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 `prodcureProof`. 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 proof recursively and resolved with a suiting instance of the congruence axiom. Afterwards the subproof is resolved with the original transitivity chain. Since terms can never be equal to their 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. In other words, let $s \approx t$ be the equality to be explained and suppose `produceProof` returns a proof with root ρ. Then for ρ it is the case that $F := \{(u, v) \mid u \neq v \text{ is a literal in } \rho\} \models s \approx t$ and F is a subset of the input equations.

I am not sure yet at what point I should introduce resolving against input equations. This could be done either here or at the compressor algorithm

Example 0.0.1. 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 and after resolving with all input equations, we obtain C_7.

$$
\begin{aligned}
C_1 &:= \{a \neq f(c_1, e), f(c_1, e) \neq f(c_4, e), f(c_4, e) \neq c_1, c_1 \neq c_2, c_2 \neq c_3, c_3 \neq c_4, c_4 \neq b, a = b\}\\
C_2 &:= \{c_1 \neq c_2, c_2 \neq c_3, c_3 \neq c_4, c_1 = c_4\}\\
C_3 &:= \{e \neq e, c_1 \neq c_4, f(c_1, e) = f(c_4, e)\}\\
C_4 &:= \{e = e\}\\
C_5 &:= \{c_1 \neq c_2, c_2 \neq c_3, c_3 \neq c_4, f(c_1, e) = f(c_4, e)\}\\
C_6 &:= \{a \neq f(c_1, e), f(c_4, e) \neq c_1, c_1 \neq c_2, c_2 \neq c_3, c_3 \neq c_4, c_4 \neq b, a = b\}\\
C_7 &:= \{a = b\}
\end{aligned}
$$

This could possibly be shown graphically more nicely

Algorithm 0.1: produceProof

Input: term s
Input: term t
Output: Resolution proof for $s = t$ or $null$

```
1  p ← explain(s, t, g)
2  d ← ∅
3  e ← ∅
4  proof ← null
5  while p is not empty do
6      (u, l, v) ← first edge of p
7      p ← p \ (u, l, v)
8      e ← e ∪ {u ≠ v}
9      if l = null then
10         f(a, b) ← u
11         f(c, d) ← v
12         p1 ← produceProof(a, c, g)
13         p2 ← produceProof(b, d, g)
14         con ← {a ≠ c, b ≠ d, f(a, b) = f(c, d)}
15         int1 ← resolve con with root(p1)
16         int2 ← resolve int1 with root(p2)
17         d ← d ∪ int2
18 if #e > 1 then
19     proof ← e ∪ {s = t}
20     while d is not empty do
21         int ← some element in d
22         d ← d \ {int}
23         proof ← resolve t with int
24     return proof
25 else if d = {ded} then
26     return ded
27 else
28     if e = {(u, l, u)} then
29         return {u = u}
30     else
31         return null
```

Congruence Compressor

In Section **??** processing of a proof was defined. The most important application of proof processing for this work 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 less resolution steps

further down the proof. There is however a tradeoff in overall proof length when introducing new subproofs. The subproof with a shorter conclusion can still be longer, i.e. involve more resolution nodes, than one with a bigger conclusion.

Example 0.0.2. This example is not finished, it should demonstrate the long transitivity chain produces short proof issue; possibly the example could also be made so it shows that un dag-ifying a proof englarges it, i.e. replacing nodes with new nodes, where the old one remains in the proof. Consider the set of equations $E = \{(f(f(a,b),f(a,a)),a),(a,b),(b,f(f(b,a),f(b,b)))\}$ and the target equality $f(f(a,b),f(a,a)) \approx f(f(b,a),f(b,b))$. Using equations in E, one can prove the the target equality in two ways. Either one uses the instance of the transitivity axiom $\{f(f(a,b),f(a,a)) \not\approx a, a \not\approx b, b \not\approx f(f(b,a),f(b,b)), f(f(a,b),f(a,a)) = f(f(b,a),f(b,b))\}$ or a repeated applications of instances of the congruence axiom, e.g. $\{a \not\approx b, f(a,a) = f(b,b)\}$. The corresponding explanations are E and $\{(a,b)\}$.

The two resulting proofs are shown in Figure 1 and 2. The proof with the longer explanation E has 7 proof nodes, whereas the proof with the singleton explanation has proof length 9.

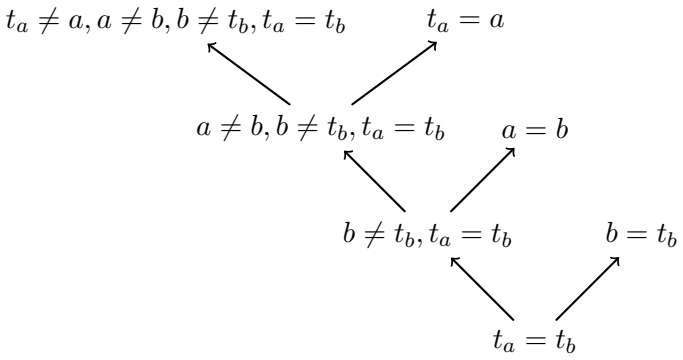

Figure 1: Long explanation, short proof

The Congruence Compressor compresses processes a proof replacing subproofs as described above. It is defined upon the following processing function, specified in pseudocode. Input equations are assumed to be true, i.e. an input equation (u, v) is treated as an axiomatic clause $\{u = v\}$. In a last step of `prodProof` all input equations are resolved away.

The compressor (Algorithm 18) 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.

I need to prove that a correct proof is maintained with fixnode and the whole compressor algorithm.

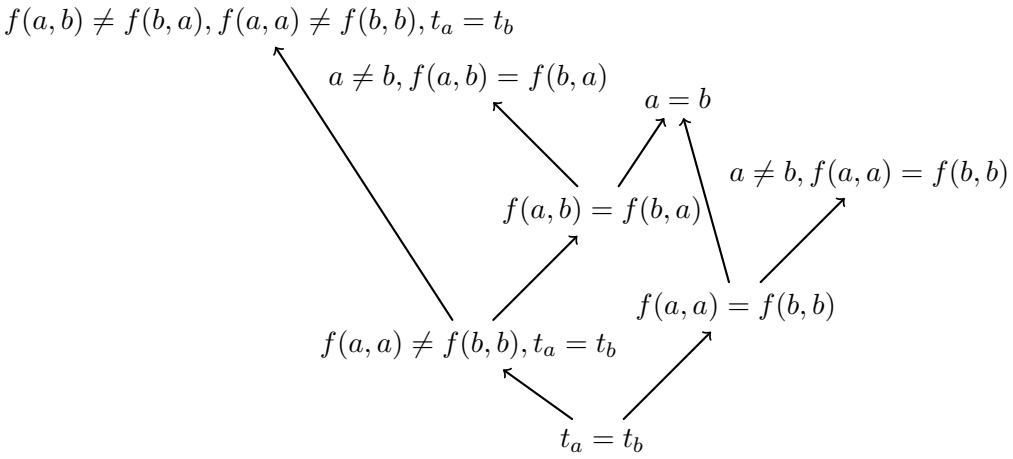

Figure 2: Short explanation, long proof

Algorithm 0.2: compress

Set of input equations E **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
  $m \leftarrow fixNode(n, (p_1, p_2))$
  $lE \leftarrow \{(a,b) \mid (a \neq b) \in m\}$
  $rE \leftarrow \{(a,b) \mid (a = b) \in m\}$
  $con \leftarrow$ empty congruence structure
  for (a,b) *in* lE **do**
    $con \leftarrow con.addEquality(a,b)$
  for (a,b) *in* rE **do**
    $con \leftarrow con.addNode(a).addNode(b)$
    $proof \leftarrow con.prodProof(s,t)$
    while *Root of* $proof$ *has literal* $u \neq v$*, s.t.* $(u,v) \in E$ **do**
      $proof \leftarrow$ resolve $proof$ with $\{u = v\}$
    if $proof \neq null$ *and* $\#proof.conclusion < \#m.conclusion$ **then**
      $m \leftarrow proof$
  return m

Algorithm 0.3: 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 ∈ p_1 and n.pivot ∈ p_2 then
        return resolve(p_1, p_2)
    else if n.pivot ∈ p_1 then
        return p_2
    else
        return p_1
```

Bibliography