In this chapter we present a method to compress proofs in length. The method manipulates SMT proofs of the theory of equality. To this end, in Section **??** we extend the resolution calculus presented in Section **??** to handle equality and its axioms. The proof compression method is based on the idea of replacing long explanations for the equality of two terms by shorter ones. In Section **??** we show that finding the shortest explanation is NP-complete. In Section **??** we present our explanation producing congruence closure algorithm, which is applied in the proof compression algorithm presented in Section 0.1. Closing this chapter, we give an outlook of possible future work.

0.1 Proof Production

In this section we describe how to produce resolution proofs from paths in a congruence graph. The method to carry out this operation is `produceProof`. The basic idea is to traverse the path, creating a transitivity chain of equalities between adjacent nodes, while keeping track of the deduced equalities in the chain. From Invariant Deduced Edges follows that for the deduced equalities there have to be paths between the respective arguments of the compound terms. These paths are transformed into proofs recursively and resolved with a suiting instance of the compatibility axiom. After this operation the subproof is resolved with the original transitivity chain. Since terms can never be equal to their (proper) subterms, the procedure will eventually terminate. The result of this procedure is a resolution proof with a root, such that the equations of the negative literals are an explanation of the target equality or $\emptyset$ to denote that the equality can not be proven. Suppose some equality $s \approx t$ can be explained and `produceProof` returns a proof with root ρ, then it is the case that $neg(\rho) \models s \approx t$ and $neg(\rho)$ is a subset of the input equations.

Example 0.1.1. Consider again the congruence graph shown in Figure **??** and suppose we want a proof for $a \approx b$. Suppose we found the path $p_1 := \langle a, f(c_1, e), f(c_4, e), c_1, c_2, c_3, c_4, b\rangle$ as an explanation and that the explanation for $f(c_1, e) \approx f(c_4, e)$ is the path $\langle c_1, c_2, c_3, c_4\rangle$. We transform p_1 and p_2 into instances of the transitivity axiom C_1 and C_2 respectively. The clause C_2 is resolved with the instance of the congruence axiom C_3, which is then resolved with the instance of the reflexive axiom C_4 resulting in clause C_5. Finally, C_1 is resolved with C_5 to obtain the final clause C_6. The proof is shown in Figure 0.1.

As mentioned in Section **??**, edges are inserted into a congruence graph in a lazy way by the congruence closure algorithm. The reason is that `produceProof` searches for explanations for edges with label ☺. Should the equality of question be an input equation that is added later to the congruence structure than it was deduced, then we would like to overwrite this label with the input equation. The impact of lazy insertion gets larger, if an implementation searches for explanations already when an edge is added to the graph. Example 0.1.2 shows how this technique can help producing shorter proofs.

Example 0.1.2. Suppose we want to add the following sequence of equations into an empty congruence structure: $\langle (a, b), (f(a, a), d), (f(b, b), e), (f(a, a), f(b, b))\rangle$. After adding the first three equations, the congruence closure algorithm detects the deduced equality $f(a, a) \approx f(b, b)$. The explanation for this equality is $\{(a, b)\}$, if we were to insert the edge $(f(a, a), f(b, b))$ into

Algorithm 0.1: produceProof

Input: term s
Input: term t
Output: Resolution proof for $E \models s \approx t$ or $\emptyset$

$p \leftarrow explain(s, t, g)$
$d \leftarrow \emptyset$
$e \leftarrow \emptyset$
while p *is not empty* **do**
    $(u, l, v) \leftarrow$ first edge of p
    $p \leftarrow p \setminus (u, l, v)$
    $e \leftarrow e \cup \{u \neq v\}$
    if $l = \text{☺}$ **then**
        $f(a, b) \leftarrow u$
        $f(c, d) \leftarrow v$
        $p_1 \leftarrow produceProof(a, c)$
        $p_2 \leftarrow produceProof(b, d)$
        $con \leftarrow \{a \neq c, b \neq d, f(a, b) = f(c, d)\}$
        $res \leftarrow$ resolve con with non $\emptyset$ roots of p_1 and p_2
        $d \leftarrow d \cup res$
if $\#e > 1$ **then**
    $proof \leftarrow e \cup \{s = t\}$
    while d *is not empty* **do**
        $int \leftarrow$ some element in d
        $d \leftarrow d \setminus \{int\}$
        $proof \leftarrow$ resolve $proof$ with int
    return $proof$
else if $d = \{ded\}$ **then**
    return ded
else
    if $e = \{(u, l, u)\}$ **then**
        return $\{u = u\}$
    else
        return $\emptyset$

the graph immediately, it would have weight 1 and label ☺. Depending on the congruence graph used, when adding the fourth equation $(f(a, a), f(b, b))$ to the congruence structure, either the edge $(f(a, a), f(b, b))$ is not added at all to the graph or is added with weight 1. In the latter case, both edges have weight 1 and equal chance to be selected by the shortest path algorithm. However, choosing the edge with label ☺ is undesirable, since it two extra resolution nodes (corresponding to the compatability axiom and an intermediate node).

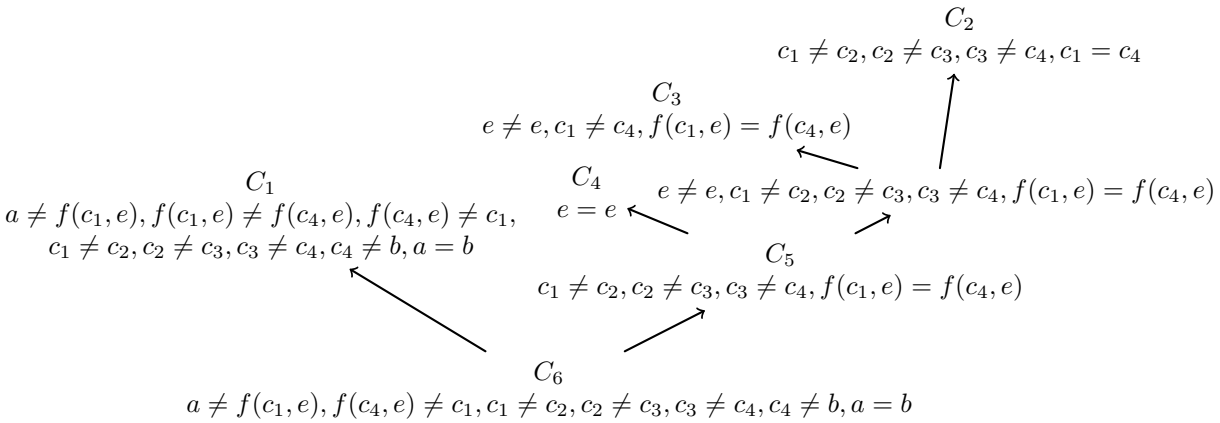

Figure 0.1: Example proof

Congruence Compressor

In this section we put our explanation producing congruence closure algorithm and the proof production method into the context of proof compression. To this end we replace subproofs with conclusions that contain unnecessary long explanations with new proofs that have shorter conclusions. Shorter conclusions lead to less resolution steps further down the proof and possibly large chunks of the proof can simply be discarded. There is however a tradeoff in overall proof length when introducing new subproofs. The subproof corresponding to a short explanation can be longer in proof length, i.e. involve more resolution nodes, than one with a longer explanation. Example 0.1.3 displays this issue. Additionally it can be the case that by introducing a new subproof, we only partially remove the old subproof. Some nodes of the old subproof might still be used in other parts of the proof. Therefore the replacement of a subproof by another, smaller one does not necessarily lead to a smaller proof. Nevertheless, our intuition is that favoring smaller conclusions should dominate such effects, especially on large proofs.

Example 0.1.3. For presentation purposes, throughout this example we will abbreviate the term $f(f(a,b), f(a,a))$ with t_a and $f(f(b,a), f(b,b))$ with t_b. Consider the set of equations $E = \{(t_a, a), (a, b), (b, t_b)\}$ and the target equality $t_a \approx t_b$. Using equations in E, one can prove the the target equality in two ways. Either one uses the instance of the transitivity axiom $\{t_a \neq a, a \neq b, b \neq t_b, t_a = t_b\}$ or a repeated applications of instances of the congruence axiom, e.g. $\{a \neq b, f(a,a) = f(b,b)\}$. The corresponding explanations are E and $\{(a, b)\}$.

The two resulting proofs are shown in Figure 0.2. The proof with the longer explanation E is only one proof node, whereas the proof with the singleton explanation has proof length 5.

The Congruence Compressor compresses processes a proof replacing subproofs as described above. It is defined upon the processing function $f : V \times V \times V \to V$ specified in pseudocode in Algorithm 13. The function $g_f : V \to V$ for axioms is simply the identity (i.e. axioms are not modified). The idea of the processing function is simple. Axioms are not changed by the function. For all other nodes the `fixNode` method is called, to maintain a correct proof.

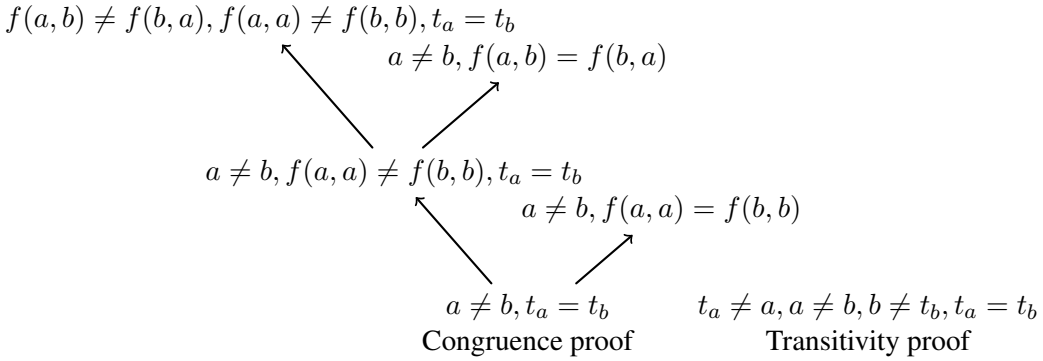

Figure 0.2: Short explanation, long proof

For a clause C, the method adds $neg(C)$ (as defined in **??**) to the empty congruence structure and checks whether these equations induce a proof for one of the equations in the $pos(C)$ that has a shorter conclusion than the original subproof. If there is such a proof, we replace the old subproof by the new one. Example **??** displays this procedure.

In line 2 it is decided whether the explanation finding congruence closure algorithm should be used to find a replacement for the current node. A trivial criteria is true for every node. Testing every node will result in a slow algorithm, but the best possible compression. Some nodes do not need to be checked, since they contain optimal explanations by definition or there is no hope of finding an explanation at all. The following definition classifies nodes to define a more sophisticated decision criteria.

Definition 0.1.1 (Types of nodes)**.** An axiom is a *theory lemma* if it is an instance of one of the congruence axioms. Otherwise it is *input derived*. The classification of internal nodes is defined recursively. An internal node is input derived, if one of its premises is input derived. Otherwise it is a theory lemma. We call a node a *low theory lemma* if it is a theory lemma and has a child that is input derived.

We suspect that most redundancies in proofs are to be found in low theory lemmas, since they reflect the explanations found by the proof producing solver. Therefore an alternative criteria is to only find replacements for low theory lemmas. The question whether a node is a low theory lemma is not trivial to answer while traversing the proof in a top to bottom fashion. Therefore a preliminary traversal is necessary to determine the classification of nodes. Further criteria for deciding whether or not to replace could be size of the subproof or a global metric that tries to predict the global compression achieved by replacement.

The compressor (Algorithm 13) 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 using pivot ℓ in the original proof and which updated premises are pr_1 and pr_2 . Depending on the presence of ℓ in pr_1 and pr_2, n is either replaced by the resolvent of pr_1 and pr_2 or by one of the updated premises. In case both updated premises do not contain the original pivot element, replacing the

Algorithm 0.2: compress

Global: set of input equations E
Input: resolution node n
Input: pr : tuple of resolution nodes (p_1, p_2)
Output: resolution node

$m \leftarrow fixNode(n, (p_1, p_2))$
if *m fulfills criteria* **then**
  $lE \leftarrow \{(a, b) \mid (a \neq b) \in m\}$
  $rE \leftarrow \{(a, b) \mid (a = b) \in m\}$
  $con \leftarrow$ empty congruence structure
  for (a, b) *in* lE **do**
    $con \leftarrow con.addEquality(a, b)$
  for (a, b) *in* rE **do**
    $con \leftarrow con.addNode(a).addNode(b)$
    $proof \leftarrow con.prodProof(s, t)$
    if $proof \neq \emptyset$ *and* $|proof.conclusion| < |m.conclusion|$ **then**
      $m \leftarrow proof$
return m

node by either one of them maintains a correct proof. Since we are interested in short proofs, we return the one with the shorter clause. This method of maintaining a correct proof was proposed in [1] in the context of similar proof compression algorithms.

Algorithm 0.3: fixNode

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

if *($n.premise_1 = p_1$ 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 if $n.pivot \in p_2$ **then**
    return p_1
  else
    return node with smaller clause

Example 0.1.4. Consider the proof presented graphically in Figure 0.3. It uses the same abbreviations for t_a and t_b as in Example 0.1.3. Furthermore, the proof uses the long explanation for

$t_a \approx t_b$. The length of the proof is 12. The proof contains one propositional variable A. The compression algorithm traverses the proofs and detects the redundant explanation in node O_1. The subproof N_1 corresponding to the explanation $\{(a, b)\}$ is created and O_1 is replaced by it. The construction of this subproof is discussed in Example 0.1.3. When iterating over node O_2, the algorithm detects that the pivot literal $t_a = a$ is not present in N_1 and `fixNode` replaces O_2 by N_1. At node O_3, both premises contain the pivot $a = b$, therefore O_3 is replaced by resolvent of N_1 and its other original premise. No other subproof is altered by the algorithm. The resulting proof is displayed in Figure 0.4 and has length 11 which is shorter than the original one, even though the replaced subproof is larger. Note that the clause $\{\neg A\}$ is part of the replaced subproof. It is also part of the subproof with conclusion $\{a = b\}$, which remains in the new proof.

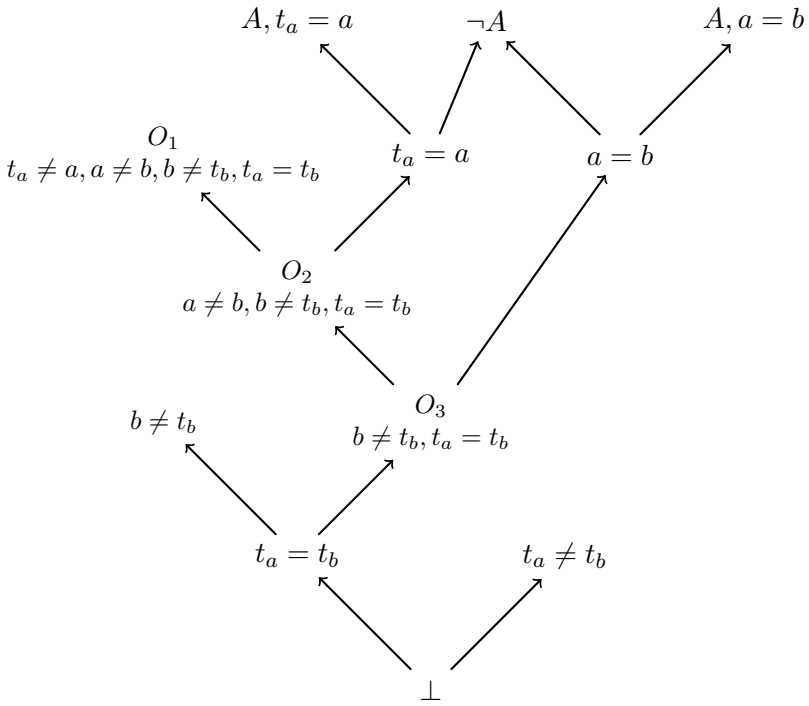

Figure 0.3: Original Proof

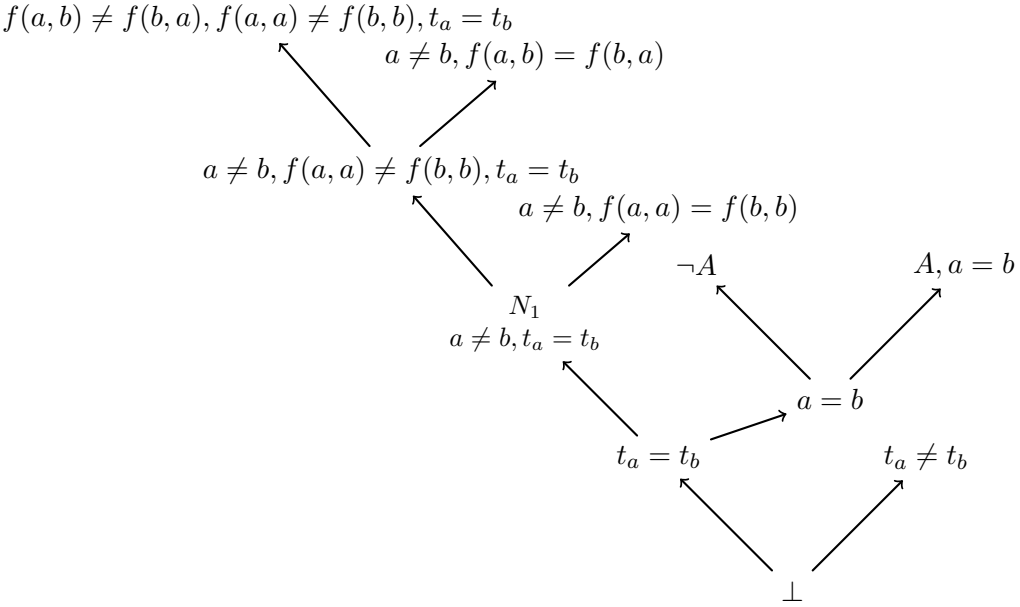

Figure 0.4: Compressed Proof

Bibliography

[1] Omer Bar-Ilan, Oded Fuhrmann, Shlomo Hoory, Ohad Shacham, and Ofer Strichman. Linear-time reductions of resolution proofs. In *Haifa Verification Conference*, pages 114–128, 2008.