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

Congruence Closure is already used by most theorem provers for solving decision procedures. However, the implementations of congruence closure are usually not formally proven for completeness.

This thesis describes the implementation and correctness proofs for a union-find algorithm with an explain operation and congruence closure with an explain operation, which is partly based on the implementation of the union-find algorithm. The proofs are formalized in the interactive theorem prover Isabelle/HOL. The correctness proof for the explain operation is only described, not formally proven yet. The implementation is based on the paper by Nieuwenhuis and Oliveras [3]. This result can be used in order to develop an automatic proof strategy in Isabelle, for automatic proofs of theorems.

Contents

1 Introduction

The equivalence closure of a relation, or analogously a set of equations $a = b$, with a and b being two constants, is the smallest superset of the relation which is reflexive, symmetric and transitive. Algorithms for maintaining the equivalence closure have been known for a long time [1], the most widely used being the union-find algorithm, because of its simplicity and its almost constant runtime [2].

The congruence closure of a set of equations is similar, but it considers not only constants, but also functions. Additionally to reflexivity, symmetry and transitivity, it satisfies monotonicity, i.e. $f(x_1, ..., x_n) = f(y_1, ..., y_n)$ if $x_i = y_i$ for all i between 1 and n. [3] For example the equation $c = e$ belongs to the congruence closure of $f(a, b) = c$, $f(d, b) = e$ and $a = d$. Several approaches to solve this problem have been described, with different runtimes and applications, for example [4, 5, 3].

These algorithms are used in decision procedures such as satisfiability modulo theories (SMT) solvers. [9] In these settings, it is required to generate an explanation, i.e. find the set of input equations which caused the congruence. These can then be used to generate a certificate of the congruence, which can be verified by a program. Nieuwenhuis and Oliveras have presented an efficient version of the congruence closure algorithm with an explain operation, and two versions of the union-find algorithm with an explain operation, in a conference paper [3], which was later extended, see [6].

The descriptions of the algorithms contain informal proofs, but the algorithms can be verified by an interactive theorem prover, like Isabelle, in order to strengthen our confidence in their correctness. In this thesis we will implement the algorithms of the paper [3] and prove their correctness in the theorem prover Isabelle/HOL. The description of the congruence closure algorithms contains an informal proof, but to my knowledge, this thesis presents the first proof of the union-find explain algorithm and of the congruence closure algorithm in an automatic theorem prover. Our implementation is based on the union-find formalisation by Lammich [7] in Isabelle/HOL. Given that the focus of this paper is on the verification of the algorithms, some optimisations are left out of the implementation, such as path compression for union-find.

1.1 Outline

This thesis is organized as follows: Chapter 2 discusses some related work and gives a brief overview of the notation used by Isabelle.

In Chapter 3 the union-find implementation by Lammich [7] is described, and the explain operation is presented together with its correctness and termination proofs.

Chapter 4 looks at the congruence closure implementation and its correctness proof. It also describes the explain operation for congruence closure, and discusses a possible proof outline for its correctness. For reasons of time, the actual proof has not been finished yet.

The last chapter summarizes the results and gives an outlook on the proofs which are open for future work.

The Isabelle code of this thesis is available on GitHub[1].

[1]https://github.com/reb-ddm/congruence-closure-isabelle

2 Preliminaries

2.1 Isabelle/HOL

Isabelle[14] is an automated theorem prover, which can be used among other things for the verification of algorithms. It provides different types of logic, the most used one being Higher-Order Logic (HOL), which is the one used in this thesis. For reasons of self-containedness of this thesis, this chapter introduces the notation used by Isabelle/HOL.

Lists

The syntax for the empty list in Isabelle is `[]`, and longer lists are constructed with the infix `#` operator, with which one element is appended to the front of the list. To concatenate two lists, the `@` operator is used. `set` is a function which converts a list to a set. Lists are indexed with the `!` operator, and the syntax for updating a list l at index i is the following: `l[i := new_value]`.

Functions

In Isabelle, the termination of functions must be proven. In the case of simple functions, Isabelle can prove it automatically. This is done for functions which are declared with the **`fun`** keyword. For example the declaration **`fun`** `f :: 'a ⇒ 'b` describes a function f with a parameter of the type $'a$ and it returns a value of the type $'b$. Afterwards, the recursive equations of the function are defined.

Partial functions can also be defined in Isabelle, with the **`function`** keyword. The definition looks like this:

```
function (domintros) g
 where "..."
 by pat_completeness auto
```

Isabelle automatically defines a predicate `g_dom` where `g_dom (a)` means that the function terminates with the parameter a.

The option `domintros` provides inductive introduction rules for `g_dom`, based on the defining equations of g.

After the function definition, it needs to be proven that the patterns used in the definition are complete and compatible. In our case, the method `pat_completeness auto` always automatically proves this goal.

For a more detailed description of functions in Isabelle, see [15].

Records

Records are similar to tuples, where each component has a name. For example, for the implementation of the union-find explain operations, we need three lists, which are grouped together in a record. The syntax is `ufe = ⦇uf_list = l, unions = u, au = a⦈`, and to select for example the first component, we can write `uf_list ufe`. The meaning of the three lists will be described in Section 3.3.

For more information on records, see [14], chapter 8.3.

equivalence closure

The theory "Partial_Equivalence_Relation"[17] defines the symmetric closure `symcl` of a relation and the reflexiv-transitive closure is already part of the Isabelle/HOL distribution, and its syntax is `R*`. The two definitions can be combined to have an abstraction of the equivalence closure. For example the equivalence closure of the relation R is `(symcl R)*`.

Datatypes

New datatypes can be defined with the **`datatype`** keyword. New datatypes consist of constructors and existing types. A concrete syntax for the new datatypes can be defined in brackets.

For example we define a new datatype for the two types of input equations used in the congruence closure algorithm. Equations of the type $a = b$ will be written as `a ≈ b`, and equations of the type $F(a, b) = c$ are written as `F a b ≈ c`.

```
datatype equation = Constants nat nat ("_ ≈ _")
  | Function nat nat nat ("F _ _ ≈ _")
```

In this thesis, we will use the notation `a ≈ b` and `F a b ≈ c` in Isabelle listings, and $a = b$ and $F(a, b) = c$ outside of the listings.

option

The type `option` models optional values. The value of a variable with type `'a option` is either `None` or `Some a` where a is a value with type $'a$.

3 Union-Find with Explain Operation

3.1 Union-Find Algorithm

In this thesis, we will use lists to model the union-find forest, where at each index i the list contains the parent $l!i$ of the element i in the forest. The theory "Union_Find" also uses this representation.

If i does not have a parent, e.g. it is a root, then the list contains the element i itself at the index i, e.g. $l!i = i$.

3.2 Union-Find in Isabelle

The union find algorithm was already formalized in Isabelle, and the code can be found in the "Archive of Formal Proofs" (AFP)[13].

For example the following is the definition of the *find* operation from the Theory "Union_Find"[16]. It is called `rep_of`, because it finds the representative of an element in the union-find forest.

The domain of `rep_of` is used in the theory "Union_Find" to define the following invariant for valid union-find lists. [16].

```
definition
  "ufa_invar l ≡ ∀i<length l. rep_of_dom (l,i) ∧ l!i<length l"
```

3.3 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 [3].

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 2.
- `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 $p1 = p2$. □

3.4 Implementation

3.4.1 Union

The *union* operation was already implemented for the `uf_list` in the theory `Union_Find` [16] (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`[16]. 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]⦈
```

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)"
```

3.4.2 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[18].

```
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 a path 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[18] 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)"
```

3.4.3 Explain

We implement the explain function following the description of the first version of the union-find algorithm in the paper[3].

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
```

3.5 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.

3.5.1 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[16], 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.

3.5.2 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 □

3.5.3 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 3.5.2 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 Congruence Closure Algorithm

4.2 Implementation

For the implementation of the congruence closure algorithm, we follow the description in the paper. [3]

4.2.1 Modified Union Find Algorithm

In order to implement an explain operation with a reasonable runtime for the congruence closure data structure, the paper [3] introduces an alternative union-find algorithm. We will not discuss in detail the runtimes of the functions, since the main objective of this thesis is to prove the correctness of the algorithms, and for this purpose we disregarad sThe `find` function remains the same, but for `union` and `explain` a new data structure is introduced, the *proof forest*, i.e. a forest which has as nodes the variables, and as edges the unions that were made. The forest structure is preserved by `union`, because redundant unions are ignored.

add_edge

The proof forest 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 e to e are reversed. In the implementation, the proof forest is represented by a list which stores the parent of each node, exactly as in the union-find list. My implementation for adding an edge, which corresponds to the `union` operation, 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
```

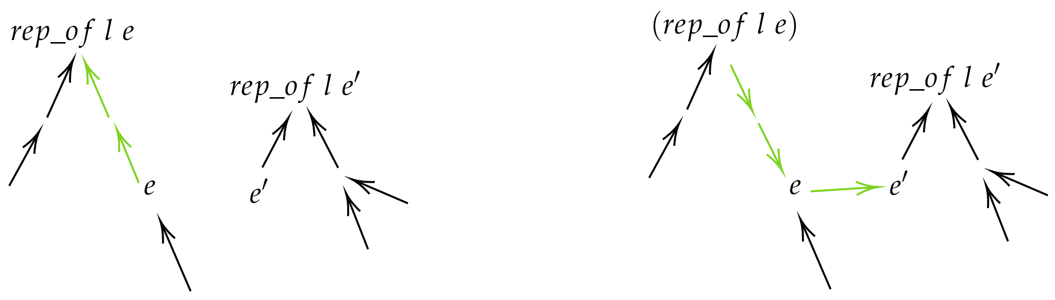
We can show that `add_edge e e'` terminates, if the invariant `ufa_invar` holds for the proof forest and e and e' do not belong to the same equivalence class.

```
lemma add_edge_domain:
  assumes "ufa_invar l" "rep_of l e ≠ rep_of l e'"
  shows "add_edge_dom (l, e, e')"
```

Proof. It can be proven by induction on the length of the path p from the root of e to e.

In the base case there is only one node in the path, therefore e must be equal to its root, therefore $pf!e = e$, and the algorithm terminates immediately.

In the other case e is not a root, then there is a path p' from the root to the parent of e which is shorter than the path from the root to e. The path p' is also present in the $pf[e := e']$, because the path does not contain e. Also, the representative of e in $pf[e := e']$ is equal to the representative of e', and the representative of the parent of e is still the old representative of e, 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 is not necessary for the union-find algorithm by itself, but it will be needed by the `explain` operation for congruence closure. There are two possible types of labels: either an equation $a = b$ was input, or two equations of the type $F(a_1, a_2) = a$ and $F(b_1, b_2) = b$, where a_1 and b_1 were already in the same equivalence class before this union, as well as a_2 and b_2. In both theses cases a union between the equivalence classes of a and b must be made. The labeling is implemented by using an additional list, which at each index contains the label of the outgoing edge, or `None` if there is no outgoing edge. It is similar to the associated unions list of union-find, but it contains directly the labels instead of an index to another list.

The labels have the type `pending_equation`, which can be either one or two equations.

```
datatype pending_equation = One equation
  | Two equation equation
```

The name `pending_equation` derives from the fact that it is also the type of the elements of the pending list, which will 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 either `One` ($a = b$) or `Two` ($F(a_1, a_2) = a$) ($F(b_1, b_2) = b$).

Each time an edge gets added to the proof forest, the labels need to be updated as well. The function `add_label` adds a label to the new edge, and modifies the labels for the edges which are modified by `add_edge`:

```
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 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')"
```

4.2.2 Congruence Closure Data Structure

For the congruence closure algorithm there are five important data structures, which are described in the following. More details on this topic can be found in [3].

- `cc_list`: the union-find list, corresponds to the `uf_list`.
- `use_list`: a two-dimensional list which contains for each representative a a list of input equations $F(b_1, b_2) = b$ where the representative of b_1 or b_2 is a.
- `lookup`: a lookup table indexed by pairs of representatives b and c, which stores an input equation $F(a_1, a_2) = a$ such that b is the representative of a_1 and c is the representative of a_2, or `None` if no such equation exists.
- `pending`: equations of the type `One` ($a = b$) or `Two` ($F(a_1, a_2) = a$) ($F(b_1, b_2) = b$) where a and b need to be merged, and a_1 and b_1 are already in the same congruence class, as well as a_2 and b_2.

- `proof_forest`: the proof forest as described in the previous subsection.
- `pf_labels`: the labels of the proof forest as described in the previous subsection
- `input`: a set of the input equation, which will be useful for some proofs in the next sections.

In the following, we shall sometimes refer to `cc_list` as l, the use list as u, the lookup table as t, the pending list as pe, the proof forest as pf, the labels list for the proof forest as pfl and the input as ip.

4.2.3 Congruence Closure Algorithm

With this data structure we can implement the merge function as described in [3].

The merge function adds the equation to pending, and then calls propagate. If the input equation is of the type $F(a_1, a_2) = a$, then there are two possibilities: if there is already an equation $F(b_1, b_2) = b$ in the lookup table at the $(rep\_of(a_1), rep\_of(a_2))$, then we know that $a_1 = b_1$ and $a_2 = b_2$, and we add $F(b_1, b_2) = b$ and $F(a_1, a_2) = a$ to pending. On the other hand, if the respective lookup entry is `None`, then the equation is added to the lookup table, at the index $(rep\_of(a_1), rep\_of(a_2))$ so that the next time an equation with congruent parameters is input, they will be added together to pending.

For this case distinction there is a function `lookup_Some`, which returns `True` if there is an entry in lookup at the index $(rep\_of(a_1), rep\_of(a_2))$ and False otherwise, and a function `update_lookup`, which adds the equation to lookup at the index $(rep\_of(a_1), rep\_of(a_2))$.

```
fun merge :: "congruence_closure ⇒ equation ⇒ congruence_closure"
  where
"merge ⦇cc_list = l, use_list = u, lookup = t, pending = pe, proof_forest =
    pf, pf_labels = pfl, input = ip⦈
(a ≈ b) =
  propagate
   ⦇cc_list = l, use_list = u, lookup = t, pending = One (a ≈ b)#pe,
      proof_forest = pf, pf_labels = pfl, input = insert (a ≈ b) ip⦈"

| "merge ⦇cc_list = l, use_list = u, lookup = t, pending = pe, proof_forest =
     pf, pf_labels = pfl, input = ip⦈
(F a₁ a₂ ≈ a) =
(if (lookup_Some t l (F a₁ a₂ ≈ a))
  then propagate ⦇cc_list = l, use_list = u, lookup = t,
         pending = link_to_lookup t l (F a₁ a₂ ≈ a)#pe, proof_forest = pf,
              pf_labels = pfl, input = insert (F a₁ a₂ ≈ a) ip⦈
  else ⦇cc_list = l,
        use_list = (u[rep_of l a₁ := (F a₁ a₂ ≈ a)#(u ! rep_of l a₁)])[rep_of
            l a₂ := (F a₁ a₂ ≈ a)#(u ! rep_of l a₂)],
        lookup = update_lookup t l (F a₁ a₂ ≈ a),
        pending = pe, proof_forest = pf, pf_labels = pfl, input = insert (F a₁
             a₂ ≈ a) ip⦈
)"
```

The main part of the algorithm is executed in propagate, which recursively takes one item from pending and performs the union of the representative classes. As previously mentioned, the pending item could be either an equation of the type $a = b$, or two equations of the type $F(a_1, a_2) = a$ and $F(b_1, b_2) = b$, where a_1 and a_2 are already in the same representative class as b_1 and b_2 respectively. In both cases the representative classes of a and b need to be merged. The functions `left` and `right` simply retrieve a and b from either of the two types of pending equations. If a and b are already in the same representative class, nothing needs to be done, otherwise the union is performed. For more clarity, I defined the union separately as `propagate_step`.

```
function propagate :: "congruence_closure ⇒ congruence_closure"
  where
"propagate ⦇cc_list = l, use_list = u, lookup = t, pending = [], proof_forest
     = pf, pf_labels = pfl, input = ip⦈ =
⦇cc_list = l, use_list = u, lookup = t, pending = [], proof_forest = pf,
    pf_labels = pfl, input = ip⦈"
| "propagate
⦇cc_list = l, use_list = u, lookup = t, pending = (eq # pe), proof_forest =
    pf, pf_labels = pfl, input = ip⦈ =
(let a = left eq; b = right eq in
  (if rep_of l a = rep_of l b
   then propagate ⦇cc_list = l, use_list = u, lookup = t, pending = pe,
       proof_forest = pf, pf_labels = pfl, input = ip⦈
   else
     propagate (propagate_step l u t pe pf pfl ip a b eq)
))"
  by pat_completeness auto
```

Concerning the union, we disregard some optimisations, namely the path compression and the optimisation which considers the size of the representative classes in order to choose in which direction to add the union edge. These optimisations are not relevant for the correctness of the algorithm, and they could later be added to a refinement of the algorithm.

The union consists of the previously discussed `ufa_union`, `add_edge` and `add_label`, as well as a loop which moves all elements from the use list of the representative of a to either the representative of b, or to pending. This is necessary, because the old representative of a is not a representative any more, and its new representative is $rep\_of(l, b)$.

```
abbreviation propagate_step
```

```
  where
"propagate_step l u t pe pf pfl ip a b eq ≡
  propagate_loop (rep_of l b) (u ! rep_of l a)
   ⦇cc_list = ufa_union l a b,
   use_list = u[rep_of l a := []],
   lookup = t,
   pending = pe,
   proof_forest = add_edge pf a b,
   pf_labels = add_label pfl pf a eq,
   input = ip⦈"
```

The loop is defined as a recursive function, which considers each element of the use list of $rep\_of(l, a)$, and either adds it to the use list and the lookup table, or if there is already an entry in lookup, then that entry together with the current equation are added to pending.

```
fun propagate_loop
  where
"propagate_loop rep_b (u1 # urest)
⦇cc_list = l, use_list = u, lookup = t, pending = pe, proof_forest = pf,
    pf_labels = pfl, input = ip⦈
=
  propagate_loop rep_b urest (
   if (lookup_Some t l u1)
   then
     ⦇cc_list = l, use_list = u, lookup = t,
         pending = link_to_lookup t l u1#pe,
         proof_forest = pf, pf_labels = pfl, input = ip⦈
   else
     ⦇cc_list = l,
         use_list = u[rep_b := u1 # (u ! rep_b)],
         lookup = update_lookup t l u1,
         pending = pe, proof_forest = pf, pf_labels = pfl, input = ip⦈
)"
| "propagate_loop _ [] cc = cc"
```

The function `are_congruent` returns `True` if an equation is in the congruence closure of all the input equations so far. It simply checks if the elements have the same representative or if they have the same representative as the correspondent entry in lookup.

```
fun are_congruent :: "congruence_closure ⇒ equation ⇒ bool"
  where
"are_congruent ⦇cc_list = l, use_list = u, lookup = t, pending = pe,
    proof_forest = pf, pf_labels = pfl, input = ip⦈ (a ≈ b) =
   (rep_of l a = rep_of l b)"
| "are_congruent ⦇cc_list = l, use_list = u, lookup = t, pending = pe,
    proof_forest = pf, pf_labels = pfl, input = ip⦈ (F a₁ a₂ ≈ a) =
   (case lookup_entry t l a₁ a₂ of
     Some (F b₁ b₂ ≈ b) ⇒ (rep_of l a = rep_of l b)
   | None ⇒ False
)"
```

4.3 Correctness Proof

4.3.1 Invariants

At this point we can already prove some properties of the congruence closure data structure. Our approach this time is different than the one for the union-find algorithm. Instead of defining an induction rule like in the union find section and then prove the properties through the induction rule, we define the properties as invariants and then prove that they remain invariant after applying merge. For each invariant, we need to follow the same steps:

1. Prove that the invariant holds for the initial empty congruence closure.
2. Prove that if the invariant holds before the merge operation, it also holds after the merge operation. Below is a list of what needs to be proven:
   a) The invariant holds after one step in the `propagate_loop`. We shall refer to the two possible cases as `loop1` and `loop2`.
   b) The invariant holds after the entire `propagate_loop`.
   c) The invariant holds for the parameters of `propagate_loop` in `propagate_step`. We shall refer to the this case as `mini_step`.
   d) It holds after `propagate_step`.
   e) It holds after `propagate`.
   f) And finally, it holds after `merge`.

After having defined all the invariants and proven the above properties, we can put all the invariants together in the invariant `cc_invar` and prove the following two theorems, which state exactly the two properties described above for the invariant `cc_invar`:

```
theorem cc_invar_initial_cc: "cc_invar (initial_cc n)"

theorem cc_invar_merge:
  assumes "cc_invar cc" "valid_vars eq (nr_vars cc)"
  shows "cc_invar (merge cc eq)"
```

Let us now look at the concrete invariants. Each list in the data structure has an invariant which states that all the elements which are in the list are in bounds. This is easy to prove if we assume that all the input equations contain only valid elements.

One of the invariants is the usual `ufa_invar` that we know from the union-find algorithm. The `ufa_invar` holds for the `cc_list` and the `proof_forest`. These two are only modified before entering the in the `mini_step`, and we already proved previously that the `ufa_invar` holds after `ufa_union` (section union find from AFP TODO) and after `add_edge` (Subsection 4.2.1). Therefore it also holds after `merge`.

We define a new invariant `inv_same_rep_classes`, which states, as the name suggests, that the union-find forest and the proof forest represent the same equivalence classes:

```
rep_of l i = rep_of l j ⟷ rep_of pf i = rep_of pf j
```

In order to prove this, given that the two lists are only modified during the `mini_step`, it is sufficient to show that `ufa_union l x y` and `add_edge pf x y` have the same behaviour. The theory `Union_Find` from the AFP[16] already provides the following lemma for `ufa_union`:

```
lemma ufa_union_aux:
  "rep_of (ufa_union l x y) i =
    (if rep_of l i = rep_of l x then rep_of l y else rep_of l i)"
```

We can show a similar lemma for `add_edge`:

```
lemma rep_of_add_edge_aux:
   assumes "rep_of l x ≠ rep_of l y"
   shows "rep_of (add_edge l x y) i =
     (if rep_of l i = rep_of l x then rep_of l y else rep_of l i)"
```

The additional assumption `"rep_of l x ≠ rep_of l y"` does not cause problems, because `add_edge` is only used within `propagate_step`, and `propagate_step` is only executed if `"rep_of l x ≠ rep_of l y"`.

Proof. We already showed that the function terminates in Subsection 4.2.1, therefore we can prove it by induction on `add_edge`.

TODO describe the proof. □

Additionally, for each data structure there is an invariant which states the properties which were informally described in Subsection 4.2.2.

For the use list, the invariant states that for each representative a, its use list only contains equations of the type $F(b_1, b_2) = b$, where a is the representative of either b_1 or b_2.

Proof. In order to prove the use list invariant, we can follow the steps described earlier. For the correctness proof after the `propagate_loop`, I needed to add an additional assumption that the second parameter only contains equations of the type $F(a_1, a_2) = a$ and the representative of either a_1 or a_2 is $rep\_of(l, b)$ (where b is the right side of the equation which is being propagated). This follows from the facts that the parameter of `propagate_loop` was $(u!rep\_of(l, a))$, and the new representative of a after the union is b.

With this assumption I could show that each time an equation gets added to the use list in the `propagate_loop`, it is a valid equation.

In the proof after the merge operation, the use list is only modified in the third case, and only equations of a valid form are added to $rep\_of(l, a_1)$ and r$rep_of(l, a_2)$. Therefore all the necessary properties hold for these new equations.

For the remaining cases, use list is either unchanged, or something is removed from it, therefore the invariant trivially holds. □

The invariant for lookup is similar, it states that each entry in the lookup table at index (i, j), for representatives i and j, is either `None` or is an equation of the form $F(a_1, a_2) = a$ where the representative of a_1 is i and the representative of a_2 is j.

Proof. Each time an equation is added to lookup, it has the desired form and it is added to the index $(rep\_of(l, a_1), rep\_of(l, a_2))$. This happens in the `propagate_loop` and in `merge`. In the `propagate_loop`, the added equation derives from the use list, for which we proved with the previous invariant that its equations have the desired form. In merge, only the equations of the type $F(a_1, a_2) = a$ are added to lookup. □

For pending, the invariant states that the equations are either of the form `One` $(a = b)$ or `Two` $(F(a_1, a_2) = a)$ $(F(b_1, b_2) = b)$ where $rep\_of(l, a_1) = rep\_of(l, b_1)$ and $rep\_of(l, a_2) = rep\_of(l, b_2)$: It is important to know that they are in the same representative class, because TODO

Proof. We need to show that in the `propagate_loop` the equation $u1$ we add to pending has a valid format. We know that $u1$ derives from the use list, therefore it is of the form $F(a_1, a_2) = a$. Then we link to it the lookup entry at the index $(rep\_of(l, a_1), rep\_of(l, a_2))$. From the lookup invariant we know that there is an entry of the form $F(b_1, b_2) = b$ at this index where $rep\_of(l, a_1) = rep\_of(l, b_1)$ and $rep\_of(l, a_2) = rep\_of(l, b_2)$. This shows that they are valid equations for pending.

The same holds for the equations added to pending in merge. □

There is also an invariant which states that the `cc_list`, the first dimension of the use list, both dimensions of lookup, the proof forest and the `pf_labels` have the same length. This was trivial to prove, given that the algorithm never changes the length of the lists, and initially the lists have the same length.

All the above-mentioned invariants hold trivially for the initial case, given that all the data structures are empty or contain only `None` in the beginning.

The remaining invariants will be described later on, when they become relevant.

4.3.2 Abstract Formalisation of Congruence Closure

In order to prove the correctness of the algorithm, we define an abstraction of congruence closure. We cannot use any previouly defined definitions, because the data structure that we use can only represent a subset of all possible equations, for example it cannot represent equations of the type $a = F(b, c)$ or $F(F(a, b), c) = d$. For this reason, we define an inductive set which represents the congruence closure of a set of equations and only uses the our restricted definition of equation.

```
inductive_set Congruence_Closure :: "equation set ⇒ equation set" for S
  where
    base: "eqt ∈ S ⟹ eqt ∈ Congruence_Closure S"
  | reflexive: "(a ≈ a) ∈ Congruence_Closure S"
  | symmetric: "(a ≈ b) ∈ Congruence_Closure S ⟹ (b ≈ a) ∈
      Congruence_Closure S"
  | transitive1: "(a ≈ b) ∈ Congruence_Closure S ⟹ (b ≈ c) ∈
      Congruence_Closure S
⟹ (a ≈ c) ∈ Congruence_Closure S"
  | transitive2: "(F a₁ a₂ ≈ b) ∈ Congruence_Closure S ⟹ (b ≈ c) ∈
      Congruence_Closure S
⟹ (F a₁ a₂ ≈ c) ∈ Congruence_Closure S"
  | transitive3: "(F a₁ a₂ ≈ a) ∈ Congruence_Closure S
⟹ (a₁ ≈ b₁) ∈ Congruence_Closure S ⟹ (a₂ ≈ b₂) ∈ Congruence_Closure S
⟹ (F b₁ b₂ ≈ a) ∈ Congruence_Closure S"
  | monotonic: "(F a₁ a₂ ≈ a) ∈ Congruence_Closure S ⟹ (F a₁ a₂ ≈ b) ∈
      Congruence_Closure S
⟹ (a ≈ b) ∈ Congruence_Closure S"
```

The following proof rule follows directly from the definition of Congruence Closure, and proved to be very useful for multiple proofs:

```
lemma Congruence_Closure_eq[case_names left right]:
  assumes "⋀ a. a ∈ A ⟹ a ∈ Congruence_Closure B"
    "⋀ b. b ∈ B ⟹ b ∈ Congruence_Closure A"
  shows "Congruence_Closure A = Congruence_Closure B"
```

It is used to prove equality between congruence closures of A and B. It states that it is sufficient to prove that all elements of set A are in the congruence closure of B and vice versa, instead of having to prove that all elements of the congruence closure of A are in the congruence closure of B.

4.3.3 Correctness

To prove the correctness of the congruence closure implementation, we need to show that the invariants imply that `are_congruent cc eq` returns `True` if and only if the equation *eq* lies in the congruence closure of the input equations.

```
theorem are_congruenct_correct:
  assumes "cc_invar cc" "pending cc = []"
  shows "eq ∈ Congruence_Closure ((input cc)) ⟷ are_congruent cc eq"
```

The paper [3] proves this by stating athe folllowing invariant which holds throughout the algorithm,

```
Congruence_Closure(representativeE ∪ pending) = Congruence_Closure (input)
```

where representativeE can be seen as the set of equations derived from our union-find list and the equations in lookup. It is the union of the following two sets:

- `representative_set` is defined such that its congruence closure contains all the equations between two elements which have the same representative.
- `lookup_entries_set` is the set of all the entries in lookup at indexes which are representatives.

```
abbreviation representatives_set :: "nat list ⇒ equation set"
  where
   "representatives_set l ≡ {a ≈ rep_of l a |a. l ! a ≠ a}"

abbreviation lookup_entries_set :: "congruence_closure ⇒ equation set"
  where
   "lookup_entries_set cc ≡ {F a' b' ≈ rep_of (cc_list cc) c | a' b' c c₁ c₂ .

                  cc_list cc ! a' = a' ∧ cc_list cc ! b' = b'
                  ∧ lookup cc ! a' ! b' = Some (F c₁ c₂ ≈ c)}"

definition representativeE :: "congruence_closure ⇒ equation set"
  where
   "representativeE cc = representatives_set (cc_list cc) ∪
       lookup_entries_set cc"
```

The formal definitioin of the aforementioned invariant is the following, where `pending_set` converts the pending list to a set of equations of the type $a = b$:

```
definition inv2 :: "congruence_closure ⇒ bool"
  where
   "inv2 cc ≡
Congruence_Closure (representativeE cc ∪ pending_set (pending cc)) =
    Congruence_Closure (input cc)"
```

The set of input equations is only modified by the `merge` function, but remains constant throughout the `propagate` function, therefore for the proof we just need to show that the congruence closure of the representativeE set and pending remain unchanged after the propagate function.

The main challenge is to prove that the invariant holds after the `mini_step`. We will show that `Congruence Closure (representativeE ∪ pending)` before the `propagate_step` is equal to `Congruence Closure (representativeE ∪ pending ∪ (u ! rep_of l a))` after the `mini_step`. Then we prove that `Congruence Closure (representativeE ∪ pending ∪ (u ! rep_of l a))` is equal to `Congruence Closure (representativeE ∪ pending)` after the `propagate_loop`. These two lemmas imply that the congruence closure of the representativeE set and pending remain unchanged after the propagate function.

We will first prove the second statement, given that it is much easer to show.

Proof. We need to shhow that `Congruence Closure (representativeE ∪ pending ∪ (u ! rep_of l a))` is equal to `Congruence Closure (representativeE ∪ pending)` after the `propagate_loop`. In each step of the loop, one element from $(u!rep\_of(l,a))$ is moved either to pending or to lookup. Therefore after the loop each element of $(u!rep\_of(l,a))$ is either in pending or in representativeE. □

The first lemma is more difficult to prove. The following is the statement of the lemma:

```
lemma inv2_mini_step:
  assumes "a = left eq" "b = right eq"
  "cc_invar ⦇cc_list = l, use_list = u, lookup = t, pending = (eq # pe),
  proof_forest = pf, pf_labels = pfl, input = ip⦈"
    shows "Congruence_Closure
(representativeE
⦇cc_list = l, use_list = u, lookup = t, pending = (eq # pe),
proof_forest = pf, pf_labels = pfl, input = ip⦈
∪ pending_set (eq # pe))
=
Congruence_Closure (representativeE
⦇cc_list = ufa_union l a b,
   use_list = u[rep_of l a := []],
   lookup = t,
   pending = pe,
   proof_forest = add_edge pf a b,
   pf_labels = add_label pfl pf a eq,
   input = ip⦈
∪ pending_set pe
∪ set (u ! rep_of l a))"
```

Proof. There are two inclusions which need to be shown. We can use the rule `Congruence_Closure_eq` from Subsection 4.3.2, which means that it is sufficient to show that each equation in the set on the left hand side is in the congruence closure of the right hand side and vice versa.

"⊆" It needs to be shown that the equations of the `representatives_set`, in lookup and in pending are in the Congruence Closure of the right-hand side.

Regarding the `representatives_set`, all the elements which had the same representative before a union also have the same representative after a union.

For the pending set, we need to prove that the equation that is removed from pending is still in the congruence closure after the `mini_step`. This holds, because the equation which is removed is $a = b$, and a and b are in the same equivalence class after the `ufe_union`.

The problematic case are the equations in lookup. Given that after the union there is one element $c = rep\_of(l, a)$ which is not a representative any more, the entries in lookup which have as first or second index c are not in lookup anymore after the union. The goal is to prove that these equations are exactly the equations which are present in $(u!rep\_of(l, a))$, but until now it was only proven that the equations in the use list are valid, not that they are exhaustive. A new invariant `use_list_inv2` is needed which states that all elements which are present in the lookup table at index (i, j) are also in the corresponding use lists of i and j. I will introduce this invariant later.

"⊇" We need to show that the equations of the `representatives_set`, `lookup_entries_set`, pending set and of $(u!rep\_of(l, a))$ on the right-hand side are in the congruence closure of the left-hand side.

The `representatives_set`, contains equations of the type $c = rep\_of(ufa\_union(l, a, b), c)$. If the representative after the union is the same before the union, the same equation is in the `representatives_set` of the left hand side. The only representative that is different than before the union is the representative of a, which has as new representative $rep\_of(l, b)$. The left-hand side contains the equations $b = rep\_of(l, b)$ and $a = b$ (which is in pending). By transitivity, the congruence closure also contains $a = rep\_of(l, b)$ and $rep\_of(l, b)$ is exactly the same as $rep\_of(ufa\_union(l, a, b), a)$.

Regarding lookup, all the elements which are roots after the union, are also roots before the union, therefore all elements in the `lookup_entry_set` of the right-hand side are also in the left-hand side.

It is evident that the equations in pending on the right-hand side are also in pending in the left-hand side.

It is more difficult to show that the equations in $(u!rep\_of(l, a))$ are also present in the lookup table of the left-hand side. Like before, we need a new invariant `lookup_invar2`, which I will describe below. □

Thus we need two new invariants of this form:

- `lookup_invar2`: The elements in the lookup table are also present in the use list.
- `use_list_invar2`: The elements in the use list are also present in the lookup table.

Unfortunately, these two invariants are not exactly true, because if there are two different equations where the elements have the same representatives, then they can't both be present in the lookup table, because it only stores one equation for each pair of representatives. In fact, the set of equations in lookup and in the use list are not exactly the same, but for each equation in one of them, there is a "similar" equation in the other one.

The difficulty was to find a suitable definition of "similar" which is not too strong, otherwise it wouldn't be true, but also not too weak, otherwise it is not possible to prove the invariant `inv2`.

The right definition of similar turned out to be the following:

Definition. Two equations $F(a_1, a_2) = a$ and $F(b_1, b_2) = b$ are *similar*, if $rep\_of(l, a_1) = rep\_of(l, b_1), rep\_of(a_2) = rep\_of(b_2)$ and $(a = b) \in Congruence\_Closure(representatives\_set \cup pending)$.

Simply stating that a and b have the same representative would be too strong, because during the propagate function, they are added to pending in order to be merged later, and are not merged yet. If we use `representativeE` instead of `representatives_set`, the inariant is not strong enough in order to prove `inv2_mini_step`.

Here follows the proof for `lookup_invar2`:

Proof. We need to show that if before the `merge` the invariant holds, then it also holds after the `merge`. The invariant states that for each equation in lookup at the index (i, j) (where i and j are representatives) there is a similar equation in use list i and one in use list j.

The main aim is to show that it holds after `propagate`. We assume that before `propagate` the invariant holds. In particular, each equation $F(c_1, c_2) = c$ in $u_a = (u!rep\_of(l, a))$ there is a similar equation in $(u!rep\_of(l, c_1))$ and in $(u!rep\_of(l, c_2))$.

In the `propagate_loop` u_a is emptied, while the other use lists are not modified, and u_a is handed over as a parameter to the `propagate_loop`.

From now on let l be the `cc_list` after the `ufe_union`.

In `propagate_loop1` the lookup table and the use lists are not modified, thus there is nothing to show.

In `propagate_loop2` we take an equation $u1$ of the form $F(c_1, c_2) = c$ from u_a.

$u1$ is then added to lookup at the index $(rep\_of(l, c_1), rep\_of(l, c_2))$. We need to show that after this step, an equation similar to $u1$ is present both in the use list of

$rep\_of(l, c_1)$ and the use list of $rep\_of(l, c_2)$. This holds if one (or both) of them is equal to $rep\_of(l, b)$, because $u1$ is also added to $u!rep\_of(l, b)$ by the function.

If they are not equal to $rep\_of(l, b)$, there is a similar equation in the corresponding use list, because the use list has not been changed. □

Here follows the proof for `use_list_invar2`:

Proof. We need to show that if before the `merge` the invariant holds, then it also holds after the `merge`. The invariant states that for each equation $F(c_1, c_2) = c$ in use list at the index i (where i is a representative) there is a similar equation in lookup$(rep_of(l, c_1), rep_of(l, c_2))$.

The difficulty in this proof was when after the `mini_step`, because of the union, $rep\_of(l, a)$ is not a root any more, therefore the lookup entries at the indexes of $rep\_of(l, a)$ are not valid anymore. However, there could be equations in a use list at an index which is not $rep\_of(l, a)$, which contains equations where the second parameter has the same representative of a. These equations do not have a similar equation in lookup at this moment in the algorithm, they have only a similar equation in $u_a = u!rep_o fla$, therefore the invariant does not hold for these equations. Nevertheless, we will show that it holds after the `propagate_loop` was executed.

From now on let l be the `cc_list` after the `ufe_union`.

The `propagate_loop` removes an equation $F(c_1, c_2) = c$ from u_a, and it enters `propagate_loop1` when lookup contains an equation $F(d_1, d_2) = d$ at the index $(rep\_of(l, c_1), rep\_of(l, c_2))$. Then the equation $c = d$ is added to pending. Note that $F(c_1, c_2) = c$ and $F(d_1, d_2) = d$ are similar at this point, because $rep\_of(l, c_1) = rep\_of(l, d_1)$ and $rep\_of(l, c_2) = rep\_of(l, d_2)$ follows from the `lookup_invar`.

We know that for all the equations in the use list, there is a similar equation in lookup or u_a. For those equations where the similar equation is exactly the one that we are removing from u_a, know that that there is another equation $F(d_1, d_2) = d$ in lookup which is similar to $F(c_1, c_2) = c$.

This case is exactly the reason why we can only prove that there is a "similar" equation in lookup, and not exactly the same.

The `propagate_loop2` is entered, when lookup contains `None` at the index $(rep\_of(l, c_1), rep\_of(l, c_2))$.

$F(c_1, c_2) = c$ is added to $(u!rep\_of(l, b))$ and lookup. Each equation which was in use list has a similar equation in lookup or u_a. For those equations where the similar equation is exactly the one that we are removing from u_a, we know that it was also added to lookup.

There is also a new element in the use list, which is $F(c_1, c_2) = c$, and it has a similar equation in lookup, which is $F(c_1, c_2) = c$ itself. □

With these two invariants the proof for `inv2` is completed.

Given that the pending list is always empty after the termination of propagate, the correctness proof `are_congruent_correct` follows directly from this invariant.

To completely finish the proof, one would need to prove that propagate terminates, which is currently left open for future work.

4.4 Implementation of the Explain Operation

I implemented the explain operation for congruence closure, leaving the proof of termination and correctness open for future work. Nevertheless I would like to dedicate this section on the description of the implementation and a proposal of how the correctness could be proven.

TODO

5 Conclusion

5.1 Future work

List of Figures

Bibliography

[1] B. A. Galler and M. J. Fisher. "An improved equivalence algorithm." In: *Communications of the ACM* 7.5 (May 1964), pp. 301–303. DOI: 10.1145/364099.364331.

[2] R. E. Tarjan. "A class of algorithms which require nonlinear time to maintain disjoint sets." In: *Journal of Computer and System Sciences* 18.2 (Apr. 1979), pp. 110–127. DOI: 10.1016/0022-0000(79)90042-4.

[3] R. Nieuwenhuis and A. Oliveras. "Proof-Producing Congruence Closure." In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, pp. 453–468. DOI: 10.1007/978-3-540-32033-3_33.

[4] R. E. Shostak. "An algorithm for reasoning about equality." In: *Communications of the ACM* 21.7 (July 1978), pp. 583–585. DOI: 10.1145/359545.359570.

[5] G. Nelson and D. C. Oppen. "Fast Decision Procedures Based on Congruence Closure." In: *Journal of the ACM* 27.2 (Apr. 1980), pp. 356–364. DOI: 10.1145/322186.322198.

[6] R. Nieuwenhuis and A. Oliveras. "Fast congruence closure and extensions." In: *Information and Computation* 205.4 (Apr. 2007), pp. 557–580. DOI: 10.1016/j.ic.2006.08.009.

[7] P. Lammich. "Refinement to Imperative HOL." In: *Journal of Automated Reasoning* 62.4 (Oct. 2017), pp. 481–503. DOI: 10.1007/s10817-017-9437-1.

[8] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. "CVC4." In: *Computer Aided Verification*. Springer Berlin Heidelberg, 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14.

[9] L. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[10] P. Corbineau. "Autour de la clôture de congruence avec Coq." MA thesis. Université Paris-Sud, 2001.

[11] D. Selsam and L. de Moura. "Congruence Closure in Intensional Type Theory." In: *Automated Reasoning*. Springer International Publishing, 2016, pp. 99–115. DOI: 10.1007/978-3-319-40229-1_8.

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

[13] *Archive of Formal Proofs*. `https://www.isa-afp.org/`. Accessed July 18, 2022.

[14] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL A Proof Assistant for Higher-Order Logic. A Proof Assistant for Higher-Order Logic*. Springer London, Limited, 2003, p. 226. ISBN: 9783540459491.

[15] A. Krauss. "Defining Recursive Functions in Isabelle/HOL." In: (May 2012).

[16] 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.

[17] P. Lammich. "Collections Framework." In: *Archive of Formal Proofs* (Dec. 2009). `https://isa-afp.org/entries/Collections.html`, Formal proof development. ISSN: 2150-914x.

[18] 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`.