then**
    add edge $(s, (eq, \emptyset), t, 1)$ to g
else
    $f(a, b) \leftarrow s$
    $f(c, d) \leftarrow t$
    $p1 \leftarrow$ shortest path between a and c in g
    $p2 \leftarrow$ shortest path between b and d in g
    $w \leftarrow \#(p1.inputEqs \cup p2.inputEqs)$
    add edge $(s, (null), t, w)$

Algorithm 0.9: explain

Input: term s
Input: term t
Input: equation graph g
Output: Path in g
return shortest path between s and t in g

Proof Forest

A proof forest is a collection of proof trees. A proof tree is a labeled tree with vertices in $\mathcal{T}$ and edge labels in $\mathcal{E}$.

Algorithm 0.10: explain

Input: term s
Input: term t
Input: proof forest g
if *s and t are in the same proof tree P* **then**
    Let nca be the nearest common ancestor of s and t in P
    $p1 \leftarrow$ path from s to nca
    $p2 \leftarrow$ path from nca to s
    return $p1 :: p2$
else
    return the empty path

See how BarceLogic ppl prove stuff, -) tree is still tree after inserting -) path to NCA forms explanation

Algorithm 0.11: insert (proof forest)

Input: term s
Input: term t
Input: equation $eq \in \mathcal{E}$

if s *is not in* g **then**
    add tree with single node s
if t *is not in* g **then**
    add tree with single node t
$sSize \leftarrow$ size of tree of s
$tSize \leftarrow$ size of tree of t
if $sSize \leq tSize$ **then**
    $(u, v) \leftarrow (s, t)$
else
    $(u, v) \leftarrow (t, s)$
reverse all edges on the path between u and its root node
insert edge (v, eq, u)

Proof Production

In this section we describe how to produce proofs from explanations. The basic idea is to traverse the path corresponding to the explanation, creating a transitivity chain and keeping track of the equalities in the chain that were derived using the congruence axiom. For the congruence equalities there have to be an explanations for the arguments of the compound terms to be equal. These explanations are transformed into proof recursively and resolved with the transitivity chain. Since terms can never be equal to their subterms, the procedure will eventually terminate.

Algorithm 0.12: produceProof

```
Input: term s
Input: term t
Output: Resolution proof for s = t or null
p ← explain(s, t, g)
d ← ∅
e ← ∅
proof ← null
while p is not empty do
    (u, l, v) ← first edge of p
    p ← p \ (u, l, v)
    e ← e ∪ {u ≠ v}
    if l = null then
        f(a, b) ← u
        f(c, d) ← v
        p_1 ← produceProof(a, c, g)
        p_2 ← produceProof(b, d, g)
        con ← {a ≠ c, b ≠ d, f(a, b) = f(c, d)}
        int_1 ← resolve con with root(p_1)
        int_2 ← resolve int_1 with root(p_2)
        d ← d ∪ int_2
if #e > 1 then
    proof ← e ∪ {s = t}
    while d is not empty do
        int ← some element in d
        d ← d \ {int}
        proof ← resolve t with int
    return proof
else if d = {ded} then
    return ded
else
    if e = {(u, l, u)} then
        return {u = u}
    else
        return null
```

Congruence Compressor

In Section **??** processing of a proof was defined. The most important kind of proof processing for us is proof compression. We want to make use of the short explanations found by the congruence closure algorithm described above. To this end we replace subproofs with new proofs that have shorter conclusions. Shorter conclusions lead to the need for less resolution steps further down the proof.

The Congruence Compressor does exactly this. It is defined upon the following processing function, specified in pseudocode.

Algorithm 0.13: compress

Input: resolution node n
Input: pr : tuple of resolution nodes (p_1, p_2) or null
Output: resolution node

```
if pr = null then
    return n
else
    m ← fixNode(n, (p1, p2))
    lE ← {(a, b) | (a ≠ b) ∈ m}
    rE ← {(a, b) | (a = b) ∈ m}
    con ← empty congruence structure
    for (a, b) in lE do
        con ← con.addEquality(a, b)
    for (a, b) in rE do
        con ← con.addNode(a).addNode(b)
        proof ← con.prodProof(s, t)
        if proof ≠ null and #proof.conclusion < #m.conclusion then
            m ← proof
    return m
```

The compressor (Algorithm 15) uses the method `fixNode` to maintain a correct proof. The method modifies nodes with premises that have earlier been replaced by the compressor. Nodes with unchanged premises are not changed. Let n be a proof node that was derived by resolving pr_1 and pr_2 using pivot ℓ. It assumed that the values pr_1, pr_2 and ℓ are stored together with the node and can be accessed in constant time. Note that the method returns p_1 in case non of the new premises contains the pivot. We might as well choose p_2 to maintain obtain a correct node.

Algorithm 0.14: fixNode

Input: resolution node n
Input: pr : tuple of resolution nodes (p_1, p_2) or null
Output: resolution node

if $pr = null$ *or* $($n.premise_1 = p_1$ *and* $n.premise_2 = p_2$) **then**
  return n
else
  if $n.pivot \in p_1$ *and* $n.pivot \in p_2$ **then**
    return $resolve(p_1, p_2)$
  else if $n.pivot \in p_1$ **then**
    return p_2
  else
    return p_1

Bibliography

[1] Leo Bachmair and Ashish Tiwari. Abstract congruence closure and specializations. In *CADE*, pages 64–78, 2000.

[2] Henk Barendregt. The impact of the lambda calculus in logic and computer science. *Bulletin of Symbolic Logic*, 3(02):181–215, 1997.

[3] Richard Bellman. On a routing problem. Technical report, DTIC Document, 1956.

[4] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Math. Program.*, 73:129–174, 1996.

[5] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, pages 345–363, 1936.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1989.

[7] Haskell B. Curry. *Combinatory Logic*. Amsterdam, North-Holland Pub. Co., 1958.

[8] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

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

[10] Pascal Fontaine. *Techniques for verification of concurrent systems with invariants*. PhD thesis, PhD thesis, Institut Montefiore, Université de Liege, Belgium, 2004.

[11] Lester Randolph Ford. Network flow theory. 1956.

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

[13] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *RTA*, pages 453–468, 2005.

[14] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Inf. Comput.*, 205(4):557–580, 2007.

[15] Moses Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische Annalen*, 92(3):305–316, 1924.

[16] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.