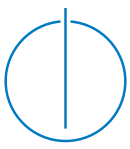
DEPARTMENT OF INFORMATICS

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

First of all, I want to thank Manon and Tony for being great friends. Moreover, I am thankful to my parents for proof-reading this thesis and for their constant love and support. Lastly, I want to thank my supervisor Lukas Stevens for answering all the questions about the thesis and Isabelle that I had.

Abstract

Congruence closure is a key algorithm of decision procedures in automated theorem proving. In these settings, it is important to have trustworthy implementations, that are formally proven for correctness. This thesis describes a formal implementation and verification of a congruence closure algorithm in the interactive theorem prover Isabelle/HOL. The implementation is based on the union-find algorithm. In the context of decision procedures, it is necessary to also compute a certificate which proves that two given terms are congruent. This is done by the *explain* operation. The algorithms in this thesis are based on the description of Nieuwenhuis and Oliveras [1].

First, an *explain* operation for union-find is implemented with proofs for correctness and termination. It extends the formalization of the union-find data structure in Isabelle/HOL by Lammich [2]. Then, the congruence closure algorithm is implemented and verified for correctness and termination. Lastly, an implementation of the *cc_explain* operation for congruence closure with a termination proof is presented. The correctness proof of *cc_explain* is not part of this thesis.

Contents

1 Introduction

Isabelle is an interactive theorem prover, with which it is possible to formalize mathematical formulas and proofs [3]. A common use case for it is the verification of algorithms. It provides different types of logic, the most used one being Higher-Order Logic (HOL).

Congruence closure is used in automated theorem proving in order to determine whether an equation is implied by a given set of equations. Therefore, it is a central part of decision procedures, such as satisfiability modulo theories (SMT) solvers [4]. Isabelle itself uses automated decision procedures to solve proof goals. Hence, the code of this thesis can be used as a basis for a new automated proof strategy in Isabelle. It is important for algorithms of theorem provers to be trustworthy, therefore the algorithms of this thesis are verified for correctness. The proofs are formalized in the interactive theorem prover Isabelle/HOL. The algorithms are based on the paper by Nieuwenhuis and Oliveras [1].

This thesis describes the implementation in Isabelle/HOL of algorithms that find the congruence closure of a set of equations. We consider equations containing constant symbols and function symbols. In the following, a, b, c, d, i, x and y will denote constants and f and F functions. We consider only uninterpreted functions, that is, the only property we know of the functions is how many arguments the function takes. A constant or a function applied to constants, e.g., $f(a, b)$, is called a *term* and an equality between two terms, e.g., $f(a, b) = c$, is called an *equation*.

We will first introduce the union-find algorithm, which maintains the congruence closure of equations containing only constants, e.g. $a = b$. When we only consider constants, the congruence closure is also called equivalence closure. The equivalence closure of a relation is the smallest reflexive, symmetric and transitive superset of the relation. We denote that a is in relation with b by writing $a = b$. For example, the equivalence closure of $a = b$ and $c = b$ contains $a = c$.

Then, we consider an algorithm which maintains the congruence closure of equations containing also function symbols. The congruence closure of a set of equations satisfies, in addition to reflexivity, symmetry and transitivity, also monotonicity, i.e., $f(x_1, ..., x_n) = f(y_1, ..., y_n)$ if $x_i = y_i \; \forall i \in [1..n]$ [1]. For example, the congruence closure of $f(a, b) = c$, $f(d, b) = e$ and $a = d$ contains the equation $c = e$. Several approaches to solve this problem have been described [5, 6, 7, 1]. They differ in their runtime

and application area. Most implementations of congruence closure are based on the union-find algorithm. Dating back to 1964 [8], the union-find algorithm is nowadays the most widely used algorithm for maintaining the equivalence closure of a set, due to its simplicity and almost constant runtime [9].

In the context of decision procedures it is also required to understand which subset of the given equations is responsible for the congruence. For this reason, we also implement an *explain* operation, which returns the set of input equations that caused the congruence. This can be used by an external program in order to generate a certificate of the congruence and verify that it is in fact contained in the congruence closure of the input equations. Nieuwenhuis and Oliveras have presented an efficient version of the congruence closure algorithm and two versions of the union-find algorithm, each with their own *explain* operation. Their conference paper [1] was later extended, see [10]. We will call the *explain* operation of the union-find algorithm *explain*, and the one for congruence closure *cc_explain*.

Our implementation is based on the union-find formalization by Lammich [2] in Isabelle/HOL. We implement three algorithms: the *explain* operation for union-find, the congruence closure algorithm and the *cc_explain* operation for congruence closure.

To my knowledge, this thesis presents the first verified formalization of these algorithms in Isabelle/HOL. Given that the focus of this thesis is on the verification of the algorithms, a few optimizations are left out of the implementation, such as path compression for union-find.

1.1 Outline

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

In Chapter 3 the union-find implementation by Lammich [2] is described and the *explain* operation for union-find is presented together with its correctness and termination proofs.

Chapter 4 looks at the congruence closure implementation and shows that it is correct and that it terminates. It also describes the *cc_explain* operation for congruence closure with its termination proof. The correctness proof is not part of this thesis. However, a proposed outline of the proof is presented.

The last chapter summarizes the results and gives an outlook on the possible future work. The Isabelle code of this thesis is available on GitHub[1]. The code also contains the examples of this thesis in the files `Examples_Thesis.thy` and `CC_Examples_Thesis.thy`.

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

2 Preliminaries

2.1 Isabelle/HOL

This chapter introduces the notation used by Isabelle/HOL.

Lists

The syntax for the empty list in Isabelle is `[]`, and the infix operator `#` is used to append one element to the front of the list. The `@` operator is used in order to concatenate two lists. The function `set` converts a list to a set. Lists are indexed with the `!` operator, and the syntax for updating a list l at index i is: `l[i := new_value]`. The list `[0..<n]` represents a list that contains all the numbers from 0 to $n - 1$.

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. Isabelle automatically defines induction rules for each function.

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 `g` 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.

Partial simplification rules and a partial induction rule are also automatically defined by Isabelle, but they can only be applied if we assume or prove that the function terminates with the given parameters.

For a more detailed description of functions in Isabelle, see [11].

Records

Records are similar to tuples, where each component has a name. For example, for the implementation of the union-find *explain* operation, we need three lists, called `uf_list`, `unions` and `au`. They are grouped together in the record `ufe = ⦇uf_list = l, unions = u, au = a⦈`. In order 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 [3], chapter 8.3.

Equivalence Closure

The theory "Partial_Equivalence_Relation"[12] 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 x` where x is a value with type $'a$. The function `the` applied to `Some x` returns `x`, and it returns `undefined` if the parameter is `None`.

If `'a` is an ordered type, the order is extended to `'a option`, where `None` $\leq$ `x` for all `x`, and `Some x` $\leq$ `Some y` iff `x` $\leq$ `y`. This is defined in the Theory "Option_ord" of the HOL library, which is included with the standard Isabelle/HOL distribution.

2.2 Related work

Efficient union-find algorithms have been known for a long time [8, 9]. Given its importance as an algorithm, it was already formalized and verified in some of the most important theorem provers, such as Isabelle and Coq [13]. The code in this thesis uses the union-find formalization in Isabelle by Lammich, which was first published in a journal [2] and later presented at a conference [14]. It includes the functions for *union* and *find*, as well as an invariant that characterizes the validity of the union-find data structure. It will be described in more detail in Section 3.2.

Based on the union-find implementation, efficient congruence closure algorithms have been developed by Shostak [5], Nelson and Oppen [6] and Downey et al. [7]. Nieuwenhuis and Oliveras [1] extended the algorithm by a *cc_explain* operation, which is necessary in the context of decision procedures, for example, for theorem provers.

Congruence closure is implemented in most automated theorem provers. Pierre Corbineau implemented a `congruence` tactic for the theorem prover Coq [15, 16], based on the algorithm of Downey et al. [7], and with a *cc_explain* operation that is similar to the union-find *explain* presented in this thesis and not as efficient as the *cc_explain* operation introduced by Nieuwenhuis et al. [1]. He formally proved the correctness and termination of the congruence closure algorithm in the theorem prover Coq.

The theorem prover Lean also has a congruence closure-based decision procedure [17], which is additionally able to handle dependent types. To my knowledge, it is not proven for correctness.

Isabelle/HOL includes a tool called `sledgehammer` [18], which uses external SMT solvers, e.g., Z3 [4] and CVC4 [19], whose implementation is based on congruence closure. However, there is to my knowledge no built-in congruence closure proof method for Isabelle yet. The verified algorithms of this thesis can be used in the future in order to build such a proof method. The thesis focuses on the formalization of the congruence closure algorithm in the functional programming language of Isabelle, leaving the implementation of the proof method open for future work. For this, it will be necessary to implement an appropriate transformation of the input equations to the form used by our algorithm. This transformation is described shortly in Section 4.1 and more in detail in [1], but the concrete implementation is not part of this thesis. The optimizations that were omitted in this thesis should also be included in an implementation of the proof method.

3 Union-Find with Explain Operation

3.1 Union-Find Algorithm

Given a set of n elements, the union-find data structure keeps track of a partition of these elements in disjoint sets. It provides efficient operations for finding the set that an element belongs in and for modifying the sets.

In the beginning, each element is in its own set, then the sets are merged by subsequent *union* operations between pairs of elements. Each set in the partition has a representative, which is one particular element in the set. We will denote the representative of an element a in the union-find forest l as $rep\_of(l, a)$. The *find* operation returns the representative of the set that a certain element belongs to. If two elements have the same representative, that means that they belong to the same set. The elements of the union-find data structure are represented by the natural numbers $0, 1, ..., n - 1$.

One application of this algorithm is to maintain the equivalence closure of equations. Given a set of n variables, we can initialize the union-find algorithm with n elements, and for each equation $a_1 = a_2$ we perform a *union* between a_1 and a_2. Each set in the partition represents an equivalence class.

The equivalence classes are modeled with a forest, which is a graph where each connected component is a tree. The connected components of the graph represent the equivalence classes. Initially, the graph contains n vertices and no edges, then each `union` adds a directed edge. Each tree in the forest has a root, which is also the representative of the equivalence class, and each edge in the tree is directed towards the root. In order to keep this invariant, at each *union* between a and b, the new edge is added between the $rep\_of(l, a)$ and $rep\_of(l, b)$.

The union-find forest is represented by a list l of length n, where at each index i the list contains the parent of the element i in the forest. Mimicking the Isabelle syntax for lists, we will denote the representative of an element a as $l!a$. If i does not have a parent, then the list contains the element i itself at the index i, i.e., $l!i = i$. This means that i is a root.

The original union-find algorithm [9] contains two optimizations: path compression in the *find* method, and choosing the representative of the bigger class to be the new representative of the merged class in *union*. These optimizations are irrelevant for the correctness of the algorithm, therefore we leave them out of this implementation in

order to simplify the proofs.

Most of the proofs about the following algorithms contains as an assumption that all variables are in bound, i.e. $x < n$ for each variable x. More precisely, the parameter n is characterized by the length of the list l, therefore the assumption is $x < length(l)$. For reasons of conciseness, assumptions of this kind will not be mentioned in the thesis. The same appliles to the theorems about congruence closure in the next chapter. For the exact formulation of the lemmas, refer to the Isabelle code.

3.2 Union-Find in Isabelle

The union-find algorithm was already formalized in Isabelle by Lammich [2], and the code can be found in the "Archive of Formal Proofs" (AFP) under "A Separation Logic Framework for Imperative HOL", in the theory "Union_Find"[20, 21]. The following is a brief description of the implementation.

The function `rep_of` finds the representative of an element in the forest. It is analogous to the `find` operation, except that it does not do path compression.

```
function (domintros) rep_of
  where "rep_of l i = (if l!i = i then i else rep_of l (l!i))"
  by pat_completeness auto
```

The domain of `rep_of` is used to define the following invariant for valid union-find lists. This invariant states that the `rep_of` function terminates on all valid indexes of the list, which is equivalent to affirming that the union-find forest does not contain any cycles.

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

The `union` operation simply adds an edge between the representatives of the two elements.

```
abbreviation "ufa_union l x y ≡ l[rep_of l x := rep_of l y]"
```

The theory contains several lemmas, including a lemma which states that the invariant `ufa_invar` holds for the initial union-find forest without edges, and that it is preserved by the `ufa_union` operation.

3.3 Union-Find Data Structure

The section below describes the implementation of a union-find data structure. It contains the union-find forest described in Section 3.1, as well as two other lists that are useful for the implementation of the *explain* operation. They are described in the following. For a more detailed explanation refer to [1].

- `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 3.1.
- `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. Each *union* adds exactly one edge to the forest and when it is added, we also label this edge with the arguments of the *union*. Similarly to the `uf_list`, `au` is indexed by the element. For each element b which is not a root, `au` contains the input equation which caused the creation of this edge between b 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 which are roots, the `au` entry is `None`.

Example 1. For a union-find algorithm with 4 variables, the initial empty data structure 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. 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. r is an ancestor of v, hence it is closer to the root. p is a list which contains all the nodes visited on the path from r to v. This definition proved to be very useful for several proofs, as will become clearer in the following sections.

The theory "Path" contains various lemmas about paths, including lemmas about concatenation of adjacent paths. For a complete overwview of the proofs, refer to the Isabelle code. Most theorems about paths could be proven by rule induction on `path`. The most interesting and useful lemma is about the uniqueness 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 $p1$.

A path cannot have length 0 by our definition, therefore 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 we prove a lemma which shows 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 we show 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 $p1'$ and $p2'$ from u to the parent of v.

The following is a graphical representation of the paths:

$u \overset{p1'}{\underset{p2'}{\leftleftarrows}} l!v \leftarrow v.$

We can apply the induction hypothesis, which tells us that $p1' = p2'$. 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 `ufa_union` operation needs to be extended in order to appropiately update the other two lists, therefore we introduce the function `ufe_union`. The `"e"` in the name of the function indicates that it was implemented in order to support an *explain* function.

The function `ufe_union` only modifies the data structure if the parameters are not already in the same equivalence class. The `uf_list` is modified with `ufa_union`. The current union (x, y) is appended to 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 1 and 0, the data structure from Example 1 looks as follows:

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

It has the following graphical representation: $0 \xleftarrow{(1,\,0)} 1 \quad 2 \quad 3$.

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)"
```

Example 3. Let `initial_ufe n` be the empty union-find list with n variables. The result of `apply_unions [(1,0), (3,2), (3,1)] (initial_ufe 4)` looks like this:
$3 \xrightarrow{(3,\,2)} 2 \xrightarrow{(3,\,1)} 0 \xleftarrow{(1,\,0)} 1$.

3.4.2 Helper Functions for Explain

We implement the `explain` function following the description of the first version of the union-find algorithm in the paper by Nieuwenhuis et al. [1]

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 constants 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 is different than the aforementioned graph. It does not have as edges the unions, because `ufa_union l x y` adds an edge between $rep\_of(l, x)$ and $rep\_of(l, y)$, instead of an edge between x and y.

Let (a, b) be the last union made between the equivalence class of x and the one of y. The last union is the edge with the maximal label on the path between x and y. Given the union-find forest, we can calculate the desired output of `explain` by first adding (a, b) to the output and then recursively calling 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).

(a, b) is calculated by finding the lowest common ancestor lca of x and y, and then finding the newest union on the paths from x to lca and from y to lca.

This section describes the helper functions needed for the implementation of `explain`, which calculate the lowest common ancestor, using the function `path_to_root`, and the newest union on a path.

path_to_root

The function `path_to_root l x` computes the path from $rep\_of(l, x)$ 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 front of 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
```

Example 4. If we consider l to be the union-find list of Example 3, then `path_to_root l 3 = [0, 2, 3]`.

It was straightforward 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 directly by computation 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. 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 the common ancestor which is farthest away from the root.

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", which is included in the standard Isabelle distribution.

```
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))"
```

Example 5. If we consider l to be the union-find list of Example 3, then `lowest_common_ancestor l 3 1 = 0`.

Regarding the correctness proof, there are 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 states that any other common ancestor of x and y has a shorter distance from the root. The proof assumes that x and y have the same root.

First, we define that a common ancestor of x and y is a node ca that has a path to x and a path to y.

```
abbreviation "common_ancestor l x y ca ≡
(∃ p . path l ca p x) ∧
(∃ p . path l ca p y)"
```

The lowest common ancestor lca is a common ancestor where each other common ancestor has a shorter distance from the root r.

```
abbreviation "Lowest_common_ancestor l x y lca ≡
(common_ancestor l x y lca ∧
(∀r ca₂ p₁ p₂. (path l r p₁ lca ∧ path l r p₂ ca₂ ∧ common_ancestor l x y ca₂
⟶ length p₁ ≥ length p₂)))"
```

Now we can define the correctness theorem.

```
theorem lowest_common_ancestor_correct:
  assumes "ufa_invar l"
    and "rep_of l x = rep_of l y"
  shows "Lowest_common_ancestor l x y (lowest_common_ancestor l x y)"
```

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 path p_x, 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. We assume that there is a common ancestor lca_2 with a longer path from the root than lca. We 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" 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 l a x y` finds the newest edge on the path from y to x. It is assumed that y is an ancestor of x. The function simply scans all the elements

on the path from y to x and returns the one with the largest index in a, 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
```

Example 6. Let l be the union-find list of Example 3. If we consider the edge labels in the associated unions list instead of the unions they represent, the union-find graph looks like this:
$3 \xrightarrow{1} 2 \xrightarrow{2} 0 \xleftarrow{0} 1$.
From this representation we can see that the newest label on the path from 3 to 0 is 2.

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 the function is correct.

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

3.4.3 Explain

We define the `explain` operation, as described in Subsection 3.4.2. If $rep\_of(l,x) \neq rep\_of(l,y)$, then x and y are not be in the same equivalence class and we cannot produce an explanation. The function terminates when $x = y$. Otherwise, the lowest

common ancestor between x and y is determined. Next, we first compute the newest edge on the x branch, then the one on the y branch, and afterwards we make a case distinction to choose the larger one.

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

Example 7. Let ufe be the union-find data structure of Example 3. We compute the output of `explain ufe 3 1`

We already saw in the previous examples that `lca = 0`, `newest_index_x = 2` and it is easy to see that `newest_index_x` $\geq$ `newest_index_y`. The list of unions is `[(1,0), (3,2), (3,1)]`, therefore a_x `= 3` and b_x `= 1`. The two recursive calls terminate immediately, hence `explain ufe 3 1 = {(3, 1)}`.

3.5 Proofs

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

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., variables which are in bounds. The function `valid_unions` simply tests if all the variables of the `unions` list are in bounds.

```
abbreviation "ufe_valid_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_valid_invar:
  assumes "ufe_valid_invar ufe"
  shows "ufe_valid_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`. We use a lemma from the Theory "Union Find", which states that `ufa_invar` holds after having applied `ufa_union`, and we show that it holds for the initial empty data structure.

```
theorem ufe_valid_invar_imp_ufa_invar: "ufe_valid_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. With this rule we can show that a certain predicate holds for ufe. The base case that needs to be proven is that it holds for the initial data structure. 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_valid_invar ufe"
  assumes "P (initial_ufe (length (uf_list ufe)))"
  assumes "⋀pufe x y. ufe_valid_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 the majority of the proofs about `explain`.

3.5.2 Termination Proof

An important result was to show that the function always terminates if `ufe_valid_invar` holds. We will show this using `apply_unions_induct`, therefore we first need to show an auxiliary lemma. It states that if the function terminates before `ufe_union` is applied, then it also terminates afterwards, assuming that x and y are in the same representative class.

```
lemma explain_domain_ufe_union_invar:
  assumes "explain_dom (ufe, x, y)"
    and "ufe_valid_invar ufe"
```

```
  and "rep_of (uf_list ufe) x = rep_of (uf_list ufe) y"
shows "explain_dom (ufe_union ufe x₂ y₂, x, y)"
```

Proof. We can use the partial induction rule of `explain`, given that our first assumption is that `explain` terminates.

We show only the case when `newest_index_x` $\geq$ `newest_index_y`, because the other case is symmetric to it. The Isabelle code also contains proofs about the symmetry of `explain`, which are used in order to avoid duplicate proofs for the two cases of the `explain` function, but they will not be discussed here, as they are not essential to prove the correctness of the function.

Initially, we remark that the lowest common ancestor and the newest index on a path do not change after a union was applied. Therefore, we will refer to the variables with the same names as in the function definition, e.g., lca, ax, etc., without specifying if we refer to, e.g., `lowest_common_ancestor l x y` or `lowest_common_ancestor (ufe_union ufe x₂ y₂) x y`.

We assume that x and y are in the same representative class after the union. Given that (a_x, b_x) is the newest branch on the path fom a_x to the lowest common ancestor lca, we know that every edge on the path from x to a_x was also present before the union. Therefore $rep\_of(l, a_x) = rep\_of(l, x)$ holds before the union, and we can apply the induction hypothesis and conclude that the recursive call terminates, i.e., `explain_dom (ufe_union ufe x₂ y₂, x, a`$_x$`)`. The edge labeled with (a_x, b_x) is also newer than the newest branch on the path fom y to lca, therefore $rep\_of(l, y) = rep\_of(l, b_x)$, and the induction hypothesis shows that `explain_dom(ufe_union ufe x₂ y₂, b`$_x$`, y)`. The two recursive calls terminate, therefore `explain` terminates. □

Using this result we can prove the termination of `explain`:

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

Proof. We prove it by using `apply_union_induct`.

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

For the induction step, if x and y are not in the same representative class after the union, the function terminates immediately. Otherwise, we can show that x and a_x are in the same representative class before the union, and b_x and y as well. Therefore, we can apply the lemma `explain_domain_ufe_union_invar` to the recursive calls of the function and conclude that `explain` terminates. □

3.5.3 Correctness Proof

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` is the following:

```
theorem explain_valid:
  assumes "ufe_valid_invar ufe"
    and "k ∈ (explain ufe x y)"
  shows "k ∈ 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 for `explain` that Isabelle automatically generates for partial functions. We can prove that k is a valid union, given that each element in `explain ufe x y` originally derives from the `unions` list, which is the list of input equations. In order to use this argument, we need to prove that the index we use for the `unions` list is in bounds. We index the list with `newest_index_x`, which is the one computed with `find_nearest_on_path`.

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

`find_nearest_on_path` returns one of the entries of the *associated unions* list. Therefore we prove a lemma, which shows that the entries in the *associated unions* list are in bounds.

```
lemma au_valid:
  assumes "ufe_valid_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` by `ufe_union` are valid.

Thus we can prove the lemma about the validity of the `explain` function. It remains to show the correctness.

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

Proof. The lemma was shown using the induction rule of `explain`.

For the case where `x = y`, the algorithm returns the empty set. Because of reflexivity, (x, y) is in the equivalence closure of the empty set.

As before, for the remaining cases we consider only the case where `newest_index_x` $\geq$ `newest_index_y`. From the induction hypothesis, we know that $(x, a_x) \in$`(symcl (explain ufe x y))*` and $(b_x, y) \in$ `(symcl (explain ufe x y))*`.

Because of the definition of `explain`, it holds that $(a_x, b_x) \in$ `(explain x y)`. Therefore from the transitivity of the equivalence closure it follows that $(x, y) \in$ `(symcl (explain ufe x y))*`. □

4 Congruence Closure

Congruence closure considers equations containing constants and function symbols. Each function symbol is associated with an arity, which is the number of parameters it accepts. The congruence closure $CC(S)$ of a set of equations S is the smallest reflexive, symmetric, transitive and monotonic superset of the equations. It is inductively defined as follows [22]:

(i) $S \subseteq CC(S)$.

(ii) $\forall a : a = a \in CC(S)$ (reflexivity).

(iii) if $a = b \in CC(S)$, then $b = a \in CC(S)$ (symmetry).

(iv) if $a = b \in CC(S)$ and $b = c \in CC(S)$, then $a = c \in CC(S)$ (transitivity).

(v) $\forall f$ of arity k : if $\forall i \in [1..k] : a_i = b_i \in CC(S)$, then $f(a_1, ..., a_k) = f(b_1, ..., b_k)$ (monotonicity).

This chapter describes the implementation of an algorithm which maintains the congruence closure of a set of equations. The algorithm was presented by Nieuwenhuis and Oliveras [1].

Similarly to the union-find algorithm, the congruence closure algorithm starts with an empty data structure and equations are added to it incrementally with the `merge` function. The function `are_congruent` returns *True* if a given equation is contained in the congruence closure.

The data structure contains a union-find forest, in order to maintain the equivalence classes of the variables. Additionally, there is another forest, called the *proof forest*. It is similar to the union-find forest as it maintains the same equivalence classes, but it differs in the positioning of the edges.

Moreover, it contains two data structures for storing equations containing function symbols. The following sections describe the *proof forest* and then the congruence closure algorithm, as well as their correctness proofs.

As before, the optimizations of path compression and considering the sizes of the representative classes are left out. These optimizations are not relevant for the correctness of the algorithm, and they could later be added to a refinement of the algorithm.

4.1 Input equations

Arbitrary equations can be transformed to equations which have one of the following two forms: either constant equations of the form $a = b$ or equations of the form $F(a, b) = c$, where a, b and c are constants from a set of n constants, which are indexed from 0 to $n - 1$. F is a specific function of arity 2.

The transformation is done by currying and by introducing new constant symbols. For a detailed explanation of how this is done, refer to [10]. In order to understand this thesis, it is irrelevant to know how the transformation is done. The congruence closure of the transformed equations is equal to the congruence closure of the original equations, therefore in the following, we will only consider the transformed equations. The datatype we use for these equations is the one described in Subsection 2.1.

4.2 Congruence Closure Implementation

4.2.1 The proof forest

In order to implement a *cc_explain* operation with a reasonable runtime, the paper by Nieuwenhuis et al. [1] introduces an additional data structure, the *proof forest*. It is a forest which has as nodes the variables and as edges the unions that were made. Each time `ufa_union` is called on the union-find forest, the proof forest is modified with `add_edge`. In order to avoid the creation of cycles, redundant unions are ignored.

add_edge

Analogously to the union-find forest, the proof forest has directed edges and for each equivalence class there is a representative element, or root. We denote the root of an element x in the proof forest pf as $rep\_of(pf, x)$. All edges are directed towards the root. To keep this invariant, each time and edge from x to y is added, all the edges on the path from $rep\_of(pf, x)$ to x 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. As before, the parent of an element x in the proof forest pf is denoted as $pf!x$. The 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 x y = (if pf ! x = x
                  then (pf[x := y])
                  else add_edge (pf[x := y]) (pf ! x) x)"
  by pat_completeness auto
```

Example 8. Assuming the proof forest looks like this
$2 \longleftarrow 3 \quad 1 \longrightarrow 0$
After adding an edge between 3 and 1, the edges from 3 to its root 2 are inverted.
$2 \longrightarrow 3 \longrightarrow 1 \longrightarrow 0$

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

```
lemma add_edge_domain:
  assumes "ufa_invar pf" "rep_of pf x ≠ rep_of pf y"
  shows "add_edge_dom (pf, x, y)"
```

Proof. It can be proven by induction on the length of the path p from $rep\_of(pf, x)$ to x.

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

In the other case x is not a root. This implies that there is a path p' from $rep\_of(pf, x)$ to $pf!x$ which is shorter than the path from $rep\_of(pf, x)$ to x. The path p' is also present in the $pf[x := y]$, because the path does not contain x. Also, the representative of x in $pf[x := y]$ is equal to the representative of y, and $rep\_of(pf[x := y], pf!x)$ is still equal to the old representative $rep\_of(pf, pf!x)$, 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. □

In order to prove the correctness of `add_edge`, we show that `ufa_union l x y` and `add_edge l x y` have the same behaviour. We can conclude from this that the equivalence classes of the union-find forest are the same as those of the proof forest. The theory `Union_Find` from the AFP [21] 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`. The additional assumption `rep_of pf x ≠ rep_of pf y` does not cause problems, because `add_edge` is only executed by the congruence closure algorithm when `rep_of pf x ≠ rep_of pf y`.

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

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

The proof uses various lemmas about the behaviour of `rep_of` after a function update, which depends on where the function update was, and which element we want to find the representative of. The function update corresponds to adding an edge in the forest. These lemmas are proven by analysing how the paths in the forest change after a function update. With these lemmas the induction is easily proven. □

We also show that `add_edge` reverses all the edges from $rep\_of(pf, x)$ to x, and it adds an edge from x to y, i.e., after `add_edge` the forest contains a path from y to $rep\_of(pf, x)$. This path is the original path from $rep\_of(pf, x)$ to x reversed and with one edge added between x and y. `rev` is a function which reverses a list, and `path_to_root` is the function described in Subsection 3.4.2. The proof can be shown by computation induction on `add_edge`.

```
lemma add_edge_correctness:
  assumes "ufa_invar pf"
    "rep_of pf x ≠ rep_of pf y"
  shows "path (add_edge pf x y) y ([y] @ rev (path_to_root pf x)) (rep_of pf
      x)"
```

The proof forest has a similar structure as the union-find forest, therefore we prove that `add_edge` preserves the `ufa_invar` invariant from Section 3.2. This allows us to apply all the lemmas that were proven for the union-find forest also to the proof forest.

```
lemma add_edge_ufa_invar_invar:
  assumes "ufa_invar pf"
    "rep_of pf x ≠ rep_of pf y"
  shows "ufa_invar (add_edge pf x y)"
```

Proof. In order to prove this lemma, we show another lemma which states that the invariant holds after a function update, if the update does not cause the formation of a cycle. Then we show that each function update of `add_edge` does not form a cycle. □

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 `cc_explain` operation. There are two possible types of labels: either one equation $a = b$, or two equations of the type $F(a_1, a_2) = a$ and $F(b_1, b_2) = b$, where a_1 and b_1 were already in the same equivalence class before this union, as well as a_2 and b_2. Both these cases can cause the union between the equivalence classes of a and b. The labeling is implemented by using an additional list, called `pf_labels`. It contains at each index 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. 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. The datatype is defined as follows:

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

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 `pf_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 revered by `add_edge`. `pfl` represents the `pf_labels` list.

```
function (domintros) add_label :: "pending_equation option list ⇒ nat list
    ⇒ nat ⇒ pending_equation ⇒ pending_equation option list"
  where
"add_label pfl pf x lbl =
   (if pf ! x = x
      then (pfl[x := Some lbl])
      else add_label (pfl[x := Some lbl]) pf (pf ! x) (the (pfl ! x)))"
  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 [1].

- `cc_list`: the union-find list, corresponds to the `uf_list`.
- `use_list`: a two-dimensional list which contains for each index a a list of input equations $F(b_1, b_2) = b$ where the representative of b_1 or b_2 is a. Only indexes which symbolize representative elements contain equations.
- `lookup`: a lookup table indexed by pairs of representatives b and c. At index (b, c) it 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 Subsection 4.2.1.
- `pf_labels`: the labels of the proof forest as described in Subsection 4.2.1.
- `input`: a set of the input equations, which will be useful for some proofs in the next sections.

In the following, we shall 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, unless otherwise stated.

4.2.3 Congruence Closure Algorithm

With this data structure we can implement the `merge` function as described in [1]. It takes as parameter the current congruence closure data structure and an equation which it adds to the data structure. The `propagate` function used by `merge` will be described later. It performs unions between the constant symbols in the pending list.

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 index $(rep\_of(l, a_1), rep\_of(l, a_2))$, then we know that $a_1 = b_1$ and $a_2 = b_2$. Therefore, we add $F(b_1, b_2) = b$ and $F(a_1, a_2) = a$ to pending, so that the equivalence classes of a and b will be merged. On the other hand, if the respective lookup entry is `None`, then the equation is added to the lookup table at the index $(rep\_of(l, a_1), rep\_of(l, a_2))$, so that the next time an equation with congruent parameters is input, they will be added together to pending.

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

```
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`. It 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, the union is defined 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
```

The union consists of the previously discussed `ufa_union`, `add_edge` and `add_label`, as well as the function `propagate_loop` which moves all elements from the use list of $rep\_of(l, a)$ to either $rep\_of(l, b)$, 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⦈"
```

`propagate_loop` is defined as a recursive function. It considers each element of the use list of $rep\_of(l, a)$, and either adds it to the use list of $rep\_of(l, b)$ 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"
```

Example 9. Let cc be a congruence closure data structure with an empty use list and lookup, with a union-find list l and with the following proof forest:

$2 \xleftarrow{3=2} 3 \quad 1 \xrightarrow{1=0} 0$

This shows that 2 and 3 are in one equivalence class and 1 and 0 are in the other.

If we apply `merge` with the equation $F(0, 2) = 1$, then the algorithm considers the lookup entry at the index $(rep\_of(l, 0), rep\_of(l, 2))$, which is empty, therefore the equation is added to the use list and to lookup.

If we then apply `merge` with the equation $F(1,3) = 3$, then the lookup entry at index $(rep\_of(l,1), rep\_of(l,3))$ contains $F(0,2) = 1$, and the two equations get added to pending.

Then, `propagate` is executed, which first performs the union of 3 and 1 in the union-find list and it adds a labeled edge to the proof forest, which then looks like this:

$$2 \xrightarrow{3=2} 3 \xrightarrow[F(0,2)\,=\,1]{F(1,3)\,=\,3} 1 \xrightarrow{1\,=\,0} 0$$

After the union, the representative of 3 and 2 has changed, therefore the equations $F(1,3) = 3$ in the use list of the old representative of 2 is moved to the use list of the new representative by `propagate_loop`, and it is also added to the lookup table with the new representative as index.

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. In the case of equations containing F, it verifies if they have the same representative as the corresponding 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 Proofs

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 as for the union-find proofs and 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. They are listed here in order to introduce a name for each step:

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. It includes the following intermediate steps:
   a) The invariant holds after one step in the `propagate_loop`. We shall refer to the two cases of the function 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 this case as `mini_step`.
   d) It holds after `propagate_step`.
   e) It holds after `propagate`.
   f) And finally, it holds after `merge`.

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. The corresponding proofs are easy to show 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 `mini_step`, and we already proved previously that the `ufa_invar` holds after `ufa_union` (in Section 3.2) and after `add_edge` (Subsection 4.2.1). Therefore it also holds after `merge`.

We define a new invariant `same_eq_classes_invar`, 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` $\longleftrightarrow$ `rep_of pf i = rep_of pf j`

This invariant is important for the proofs that consider `add_edge`. That is because we only showed that `add_edge pf x y` terminates if `rep_of pf x` $\neq$ `rep_of pf y`. `propagate` only executes `add_edge` when `rep_of l x` $\neq$ `rep_of l y`. The invariant shows that these two statements are equivalent, therefore `add_edge` always terminates when used inside of the algorithm. In order to prove the invariant, given that the two lists l and pf 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, which is what we showed in Subsection 4.2.1.

Furthermore, for each data structure there is an invariant which states the properties which were informally described in Subsection 4.2.2. The nvariants are presented in the following.

For the use list, the invariant `use_list_invar` 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. For the correctness proof after the `propagate_loop`, we need 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 second parameter of `propagate_loop` is $u!rep\_of(l, a)$, the invariant holds before the `propagate_loop`, and the new representative of a after the union is $rep\_of(l, b)$.

With this assumption we can 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 of the function and only equations of the form $F(a_1, a_2) = a$ are added to $rep\_of(l, a_1)$ and $rep\_of(l, a_2)$. 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 `lookup_invar` 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 `pending_invar` 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 $rep\_of(l, a_2) = rep\_of(l, b_2)$:

Proof. We need to show that in the `propagate_loop` the equation $u1$ that we add to pending has a valid form. 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 proof is analogous for the equations added to pending in `merge`. □

An equivalent invariant holds for the labels in `pf_labels`. They are only modified in `propagate_step`, where the added label comes from `pending`. Therefore, the invariant follows from `pending_invar`. This invariant is called `pf_labels_invar`.

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.

The remaining invariants will be described at a later point, when they become relevant.

4.3.2 Abstract Formalization of Congruence Closure

In order to prove the correctness of the algorithm, we define an abstraction of congruence closure. We cannot use any previously 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 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. It is used to prove equality between congruence closures of the two sets 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.

```
lemma Congruence_Closure_eq:
```

```
  assumes "⋀ a. a ∈ A ⟹ a ∈ Congruence_Closure B"
    "⋀ b. b ∈ B ⟹ b ∈ Congruence_Closure A"
  shows "Congruence_Closure A = Congruence_Closure B"
```

4.3.3 Correctness

To prove that the congruence closure implementation is correct, we show that it is sound and complete, that is, `are_congruent cc eq` returns *True* if and only if the equation *eq* lies in the congruence closure of the input equations.

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

The paper by Nieuwenhuis et al. [1] proves this by stating the folllowing invariant which holds throughout the algorithm. We will call it the `correctness_invar`:

```
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:

- `cc_list_set`, which 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 cc_list_set :: "nat list ⇒ equation set"
  where
    "cc_list_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 = cc_list_set (cc_list cc) ∪ lookup_entries_set cc"
```

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

```
definition correctness_invar :: "congruence_closure ⇒ bool"
  where
    "correctness_invar 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 the set `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 the latter 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 show that `Congruence Closure (representativeE ∪ pending ∪ (u ! rep_of l a))` before the `propagate_loop` 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 elements of pending and representativeE are never removed from the set, therefore they are present also after the loop. □

The first lemma is more difficult to prove. The following is the statement of the lemma. It states that the set `Congruence Closure (representativeE ∪ pending)` before the `propagate_step` is equal to `Congruence Closure (representativeE ∪ pending ∪ (u ! rep_of l a))` after the `mini_step`.

```
lemma correctness_invar_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, therefore 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. Note that the equation eq of the left hand side is equal to $a = b$, because of the first two assumptions.

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

Regarding the `cc_list_set`, all the elements which had the same representative before a union also have the same representative after a union, therefore the equations are also present in the right hand side.

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 `ufa_union`.

The more problematic cases are the equations in lookup. Given that after the `ufa_union` $rep\_of(l, a)$ is not a root anymore, the entries in lookup which have as first or second index $rep\_of(l, a)$ are not in the `lookup_set` after the union. The goal is to prove that these entries are exactly the equations which are present in $u!rep\_of(l, a)$. Until now it was only proven that the equations in the use list are valid, not that they are exhaustive. For this, we introduce a new invariant `use_list_invar2`. It states that all elements that are present in the lookup table at index (i, j) are also in the corresponding use lists of i and j. We will describe this invariant later on.

"$\supseteq$" We need to show that the equations of the `cc_list_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 `cc_list_set` contains equations of the type $c = rep\_of(ufa\_union(l, a, b), c)$ for each element c which is not a root. If the representative of c after the union is the same as before the union, the same equation is in the `cc_list_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 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. As before, we introduce a new invariant `lookup_invar2`. It states that all equations that are in the use list of a root are also present in the lookup table. □

We introduce two new invariants of this form:

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

Unfortunately, these two invariants do not hold in this exact form. If there are two different equations $F(a_1, a_2) = a$ and $F(b_1, b_2) = b$, with $rep\_of(l, a_1) = rep\_of(l, b_1)$ and $rep\_of(l, a_2) = rep\_of(l, b_2)$, then they cannot 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 finish the proof for the invariant `correctness_invar`.

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

(i) $rep\_of(l, a_1) = rep\_of(l, b_1)$,

(ii) $rep\_of(l, a_2) = rep\_of(l, b_2)$,

(iii) $(a = b) \in Congruence\_Closure(representatives\_set \cup pending)$.

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, but they are not merged yet. If we use `representativeE` instead of `cc_list_set`, the invariant is not strong enough in order to prove `correctness_invar_mini_step`.

The final invariants are the following:

- `lookup_invar2`: 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.
- `use_list_invar2`: 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 at the index $(rep\_of(l, c_1), rep\_of(l, c_2))$.

Here follows the proof for `lookup_invar2`:

Proof. We need to show that if the invariant `lookup_invar2` holds before `merge`, then it also holds after the function.

The main aim is to show that it holds after `propagate`. We assume that before `propagate` the invariants `lookup_invar2` and `use_list_invar2` hold. In particular, we observe that for each equation $F(c_1, c_2) = c$ in $u!rep\_of(l, a)$, which we shall refer to with u_a, there is a similar equation in lookup at the index $(rep\_of(l, c_1), rep\_of(l, c_2))$. Hence, there are similar equations 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. Then, 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 `loop1` the lookup table and the use lists are not modified, thus there is nothing to show.

In `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 $rep\_of(l, c_2)$.

Earlier, we observed that this was true before `loop2`. The only use list which was modified in this step is u_a. Therefore the claim holds if none of those two use lists are u_a.

If one (or both) of the use lists are u_a, then the representative of the corresponding element was $rep\_of(l, a)$ before the union, therefore after the union it is $rep\_of(l, b)$. Given that $u1$ is also added to $u!rep\_of(l, b)$ by the function, we can conclude that there is a similar equation also in this use list. □

Here follows the proof for `use_list_invar2`:

Proof. We need to show that if the invariant `use_list_invar2` holds before `merge`, then it also holds after the function.

The main challenge of this proof was to show that it holds after the `mini_step`. The `mini_step` performs a `ufa_union` and afterwards $rep\_of(l, a)$ is not a root anymore. Therefore, if there are some equations $F(c_1, c_2) = c$ in the use list, where the representative of c_1 or c_2 is $rep\_of(l, a)$, they have a new representative after the union. Due to

this, the corresponding *similar* equations in the lookup table are not at the right index anymore.

We shall again refer to $u!rep\_of(l,a)$ with u_a. If $F(c_1,c_2) = c$ was in u_a, then it is removed from the use list after the `mini_step`. In this case, the invariant holds trivially. If c_1 and c_2 are both not in the same equivalence class as a, then the corresponding lookup entry has not changed and the invariant holds. However, there could be equations in the use list at an index which is not $rep\_of(l,a)$, where c_1 or c_2 has the same representative as a. These equations do not have a *similar* equation in lookup after the `mini_step`, but we know from `lookup_invar2` that they have a similar equation in u_a. Therefore, in order show that `use_list_invar2` holds after the `propagate_loop`, it is sufficient to show that for each equation in u_a a similar equation is added (or already present) in the lookup table after the `propagate_loop`.

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 `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`, and $c = d$ is added to pending. This case is exactly the reason why we can only prove that there is a *similar* equation in lookup, and not exactly the same.

Regarding `loop2`, it is entered when lookup contains `None` at the index $(rep\_of(l,c_1), rep\_of(l,c_2))$. Then $F(c_1,c_2) = c$ is added to $u!rep\_of(l,b)$ and to lookup. Obviously, an equation is *similar* to itself, therefore after this lookup contains a similar equation to $F(c_1,c_2) = c$. □

With these two invariants the proof for `correctness_invar` is completed. The proof for `are_congruent_correct` follows from the `correctness_invar` and the definition of `are_congruent`, because the pending list is always empty after the termination of `propagate`.

All the invariants are put together in the invariant `cc_invar`, and using all the previously described proofs about the invariants, we can prove that `cc_invar` holds for the initial empty data structure and that it holds after `merge`.

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

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

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

Proof. We already proved for each individual invariant that they hold after propagate. The proof for `merge` uses the fact that `propagate` terminates, which will be proven in the following section. □

4.3.4 Termination

We already proved that the functions `add_edge` and `add_label` terminate and the only missing proof is the termination of `propagate`. All the remaining functions are simple enough for Isabelle to prove their termination automatically.

In order to prove the termination of `propagate`, we show that the number of equivalence classes strictly decreases in each step of `propagate`, therefore the function terminates at the latest when all the elements belong to the same equivalence class.

The number of equivalence classes is defined as the number of roots in the union-find forest. The `root_set` is the set of all roots. The function `card` returns the cardinality of a set.

```
abbreviation root_set
  where
    "root_set l ≡ {i | i. i < length l ∧ l ! i = i}"

definition nr_eq_classes :: "nat list ⇒ nat"
  where
    "nr_eq_classes l = card (root_set l)"
```

With this, we can show that after a union, $rep\_of(l, a)$ is not a root anymore, therefore there is one less root in the forest.

```
lemma ufa_union_decreases_nr_eq_classes:
  assumes "ufa_invar l" "a < length l"
    "rep_of l a ≠ rep_of l b"
  shows "nr_eq_classes (ufa_union l a b) = nr_eq_classes l - 1"
```

The termination proof for `propagate` follows from this lemma, with an additional assumption that there is at least one variable, so that there is at least one equivalence class.

```
lemma propagate_domain:
  assumes "cc_invar cc" "nr_vars cc > 0"
  shows "propagate_dom cc"
```

Proof. We prove it by induction on the amount of equivalence classes in the union-find forest.

If the pending list is empty, the function terminates. Otherwise, the first element *eq* is taken from the pending list. We define a as `left eq` and b as `right eq`, as in the definition of the `propagate` function.

If a and b are already in the same equivalence class, then the union-find list is not modified, therefore we cannot use the induction hypothesis. Therefore we apply an induction on the length of the pending list.

If a and b are not in the same equivalence class, they are merged by the `propagate_step`, therefore the number of equivalence classes decreases by one according to the lemma `ufa_union_decreases_nr_eq_classes` and we can prove the goal by using the induction hypothesis. □

5 The CC_Explain Operation

We will implement the *cc_explain* operation for congruence closure, leaving the formal proof of correctness open for future work. This section describes the implementation, a validity and termination proof and a proposal of how the correctness could be proven.

The *cc_explain* operation is analogous to the *explain* operation for union-find. For two constants a and b that are congruent, it returns the subset of input equations which explain why a and b are congruent.

5.1 Implementation

The function `cc_explain` has three parameters: two constants and the congruence closure data structure. It assumes that the two constants are congruent in the data structure and returns the set of input equations which caused these two constants to be congruent. The algorithm finds the path in the proof forest between the two constants and returns the labels of the edges on the path. For each edge labeled with two equations $F(a_1, a_2) = a$ and $F(b_1, b_2) = b$, we need to add to the output also the explanation for $a_1 = b_1$ and $a_2 = b_2$. Therefore the function recursively calls the `cc_explain` function with the parameters (a_1, b_1) and (a_2, b_2). In order to avoid adding redundant equations to the output, there is an additional union-find data structure. The additional union-find is local to the `cc_explain` operation and it keeps track of the equations that are already part of the output.

In order to initialize the additional union-find, we define the auxiliary function `cc_explain_aux`. It has three parameters: the congruence closure data structure, the additional union-find and a list of pairs of constants. The output of `cc_explain_aux` contains the explanation for all the pairs of constants in the list, except those which are already equivalent in the additional union-find. The initial union-find is a forest without edges.

```
abbreviation cc_explain :: "congruence_closure ⇒ nat ⇒ nat ⇒ equation set"
  where
    "cc_explain cc a b ≡ cc_explain_aux cc [0..<nr_vars cc] [(a, b)]"
```

The `cc_explain_aux` function takes one pair of elements from the list, then it computes the lowest common ancestor between the two elements. Afterwards, it calls the function

explain_along_path, which has three return values. They are named *output*, *pending* and *new_l*. The *output* is simply the set of labels on the path from the constant to the lowest common ancestor. For each edge on this path labeled with Two $(F(a_1, a_2) = a)$ $(F(b_1, b_2) = b)$, the pairs (a_1, b_1) and (a_2, b_2) are added to the *pending* list. The *new_l* is the additional union-find data structure, modified in order to keep track of the equations that are already in the output.

```
function (domintros) cc_explain_aux :: "congruence_closure ⇒ nat list ⇒ (nat
    * nat) list ⇒ equation set"
  where
    "cc_explain_aux cc l [] = {}"
  | "cc_explain_aux cc l ((a, b) # xs) =
(if are_congruent cc (a ≈ b)
then
  (let c = lowest_common_ancestor (proof_forest cc) a b;
    (output1, new_l, pending1) = explain_along_path cc l a c;
    (output2, new_new_l, pending2) = explain_along_path cc new_l b c
  in
    output1 ∪ output2 ∪ cc_explain_aux cc new_new_l (xs @ pending1 @ pending2)
      )
else cc_explain_aux cc l xs)"
  by pat_completeness auto
```

The additional union-find does not use ufa_union for the union, instead it simply adds the same edge as the one in the proof forest. For a more efficient implementation, the union-find can be replaced by a classical union-find data structure by showing that it has the same equivalence classes as this version. The version used in this implementation was chosen in order to simplify the proofs.

Nieuwenhuis et al. [1] also implement a *highest_node* function, in order to find the element of the additional union-find which is highest in the proof forest. In our version of union-find, this corresponds to the rep_of operation, because we do not use the optimization of checking which equivalence class is bigger. Instead, we just make the union in the given order. When adding this optimization, a *highest_node* function must be also implemented.

The function explain_along_path cc l a c starts at the node a in the proof forest, and recursively traverses all the edges from a to c. It skips those edges that have already been traversed sometime before in the algorithm, i.e., the edges which are present in the additional union-find l. Therefore it starts at the element $rep\_of(l, a)$ and considers the edge to its parent. It adds the label of the edge to the output and adds the edge to l. If necessary, it updates the pending list. The function terminates when it reaches the equivalence class of c in l.

```
function (domintros) explain_along_path :: "congruence_closure ⇒ nat list ⇒
    nat ⇒ nat ⇒
```

```
   (equation set * nat list * (nat * nat) list)"
  where
   "explain_along_path cc l a c =
(if rep_of l a = rep_of l c
then
  ({}, l, [])
else
  (let b = (proof_forest cc) ! rep_of l a in
   (
   case the ((pf_labels cc) ! rep_of l a) of
      One a' ⇒
       (let (output, new_l, pending) = explain_along_path cc (l[rep_of l a
           := b]) b c
        in ({a'} ∪ output, new_l, pending))
      | Two (F a₁ a₂ ≈ a') (F b₁ b₂ ≈ b') ⇒
       (let (output, new_l, pending) = explain_along_path cc (l[rep_of l a
           := b]) b c
        in ({(F a₁ a₂ ≈ a'), (F b₁ b₂ ≈ b')} ∪ output, new_l, [(a₁, b₁), (a₂, b
           ₂)] @ pending))
   )
  )
)"
  by pat_completeness auto
```

Example 10. Let *cc* be the congruence closure data structure of Example 9. We want to compute `cc_explain cc 3 1`.

The lowest common ancestor of 3 and 1 is 1. We call the function `explain_along_path cc l 3 1`, which considers all the edges on the path between 3 and 1. In this case it is one edge labeled $F(1,3) = 3$ and $F(0,2) = 1$. The two equations are added to the output and the pairs $(1,0)$ and $(2,3)$ are added to pending.

For the two remaining pairs in pending, the explanations contain one edge each, and the final result contains all the equations that are present in the proof forest.

The structure of the additional union-find can be formalized with an invariant, which states that all the edges in the additional union-find are also present in the proof forest. It also affirms that the `ufa_invar` holds for the union-find list, and that the list l has the same length as the `proof_forest` list pf.

```
definition explain_list_invar :: "nat list ⇒ nat list ⇒ bool"
  where
   "explain_list_invar l pf ≡ (∀ i < length l. l ! i ≠ i ⟶ l ! i = pf ! i)
       ∧
(length l = length pf) ∧ ufa_invar l"
```

Each time an edge is added to the union-find in `explain_along_path`, it is an edge which is also present in the proof forest, therefore it is easy to prove that this is an

invariant.

This invariant implies the following useful lemma, which states that each path in the additional union-find is also present in the proof forest.

```
lemma explain_list_invar_paths:
  "path l a p b ⟹ explain_list_invar l pf ⟹ path pf a p b"
```

5.2 Termination

We need to prove the termination of the two functions `explain_along_path` and `cc_explain_aux`.

The function `explain_along_path` starts at the node a and recursively considers the parent node until it reaches c. Therefore the function terminates if c is an ancestor of a.

```
theorem explain_along_path_domain:
  assumes "cc_invar cc"
    "explain_list_invar l (proof_forest cc)"
    "path (proof_forest cc) c p a"
  shows "explain_along_path_dom (cc, l, a, c)"
```

Proof. The proof is by induction on the length of the path p.

If the path has length 1 or if `rep_of l a = rep_of l c`, the function terminates immediately.

If the path is longer, then the union-find l contains the following path:
$a \longrightarrow rep\_of(l, a)$
and the proof forest pf contains the same path, which continues until it reaches c:
$a \longrightarrow rep\_of(l, a) \rightarrow pf!rep\_of(l, a) \longrightarrow c$.

The long arrows reperesent paths of arbitrary length, including paths with no edges, and the short arrow represents exactly one edge.

Therefore there is a shorter path from c to $pf!rep\_of(l, a)$ in the proof forest and we can apply the induction hypothesis for the recursive call of `explain_along_path`. Given that the recursive call terminates, the function also terminates. □

The function `explain_along_path` is only executed in `cc_explain_aux` when c is the lowest common ancestor of a and b, therefore the assumption that there is a path from c to a always holds when the function is executed.

In order to prove the termination of `cc_explain_aux`, we use the same idea as for the termination proof of `propagate` in Subsection 4.3.4. We consider the number of equivalence classes of the additional union-find, and prove that the number of equivalence classes decreases in each call of `explain_along_path`.

First, we prove that the number of equivalence classes decreases with a function update, if the updated index is a root:

```
lemma list_upd_eq_classes:
  assumes "a ≠ b" "l ! a = a" "a < length l"
  shows "nr_eq_classes (l[a := b]) < nr_eq_classes l"
```

By examining the function `explain_along_path`, we see that each time the union-find list is updated, the updated index is a root, therefore the number of equivalence classes decreases. However, if `rep_of l a = rep_of l c`, the union-find list is never updated, therefore we can only show that the number of equivalence classes does not increase after `explain_along_path`:

```
lemma explain_along_path_eq_classes:
  assumes "cc_invar cc"
    "explain_list_invar l (proof_forest cc)"
    "path (proof_forest cc) c p a"
  shows "nr_eq_classes (fst (snd (explain_along_path cc l a c))) ≤
      nr_eq_classes l"
```

We can show that it strictly decreases if the pending list is not empty after the function `explain_along_path`, because we always update the union-find list when we add elements to pending:

```
lemma explain_along_path_eq_classes_if_pending_not_empty:
  assumes "cc_invar cc"
    "explain_list_invar l (proof_forest cc)"
    "path (proof_forest cc) c p a"
    "snd (snd (explain_along_path cc l a c)) ≠ []"
  shows "nr_eq_classes (fst (snd (explain_along_path cc l a c))) <
      nr_eq_classes l"
```

With these two lemmas, we can prove that `cc_explain_aux` terminates:

```
theorem cc_explain_aux_domain:
  assumes "cc_invar cc"
    "explain_list_invar l (proof_forest cc)"
  shows "cc_explain_aux_dom (cc, l, xs)"
```

Proof. We show it by using a nested induction: the outer induction is on the number of equivalence classes, and the inner induction is on the length of *xs*. If *xs* is empty, then the function terminates.

If `are_congruent (a ≈ b)`, then we do a case distinction: if both `pending1` and `pending2` are empty, then the length of *xs* decreases and we can use the induction hypothesis of the inner induction. In this case we need the previously proven lemma `explain_along_path_eq_classes`, because we also need to show that the number of equivalence classes does not increase. If one of the pending lists is not

empty, we use `explain_along_path_eq_classes_if_pending_not_empty` to show that the `nr_eq_classes` has decreased, and we use the first induction hypothesis.

If `are_congruent (a ≈ b)` does not hold, then the length of *xs* decreases and we can use the second induction hypothesis. □

5.3 Validity

Similarly to the `explain` operation for union-find (see Subsection 4.3.3), we can prove that the equations in the output of `cc_explain` are a subset of the input equations.

```
theorem cc_explain_valid:
  assumes "cc_invar cc" "validity_invar cc"
  shows "cc_explain cc a b ⊆ input cc"
```

We assume the `cc_invar` which we have previously proven, and we introduce a new invariant, the `validity_invar`. It states that all the equations in `pf_labels`, `lookup`, `use_list` and `pending` are equations from the `input` set. In order to show that this is an invariant, we remark that all the new equations added in `merge` to any data structure are also added to the `input` set. In `propagate`, no new equations are added, the equations are simply moved around between different lists.

With this invariant, we can prove that the output of `explain_along_path` is valid.

```
lemma explain_along_path_valid:
  assumes "cc_invar cc" "validity_invar cc"
    "explain_list_invar l (proof_forest cc)"
    "path (proof_forest cc) c p a"
  shows "fst (explain_along_path cc l a c) ⊆ input cc"
```

Proof. We already proved that `explain_along_path` terminates, therefore we can use the induction rule of the function.

If `rep_of l a = rep_of l c`, then the output of `explain_along_path` is the empty set, therefore the proof is trivial.

Else, the new equations added to the output derive from the proof forest, more specifically from `((pf_labels cc)! rep_of l a)`. The `validity_invar` states that all edge labels are valid, that is, the entries in `pf_labels` are valid for indexes which are not roots. If the index is a root, there is no outgoing edge, therefore those entries are not valid. In order to use the invariant, we need to prove that $rep\_of(l, a)$ is not a root in the proof forest. Given that we assumed that there is a path from c to a in the proof forest, we know that c is nearer to the root than a and we can show that if $rep\_of(l, a)$ was a root in the proof forest, then c can only be equal to the root, therefore `rep_of l a = rep_of l c`, which is a contradiction. □

From this lemma we can easily show the theorem `cc_explain_valid` by computation induction on `cc_explain_aux`. We can use the induction rule because we have already proven the termination of `cc_explain_aux`.

5.4 Correctness

Proving the correctness of `cc_explain` is more complicated than expected. The most intuitive strategy would be to use the induction rule of `cc_explain_aux`, but it is not strong enough to prove it. Our goal is to show that the two variables are congruent under the congruence closure of the output of `cc_explain`.

```
theorem cc_explain_correct:
  assumes "are_congruent cc (a ≈ b)" "cc_invar cc"
  shows "(a ≈ b) ∈ Congruence_Closure (cc_explain cc a b)"
```

It is possible to show for `explain_along_path` that $a = b$ is in the congruence closure of the output, pending and the `representative_set l`. Intuitively, the function adds the necessary equations to output and pending, except if the elements of the equations are already in the same equivalence class in the additional union-find.

```
lemma explain_along_path_correctness:
  assumes "explain_along_path_dom (cc, l, a, c)"
    "explain_along_path cc l a c = (output, new_l, pend)"
    "path pf c pAC a"
    "cc_invar cc"
    "explain_list_invar l (proof_forest cc)"
  shows "(a ≈ c) ∈ Congruence_Closure (cc_list_set l ∪ output
∪ pending_set_explain pend)"
```

Given that at after the termination of the function `pending` and the initial union-find l are empty, this should imply that $a = b$ is in the congruence closure of `cc_explain`. However, this argument is not applicable. I will illustrate this with an example.

Example 11. We consider the following proof forest:

$$\mathrm{a} \xrightarrow[F(b,c)=b]{F(a,c)=a} \mathrm{b} \xleftarrow{\mathrm{b = c}} \mathrm{c} \qquad \mathrm{d}$$

The `cc_invar` holds, in particular `pf_labels_invar` holds. This invariant states that for the labels in the proof forest of the type $F(a_1, a_2) = a_3$ and $F(b_1, b_2) = b_3$ it holds that $rep\_of(l, a_1) = rep\_of(l, b_1)$ and $rep\_of(l, a_2) = rep\_of(l, b_2)$.

If we call `cc_explain cc a b` it terminates and returns the two equations $F(a, c) = a$ and $F(b, c) = b$. However, $a = b$ is not in the congruence closure of these two equations. The problem in this case is that when `explain_along_path cc l a b` is called, it simply adds (a, b) to pending, adds the two equations to output and adds the edge between a and b to the additional union-find. In the recursive call of the function, (a, b) is taken

from pending. The algorithm sees that it has already considered this edge, and returns an empty output.

The previous example shows that the invariant `cc_invar` together with the lemma `explain_along_path_correctness` are not enough to prove that `cc_explain` is correct. However, it does not show that `cc_explain` is incorrect, because the proof forest in the example cannot be produced by subsequent merges. For the labels in the proof forest of the type $F(a_1, a_2) = a_3$ and $F(b_1, b_2) = b_3$, it not only holds that $rep\_of(l, a_1) = rep\_of(l, b_1)$ and $rep\_of(l, a_2) = rep\_of(l, b_2)$, but also that those representatives have been equal before the addition of the edge. Therefore, `explain_along_path` will never add equations to pending which are only congruent if the output equations are congruent.

An idea for the proof would be to define an additional invariant that expresses what was explained in the previous paragraph. Alternatively, we could define the correctness of `cc_explain` as an invariant, and prove that it is an invariant of merge.

In order to prove that it is an invariant, we need to show that after a `merge` operation, the path between two elements remains the same in the proof forest. This implies that `cc_explain` considers the same edges before and after the `merge`. The difficulty with this approach is that some edges could be inverted by `add_edge`, which means that the lowest common ancestor of two elements could also change. In other words, the `cc_explain` function before and after `merge` considers the same edges but in a completely different order. As a result, we also need to show that the order of the edges which `cc_explain` considers does not matter.

6 Conclusion

This thesis described the implementation of an *explain* operation for the union-find algorithm implemented by Lammich [2] in Isabelle/HOL. It can be used to generate certificates of equality for equations between constants. We formally proved that the algorithm terminates and returns a correct output. The implementation can be used independently of the congruence closure algorithm.

The paper by Nieuwenhuis et al. [1] shows the minimality of the set of equations produced by the *explain* operation. For future work, this property could be formalized and verified in Isabelle.

The other main focus of this thesis was the implementation of the functions `merge` and `are_congruent`, as described in [1]. They maintain the congruence closure of a set of equations. Invariants of the `merge` operation were identified and proven. The algorithm terminates and is sound and complete.

The *cc_explain* operation for congruence closure can produce certificates which validate the congruence of two terms. We proved the termination of the function and we discussed its correctness. The formalization of the correctness proof for the *cc_explain* function is still open for future work.

The code of this thesis can be used in the future in order to implement an automatic proof strategy for Isabelle/HOL. It can be refined by using the imperative code framework of Isabelle [23] and the optimizations which were left out of this thesis can be included in the refinement. In order to use it as a proof strategy that works for arbitrary terms, it is also necessary to implement the initial transformations of the equations to the form used by this algorithm.

The applications which use the *explain* and *cc_explain* operations usually need to reconstruct the proof for certain equations and they could do so more easily if the output of the operations was not given in the form of a set of equations, but rather as a tree of equations. This modification of the output format is open for future work.

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] P. J. Downey, R. Sethi, and R. E. Tarjan. "Variations on the Common Subexpression Problem." In: *Journal of the ACM* 27.4 (Oct. 1980), pp. 758–771. DOI: 10.1145/322217.322228.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] S. Conchon and J.-C. Filliâtre. "A persistent union-find data structure." In: *Proceedings of the 2007 workshop on Workshop on ML - ML '07*. ACM Press, 2007. DOI: 10.1145/1292535.1292541.

[14] M. P. L. Haslbeck and P. Lammich. "Refinement with Time - Refining the Run-Time of Algorithms in Isabelle/HOL." en. In: (2019). DOI: 10.4230/LIPICS.ITP.2019.20.

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

[16] *congruence tactic in the Coq documentation*. https://coq.inria.fr/distrib/V8.11.2/refman/proof-engine/tactics.html#coq, Accessed 25.07.2022.

[17] 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.

[18] J. Blanchette. *Hammering Away - A User's Guide to Sledgehammer for Isabelle/HOL*. Dec. 2021.

[19] 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.

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

[21] 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.

[22] D. Kapur. "A Modular Associative Commutative (AC) Congruence Closure Algorithm." en. In: Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. DOI: 10.4230/LIPICS.FSCD.2021.15.

[23] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. "Imperative Functional Programming with Isabelle/HOL." In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 134–149. DOI: 10.1007/978-3-540-71067-7_14.