DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Formalisation of a Congruence Closure Algorithm in Isabelle/HOL

Rebecca Ghidini

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Formalisation of a Congruence Closure Algorithm in Isabelle/HOL

Formalisierung eines Kongruenzhüllen-Algorithmus in Isabelle/HOL

Author:	Rebecca Ghidini
Supervisor:	Prof. Dr. Tobias Nipkow
Advisor:	Lukas Stevens
Submission Date:	15.09.2022

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.09.2022 Rebecca Ghidini

Acknowledgments

Thanks to Timmm and Manon.

Abstract

Contents

1 Introduction

1.1 Outline

Citation test [Lam94].

```
apply(simp)
apply(auto)
done
```

Figure 1.1: An example for a source code listing.

2 Preliminaries

2.1 Union Find with Explain Operation

2.2 Congruence Closure with Explain Operation

2.3 Isabelle/HOL

2.3.1 Union Find in Isabelle

3 Explain Operation for Union Find

3.1 The Union Find Data Structure

In this section I will present th implementation of the modified Union Find data structure, as well as the *Explain* operation and its correctness proof, as described in [NO05].

The data structure for the Union, Find and Explain operations consists of the following three lists:

- `uf_list`: This is the usual union-find list, which contains the parent node of each element in the forest data structure. It is the one described in Section idk.
- `unions`: This list simply contains all the pairs of input elements.
- `au`: This is the *associated unions* list, it contains for each edge in the union-find forest a label with the union that corresponds to this edge. Similarly to the `uf_list`, it is indexed by the element, and for each element e which has a parent in the `uf_list`, `au` contains the input equation which caused the creation of this edge between e and its parent. The equations are represented as indexes in the `unions` list. The type of the entries is `nat option`, so that for elements without a parent, the `au` entry is None.

Example 1. For a union-find algorithm with 4 variables, the initial empty union find looks as follows:

```
(uf_list = [0, 1, 2, 3], unions = []), au = [None, None, None, None]
```

Each element is its own parent in the `uf_list`, which means that it is a root, the `unions` list ist empty because no unions were made yet, and there are no edges in the tree, therefore there are no labels in `au`.

In order to reason about paths in the union-find forest, I defined the following `path` predicate.

```
inductive path :: "nat list => nat => nat list => nat => bool" where
single: "n < length l ==> path l n [n] n" |
```

```
step: "r < length l ==> l ! u = r ==> l ! u != u ==> path l u p v
       ==> path l r (r # p) v"
```

`path l r p v` defines a path from r to v, where r is closer to the root, and p contains all the nodes visited on the path from r to v. This definition proved to be very useful for many proofs, as will become clearer later in this thesis.

I proved many lemmas about paths, including lemmas about concatenation of adjecent path, and division of one path into two subpaths, and that the length of a path is at least 1, as well as many others, many of which could be proven by rule induction on `path`. The most interesting and useful lemma was about the unicity of paths between two nodes:

```
theorem path_unique: "ufa_invar l ==> path l u p1 v ==> path l u p2 v
       ==> p1 = p2"
```

Proof. The lemma is proven by induction on the length of $p1$.

For the base case we assume that the length of $p1$ is 1. There is only one node in the path, therefore $v = u$. Then I proved a lemma which showed that if the `ufa_invar` holds, each path from v to v has length 1, or, in other words, there are no cycles in the graph. For this I showed that if there was a cycle, the function `rep_of` would not terminate, because there would be an infinite loop.

For the induction step, we assume that the length of $p1$ is greater than 1. Therefore, we can remove the last node from $p1$ and the last node from $p2$ to get two paths from u to the parent of v, where the first one is shorter that $p1$, and we can apply the induction hypothesis, which tells us that the two paths are equal. Adding the node v to those two paths gives us back the original paths $p1$ and $p2$, therefore we conclude that $p1 = p2$. □

I was also able to prove that two paths of the same length which end at the same node are equal.

```
lemma path_unique_if_length_eq:
assumes "path l x p1 v"
and "path l y p2 v"
and "ufa_invar l"
and "length p1 = length p2"
shows "p1 = p2 and x = y"
```

Proof. This lemma was shown by rule induction on path.

For the base case I proved a lemma that shows that each path of length 1 is of the form p l n [n] n, using rule inversion.

Then each time a node is added to the beginning of the path, there is only one possibility to add a node, namely its parent in the list. □

3.2 Implementation

3.2.1 Union

The *union* operation was already implemented for the `uf_list` in the theory `Union_Find` [LM12] (chapter 18, Union-Find Data-Structure), it only needed to be extended in order to appropiately update the other two lists:

```
fun ufe_union :: "ufe_data_structure => nat => nat => ufe_data_structure"
where
"ufe_union (uf_list = l, unions = u, au = a) x y = (
if (rep_of l x != rep_of l y) then
(uf_list = ufa_union l x y,
unions = u @ [(x,y)],
au = a[rep_of l x := Some (length u)])
else (uf_list = l, unions = u, au = a))"
```

Example 2. After a union of 0 and 1, the data structure from Example 1 looks as follows:

```
(uf_list = [1, 1, 2, 3], unions = [(0, 1)]), au = [Some 0, None, None, None]
```

This means that there is an edge between 1 and 0, labeled with the union at index 0, which is $(0,1)$.

The algorithm only modifies the data structure if the parameters are not already in the same equivalence class. The union find tree is modified with the `ufa_union` from the theory `Union_Find`[LM12]. The current union (x,y) is added at the end of the `unions` list. `au` is updated such that the new edge between `rep_of l x` and `rep_of l y` is labeled with the last index of `unions`, which contains the current pair of elements (x,y). —

```
fun apply_unions::"(nat * nat) list => ufe_data_structure => ufe_data_structure"
where
"apply_unions [] p = p" |
"apply_unions ((x,y)#u) p = apply_unions u (ufe_union p x y)"
```

has a list of pairs as parameters, and applies for each of the pairs x,y the union, starting from an initial data structure p

3.2.2 Helper Functions for Explain

```
where
ath_to_root l x = (if l ! x = x then [x] else path_to_root l (l ! x) @ [x])"
 pat_completeness auto
```

computes the path from the root to x

It was easy to show that it has the same domain as the `rep_of` function, as it has the same recursive calls/case distinctions.

```
lemma path_to_root_domain: "rep_of_dom (l, i) <--> path_to_root_dom (l, i)"
```

the correctness follows easily by induction

```
theorem path_to_root_correct:
assumes "ufa_invar l"
and "x < length l"
shows "path l (rep_of l x) (path_to_root l x) x"

text Finds the lowest common ancestor of x and y in the
tree represented by the array l.
fun lowest_common_ancestor :: "nat list  nat  nat  nat"
where
"lowest_common_ancestor l x y =
last (longest_common_prefix (path_to_root l x) (path_to_root l y))"
```

uses `longest_common_prefix` from HOL-Library.Sublist [cit]. It is a basic algorithm that computes both paths from the root to the two nodes, and chooses the last element these two paths have in common. There are probably more efficient versions which can be used in the refinement.

For this I defined an abstract definition of `lowest_common_ancestor` and proved that it is equivalent to `lowest_common_ancestor`. The most useful result is the proof, that `lowest_common_ancestor` is a common ancestor, aka there is a path from the ancestor to x and to y. I also proved, that any other node which has a path to x and to y, aka which is a common ancestor, has a shorter distance from the root. There is an assumption that x and y are in the same eq. class, otherwise it shouldn't be invoked with the parameters. For the proof, I used the fact that `path_to_root` is a path from the root to x respectively y, and I used lemmas about splitting paths, which resulted in the following: path l lcp lca and path l l lca x bzw y. This shows that lca is a common ancestor. For the minimality, I proved it by contradiction. If there was a common ancestor with a longer path from the root, then we can show that there is a path from root to x passing through ca, and the same for y. Because of the uniqueness of paths, these paths are equal to `path_to_root x` bzw y. But if ca had a longer path from the root, we can show that there is a longer common prefix than lcp. It was already proven in Sublist[cit] that it is not possible.

```
text Finds the newest edge on the path from x to y
(where y is nearer to the root than x).
function (domintros) find_newest_on_path :: "nat list  nat option list  nat  nat  nat opti
where
"find_newest_on_path l a x y =
(if x = y then None
else max (a ! x) (find_newest_on_path l a (l ! x) y))"
by pat_completeness auto
```

This function terminates, if there is a path from y to x

```
lemma find_newest_on_path_domain:
"path l y p x  find_newest_on_path_dom (l, a, x, y)"
this is easily shown by induction.
```

I defined `Newest_on_path` as the maximal value in a for indexes in p.

```
abbreviation "Newest_on_path l a x y newest
 p . path l y p x  newest = (MAX i  set [1..<length p]. a ! (p ! i))"

theorem find_newest_on_path_correct:
assumes "path l y p x" "x  y"
shows "Newest_on_path l a x y (find_newest_on_path l a x y)"
```

proof computation induction on `find_newest_on_path` pretty easily shown

3.2.3 Explain

3.3 Proofs

3.3.1 Invariant and Induction Rule

3.3.2 Termination Proof

3.3.3 Correctness Proof

4 Congruence Closure with Explain Operation

4.1 Implementation

For the implementation of the congruence closure algorithm, I followed the implementation described in the paper. [NO05]

4.1.1 Modified Union Find Algorithm

In order to implement an explain operation with reasonble runtime for the congruene closure data structure, the paper [NO05] introduced an alternative union find algorithm. The find algorithm remains the same, but a new data structure is introduced, called the proof forest, namely a forest which has as nodes the variables, and as edges the unions that were made. The forest structure is preserved, because redundant unions are ignored.

add_edge

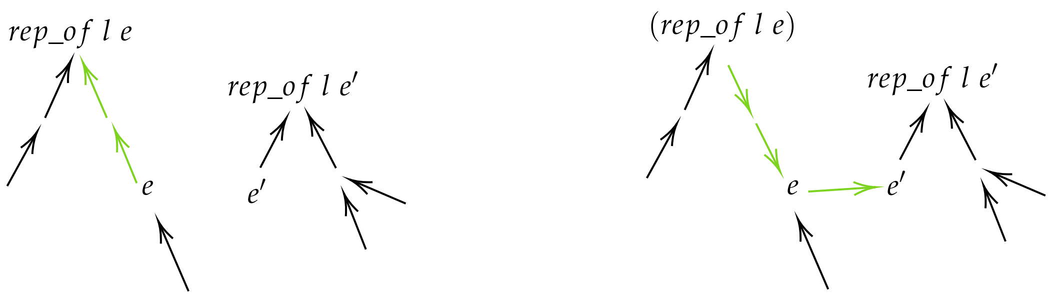
The tree has directed edges, and for each equivalence class there is a representative node, where all the edges are directed towards. To keep this invariant, each time and edge from e to e′ is added, all the edges on the path from the root of to e are reversed. In my implementation, the forest is represented by an array which stores the parent of each node, exactly as in the union find array. My implementation for each added edge is the following.

```
function (domintros) add_edge :: "nat list => nat => nat => nat list"
where
"add_edge pf e e' = (if pf ! e = e
                      then (pf[e := e'])
                      else add_edge (pf[e := e']) (pf ! e) e)"
by pat_completeness auto
```

I was able to show that the `add_edge e e'` terminates, if the `ufa_invar` holds for the proof forest and e and e' do not belong to the same equivalence class.

```
lemma add_edge_domain:
assumes "ufa_invar l" "rep_of l y != rep_of l y'"
shows "add_edge_dom (l, y, y')"
```

Proof. I proved it by induction on the length of the path p from the root of y to y. The base case is when there is only one node in the path, therefore y must be equal to its representative, therefore `pf ! y = y`, and the algorithm terminates immediately. On the other hand, if y is not a root, there is a path p′ from the root to the parent of y which is shorter than the path from the root to y. Given that only the y is modified in the recursive step, and y is not on the path p', the path p′ is also present in the updated union find list. Also, the representative of y in the new list is equal to the representative of y′, and the representative of the parent of y is still the old representative of y, therefore they are not in the same representative class, and we can apply the induction hypothesis and conclude that the recursive call terminates, therefore the function terminates. □



add_label

Additionally, each edge is labeled with the input equation or the input equations which caused the adding of this edge. This step is not necessaary for the union find algorithm by itself, but only for this algorithm when it is used within the congruence closure algorithm, because there are two possible reasons for the union of two elements a and b: either an equation $a = b$ was input, or two equations of the type $F(a_1, a_2) = a$ $F(b_1, b_2) = b$, where a1 and b1 bzw a2 and b2 were already in the same equivalence class before this union. Therefore we need to store the information about these input equations, in order to reconstruct the explanation in the end via the explain function. I implemented the labeling by using an additional list, which at each index contains the label of the outgoing edge, or `None` if there is no outgoing edge. The type of the label is `pending_equation`, which can be either `One equation` or `Two equation equation`, aka one or two equations. The name `pending_equation` derived from the fact that they are also the elements of the pending list, which is going to be described in the next section. Theoretically this allows also for invalid equations for example two equations of the type $a = b$ and $c = d$, but we will prove in the next sections, that the equations in the labels list are always of a valid type.

Each time an edge, gets added to the proof forest, the labels need to be updated as well, not only the labels of the new edge, but also of the outgoing edges. The function which implements this is the following:

```
function (domintros) add_label :: "pending_equation option list => nat list => nat
=> pending_equation => pending_equation option list"
  where
"add_label pfl pf e lbl = (if pf ! e = e
              then (pfl[e := Some lbl])
              else add_label (pfl[e := Some lbl]) pf (pf ! e) (the (pfl ! e)))"
by pat_completeness auto
```

Similarly to the `path_to_root` function, `add_label` has the same recursive calls/case distinctions as `rep_of`, therefore it has the same domain.

```
lemma rep_of_dom_iff_add_label_dom: "rep_of_dom (pf, y) <-->
add_label_dom (pfl, pf, y, y')"
```

4.1.2 Congruence Closure Data Structure

4.1.3 Congruence Closure Algorithm

4.2 Correctness Proof

4.2.1 Invariants

4.2.2 Abstract Formalisation of Congruence Closure

4.2.3 Correctness

4.3 Implementation of the Explain Operation

5 Conclusion

5.1 Future work

List of Figures

Bibliography

[Lam94] L. Lamport. *LaTeX : A Documentation Preparation System User's Guide and Reference Manual*. Addison-Wesley Professional, 1994.

[LM12] P. Lammich and R. Meis. "A Separation Logic Framework for Imperative HOL." In: *Archive of Formal Proofs* (Nov. 2012). `https://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html`, Formal proof development. ISSN: 2150-914x.

[NO05] R. Nieuwenhuis and A. Oliveras. "Proof-Producing Congruence Closure." In: *Elsevier* (2005).