DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Proof of the Amortized Time Complexity of an Efficient Union-Find Data Structure in Isabelle/HOL

Adrián Löwenberg Casas

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Proof of the Amortized Time Complexity of an Efficient Union-Find Data Structure in Isabelle/HOL

Beweis der amortisierten Laufzeit einer effizienten Union-Find-Datenstruktur in Isabelle/HOL

Author:	Adrián Löwenberg Casas
Supervisor:	Prof. Tobias Nipkow, Ph.D.
Advisor:	Maximilian P.L. Haslbeck, M.Sc.
Submission Date:	September 16th, 2019

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

Munich, Adrián Löwenberg Casas

Acknowledgments

I want to thank my advisor Maximilian Haslbeck for his infinite patience with me and his valuable recommendations. I am very grateful to Dr. Peter Lammich and Simon Wimmer for introducing me to Isabelle and the world of computer-assisted proofs, as well as to Armaël Guéneau for helping me set up Coq.

A very special thank you goes to my father for introducing me to programming when I was a young teenager but especially for his unconditional support.

Abstract

Union-Find is a classical data structure whose complexity analysis is famously non-trivial. In this thesis we prove the α-bound amortized time complexity of an efficient imperative implementation of this data structure. We first revise the history of this emblematic result by Tarjan [Tar75] and arrive at the modern proof by Alstrup et al. [Als+14].

To reproduce this proof in a formal context within Isabelle/HOL, we first gather the mathematical and technical tools required, most prominently a more comprehensive theory about the Ackermann function than the one already available in the Isabelle/HOL distribution, properties about its inverses, as well as the framework implementing Separation Logic with Time Credits for Imperative/HOL, which already contained a non optimal implementation of this data structure. We then follow closely the work of Charguéraud and Pottier [CP17], which formalized this proof in a similar framework in Coq.

In the end, we prove the asymptotically optimal bound of the operations in an efficient implementation of the Union-Find data structure. The whole proof in Isabelle is available under [Löw19]. As with any other program in Imperative/HOL, the implementation can be exported to several languages.

Contents

1 Introduction

One of the main subjects of computer science is the study of algorithms and data structures. When a new algorithmic idea is developed two problems arise, firstly if the algorithm solves the problem *(functional correctness)* and secondly whether the running time is always reasonable *(worst-case running time analysis)* for every operation, or for a sequence of operations *(amortized running time analysis)*. In this thesis we provide computer assisted formal proofs of both properties for the Union-Find data structure.

In the first chapter we present the data structure abstractly and our implementation of it. In the second chapter we introduce the Ackermann function, its properties and those of its inverse, which is asymptotically the amortized running time for the operations provided by the data structure. Finally, we outline the proof of the functional correctness and amortized time complexity of the data structure developed in Isabelle/HOL and explain the significance of the main results.

1.1 Union-Find and Partial Equivalence Relations

The Union-Find data structure mathematically models a partial equivalence relation over a finite domain. This is implemented efficiently by disjoint set forests, which are forests of rooted trees, each tree representing an equivalence class, with every node being an element and the root a representative of the equivalence class. Classically, the operations supported are *Union*, where two equivalence classes are merged into one, and *Find*, where it is checked whether two given elements belong to the same class.

In our implementation, we represent the disjoint set forest by an array representing the child-parent relationship, every index in the array represents a node, and the content of the array at that index is the parent of the node. A root node is modeled as having itself as a parent.

Note that this approach requires every node to be just a number, when in practice a user of the data structure might want to store arbitrary data in it. This is however easily solvable by the user by associating the index of the node with the data required by the use case, which does not need to be stored in the array.

Additionally, we use another array of the same length to store the *ranks* of the nodes, its significance will reveal itself during the running time analysis.

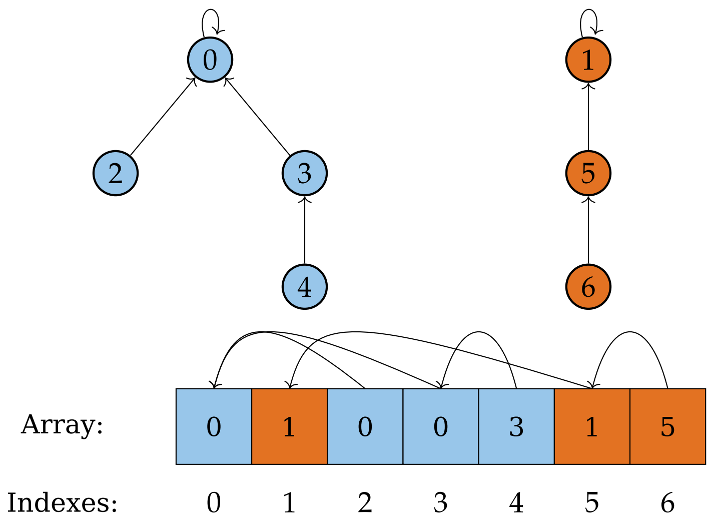

Figure 1.1: Two equivalence classes and their representation as a disjoint set forest.

Using arrays is arguably more efficient than a pointer-based implementation. Of course, this fixes the domain size to the length of the array at the moment of instantiating the data structure. This restriction could easily be removed by using a dynamic array instead. If its operations occur in amortized constant time, the resulting amortized time of the *Union* and *Find* operations is the same. We will try to give hints to where adjustments would need to be made to the mathematical analysis to accommodate the varying domain. This and other ideas for future work will be marked by ✍.

1.1.1 Path Compression and Union by Rank

The efficiency of this implementation relies on two heuristics that dramatically improve running time. Very roughly, as with any data structure involving trees, we want to keep the trees as flat as possible, to minimize the number of iterations to the root.

- **Union by rank**: When merging two trees, in order for the resulting tree to be as flat as possible, the root of the tree with more nodes should be the new root. It is possible to keep track of the size of every tree, as did Lammich [Lam19] in his implementation, and according to Tarjan et al. [TL84] the resulting time complexity would be the same, however we follow Charguéraud and Pottier [CP17] and use the *rank* approximation, which is an upper bound on the size of the tree. On initialization, every node is a childless root and has a rank of zero. When merging two trees, the root of the tree with the largest rank becomes the new root and its rank is incremented by one.
- **Path compression**: Every time the representative of a node is searched for, an iteration from said node to the root is performed. To achieve a flatter tree, the path is compressed, this means that every node visited during the iteration is updated to have the root as its parent. This is without penalty to the running time of the original representative search operation. Every subsequent iteration to the root from any updated node is therefore only one step.

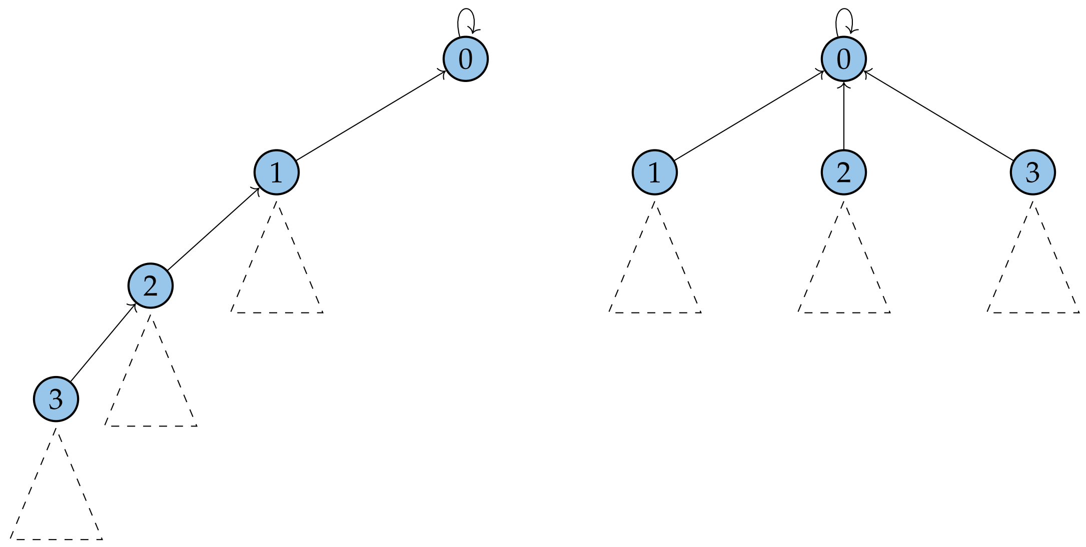

Figure 1.2: The state before and after performing path compression at the node 3.

1.1.2 Implementation

Our implementation is based on the one by Lammich, for which Haslbeck and Lammich provided a non-optimal amortized time complexity bound [HL19]. This implementation did not compress on *Union*, so its running time could not be optimal. The code has been adapted to use ranks and to compress on every occasion. The data structure is represented by two arrays, the disjoint set forest and the rank array, and such is the implementation in Imperative/HOL:

```
type_synonym uf = nat array × nat array
```

Listing 1.1: The Datatype Representing the Data Structure

Initialization

On initialization the size of the arrays has to be fixed, so it is provided as a parameter to `uf_init`:

```
definition uf_init :: nat ⇒ uf Heap where
uf_init n ≡ do {
      l ← Array.of_list [0..<n];
      szl ← Array.new n (0::nat);
      return (szl,l)
}
```

Listing 1.2: The Initialisation Function

Find

The *Find* operation, or `uf_cmp`, is implemented here in a modular way, which eases the correctness proof and the running time analysis. The usual implementations in the literature, prominently in CLRS [Cor+09] and also the one by Charguéraud and Pottier [CP17] use an arguably more natural pointer structure to represent the forest, search for a representative recursively, and do the path compression on backtracking. Our implementation relies on the `uf_rep_of_c` function, which searches for the representative first (`uf_rep_of`), and on a second pass, it compresses the equivalence class (`uf_compress`). The `uf_cmp` function retrieves both representatives and returns whether they are equal.

```
partial_function (heap) uf_rep_of :: nat array ⇒ nat ⇒ nat Heap
where [code]:
uf_rep_of p i = do {
      n ← Array.nth p i;
      if n=i then return i else uf_rep_of p n
}
```

Listing 1.3: The Representative Search Function

```
partial_function (heap) uf_compress :: nat ⇒ nat ⇒ nat array ⇒ unit Heap
where [code]:
uf_compress i ci p = (
      if i=ci then return ()
      else do {
            ni ← Array.nth p i;
            uf_compress ni ci p;
            Array.upd i ci p;
            return ()
})
```

Listing 1.4: The Iterated Path Compression Function

```
definition uf_rep_of_c :: nat array ⇒ nat ⇒ nat Heap where
uf_rep_of_c p i ≡ do {
      ci ← uf_rep_of p i;
      uf_compress i ci p;
      return ci
}
```

Listing 1.5: The Representative Search and Compression Function

```
definition uf_cmp :: uf ⇒ nat ⇒ nat ⇒ bool Heap where
uf_cmp u i j ≡ do {
      let (s,p) = u;
      n ← Array.len p;
      if (i≥n ∨ j≥n) then return False
      else do {
            ci ← uf_rep_of_c p i;
            cj ← uf_rep_of_c p j;
            return (ci=cj)
      }
}
```

Listing 1.6: The Find Operation

Union

The *Union* operation, here `uf_union`, also makes use of the `uf_rep_of_c` function, as it needs to find the roots of the trees representing the equivalence classes of its arguments to merge them. Beginning in line 8 of the `uf_union` code, the ranks of the roots are retrieved, and then the root with the highest rank becomes the new root and its rank is incremented.

```
 1 definition uf_union :: uf ⇒ nat ⇒ nat ⇒ uf Heap where
 2 uf_union u i j ≡ do {
 3       let (r,p) = u;
 4       ci ← uf_rep_of_c p i;
 5       cj ← uf_rep_of_c p j;
 6       if (ci=cj) then return (r,p)
 7       else do {
 8             ri ← Array.nth r ci;
 9             rj ← Array.nth r cj;
10             if ri<rj then do {
11                   Array.upd ci cj p;
12                   (if (ri=rj) then do {
13                         Array.upd cj (ri+1) r
14                   } else return r);
15                   return (r,p)
16             } else do {
17                   Array.upd cj ci p;
18                   if (ri=rj) then do {
19                         Array.upd ci (ri+1) r;
20                         return (r,p)
21                   } else return (r,p)
22             }
23       }
24 }
```

Listing 1.7: The Union Operation

1.2 Applications

The Union-Find data structure is important and used in several foundational algorithms, as equivalence relations are a very flexible modeling tool. An equivalence relation can represent the partition of a set, for example the connected components of an undirected graph. The Union-Find data structure can be used for example in an efficient implementation of the Kruskal algorithm to check if two vertices are connected or whether a cycle is created when adding an edge. The earlier version of Union-Find implemented in Imperative/HOL is used by Haslbeck et al. to implement Kruskal [HLB19]. The version presented here is a drop-in replacement which automatically would improve the time complexity of the whole algorithm (see 1.1.2). Another important application is an efficient implementation of Huet's algorithm for unification [Kni89].

2 The Ackermann Function

In this chapter we define the Ackermann function exactly as Charguéraud and Pottier [CP17], who follow the definition by Alstrup et al. [Als+14] and, more clasically, by Tarjan [Tar75]. We then prove some important properties about it, including monotonicity in every argument and under iteration. The theory Ackermann.thy can be used together with InverseNatNat.thy completely independently from the rest of the theories presented in this thesis.

2.1 Relationship to other Ackermann Function Definitions

It is important to note that there are several definitions of the Ackermann function in the literature. The definition already existing in Isabelle (HOL/ex/Primrec.thy) follows Mendelson [Men09, pg. 345], which in turn follows the classical definition by Ackermann [Ack]. All three of these occurrences focus on the property of this function not being primitive recursive.

The definition by Charguéraud and Pottier follows Tarjan, who states it is a “slight variant of Ackermann’s function; it is not primitive recursive” [Tar75]. The reasons for this variation are not clear to us, and the statement about it not being primitive recursive seems to not have been proved, but it rather probably follows from the fundamentally similar definitions. However, this property is of no interest for the purpose of this thesis except for the qualitative statement that a function growing faster than every primitive recursive function is *very* fast-growing, so its inverse is *very* slow-growing.

2.2 Definition

The definition by Charguéraud and Pottier is modular, and enables proving the properties of the Ackermann function through properties about simpler functions.

Definition. *Ackermann*

$$\operatorname{astep} f\, x := f^{(x+1)}\, x \tag{2.1}$$

$$A\, k\, n := (\operatorname{astep}^{(k)} \operatorname{Suc})\, n \tag{2.2}$$

A is the Ackermann function, and it satisfies the following alternative equation, unfolding all definitions:

$$A\, k\, n = ((\lambda\, f\, x.\, f^{(x+1)}\, x)^{(k)} \operatorname{Suc})\, n \tag{2.3}$$

The notation $f^{(n)}\, x$ corresponds to Isabelle's $(f\, \hat{}\hat{}\, n)\, x$ and refers to function iteration:

$$f^{(n)}\, x := \begin{cases} x & n = 0 \\ f^{(n-1)}\, (f\, x) & n > 0 \end{cases} \tag{2.4}$$

And Suc *is, for the purposes of this thesis, just* $\operatorname{Suc} x := x + 1$.

The definition abstracts the recursion inherent to the Ackermann function and thus enables the use of the existing lemmas about function iteration, in particular the powerful *funpow_mono2* (see 2.4). Of course, this definition corresponds to the equations by Tarjan:

Lemma. *Tarjan characteristic equations*

$$A\, 0\, x = x + 1 \qquad \textit{Ackermann\_base\_eq} \tag{2.5}$$

$$A\, (k+1)\, x = (A\, k)^{(x+1)}\, x \qquad \textit{Ackermann\_step\_eq} \tag{2.6}$$

2.3 Explicit Equations and Bounds

The Ackermann function satisfies the following explicit equations for the first values of k, which gives an idea of how fast the function grows:

Lemma. *Ackermann_1_eq*

$$A\, 1\, x = 2x + 1 \tag{2.7}$$

Lemma. *Ackermann_2_eq*

$$A\,2\,x = 2^{x+1}(x+1) - 1 \tag{2.8}$$

It also fulfills the following lower bound for $A\,2$:

Lemma. *Ackermann_2_log_lower_bound*

$$n \leq A\,2\,(\log n) \tag{2.9}$$

Where log *is the discrete binary logarithm as defined in the Isabelle/HOL distribution (HOL-Library.Discrete).*

As well as the following, already impressive, lower bound for $A\,3$:

Lemma. *Ackermann_3_lower_bound*

$$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{x\,times} \leq A\,3\,x \tag{2.10}$$

Or, more formally: $((\hat{\ })\,2)^{(x+1)}\,0 \leq A\,3\,x$

2.4 Further Contributions in this Theory

There are some more properties of the Ackermann function which have been generalized and proved for general iterated functions, as well as many technical lemmas which enable for more direct proofs about monotonicity and inflationarity of iterated functions. A very useful lemma which already existed in the Nat.thy theory was *funpow_mono2*, which states:

Lemma. *funpow_mono2*
Assume: *mono f; $i \leq j$; $x \leq y$; $x \leq f\,x$*

$$f^{(i)}\,x \leq f^{(j)}\,y \tag{2.11}$$

We provided among others the following lemmas, which extend this lemma to the comparison of two functions, under some assumptions about the relationship between them:

Lemma. *compow_mono_in_f*
Assume: *mono f; mono g; $\forall x. f\, x \leq g\, x$*

$$f^{(i)}\, x \leq g^{(i)}\, x \tag{2.12}$$

Lemma. *compow_mono_in_f_and_i'*
Assume: *mono f; mono g; inflationary g; $\forall x\, y.\, x \leq y \rightarrow f\, x \leq g\, y$; $x \leq y$; $i \leq j$*

$$f^{(i)}\, x \leq g^{(j)}\, y \tag{2.13}$$

We also proved some properties about the asymptotics of the Ackermann function, which of course goes to infinity as any variable grows, which are proved following some more general lemmas about asymptotics of iterated strictly inflationary functions.

2.5 Inverses of Functions between Natural Numbers

In this section we present two notions of inverses for functions f : nat $\Rightarrow$ nat. To our knowledge, this notation was introduced by Chargeraud et al. [CP17] to modularize the definition of the inverse Ackermann function, as well as to ease some proofs by providing lemmas to change the proof obligations in both directions between a function and its inverse. In Isabelle/HOL, this theory provides a locale which specifies the requirements to the function. For the sake of simplicity we require the function to be strictly monotonic and to tend to infinity. Many of the lemmas, as well as the definition itself, only require monotonicity. Without the asymptotic condition however, one cannot prove the existence of the inverse at every point.

This theory can be useful independently of the current context of the inverse Ackermann function. Some important concepts in the abstract analysis of the Union-Find data structure which are not directly inverses of the Ackermann function are defined using α_f and β_f, as this provides many useful lemmas.

2.5.1 Definitions

Definition. *Upper inverse*

$$\alpha_f\, y := \min\{x \mid y \leq f\, x\} \tag{2.14}$$

$\alpha_f\, y$ is therefore the smallest x for which $y \leq f\, x$ holds. As the function is monotonic, this holds for all further x, so $y \leq f\, x$ is equivalent to $\alpha_f\, y \leq f\, x$, which makes α_f an upper inverse of f.

Definition. *Lower inverse*

$$\beta_f\, y := \max\{x \mid f\, x \leq y\} \tag{2.15}$$

Here the existence is not guaranteed, as f may start above y, so we need to require $f\, 0 \leq y$. If we have a $\beta_f\, y$, then it is the largest x, for which $f\, x \leq y$ holds. By monotonicity the property holds for all smaller x. Therefore $f\, x \leq y$ is equivalent to $x \leq \beta_f\, y$, which makes β_f a lower inverse of f.

This functions of course differ by at most one, and coincide if y is the image under f of some x. In Isabelle, they are defined by the *Least* and *Greatest* operators, and the existence is shown in separate lemmas.

Important lemmas that always apply to these functions are monotonicity and that they tend to infinity (this is of course important for the inverse Ackermann function).

2.6 Inverse Ackermann Function

2.6.1 Historical Definitions

The name α_f suggestively relates to the historical definition of α as the “functional inverse” of the Ackermann function [Tar75]. According to the Nist [Bla04] which is the most canonical source we could find the function is defined as:

$$\alpha\, m\, n := \min\{k \geq 1 \mid A\, k\, \lfloor m/n \rfloor > \log n\} \tag{2.16}$$

Another important source is Nivasch’s Ph.D. thesis [Niv09], which uses the inverse Ackermann function for a lower bound in computational geometry. The definition is very similar to ours, albeit with rather differing notation, but with very similar bounds and same asymptotic behavior. There is also a section dedicated more thoroughly to the different versions of A and α in the literature.

2.6.2 Definition

We naturally want to use our theory about inverses of natural functions to define α, as did Charguéraud and Pottier We define two versions of the function, one would

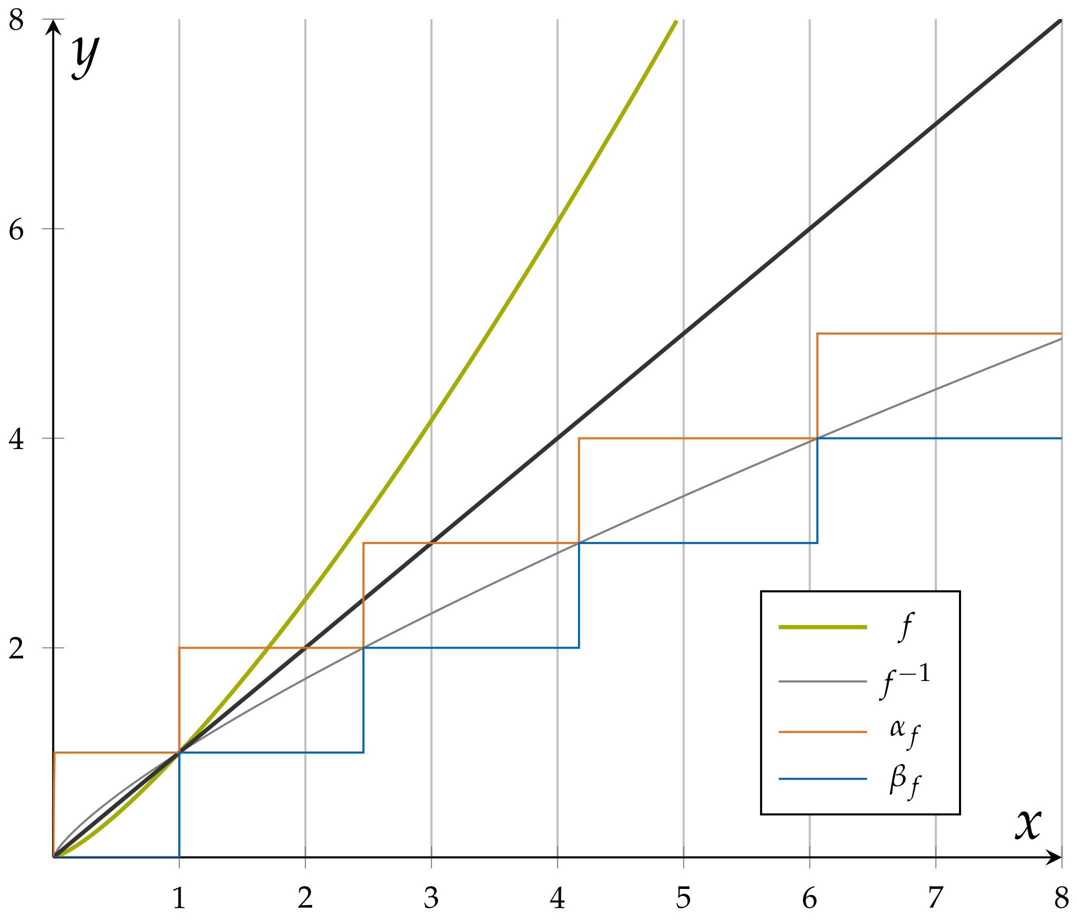

Figure 2.1: Visualization of α_f and β_f and their relationship to an exact real inverse. Only the integer points are defined in our case.

be the more natural single argument version of the inverse and the other follows Alstrup et al. [Als+14] and adds a parameter.

Definition. *Inverse Ackermann function*

$$\alpha\, n := \alpha_{\lambda k.\, A\, k\, 1}\, n \tag{2.17}$$

Which unfolded yields:

$$\alpha\, n = \min\{k \mid A\, k\, 1 \geq n\} \tag{2.18}$$

This is exactly the definition given in CLRS [Cor+09, pg. 574]. The choice of the constant 1 seems arbitrary, but fixing the second argument of A does not change much, as the first one makes the function more powerful as it grows. Following Alstrup et al. we also define a version of α for an arbitrary parameter as a second argument to A:

Definition. *Parametrized inverse*

$$\alpha_r\, n := 1 + \alpha_{\lambda k.\, A\, k\, r}\,(n+1) \tag{2.19}$$

Which unfolded yields:

$$\alpha_r\, n = 1 + \min\{k \mid A\, k\, r \geq (n+1)\} \tag{2.20}$$

This definition is very similar to Nivasch's, except for some shifting by one. Throughout the rest of the thesis, we will assume a fixed and positive r, and prove all statements related to the inverse Ackermann function using α_r, including the final Hoare-Triples, which shows that the election of a specific r is not important.

We also prove lemmas that hint at the slow rate at which α grows.

Lemma. *α_n_0_α_logn*
Assume: $16 \leq n$

$$\alpha\, n \leq 1 + (\alpha\,(\log n)) \tag{2.21}$$

According to Charguéraud and Pottier, this lemma is far-reaching, as it shows that $\alpha\, n$ and $\alpha\,(\log n)$ are asymptotically equivalent. It is in fact an exercise in CLRS to show this is the case. One could even substitute log by the iterated logarithm, or any reasonable slow-growing function, recall the fact that A grows faster than any primitive recursive function.

The following lemma is much more explicit, and was one of the initial motivations for the authors to start this project:

Lemma. *observable_universe_α*
Assume: $n \leq 10^{80}$

$$\alpha\, n \leq 4 \tag{2.22}$$

The figure 10^{80} is one of the current estimates of the number of atoms in the universe. If the estimate ever grows, even significantly, it would not change the bound, as, following CLRS, we were able to prove that $A\,4\,1 > 16^{512}$. This means that for all "practical" inputs, with a very liberal interpretation of the word, α is at most 4.

3 The Proof in Isabelle

The goal of this thesis was to prove the α-bound asymptotic time complexity of the *Union* and *Find* operations. This proof is famously non-trivial and has been improved over the years to the current standard version in CLRS, which however has lost any insight into why the inverse Ackermann function arises. This proof does not include most details, and a formalized proof would require a far too extensive level of creativity.

The paper by Alstrup et al. improves the bound slightly by limiting the argument of α to the size of the largest equivalence class, instead of the whole domain, and crucially provides some detailed proofs. They introduce some new concepts which allow for more context-sensitivity of the bounds, most importantly they link all bounds to the rank of the existing nodes (a lower bound on the size of the equivalence class) instead of to the size of the data structure, which was in earlier proofs fixed.

This is used by Charguéraud and Pottier to implement a dynamic, pointer based implementation of Union-Find in OCaml, which they verify using a similar framework implementing separation logic with time credits in Coq [GCP18]. The result they use for the Hoare-Triples is slightly weaker than the one by Alstrup et al. as the α bound refers to the current size of the whole domain, but they formalize the tighter bound as well, which we will also prove.

Our implementation is based on the work by Haslbeck and Lammich [HL19], which had already proved the correctness and worst-case logarithmic asymptotic complexity of an array-based implementation. In order to minimize the duplication of work, we started as close as possible to them. We decided also to follow the proof by Charguéraud and Pottier for the abstract analysis of the data structure. It was crucial that both proofs separate the abstract, mathematical view of the data structure as a relation, a graph or a list and its properties from the proof about the imperative program. This allowed us to mimic most of the Coq analysis in Isabelle, with some adaptations, even though the resulting implementation is vastly different, as are the frameworks for modeling the imperative semantics and the tools they provide. In principle, this means that many lemmas can be reused in future work by diverging implementations of Union-Find.

The proof in Coq is about 4KLoc, and our resulting proof is of similar length. This proof required many more fine-grained concepts about the data-structure, much theory surrounding them, and a different, stronger invariant (see the following section) than the one used in the worst-case analysis.

3.1 Porting from Coq

This section aims to give an anecdotal but hopefully practical view on the problems that arose while porting proofs from Coq to Isabelle/HOL. It should be noted as a starting remark that the authors are users only of Isabelle/HOL and have only a superficial understanding of the Coq system. A special thank you goes to Armaël Guéneau, who introduced the authors to Coq. This comparison deviates from others in the literature by providing a practical view on the current state of theorem proving in both systems rather than comparing the theoretical foundations [Yus18].

Superficially, the Gallina language used by Coq is similar to the *apply-style* scripts that are often considered bad style in the Isabelle/HOL community. It is however much richer than that, and for the writer of proofs there does not seem to be much of a difference in the expressiveness. In particular, and in contrast to *apply-style* scripts, it is possible to construct forward proofs by explicitly stating new subgoals which can later be referenced. The Isar language is of course specifically targeted to making proofs easier to read without the proof state at every point. However, in our opinion, the current proof state is in both systems necessary to follow a difficult proof in detail. The differences lay of course much more on the tactics available, and the quality of the proof library.

All in all Coq has many more specialized tactics [Zim] which allow for some fine grained manipulation of assumptions and goals. Isabelle's tactics seem to be more powerful, and the Isar language allows for complex proofs without editing the terms explicitly. However, some tactics could be useful to improve backwards reasoning in Isabelle/HOL. The clearest advantage of Isabelle/HOL over Coq is of course the Sledgehammer tool, which simplifies proof exploration and finalization enormously. Because of this aspect alone, together with the better IDE support and the greater interactivity of the proof process we would deem the current Isabelle/jEdit system the most user-friendly, both being mostly equivalent in their power to develop proofs.

3.1.1 Reasoning about Arithmetic

$\mathrm{Suc}\, x$ and $x + 1$

This is mostly a small complaint about some lemmas which are by default in the simpset in Isabelle/HOL. Many times, a forward style arithmetic proof using Isar's equational reasoning is unnecessarily tedious, as it requires to write every step explicitly. When the steps involve precise lemmas needed to move forward, it normally boils down to looking up the lemma, tailoring the next step to use it, and proving the step by applying the lemma.

In this case a simple backward proof applying the equations and rules available is easier to follow, and much easier to write. In most cases, however, providing the lemmas to higher level tactics such as auto or simp is not enough in every step because these tend to rewrite things like 1 to $\mathrm{Suc}\, 0$. Of course, a careful presentation of the lemmas helps, but is often not enough, and the proof ends up cluttered with very low level substitutions between the actual steps which make progress, which of course hinders readability. Some examples of this problem can be found in Ackermann.thy.

Coq does not seem to have any tendency to rewrite `n + 1` to `S n`, so the backwards proofs are more readable.

generalize dependent

Consider the following equation (easy for a human or a CAS, but difficult for the automatic tactics of a proof assistant):

$$2(2^i(1+x)-1)+1 = 2\cdot 2^i(1+x)-1$$

The factors 2^i and $(1+x)$ are probably being unfolded and mangled with by the tactics, when they are not key to the identity. So one could replace them by two new variables n and y, yielding:

$$2(ny-1)+1 = 2\cdot n\cdot y-1$$

which can be proved by a tactic, as the search space is much smaller (with an explicit and large i, such in the proofs about the observable universe bound of α, the tactics really do unfold too much and take seconds to finish).

In Coq, you can explicitly declare new variables to replace those expressions (do not pay much attention to the `intros`, that is a standard Coq idiom for a goal of the form $A \longrightarrow B$):

```
generalize dependent (2^i); intro n; intros.
generalize dependent (1+x); intro y; intros.
```

In Isabelle/HOL you would need to either prove the second equation first and then instantiate it or obtain some new variables relating to the old and rewriting all terms.

Generalized Rewriting

The default rewriting method in Isabelle/HOL is subst. In Coq you have `rewrite` and `replace term1 with term2`.

`rewrite` is superficially mostly equivalent to subst, except for a more pleasant syntax that reduces the need for the symmetric parameter. It is possible to use it on named assumptions instead of only on an indexed occurrence. When applied to an assumption, it also does not change the whole goal from a list of assumptions and a goal B to a goal of the form $A_1 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow B$.

Moreover, it also allows for rewriting inequalities, so if we have an assumption of the form $A_1 : B \leq C$ and we want to show $A \leq C$, the tactic `rewrite` A_1`.` will change the goal to $A \leq B$, which in Isabelle/HOL would require to apply a transitivity rule.

Generalized rewriting can be extended to many more forms of equations axiomatically [Zim], so it is a very powerful and extendable mechanism to deal with chains of equations.

`replace` enables to rewrite equations not yet proved, so it transforms `term1` to `term2`, and then generates a subgoal for this not yet proven equality. It allows to perform simple transformations which can be proved automatically without cluttering the proof with named lemmas, or the need to recall the names of existing low level lemmas. This is also useful in making backward proofs more readable.

3.1.2 Trivia

unpack

This tactic comes from the LibTactics library by Charguéraud [Cha], and destructs conjuntions and existentials in the assumptions. This is useful for example when an invariant is assumed, which is a conjunction of several properties, and when proving a specific subgoal only one of them is needed. It is also superior to the simple existential introduction rule, which only allows the existential quantifier at the outer most level.

3.2 Abstract Analysis

3.2.1 Important Definitions

The abstract analysis deals with the data structure as two lists, one with the tree structure $\mathcal{L}$ and one with the corresponding ranks $\mathcal{R}$. This view point is not enough, as we want to prove properties about the modeled disjoint set forest, equivalence class, and the transformations performed by *Union*, and by path compression.

Definition. *Disjoint set forest*

The parent of a node i is:

$$\mathcal{L}!i \tag{3.1}$$

The representative of a node is:

$$\text{rep\_of}_{\mathcal{L}}\, i := \begin{cases} i & \text{if } \mathcal{L}!i = i \\ \text{rep\_of}_{\mathcal{L}}\,(\mathcal{L}!i) & \text{otherwise} \end{cases} \tag{3.2}$$

Through the representative, we identify the equivalence classes of the disjoint set forest:

$$\text{ufa\_}\alpha_{\mathcal{L}} := \{(x,y) \mid x < |\mathcal{L}| \land y < |\mathcal{L}| \land \text{rep\_of}_{\mathcal{L}}\, x = \text{rep\_of}_{\mathcal{L}}\, y\} \tag{3.3}$$

And the height of a node:

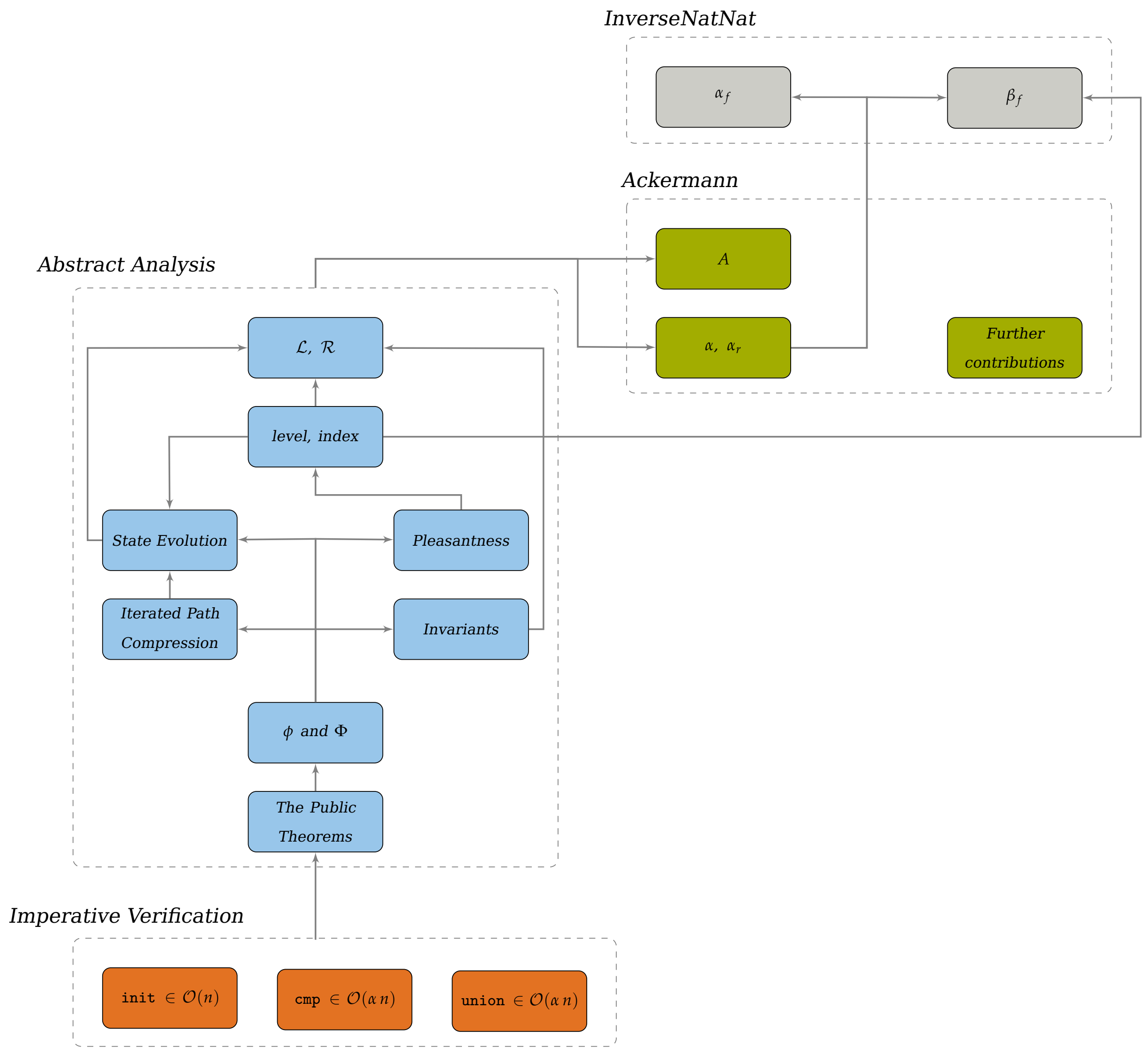

Figure 3.1: Overview of the logical dependencies between the theories presented.

An arc points to the theory or section containing a required definition or lemma.

$$\text{height\_of}_{\mathcal{L}}\, i := \begin{cases} 0 & \text{if } \mathcal{L}!i = i \\ 1 + \text{height\_of}_{\mathcal{L}}\, (\mathcal{L}!i) & \text{otherwise} \end{cases} \tag{3.4}$$

We also define the more precise child-parent relation:

$$\text{ufa\_}\beta\text{\_start}_{\mathcal{L}} := \{(x, y) \mid x < |\mathcal{L}| \wedge y < |\mathcal{L}| \wedge x \neq y \wedge \mathcal{L}!x = y\} \tag{3.5}$$

Its closures, the strict and non-strict paths in the graph:

$$\text{ufa\_}\beta^{+}_{\mathcal{L}} := (\text{ufa\_}\beta\text{\_start}_{\mathcal{L}})^{+} \quad \text{ufa\_}\beta^{*}_{\mathcal{L}} := (\text{ufa\_}\beta\text{\_start}_{\mathcal{L}})^{*} \tag{3.6}$$

Where R^{+} and R^{*} are respectively the transitive and reflexive transitive hull of a relation R.

And the descendants and ancestors of a node:

$$\text{descendants}_{\mathcal{L}}\, i := \{j \mid (j, i) \in \text{ufa\_}\beta^{*}_{\mathcal{L}}\} \tag{3.7}$$

$$\text{ancestors}_{\mathcal{L}}\, i := \{j \mid (i, j) \in \text{ufa\_}\beta^{*}_{\mathcal{L}}\} \tag{3.8}$$

✍ This relations are ultimately defined depending on $\mathcal{L}$, but any other functional relation (where a node has only one parent) satisfying the same classical properties of the hull could be used. All following lemmas would just need to replace any occurrence of $i < |\mathcal{L}|$ with $i \in \text{Dom}\, R$. In our case we did not do it, as of course $\text{Dom}\, \text{ufa\_}\beta^{*}_{\mathcal{L}} = \{0, \ldots, |\mathcal{L}| - 1\}$.

We only want to allow lists which represent a disjoint set forest, so $\text{ufa\_}\beta^{+}_{\mathcal{L}}$ cannot have any cycles. This is characterized by the first part of the invariant required:

Definition. *ufa_invar*

$$\text{ufa\_invar}\, \mathcal{L} := \forall i < |\mathcal{L}|.\, i \in \text{Dom}\, \text{rep\_of}_{\mathcal{L}} \wedge \mathcal{L}!i < |\mathcal{L}| \tag{3.9}$$

This is equivalent to not having any cycles, as every node in a cycle would not be in the domain of $\text{rep\_of}_{\mathcal{L}}$. On top of that, we do not allow parents outside the domain.

Finally, we define the full invariant enforced on the data structure (we will then show that union and path compression preserve the invariant):

Definition. *invar_rank*

$$\begin{aligned} \text{invar\_rank}\,\mathcal{L}\,\mathcal{R} :=&\text{ufa\_invar}\,\mathcal{L} \wedge & (3.10)\\ &|\mathcal{L}| = |\mathcal{R}| \wedge & (3.11)\\ &(\forall (i,j) \in \text{ufa\_}\beta\text{\_start}_{\mathcal{L}}.\, \mathcal{R}!i < \mathcal{R}!j) & (3.12)\\ &(\forall i < |\mathcal{L}|.\, \mathcal{L}!i = i \longrightarrow 2^{\mathcal{R}!i} \leq |\text{descendants}_{\mathcal{L}}\, i|) & (3.13) \end{aligned}$$

Which in words means:

- $\mathcal{L}$ models a disjoint set forest.
- The domain of the rank is the same as the domain of the child-parent relation.
- The rank of a parent is greater than the rank of any of its children.
- The rank of a root never exceeds the logarithm of the size of its descendants.

3.2.2 The Rank

We have mentioned before that the rank of a node is an upper bound on its height in the tree, this is specified by the lemma *rank_bounds_height*, which states that if there is a path from i to j of length k, then $k \leq \mathcal{R}!j$ (or, more precisely $\mathcal{R}!i + k \leq \mathcal{R}!j$). The longest path from a node is the one to the root, and the length of this path defines the height. As the height, the rank is bounded by $\log|\mathcal{L}|$.

Following [Als+14], we will for most purposes use a modified rank:

Definition. *rankr*

$$\mathcal{R}_r\, i := \mathcal{R}!i + r \tag{3.14}$$

Recall r from 2.19 is a fixed parameter.

The (modified) rank always grows along paths, and strictly along non-trivial paths, and by extension of course $\alpha_r\,(\mathcal{R}_r\, i)$ also grows along paths, as α_r is monotonic.

3.2.3 The Level and the Index

The potential function Φ refers to the "entropy" of the data structure, so a higher potential means more disorder. The potential grows when cheap operations are performed, and decreases with expensive operations. This models the credit method defined by Tarjan when he introduced amortized analysis [Tar85], a high potential means credits are saved for extra work. In our case, we will store Φ time credits in the heap, which will be enough, together with the advertised cost, to pay for each operation. This is the key to amortized analysis with time credits.

The potential of the Union-Find data structure was defined similarly in all proofs since [Tar75], but this form was introduced by Alstrup et al. [Als+14]. In order to define it, we need two subtle concepts, the index and the level (only defined for non-root nodes):

Definition. *level*

$$\mathrm{defk}\, i\, k := A\, k\, (\mathcal{R}_r\ i) \tag{3.15}$$

$$\mathrm{level}_{\mathcal{L},\mathcal{R}}\, i := 1 + (\beta_{defk}\, (\mathcal{R}_r\ (\mathcal{L}!i))) \tag{3.16}$$

which unfolded yields

$$\mathrm{level}_{\mathcal{L},\mathcal{R}}\, i = 1 + \max\{k \mid \mathcal{R}_r\, (\mathcal{L}!i) \geq A\, k\, (\mathcal{R}_r\, i)\} \tag{3.17}$$

The level is well-defined, as the following k always satisfies the inequality:

Lemma. *level_exists*

$$A\, 0\, (\mathcal{R}_r\, i) \leq \mathcal{R}_r\, (\mathcal{L}!i) \tag{3.18}$$

According to Charguéraud and Pottier, the level of a node is a measure of the distance of its rank to the rank of its parent. Where these ranks are closest we have $\mathcal{R}_r\, (\mathcal{L}!i) = 1 + (\mathcal{R}_r\, i)$, so the level is exactly one. When the ranks are furthest away, we get the following lemma:

Lemma. *level_lt_α_r*

$$\mathrm{level}_{\mathcal{L},\mathcal{R}}\, i < \alpha_r\, (\mathcal{R}_r\, (\mathcal{L}!i)) \tag{3.19}$$

The proof of this lemma is simple because we can make use of the lemmas from the InverseNatNat.thy theory, so we can easily transform the goals into a form for which the lemmas available in Ackermann.thy apply.

Definition. *index*

$$\mathrm{prei}\, i\, j := (A\, (\mathrm{level}_{\mathcal{L},\mathcal{R}}\, i - 1))^{(j)}\, (\mathcal{R}_r\, i) \tag{3.20}$$

$$\mathrm{index}_{\mathcal{L},\mathcal{R}}\, i := \beta_{prei}\, (\mathcal{R}_r\, (\mathcal{L}!i)) \tag{3.21}$$

which unfolded yields:

$$\text{index}_{\mathcal{L},\mathcal{R}}\, i = \max\{j \mid \mathcal{R}_r\,(\mathcal{L}!i) \geq (A\,(\text{level}_{\mathcal{L},\mathcal{R}}\, i - 1))^{(j)}\,(\mathcal{R}_r\, i)\} \tag{3.22}$$

The index is of course always well defined, as there is always a j satisfying the inequality:

Lemma. *index_exists*

$$(A(\text{level}_{\mathcal{L},\mathcal{R}}\, i - 1))^{(0)}(\mathcal{R}_r\, i) \leq \mathcal{R}_r\,(\mathcal{L}!i) \tag{3.23}$$

The index satisfies the following lower and upper bounds:

Lemma. *index_ge_1_le_rank*

$$1 \leq \text{index}_{\mathcal{L},\mathcal{R}}\, i \leq \mathcal{R}_r\, i \tag{3.24}$$

3.2.4 The Potential Function Φ

We define first the potential for a single node:

Definition. ϕ

$$\phi\,\mathcal{L}\,\mathcal{R}\, i := \begin{cases} \alpha_r\,(\mathcal{R}_r\, i) \cdot (1 + (\mathcal{R}_r\, i)) & \text{if}\, \mathcal{L}!i = i \\ (\alpha_r\,(\mathcal{R}_r\, i) - \text{level}_{\mathcal{L},\mathcal{R}}\, i) \cdot \mathcal{R}_r\, i - \text{index}_{\mathcal{L},\mathcal{R}}\, i + 1 & \text{if}\, \alpha_r\,(\mathcal{R}_r\, i) = \alpha_r\,(\mathcal{R}_r\,(\mathcal{L}!i)) \\ 0 & \text{otherwise} \end{cases}$$

There are according lemmas that guarantee that the subtractions will not result in a negative number. In fact, except in the last case, where ϕ is explicitly set to 0, we have $\phi\,\mathcal{L}\,\mathcal{R} \geq 1$.

To define the potential of the entire data structure, we sum over every node:

Definition. Φ

$$\Phi\,\mathcal{L}\,\mathcal{R} := \sum_{i=0}^{|\mathcal{L}|-1} \phi\,\mathcal{L}\,\mathcal{R}\, i \tag{3.25}$$

✍ *Recall again that* $\{0, \ldots, |\mathcal{L}| - 1\}$ *is in this case the domain of our equivalence relation. In an alternative implementation of the data structure, this would be a sum over its domain.*

3.2.5 State Evolution

On our way to analyzing the behavior of `uf_union` and `uf_cmp`, we first define the union of two disjoint set trees and a single step of path compression abstractly. These are the only operations that modify the state and thus need to be analyzed to ensure they do not break the invariant:

Definition. *Abstract Union*
First, we define the union of the equivalence classes of two nodes:

$$\text{ufa\_union}_{\mathcal{L}}\, x\, y := \mathcal{L}[\text{rep\_of}_{\mathcal{L}}\, x := \text{rep\_of}_{\mathcal{L}}\, y] \tag{3.26}$$

However, we only perform unions according to the rank heuristic, therefore we define the operations that, given a disjoint set forest list and a rank list, return the modified lists:

$$\text{union\_by\_rank}^{(\mathcal{L})}_{\mathcal{L},\mathcal{R}}\, x\, y := \begin{cases} \text{ufa\_union}_{\mathcal{L}}\, x\, y & \text{if } \mathcal{R}!x < \mathcal{R}!y \\ \text{ufa\_union}_{\mathcal{L}}\, y\, x & \text{otherwise} \end{cases} \tag{3.27}$$

$$\text{union\_by\_rank}^{(\mathcal{R})}_{\mathcal{L},\mathcal{R}}\, x\, y := \begin{cases} \mathcal{R}[x := 1 + \mathcal{R}!x] & \text{if } \mathcal{R}!x = \mathcal{R}!y \\ \mathcal{R} & \text{otherwise} \end{cases} \tag{3.28}$$

The operation $\mathcal{L}[x := y]$ is just the list update operation, replacing the element in $\mathcal{L}$ at position x by y.

We have yet to consider the iterated path compression, which not only links a node to the root, but also all the nodes on the path to the root. Before that, we prove that union of two trees and a single step of compression preserve the invariant. These two elementary operations are what we call in the Isabelle proof *State Evolution*. We modify the state of the data structure only through composition of these operations.

Lemma. *invar_rank_union*
Assume: invar_rank $\mathcal{L}\,\mathcal{R}$; $x, y < |\mathcal{L}|$; $x \neq y$; $x = \mathcal{L}!x$; $y = \mathcal{L}!y$

$$\text{invar\_rank}\,(\text{union\_by\_rank}^{(\mathcal{L})}_{\mathcal{L},\mathcal{R}}\, x\, y)\,(\text{union\_by\_rank}^{(\mathcal{R})}_{\mathcal{L},\mathcal{R}}\, x\, y) \tag{3.29}$$

Lemma. *invar_rank_compress*
Assume: invar_rank $\mathcal{L}\,\mathcal{R}$; $(x, y) \in \text{ufa\_}\beta\text{\_start}_{\mathcal{L}}$

$$\text{invar\_rank}\,(\mathcal{L}[x := \text{rep\_of}_{\mathcal{L}}\, y])\,\mathcal{R} \tag{3.30}$$

The change of the rank during state evolution is more or less trivial. It is monotone and only really increases for roots. The analysis of the level and the index is more subtle, and we come to the following conclusions (we refer to the Isabelle proof for the detailed statements):

- **Lemma.** *levelx_levely_compress* (3.31):
  During compression on x, either the rank or the index increase.
- **Lemma.** *level_v_grows* (3.32):
  During any state evolution step, as the rank of a non-root node x is constant while the rank of its parent may grow, the level of x can only grow.
- **Lemma.** *index_v_grows_if_level_v_constant* (3.33):
  After a state evolution step, if the level remains constant, the index can only grow.

3.2.6 Iterated Path Compression

We define two equivalent inductive predicates which encode the operation of compressing a whole path, up to the root:

Definition. *Forward and backward iterated path compression*

$$\begin{gathered}\textit{BWIPCBase:}\quad \frac{x = \mathcal{L}!x}{\text{bw\_ipc}\,\mathcal{L}\,x\,0\,\mathcal{L}} \\[2ex] \textit{BWIPCStep:}\quad \frac{(x,y) \in \text{ufa\_}\beta\text{\_start}_{\mathcal{L}} \qquad \text{bw\_ipc}\,\mathcal{L}\,y\,i\,\mathcal{L}'}{\text{bw\_ipc}\,\mathcal{L}\,x\,(i+1)\,\mathcal{L}'[x := \text{rep\_of}_{\mathcal{L}}\,x]}\end{gathered} \tag{3.34}$$

$$\begin{gathered}\textit{FWIPCBase:}\quad \frac{x = \mathcal{L}!x}{\text{fw\_ipc}\,\mathcal{L}\,x\,0\,\mathcal{L}} \\[2ex] \textit{FWIPCStep:}\quad \frac{(x,y) \in \text{ufa\_}\beta\text{\_start}_{\mathcal{L}} \qquad \text{fw\_ipc}\,\mathcal{L}\,y\,i\,\mathcal{L}'[x := \text{rep\_of}_{\mathcal{L}}\,y]}{\text{fw\_ipc}\,\mathcal{L}\,x\,(i+1)\,\mathcal{L}'}\end{gathered} \tag{3.35}$$

ipc stands in both cases for iterated path compression and, in words, $\text{ipc}\,\mathcal{L}\,x\,i\,\mathcal{L}'$ *means that in the initial state* $\mathcal{L}$*, performing path compression along the path starting at* x *leads in* i *steps to the final state* $\mathcal{L}'$*.*

The forward variant corresponds intuitively to a two-pass algorithm, similar to our implementation, in which a first pass finds the representative of x, and the second pass performs the compression. This is clearly the composition of several compression steps formulated in the previous section. This formulation therefore makes the proof of many lemmas simpler.

The backwards variant is more alike a one-pass, recursive algorithm for path compression, similar to the one by Charguéraud and Pottier, where path compression is performed while unwinding the stack created by recursively finding the representative. This formulation is not clearly the composition of several compression steps as it is not trivial that the representative of x before compression remains the same after compression.

In a non-formal proof, one could say it is obvious that paths and representatives on trees not affected by compression are not changed, or that the final state $\mathcal{L}'$ is always defined and unique, and indeed that the two predicates are equivalent. All these statements were inductively proved in this formal context.

3.2.7 Pleasantness

We define the notion of pleasantness, which is the property of a node of having a strict non-root ancestor with identical level, and we derive an upper bound on the amount of unpleasant nodes. This part of the analysis was particularly challenging, as it required a more comprehensive theory about preservation of invariants and paths than the previous lemmas, among others the preservation of ufa_invar after arbitrary compressions that do not generate a cycle, as well as a very careful use of contexts and generalizations.

Definition. *top_part*
We define a node to be in the top part of its tree if α_r of its rank is the same as α_r of the rank of the root (its representative):

$$\text{top\_part}_{\mathcal{L},\mathcal{R}}\, x := (\alpha_r\, (\mathcal{R}_r\, x)) = (\alpha_r\, (\mathcal{R}_r\, \text{rep\_of}_{\mathcal{L}}\, x)) \tag{3.36}$$

the plesantness notion:

$$\begin{aligned}\text{pleasant}_{\mathcal{L},\mathcal{R}}\, x := &\ \text{top\_part}_{\mathcal{L},\mathcal{R}}\, x \land \\ &(\exists y.\, y \neq \mathcal{L}!y \land ((\mathcal{L}!x), y) \in \text{ufa\_}\beta^*_{\mathcal{L}} \land \\ &\text{level}_{\mathcal{L},\mathcal{R}}\, x = \text{level}_{\mathcal{L},\mathcal{R}}\, y)\end{aligned} \tag{3.37}$$

*this is a sound definition of the top-part, as it is preserved when going up the graph (**Lemma.** top_part_hereditary* (3.38)*).*

We define the displeasure of a node as the number of unpleasant ancestors:

$$\text{displeasure}_{\mathcal{L},\mathcal{R}}\, x \;:=\; |(\text{ancestors}_{\mathcal{L}}\, x) \cap \{y \mid \neg\text{pleasant}_{\mathcal{L},\mathcal{R}}\, y\}| \tag{3.39}$$

The displeasure of a parent grows by exactly one if the child was unpleasant, and stays constant if the child was pleasant. (**Lemma.** *displeasure_parent_if_unpleasant* (3.40) and **Lemma.** *displeasure_parent_if_pleasant* (3.41)).

Finally, we show that the displeasure of a node is bounded by the number of distinct levels of its non-root ancestors (this number is called in the proof the *levels* of a node), and in turn this is bounded by α_r of the rank of its representative. To prove this, we need to distinguish between pleasant and unpleasant nodes, establish a bound that relates to their *levels*, and at the end relate *levels* to α_r. There are also several lemmas about the conservation of displeasure after compression.

At the end, we arrive at the main result in Alstrup et al.'s paper, from which we will derive the public theorems:

Lemma. *bounded_displeasure_alstrup*
Assume: $\text{top\_part}_{\mathcal{L},\mathcal{R}}\, x;\ x < |\mathcal{L}|$

$$\text{displeasure}_{\mathcal{L},\mathcal{R}}\, x \le \alpha_r\,(\mathcal{R}_r\,(\text{rep\_of}_{\mathcal{L}}\, x)) \tag{3.42}$$

This result ultimately follows from 3.19, and the monotonicity of the level along paths.

3.2.8 The Public Theorems

We called this theorems public, as they are in principle the only ones needed by a user of the abstract data structure to prove the correctness and the asymptotic properties of an implementation. If we regard the whole proof as a black box, these are the only really interesting results.

Theorems for the Amortized Analysis

The main theorem relates the steps required to perform an iterated path compression, the potential before and after the compression, and α_r of some measure in the data structure. These theorems implicitly assume the invariant for the lists. We will start with the strong result by Alstrup et al.:

Theorem. *amortized_cost_of_iterated_path_compression_local*
Assume: $x < |\mathcal{L}|$

$$\begin{aligned}&\exists i\, \mathcal{L}'.\, \text{bw\_ipc}\, \mathcal{L}\, x\, i\, \mathcal{L}' \wedge \\ &\qquad \Phi\, \mathcal{L}'\, \mathcal{R} + i < \Phi\, \mathcal{L}\, \mathcal{R} + 2 \cdot \alpha_r\, (\mathcal{R}_r\, (\text{rep\_of}_{\mathcal{L}}\, x))\end{aligned} \tag{3.43}$$

It is called local because the bound refers to the rank of the representative, so it is just bounded by the size of the equivalence class instead of by the size of the whole data structure.

✍ *We could use this theorem to prove a tighter bound in the Hoare-Triple, which also does not depend on the size of the domain.*

We know that the rank is bound by the logarithm of the size of its equivalence class, and in turn this is roughly bound by the size of the domain:

Theorem. *amortized_cost_of_iterated_path_compression_global*
Assume: $x < |\mathcal{L}|$

$$\begin{aligned}&\exists i\, \mathcal{L}'.\, \text{bw\_ipc}\, \mathcal{L}\, x\, i\, \mathcal{L}' \wedge \\ &\qquad \Phi\, \mathcal{L}'\, \mathcal{R} + i < \Phi\, \mathcal{L}\, \mathcal{R} + 2 \cdot \alpha_r\, (|\mathcal{L}| + (r - 1))\end{aligned} \tag{3.44}$$

This theorem will be used to prove the Hoare-Triple for `uf_rep_of_c`, which is our implementation of the iterated path compression, and the other bounds just follow as a composition of this result and constant size operations.

Much less spectacularly, we must also prove that during the union of two trees the potential only changes by a constant. We can safely assume that the union will occur on roots, as in practice this is the only case:

Theorem. *potential_increase_during_link*
Assume: $x \neq y;\ x < |\mathcal{L}|;\ y < |\mathcal{L}|;\ x = \mathcal{L}!x;\ y = \mathcal{L}!y$

$$\Phi\, (\text{union\_by\_rank}^{(\mathcal{L})}_{\mathcal{L},\mathcal{R}}\, x\, y)\, (\text{union\_by\_rank}^{(\mathcal{R})}_{\mathcal{L},\mathcal{R}}\, x\, y) \leq \Phi\, \mathcal{L}\, \mathcal{R} + 2 \tag{3.45}$$

Theorems for the Functional Correctness

The Hoare-Triples of course also provide the correctness of the implementation, so the following lemmas also need to be provided (here presented only informally):

- **Lemma.** *invar_rank_evolution* (3.46): A State Evolution step preserves the full invariant.
- **Lemma.** *ufa_union_correct* (3.47): The *Union* operation, defined as the merging of the two trees, merges the equivalence classes. This lemma and its proof come from the work by Lammich.
- **Lemma.** *bw_ipc_root_unique* (3.48): Performing backward iterated path compression on a root can only yield the same list after 0 steps. This is used as an induction basis.
- **Lemma.** *ufa_compress_aux* (3.49): A disjoint set forest remains a disjoint set forest after compression to a root (a generalized version was needed for the abstract analysis which enables arbitrary compressions within a tree).
- **Lemma.** *rep_of_invar_along_path* (3.50): The representative of all nodes sharing a path is the same.

3.3 Separation Logic with Time Credits

The concept that time credits can be “stored” in a data structure to be used later in time as a way to simplify amortized complexity analysis was introduced by Tarjan [Tar85], and later formalized by Atkey [Atk10] within separation logic [Rey02]. Separation logic is used here, very roughly, as a way to reason about mutable resources in a heap. This is a very natural way of formalizing and proving Hoare-Triples for modular programs, as every separated part of the heap can be dealt with separately.

We will not go into detail about the exact definition of the low-level concepts of separation logic, but for the sake of explaining the notation used in the Hoare-Triples, we will provide an informal explanation of the most important components:

- $\uparrow(P)$ holds if the heap is empty and P holds as a predicate.
- true and false hold respectively for every heap and for no heap.
- $p \mapsto_a xs$ is the “points-to” assertion. The memory cell at location p exists, is “owned by us” and contains an array representing the list xs.
- $P_1 * P_2$ is the separating conjunction. It holds if the heap can be split into two disjoint parts which respectively satisfy P_1 and P_2.

- $\exists_A x. P$ is just existential quantification lifted to assertions.

In our analysis, we use the framework by Zhan and Haslbeck [ZH18], which implements Separation Logic with Time Credits as an extension to the already existing separation logic for Imperative/HOL [Bul+08]. The idea of this new semantics is that for any execution to succeed, there has to be as many time credits available as "atomic steps" of computation performed.

Definition. *Hoare-Triple*

$$\langle P \rangle \quad \mathsf{c} \quad \langle Q \rangle \tag{3.51}$$

states that for every heap h satisfying P the following holds: the execution of c *is successful with a new heap h' and a return value r after t time steps, P contains $n \geq t$ time credits, Q contains $n - t$ time credits, and the new heap h' satisfies $Q\, r$.*

In our analysis we will always use the short notation $\langle P \rangle \quad \mathsf{c} \quad \langle Q \rangle_t$ which stands for $\langle P \rangle \quad \mathsf{c} \quad \langle Q * \text{true} \rangle$. As the true assertion holds for every heap, we can in this way ignore some of the remaining time credits, as $\$(a + b) = \$a * \$b$, and true holds for $\$b$. This means that the assertion Q does not need to store all of the $n - t$ remaining credits and we may "throw away" the rest. If an implementation change required some additional amount of time credits to execute, and there were enough spare credits being thrown away, there would be no need to change the analysis or the Hoare-Triples. This is sometimes referred to in the literature about Separation Logic with Time Credits as "garbage collection", as the part of the heap not longer owned and thrown away can model an idealized garbage collection [CP17].

3.4 The Hoare-Triples

In this section we will present the semantic public interface of the data structure, which guarantees the amortized time bound. Following the reasoning above, the idea of an amortized analysis through separation logic with time credits in practice is the following:

- Define an assertion which abstracts the data structure and "stores", together with the data itself, Φ time credits.
- Prove the following Hoare-Triples: assuming you have as a precondition the assertion defined above together with an advertised cost of $f\, n$ time credits, you can execute the operation and at the end you have the corresponding assertion of the modified structure.

At the end of a sequence of k operations, only the advertised cost has been paid every time, so the sequence has costed $k \cdot (f\ n)$ time credits. This means by definition that the operation has an amortized time complexity of f. How many credits were used in every operation, and therefore stored according to the following assertion is from this perspective unknown, but is irrelevant to the statement about the amortized complexity. We define the assertion abstracting our union-find data structure as follows:

Definition. *is_uf*

$$\begin{aligned}\text{is\_uf}\,\mathcal{X}\,(s, p) :=&\exists_A \mathcal{L}\,\mathcal{R}.\ p \mapsto_a \mathcal{L} * s \mapsto_a \mathcal{R} * \\ &\uparrow(\text{ufa\_}\alpha_{\mathcal{L}} = \mathcal{X} \wedge \text{invar\_rank}\,\mathcal{L}\,\mathcal{R}) * \\ &\$(4 \cdot \Phi\,\mathcal{L}\,\mathcal{R})\end{aligned} \tag{3.52}$$

The existential quantification is necessary so that is_uf *only depends on the arrays and not on the lists modeled by them. This is only a minor inconvenience in the proofs, which of course require the lists to apply the abstract lemmas.*

In words, this assertion states the following:

- There exist lists $\mathcal{L}$ and $\mathcal{R}$ which model the contents of the arrays p and s respectively.
- $\mathcal{L}$ and $\mathcal{R}$ satisfy the invariant, in particular, $\mathcal{L}$ is a disjoint set forest and $\mathcal{R}$ has been well formed according to our specification.
- The equivalence relation modeled by the disjoint set forest is exactly $\mathcal{X}$.
- We have $4 \cdot \Phi$ time credits stored, which can be used additionally to the advertised cost of an operation.

As a small disclaimer, the concrete advertised cost functions are only asymptotically optimal. No effort has been put into tightening linear factors, as this is in any case implementation dependent, and in our opinion it does not add anything to the significance of the result, making it however more difficult to make small changes to the implementation.

3.4.1 uf_init

Definition. *uf_init_time*

$$\text{uf\_init\_time}\,n := 16n + 12 \tag{3.53}$$

Lemma.

$$\text{uf\_init\_time} \in \mathcal{O}(n) \tag{3.54}$$

Theorem. *uf_init_rule*

$$\langle \$(\text{uf\_init\_time}\, n) \rangle \quad \texttt{uf\_init}\, n \quad \langle \text{is\_uf}\, \{(i,i) \mid i < n\} \rangle_t \tag{3.55}$$

3.4.2 Hoare-Triples for the Private Functions

`uf_compress` and `uf_rep_of` will ultimately only be used as a subroutine of `uf_rep_of_c`, in turn a subroutine of `uf_cmp` and `uf_union`, so these do not require the more abstract version of the Hoare-Triple, with the existentially quantified lists. Therefore the Hoare-Triples are a bit lengthier, but are really conceptually simpler.

Lemma. *uf_rep_of_rule*
Assume: ufa_invar l; $i < |\mathcal{L}|$

$$\begin{aligned} &\langle p \mapsto_a \mathcal{L} * \$(\text{height\_of}_{\mathcal{L}}\, i + 2) \rangle \quad \texttt{uf\_rep\_of}\, p\, i \\ &\qquad \langle \lambda r.\, p \mapsto_a \mathcal{L} * \uparrow (r = \text{rep\_of}_{\mathcal{L}}\, i) \rangle_t \end{aligned} \tag{3.56}$$

Lemma. *uf_compress_rule*
Assume: invar_rank $\mathcal{L}\,\mathcal{R}$; $i < |\mathcal{L}|$; $c = \text{rep\_of}_{\mathcal{L}}\, i$; bw_ipc $\mathcal{L}\, i\, d\, \mathcal{L}'$

$$\begin{aligned} &\langle p \mapsto_a \mathcal{L} * \$(1 + d + 3) \rangle \quad \texttt{uf\_compress}\, i\, c \\ &\qquad \langle \lambda \_.\, p \mapsto_a \mathcal{L}' * \uparrow (\text{invar\_rank}\, \mathcal{L}'\, \mathcal{R} \wedge |\mathcal{L}'| = |\mathcal{L}| \wedge \\ &\qquad (\forall i < |\mathcal{L}|.\, \text{rep\_of}_{\mathcal{L}}\, i = \text{rep\_of}_{\mathcal{L}'}\, i)) \rangle_t \end{aligned} \tag{3.57}$$

This lemma links the abstract concept of iterated path compression with its imperative implementation. The proof is by induction on the predicate bw_ipc *and requires some bookkeeping, but really just follows the code.*

Compressing when Looking for a Representative

The function `uf_rep_of_c` performs a representative search and iterated path compression on the path to this representative. The function is therefore the key to the adequacy of Φ as the potential function. It is here that the important theorem *amortized_cost_of_iterated_path_compression_global* is used, so we can advertise a cost of α_r. The fact that compression is performed is of course crucial to this efficiency bound.

Definition. *uf_rep_of_c_time*

$$\text{uf\_rep\_of\_c\_time}\, n := 8 \cdot \alpha_r\, (n + (r - 1)) + 16 \tag{3.58}$$

Lemma. *uf_rep_of_c_rule*
Assume: invar_rank $\mathcal{L}\,\mathcal{R}$; $i < |\mathcal{L}|$

$$\begin{aligned}&\langle p \mapsto_a \mathcal{L} * \$(4 \cdot \Phi\, \mathcal{L}\, \mathcal{R}\ + \text{uf\_rep\_of\_c\_time}\, |\mathcal{L}|)\rangle \ \ \texttt{uf\_rep\_of\_c}\, p\, i\\ &\qquad\langle \lambda\, r.\, \exists_A \mathcal{L}'.\, p \mapsto_a \mathcal{L}' *\\ &\qquad\uparrow (r = \text{rep\_of}_{\mathcal{L}}\, i \wedge \text{invar\_rank}\, \mathcal{L}'\, \mathcal{R} \wedge |\mathcal{L}| = |\mathcal{L}'| \wedge\\ &\qquad(\forall i < |\mathcal{L}|.\, \text{rep\_of}_{\mathcal{L}}\, i = \text{rep\_of}_{\mathcal{L}'}\, i)) * \$(4 \cdot \Phi\, \mathcal{L}'\, \mathcal{R})\rangle_t\end{aligned} \tag{3.59}$$

3.4.3 uf_cmp

The Hoare-Triple present here is in the abstract form. The reader is encouraged to look into the Isabelle code to see how this is refined from statements similar to the ones in the previous subsection.

Definition. *uf_cmp_time*

$$\text{uf\_cmp\_time}\, n := 2 \cdot \text{uf\_rep\_of\_c\_time}\, n + 10 \tag{3.60}$$

Lemma.

$$\text{uf\_cmp\_time}\ \in \mathcal{O}(\alpha_r\, n) \tag{3.61}$$

Theorem. *uf_cmp_rule*

$$\begin{aligned}&\langle \text{is\_uf}\, \mathcal{X}\, u\ * \$(\text{uf\_cmp\_time}\, |\text{Dom}\, \mathcal{X}|)\rangle \ \ \texttt{uf\_cmp}\, u\, i\, j\\ &\qquad\qquad\qquad\qquad\langle \text{is\_uf}\, \mathcal{X}\, u * \uparrow (r \leftrightarrow (i, j) \in \mathcal{X})\rangle\end{aligned} \tag{3.62}$$

u is here the tuple of arrays representing the data structure. It is not explicitly unfolded to highlight that the data structure should be viewed only in its abstract form here.

3.4.4 uf_union

Definition. *uf_union_time*

$$\text{uf\_union\_time}\, n := 2 \cdot \text{uf\_rep\_of\_c\_time}\, n + 20 \tag{3.63}$$

Lemma.

$$\text{uf\_union\_time} \in \mathcal{O}(\alpha_r\, n) \tag{3.64}$$

Theorem. *uf_union_rule*
Assume: $i, j \in \operatorname{Dom} \mathcal{R}$

$$\langle \text{is\_uf}\, \mathcal{X}\, u \ast \$(\text{uf\_union\_time}\, |\operatorname{Dom} \mathcal{R}|) \rangle \;\; \texttt{uf\_union}\, u\, i\, j \;\; \langle \text{is\_uf}\, (\text{per\_union}\, \mathcal{X}\, i\, j) \rangle \tag{3.65}$$

per_union *is just the function that merges the two equivalence classes belonging to i and j, exactly what* `uf_union` *is supposed to do.*

The proof of this theorem is 400 LoC long, but it is very repetitive and many of the small steps could be replaced with more tailored arguments to the sep_auto tactic. It would still not be trivial, but all lemmas required to instantiate the correct lists and prove the invariant after the transformation are present, so it is just a matter of optimizing the proof both in length and readability.

4 Conclusions

In this thesis we have proved the classical result about the α-bound amortized complexity of an imperative Union-Find data structure. We have further formalized all the mathematical analysis required to prove the state of the art bound by Alstrup et al. for a differing implementation which allows for a growth of the domain of the equivalence relation. This is, to our knowledge, together with the Coq formalization by Charguéraud and Pottier, the only result of this kind.

On top of that, we provide a more comprehensive theory about the Ackermann function and the first theory about the inverse Ackermann function in Isabelle/HOL, as well as a theory about inverses of natural number functions, which can be used in different contexts. Our formalization of the lemma 2.22 is, to our knowledge, unprecedented, and thus the first formalization of this intuitively impressive result about the actual slow rate at which α grows.

The proofs of the Hoare-Triples are still a bit too long. This is partly due to lack of optimization of the proofs, because of our limited time, but also to the still lacking support for linear arithmetic by the automation in the Separation Logic with Time Credits framework, and its too aggressive instantiation of existentially quantified variables. This is nonetheless an improvement to the situation in Coq, where, to our knowledge, no such automation is provided. The sep_auto method is, for instance, very capable of automatically proving statements about pure separation logic.

In our opinion, this result further demonstrates the feasibility of formally verifying the functional correctness and running time bounds for non-trivial algorithm implementations. As it is always the case with formal proofs, it is sometimes necessary to invest time proving intuitively trivial facts, but sometimes this process highlights some necessary hidden assumptions, which are too often ignored in non-formal contexts.

Listings

List of Figures

Bibliography

[Ack] W. Ackermann. “Zum Hilbertschen Aufbau der reellen Zahlen.” In: *Math. Ann.* 99 (), pp. 118–133.

[Als+14] S. Alstrup, M. Thorup, I. L. Gørtz, T. Rauhe, and U. Zwick. “Union-Find with Constant Time Deletions.” In: *ACM Trans. Algorithms* 11.1 (Aug. 2014), 6:1–6:28. ISSN: 1549-6325. DOI: `10.1145/2636922`.

[Atk10] R. Atkey. “Amortised Resource Analysis with Separation Logic.” In: *Programming Languages and Systems*. Ed. by A. D. Gordon. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 85–103. ISBN: 978-3-642-11957-6.

[Bla04] P. E. Black. *"inverse Ackermann function"*. in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed. Available from: `https://www.nist.gov/dads/HTML/inverseAckermann.html` (accessed 26 August 2019). 17 December 2004.

[Bul+08] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. “Imperative Functional Programming with Isabelle/HOL.” In: *Theorem Proving in Higher Order Logics*. Ed. by O. A. Mohamed, C. Muñoz, and S. Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 134–149. ISBN: 978-3-540-71067-7.

[Cha] A. Charguéraud. *Module LibTactics*. `http://gallium.inria.fr/~fpottier/ssphs/LibTactics.html`. Accessed: 2019-09-03.

[Cor+09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[CP17] A. Charguéraud and F. Pottier. “Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits.” In: *Journal of Automated Reasoning* (Sept. 2017). DOI: `10.1007/s10817-017-9431-7`.

[GCP18] A. Guéneau, A. Charguéraud, and F. Pottier. “A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification.” In: Apr. 2018, pp. 533–560. ISBN: 978-3-319-89883-4. DOI: `10.1007/978-3-319-89884-1_19`.

[HL19] M. P. L. Haslbeck and P. Lammich. “Refinement with Time – Refining the Run-time of Algorithms in Isabelle/HOL.” English. In: *ITP2019: Interactive Theorem Proving*. June 2019.

[HLB19] M. P. L. Haslbeck, P. Lammich, and J. Biendarra. “Kruskal’s Algorithm for Minimum Spanning Forest.” In: *Archive of Formal Proofs* (Feb. 2019). `http://isa-afp.org/entries/Kruskal.html`, Formal proof development. ISSN: 2150-914x.

[Kni89] K. Knight. “Unification: A Multidisciplinary Survey.” In: *ACM Comput. Surv.* 21.1 (Mar. 1989), pp. 93–124. ISSN: 0360-0300. DOI: `10.1145/62029.62030`.

[Lam19] P. Lammich. “Refinement to Imperative HOL.” In: *J. Autom. Reason.* 62.4 (Apr. 2019), pp. 481–503. ISSN: 0168-7433. DOI: `10.1007/s10817-017-9437-1`.

[Löw19] A. Löwenberg Casas. *Proof of the amortized time complexity of the Union Find data structure in Isabelle HOL*. `https://github.com/adrilow/Proof-of-the-amortized-time-complexity-of-the-Union-Find-data-structure-in-Isabelle-HOL`. 2019.

[Men09] E. Mendelson. *Introduction to Mathematical Logic*. 5th. Chapman & Hall/CRC, 2009. ISBN: 1584888768, 9781584888765.

[Niv09] G. Nivasch. “Weak Epsilon-Nets, Davenport–SchinzelSequences, and Related Problems.” PhD thesis. Tel-Aviv University, June 2009.

[Rey02] J. C. Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures.” In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. LICS ’02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 55–74. ISBN: 0-7695-1483-9.

[Tar75] R. E. Tarjan. “Efficiency of a Good But Not Linear Set Union Algorithm.” In: *J. ACM* 22.2 (Apr. 1975), pp. 215–225. ISSN: 0004-5411. DOI: `10.1145/321879.321884`.

[Tar85] R. Tarjan. “Amortized computational complexity.” In: *SIAM Journal on Algebraic and Discrete Methods* 6.2 (1985), pp. 306–318.

[TL84] R. E. Tarjan and J. van Leeuwen. “Worst-case Analysis of Set Union Algorithms.” In: *J. ACM* 31.2 (Mar. 1984), pp. 245–281. ISSN: 0004-5411. DOI: `10.1145/62.2160`.

[Yus18] A. Yushkovskiy. “Comparison of Two Theorem Provers: Isabelle/HOL and Coq.” In: *arXiv e-prints*, arXiv:1808.09701 (Aug. 2018), arXiv:1808.09701. arXiv: `1808.09701 [cs.LO]`.

[ZH18] B. Zhan and M. P. L. Haslbeck. “Verifying Asymptotic Time Complexity of Imperative Programs in Isabelle.” In: *CoRR* abs/1802.01336 (2018). arXiv: `1802.01336`.

[Zim] T. e. a. Zimmermann. *The Coq Reference Manual*. `https://coq.inria.fr/refman/coq-tacindex.html`. Accessed: 2019-09-03.