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
```

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

The section below describes the 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 TODO.
- `unions`: This list simply contains all the pairs of input elements in chronological order.
- `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 is 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, we define 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 an ancestor of v, which means that it 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.

The theory `Path` contains many lemmas about paths, including lemmas about concatenation of adjacent paths, and splitting of one path into two subpaths, and that the length of a path is at least 1, as well as 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 $p1 = p2$. □

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:

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 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 $(0, 1)$.

Next, we define a function which takes a list of unions as parameter and simply applies each of those unions to the data structure. This will be needed for the invariant and the correctness proof in the next sections.

```
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)"
```

3.2.2 Helper Functions for Explain

The explain function is based on other functions, which will be described in the following pages.

path_to_root

The function path_to_root l x computes the path from the root of x to the node x in the union-find forest l. It simply starts at x and continues to add the parent of the current node to the path, until it reaches the root.

```
function path_to_root :: "nat list ⇒ nat ⇒ nat list"
where
"path_to_root l x = (if l ! x = x then [x] else path_to_root l (l ! x) @
   [x])"
by pat_completeness auto
```

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

```
lemma path_to_root_domain: "rep_of_dom (l, i) ⟷ path_to_root_dom (l, i)
   "
```

The correctness of the function follows easily by induction.

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

lowest_common_ancestor

The function `lowest_common_ancestor l x y` finds the lowest common ancestor of x and y in the union-find forest l.

Definition. A *common ancestor* of two nodes x and y is a node which has a path to x and a path to y. The *lowest common ancestor* of two nodes x and y is a node common ancestor where its path to the root has maximal length.

The function will only be used for two nodes which have the same root, otherwise there is no common ancestor. It first computes the paths from x and y to their root, and then returns the last element which the two paths have in common. For this it uses the function `longest_common_prefix` from HOL-Library.Sublist[NW].

```
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))"
```

Regarding the correctness proof, there were two aspects to prove: the most useful result is that `lowest_common_ancestor l x y` is a common ancestor of x and y. The second aspect stated that any other common ancestor of x and y has a shorter distance from the root. The proof assumes that that x and y have the same root.

Proof. Let lca =`lowest_common_ancestor l x y`. We previously proved that `path_to_root` computes a path p_x from the root to x and p_y from the root to y. Evidently, lca lies on both paths, because it is part of their common prefix. Splitting the paths, we get a path from the root to *lca* and one from *lca* to x, and the same for y. This shows that *lca* is a common ancestor.

To prove that it is the *lowest* common ancestor, we can prove it by contradiction. If there was a common ancestor lca_2 with a longer path from the root than *lca*, then we can show that there is a path from the root to x passing through lca_2, and the same for y. Because of the uniqueness of paths, these paths are equal to `path_to_root l x` and `path_to_root l y`, respectively. That means, that there is a prefix of `path_to_root l x` and `path_to_root l y` which is longer than the one calculated by the function `longest_common_prefix`. The theory Sublist[NW] contains a correctness proof for `longest_common_prefix`, which we can use to show the contradiction. □

find_newest_on_path

The function `find_newest_on_path` finds the newest edge on the path from y to x. It is assumed that y is an ancestor of x. The function simply checks all the elements on the path from y to x and returns the one with the largest index in a, which represents the associated unions list.

```
function (domintros) find_newest_on_path  :: "nat list ⇒ nat option list
   ⇒ nat ⇒ nat ⇒ nat option"
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
```

If there is a path p from y to x, it is easily shown by induction that the function terminates.

```
lemma find_newest_on_path_domain:
assumes "ufa_invar l"
and "path l y p x"
shows "find_newest_on_path_dom (l, a, x, y)"
```

For the correctness proof we define an abstract definition of the newest element on the path: `Newest_on_path` is the maximal value in the associated unions list 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))"
```

Then it can easily be shown by computation induction on `find_newest_on_path` that our function is correct.

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

3.2.3 Explain

We implement the explain function following the description of the first version of the union-find algorithm in the paper[NO05].

The explain function takes as parameter two elements x and y and calculates a subset of the input unions which explain why the two given variables are in the same equivalence class. If we consider the graph which has as nodes the elements and as edges the input unions, then the output of explain would be all the unions on the path from x to y. However, the union-find forest in our data structure does not have as edges the unions, but only edges between representatives of the elements of the unions.

From this graph, we can calculate the desired output in the following way: first add the last union (a, b) made between the equivalence class of x and the one of y, then recursively call the explain operation with the new parameters (x, a) and (b, y) (or (x, b) and (a, y), depending on which branch a and b are on). The newst union is the label of the newest edge in the union-find forest.

(a, b) is calculated by finding the lowest common ancestor lca of x and y, and then finding the newest union on the path from x to lca and from y to lca. There is a case distinction at the end to account for the cases that the newest union is on same branch as x or as y.

TODO example

```
function (domintros) explain :: "ufe_data_structure ⇒ nat ⇒ nat ⇒ (nat *
    nat) set"
where
"explain ⦇uf_list = l, unions = u, au = a⦈ x y =
(if x = y ∨ rep_of l x ≠ rep_of l y then {}
else
(let lca = lowest_common_ancestor l x y;
newest_index_x = find_newest_on_path l a x lca;
newest_index_y = find_newest_on_path l a y lca;
(ax, bx) = u ! the (newest_index_x);
```

```
(ay, by) = u ! the (newest_index_y)
in
(if newest_index_x ≥ newest_index_y then
{(ax, bx)} ∪ explain ⦇uf_list = l, unions = u, au = a⦈ x ax
∪ explain ⦇uf_list = l, unions = u, au = a⦈ bx y
else
{(ay, by)} ∪ explain ⦇uf_list = l, unions = u, au = a⦈ x by
∪ explain ⦇uf_list = l, unions = u, au = a⦈ ay y)
)
)"
by pat_completeness auto
```

3.3 Proofs

This section introduces an invariant for the union find data structure and proves that the .explain function terminates and is correct, when invoked with valid parameters.

3.3.1 Invariant and Induction Rule

The validity invariant of the data structure expresses that the data structure derived from subsequent unions with `ufe_union`, starting from the initial empty data structure. It also states that the unions were made with valid variables, i.e. varibles which are in bounds.

```
abbreviation "ufe_invar ufe ≡
valid_unions (unions ufe) (length (uf_list ufe)) ∧
apply_unions (unions ufe) (initial_ufe (length (uf_list ufe))) = ufe"
```

With this definition, it is easy to show that the invariant holds after a union.

```
lemma union_ufe_invar:
assumes "ufe_invar ufe"
shows "ufe_invar (ufe_union ufe x y)"
```

It is also useful to prove that the old invariant, `ufa_invar`, is implied by the new invariant, so that we can use all the previously proved lemmas about `ufa_invar`. This is easily shown by computation induction on the function `apply_unions`, and by using the lemma from the Theory Union Find[LM12], which states that `ufa_invar` holds after having applied `ufa_union`, and proving that it holds for the initial ufe.

```
theorem ufe_invar_imp_ufa_invar: "ufe_invar ufe ⟹ ufa_invar (uf_list
   ufe)"
```

With this definition of the invariant, we can prove a new induction rule, which will be very useful for proving many properties of a union find data structure. The induction rule, called `apply_unions_induct`, has as an assumption that the invariant holds for the given data structure ufe, and shows that a certain predicate holds for ufe. The base case that needs to be proven is that it holds for the initial data structure, and the induction step is that the property remains invariant after applying a union.

```
lemma apply_unions_induct[consumes 1, case_names initial union]:
assumes "ufe_invar ufe"
assumes "P (initial_ufe (length (uf_list ufe)))"
assumes "⋀pufe x y. ufe_invar pufe ⟹ x < length (uf_list pufe) ⟹ y <
    length (uf_list pufe) ⟹ P pufe ⟹ P (ufe_union pufe x y)"
shows "P ufe"
```

This induction rule can be used for most of the proofs about explain.

3.3.2 Termination Proof

An important result was to show that the function always terminates if `ufe_invar` holds.

```
theorem explain_domain:
assumes "ufe_invar ufe"
shows "explain_dom (ufe, x, y)"
```

Proof. For the base case, we consider the empty data structure. There are no different variables with the same representative, therefore the algorithm terminates immediately.

For the induction step we need to show that if the function terminates for a data structure ufe, then it also terminates for `ufe_union ufe x y`. The lowest common ancestor and the newest index on path do not change after a union was applied. Therefore the entire algorithm is executed with exactly the same results at each intermediate step, therefore the recursive calls are equal, and they terminate by induction hypothesis. TODO □

3.3.3 Correctness Proof

TODO There are two properties which define the correctness of explain: foremost, the equivalence closure of `explain x y` should contain the pair (x, y) (we shall refer to this property as "correctness"), additionally, the elements in the output should only be equations which are part of the input (we shall refer to this property as "validity"). The proposition about the validity of `explain` looks as follows:

```
theorem explain_valid:
assumes "ufe_invar ufe"
and "xy ∈ (explain ufe x y)"
shows "xy ∈ set (unions ufe)"
```

We know from Subsection 3.3.2 that when the invariant holds, the function terminates. Therefore we can use the partial induction rule that Isabelle automatically generates for partial functions. We can prove that (a, b) is a valid union, given that it is in the unions list, for that we need to prove that the index found by `find_nearest_on_path` is in bounds.

```
lemma find_newest_on_path_Some:
assumes "path l y p x"
and "ufe_invar ⦇uf_list = l, unions = u, au = a⦈"
and "x ≠ y"
obtains k where "find_newest_on_path l a x y = Some k ∧ k < length u"
```

which follows from the following lemma, that shows that the entries in the associated union list are valid, aka less than the length of u

```
lemma au_valid:
assumes "ufe_invar ufe"
and "i < length (au ufe)"
shows "au ufe ! i < Some (length (unions ufe))"
```

It is easily proven, given that all the values that are added to au are valid.

Thus we have shown the validity of the explain function. It remains to show the correctness.

```
theorem explain_correct:
assumes "ufe_invar ufe"
and "rep_of (uf_list ufe) x = rep_of (uf_list ufe) y"
shows "(x, y) ∈ (symcl (explain ufe x y))*"
```

This was shown by computation induction on explain. For example for case x: (x, ax) ∈`(explain x ax)*` and $(bx, y))$ ∈`(explain bx y)*` and (ax, bx) ∈`explain x y` Therefore (x, y) ∈`(explain x y)*`

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

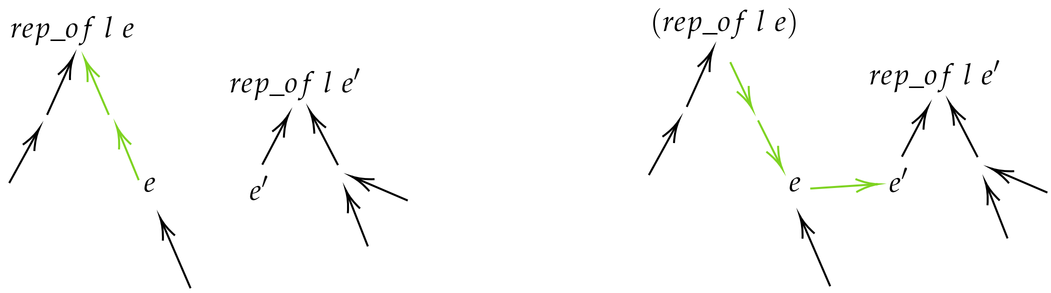
The tree has directed edges, and for each equivalence class there is a representative node, where all the edges are directed towards. To keep this invariant, each time and edge from e to e′ is added, all the edges on the path from the root of to e are reversed. In my implementation, the forest is represented 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

Proof. As usual, I left out of the assumptions the invariants, but we can assume all the previously defined invariants to hold at this point in the algorithm. There are two inclusions which need to be shown:

"$\subseteq$" This direction is trivial.

"$\supseteq$" This direction is also trivial.

□

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).

[NW] T. Nipkow and M. Wenzel. "The Supplemental Isabelle/HOL Library." In: *Isabelle/HOL sessions/HOL-Library* (). `https://isabelle.in.tum.de/library/HOL/HOL-Library/Sublist.html`.