Provably Correct Key-Value Maps in Agda

Leveraging Membership to Reach Correctness

Master's thesis in Computer science and engineering

Carl Bergman
Johan Henrik Ek

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Master's thesis 2024

Provably Correct Key-Value Maps in Agda

Leveraging Membership to Reach Correctness

Carl Bergman
Johan Henrik Ek

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Provably Correct Key-Value Maps in Agda
Leveraging Membership to Reach Correctness
Carl Bergman Johan Henrik Ek

© Carl Bergman & Johan Henrik Ek, 2024.

Supervisor: Andreas Abel, Department of Computer Science and Engineering
Examiner: Alejandro Russo, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Provably Correct Key-Value Maps in Agda
Leveraging Membership to Reach Correctness
Carl Bergman
Johan Henrik Ek
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis describes the interface and implementation of a provably correct finite map in the Agda programming language. Finite maps are fundamental data structures in computer science, and existing finite maps in Agda can be improved in terms of correctness and some implementation details. We present an extension of 'A Theory of Finite Maps' (Collins & Syme, 1995) and an Agda formalisation of that extended theory. The formalisation is designed to be structure agnostic, allowing for many underlying implementations, and its flexibility is demonstrated through a proof-of-concept simply typed lambda calculus interpreter.

The core of the formalisation is implemented and proven for AVL trees, extending the Agda standard library with additional proofs and operations. Among these are laws defining the behaviour of `delete` and `unionWith` as well as an alternative `unionWith` implementation. The implementation is also differentiated from the standard library by using erasure to remove, among other things, ordering proofs from the tree data type and operations. The performance of our implementation is compared to the standard library and the results show no meaningful performance gains for our implementation.

Finally, we discuss the inclusion of correct-by-construction membership properties in the tree structure itself. Such properties have not been introduced in our implementation. This is due to the constant need to update each proof on every operation on the collection, since any operation that modifies the structure of the tree may cause a proof to become irrelevant for the new map.

Keywords: Agda, AVL Trees, Collections, Finite maps

Acknowledgements

We would like to thank our supervisor, Andreas Abel, for his continued guidance and support throughout this project. Additionally, we would like to thank Conor McBride for his seminal paper ‘How to keep your neighbours in order’ that led Andreas to propose the subject of this thesis. Finally, we wish to thank our families and friends for their endless support.

Carl Bergman and Johan Henrik Ek, Gothenburg, 2024-06-15

Contents

List of Figures **xi**

List of Listings **xiii**

1 Introduction **1**
- 1.1 Problem and Goal 1
- 1.2 Limitations 2
- 1.3 Ethical and Ecological Implications 2
- 1.4 Code Availability 2
- 1.5 Related Work 3
  - 1.5.1 Collections 3
  - 1.5.2 Representations of Collections 3
  - 1.5.3 Verified Collections 5

2 Theory **7**
- 2.1 Agda 7
- 2.2 A Theory of Finite Maps 9
- 2.3 Correct-by-Construction 11

3 An Extension of A Theory of Finite Maps **13**
- 3.1 Modification of Collins and Symes' Axioms 15
- 3.2 Equivalence Between Finite Maps 15
- 3.3 Additional Functions and their Laws 16

4 Implementation **19**
- 4.1 Abstraction 19
- 4.2 AVL Tree Implementation 19
  - 4.2.1 Any 21
  - 4.2.2 All 23
- 4.3 Interface Implementation 23
- 4.4 Union 24

5 Result **29**
- 5.1 Proofs 29
- 5.2 Proof of Concept: Lambda Calculus Interpreter 29
- 5.3 Performance Comparison 31

5.4 Memory Profiling . . . 32

6 Discussion **35**
6.1 Correct-by-Construction Key-Value Properties . . . 35
6.2 Union and Join . . . 36
6.2.1 The Benefits of Join . . . 37
6.2.2 Union in Size-Indexed Trees . . . 37
6.3 Inductive Principle . . . 39
6.3.1 Equality and Induction . . . 40
6.4 Size and Height Parameterised AVL-Trees . . . 41
6.5 Performance Evaluation . . . 41
6.5.1 Effects of Erasure on Performance . . . 42
6.5.2 Effects of Erasure on Memory . . . 42
6.5.3 On Correctness and Performance . . . 43

7 Conclusion **45**
7.1 Future Work . . . 46

Bibliography **47**

A Definitions of Abstractions, an Ordering Function, and Some Function Implementations **I**
A.1 Definition of the BMap Record . . . I
A.2 Definition of the DMap Record . . . III
A.3 Definition of the MMap record . . . III
A.4 The Implementation of the mklim Function . . . IV
A.5 Implementation of insertWith, insert, delete, and lookup . . . IV

B Simply Typed Lambda Calculus Interpreter **VII**
B.1 Definitions and Typing Rules . . . VII
B.2 Agda Implementation . . . VIII

C Evaluation Figures **XI**

D Memory Profiling Results **XIII**

List of Figures

2.1 The axioms of Collins and Syme (1995). 10

3.1 Laws defining the behaviour of `lookup`, $_ \mapsto _ \in _$, $_ \in _$, as well as the relation between them. 14
3.2 Insert related laws. 14
3.3 The strong and weak induction principles for finite maps, respectively. 14
3.4 Additional insertion and lookup laws introduced by us. 15
3.5 Laws related to `delete`. 16
3.6 Laws related to `unionWith`. 17

5.1 Elapsed time for inserting values in the range $0..n$ into an empty map, with logarithmic x and y axes. 31
5.2 Times for merging two overlapping maps, n and m, using union. One of the maps contains 100,000 elements while the other one contains 1000, 10,000, or 100,000 elements. 32
5.3 Maximum residency for inserting values in the range $0..n$ into an empty map, with logarithmic x and y axes. 33
5.4 Maximum residency for for union where n is fixed with 100 000 elements. 34
5.5 Maximum residency for for union where m is fixed with 100 000 elements. 34

B.1 Types, Expressions, and Values for the simply typed lambda calculus. VII
B.2 Typing rules for the simply typed lambda calculus. VII

C.1 Elapsed time for inserting values in the range $0..n$ into an empty map. XI
C.2 Times for merging an empty map with a map containg 1000, 10,000, or 100,000 elements using union. XII

D.1 Maximum residency for inserting values in the range $0..n$ into an empty map. XIII
D.2 Total memory usage for inserting values in the range $0..n$ into an empty map. XIV
D.3 Total memory usage for inserting values in the range $0..n$ into an empty map, with logarithmic x and y axis. XIV
D.4 Total memory usage for for union where n is fixed with 100 000 elements. XV
D.5 Total memory usage for for union where m is fixed with 100 000 elements. XV

List of Listings

1.1 An elementary function-based collection in Agda. 4

4.1 Agda definition of AVLMapIndexed. 20
4.2 Standard library definition of Any. Reproduced from The Agda Community, 2023 20
4.3 An example proof using compare. 21
4.4 Agda definition of Any 22
4.5 Standard library type definition of All. Reproduced from The Agda Community (2023). 22
4.6 The CorrectAll abstraction extending a BasicMap. 23
4.7 Agda definition of All. 24
4.8 The AVLMap definition, hiding the ordering and height proofs. 24
4.9 Standard library unionWith. 24
4.10 Example showcasing the problem with the fold-based union. Attempting to give the argument m in the hole will produce the error: `proj₁ (insertWith k (f v) m) ⊕ h2 != h2`. 25
4.11 Type declarations of splitAt and unionWith. 26
4.12 Return types of splitAt and unionWith. 27
4.13 The type declaration of general join, gJoin. The name join is reserved for another joining function in the code base. The implementation of gJoin can be found in the git repository for the project. 27

5.1 Proof for Law 3.22 with helper function types. 30

6.1 A recurrence describing the minimum number of nodes in an AVL-tree (Adelson-Velskii & Landis, 1962) 38

1

Introduction

Collections are data structures that associate keys with values. Common examples are hash maps or search trees, for example, red-black trees or AVL trees (Adelson-Velskii & Landis, 1962). Such collections are useful tools for programming and problem-solving and are often provided directly in a programming language itself or through libraries.

Agda is a dependently typed functional programming language that is often used as a proof assistant. The ability to act as a proof assistant makes Agda a viable option for the formal verification of computer programs. For example, a compiler or interpreter can be written in such a way that it will always produce code that is correct according to some language's specifications. The typing contexts and environments in compilers and interpreters are commonly implemented using collections since they provide efficient lookup and insert operations.

Currently, there does not exist any efficient collection for Agda that is proven correct for all operations. Thus, this thesis aims to design and implement a provably correct collection for Agda as well as to explore the possibilities of introducing additional correct-by-construction aspects into the data structure. Furthermore, we will use proof erasure and evaluate the impact that it has on performance and memory usage.

1.1 Problem and Goal

We wish to explore and implement a provably correct collection in the Agda programming language and proof assistant. To allow for many different implementations, an abstraction will be defined. The abstraction shall be defined in terms of a set of operations together with laws for each operation that assert correct behaviour. Our implementation is guided by the following three (3) questions:

1. Which commonly used functions are lacking proofs and is it possible to add and prove these?

2. How much of an underlying map implementation can be abstracted while retaining the usability and flexibility of the map?

3. Can the implementation be written such that parts of the code used only for verification are erased during compilation?

An initial goal of this thesis was to explore whether correct-by-construction key-value membership properties could be integrated into the data structure itself. We found that it was hard to justify the added complexity of development and that there was no great way of doing it, which is why we removed that goal from the research questions above. Our thoughts and experiments regarding correct-by-construction key-value membership can be found in Chapter 6, Section 6.1.

1.2 Limitations

The implementation of this thesis is based on AVL trees, following previous work in the Agda standard library (The Agda Community, 2023). Only AVL trees have been proven correct for this thesis, so there might be other data structures that are more suited for the task at hand. An alternative that is explored to some degree in Chapter 6 is weight-balanced trees, but none of their properties have been proven.

The thesis focuses on a few operations, namely insertion, lookup, deletion, and union, along with membership data types. This limitation of scope is useful as it allows for a focused review of existing and new laws. Certain aspects of the implementation have also been restricted to allow for ease of proving. An example of this is StrictTotalOrder. The StrictTotalOrder from the standard library allows any instance of equality to be defined. Our implementation uses the built-in equality relation, leading to less generality and potential issues in some data types.

1.3 Ethical and Ecological Implications

Faulty software can have dire consequences, as in Ben-Ari (2001) and Wong et al. (2009). Agda is an open source language that strives for complete correctness as it is used in formal verification and mathematical reasoning. Maintaining correctness reduces the possibility of errors caused by the language itself, but a poorly written program might still produce incorrect or inconsistent results. By providing a commonly used provably correct data structure, we assist Agda developers in writing software with fewer bugs and errors. Additionally, Agda's open-source nature allows these improvements in correctness to be distributed to a large group of users, resulting in a greater impact on the programming community as a whole.

It is difficult to find any possible ecological implications for our research as the connection between the correctness of software and its ecological impact is a tenuous one at best.

1.4 Code Availability

The complete source code and programs used for the benchmarking are publicly available in the following GitHub repository: `https://github.com/MiztaOak/Provably_Correct_Finite_Maps`.

1.5 Related Work

Collections are common data structures, and therefore there exist many different implementations. In this section, we will delve into existing collections and literature related to this work.

1.5.1 Collections

Conceptually, a collection can be defined as a "collection of things", meaning something that keeps data. Collections do not require the data to be associative, so a collection can be as a set, a list, or anything else that can store a variable amount of data. However, it is considerably common to store associative data in collections. A focus of this thesis is the membership aspect, thus collections that store associative data are of primary focus, but the theory and abstraction can still be used and adapted for non-associative collection types.

One type of collection that keeps associative data is finite maps. The data stored in finite maps are usually referred to as "keys" that are associated with "values". One theory of finite maps is presented by Collins and Syme (1995). In this theory a finite map consists of a set of functions together with laws that define the behaviour of the functions, and thereby the map.

The available literature on the theory of finite maps or collections appears to be rather limited. However, we believe that the theory presented by Collins and Syme is still relevant based on its similarity to finite maps in existing libraries, e.g., Coq's standard library (The Coq Team, 1989) and Agda's standard library (The Agda Community, 2023). For that reason, it is used as the basis for the theory presented in this thesis.

1.5.2 Representations of Collections

Collections have many applications in computer science, for example, graphs, set representations, databases, caching, and many more. We place special importance on the usage of collections, specifically finite maps, for environment and context representations of programming languages. To our knowledge, the usage of data structures to maintain environments was first shown by McCarthy for usage in the LISP programming system (McCarthy, 1960).

The implementation described in McCarthy's classic paper uses a linked list to model the collection. Modern collections frequently use other data structures internally, most commonly tree structures. This is in part due to the increased efficiency of the operations, but also because in naïve implementations two lists might not be logically equal despite behaving in the same way when treating them as finite maps (Collins & Syme, 1995).

It is also possible to implement a collection using an array structure. To assert that each key is unique in an array implementation, an excellent hash function is required, unless a very large array is used. The first of which might be difficult to

implement or expensive to execute, and the second requires a lot of memory. Tree structures inherently ensure that each key is unique and also minimise memory usage by allocating space only for the keys and values that are actually utilised.

Listing 1.1: An elementary function-based collection in Agda.

```
module FunMap (K : Set) (V : Set) (compare : K → K → Dec K) where

  Map : Set
  Map = K → Maybe V

  insert : K → V → Map → Map
  insert k v m c with compare k c
  ... | yes refl = just v
  ... | no _ = nothing

  lookup : K → Map → Maybe V
  lookup k m = m k

  delete : K → Map → Map
  delete k m c with compare k c
  ... | yes refl = nothing
  ... | no _ = m c
```

Arguably, the most intuitive and elegant definition of key-value maps is a function-based approach, like Collins and Syme (1995), or the example shown in Listing 1.1. However, in practice, this can be less efficient than a tree-based approach, as composed functions essentially construct a linked list, causing common operations to have linear complexity compared to logarithmic complexity in trees.

There are many different tree structures for representing collections. While it is possible to use simple search trees it is more appropriate to use balanced search trees as the worst-case time complexity of most operations is improved. The Agda standard library uses AVL trees (Adelson-Velskii & Landis, 1962) to represent a finite map.

AVL trees are balanced by the height of the children. Each child may not differ by more than one as per the AVL invariant. Such a balancing causes the trees to achieve logarithmic complexity in terms of size for the essential operations, `insert`, `lookup`, and `delete` (Adelson-Velskii & Landis, 1962; Knuth, 1998).

Adams describes an alternate approach with weight-balanced trees (Adams, 1993). A weight-balanced tree is balanced by the size of the children and some constant factor or ratio to create a balance. According to Adams, the benefit of balancing by size compared to height is that the size can be utilised to create "smarter" constructors. A simple, or naïve, constructor constructs a tree by concatenating two subtrees t_1 and t_2 at a root node and assigning the size of the new tree to be $1 + \textsc{size}(t_1) +$

SIZE(t_2). A "smart" constructor would check and confirm that the balancing of the new tree is correct, and if not, rebalance it. Smart constructors greatly simplify his algorithms, as all rotations are performed in one place. Adams smart constructors can be likened to the `join` function of Blelloch et al. (2016), despite him providing `join` with a different name in `concat3`. This is because Blelloch et al. show that `join` can be used for the same purposes with helping functions `splitAt` and `join2`.

Different function implementations can vary in efficiency; an example of this is the union function in the Agda standard library. The existing implementation uses a fold-based approach, meaning that it inserts every element of one map into another. The complexity of such an algorithm is defined by the number of elements in the two trees. Inserting an element into a tree of size m is $O(\log m)$ (Knuth, 1998) and performing this action n times, as done for a tree of size n, results in a complexity of $O(n \log m)$. More efficient approaches are, for example, the hedge union algorithm from Adams (1993). Hedge union works by splitting one of the trees at the root node of the other and then joining them together. Blelloch et al. (2016) show that this algorithm has a time complexity of $O(m \log(\frac{n}{m} + 1))$ for two trees with heights m and n such that $m \leq n$. In addition to an overall improved time complexity, the run time is generally more consistent given different input trees. This is easily recognisable by analysing the two complexities or the functions themselves, but is also shown by us in Chapter 5 as a part of our performance evaluation. The union implementation of this thesis is based on the formalisation of Adams' hedge union algorithm by Blelloch et al. (2016).

1.5.3 Verified Collections

Melkonian (2024) has written an Agda library that contains an abstraction for finite maps. The finite map abstraction defines a set of functions and laws similar to Collins and Syme but is extended with several functions and definitions. In addition to an abstraction, there is an example of a function-based finite map in the library. The map definition in this library has been a valuable resource and source of inspiration for this project. The AVL based finite map of the standard library was used as a basis for our work as it already exists in the target language and is integrated in the Agda ecosystem. Therefore, we hope that some of the work in this thesis can be used by the extended Agda community in the future.

Libraries of languages similar to Agda, e.g., Coq (The Coq Team, 1989) or Isabelle (Nipkow et al., 2002) also have finite maps with different implementations. The AVL map in the Coq standard library is similar to the work by Adams (1993) in that most of the balancing is taken care of by a function similar to a smart constructor and by the join function. The approach of Isabelle is very similar to the Agda standard library's AVL trees, both of which are correct by construction directly in terms of the balancing instead of the heights. This means a data type that represents the balance factor, i.e., either both children are of the same height or they differ by one in some direction.

Neither Coq nor Isabelle is correct-by-construction in terms of ordering. McBride (2014) shows in his paper 'How to keep your neighbours in order' an elegant technique

to define ordering correctness in the tree itself, this technique is described in detail in Section 2.3 as it is used in our implementation. Ordering correctness à la McBride is also used in the AVL tree of the Agda standard library. Coq, like Agda, is based on the calculus of constructions (Coquand & Huet, 1988) whereas Isabelle is a more generic proof assistant. The proofs observed in Isabelle are in higher-order logic. While Coq and Isabelle contains proven map implementations, we are not able to use their proofs other than for brief analysis and inspiration. The reason for this is that Isabelle and Coq tend to utilise automation in proving, whereas Agda focuses on a more direct approach where the developer manually writes the proofs. In short, the more automated proving process based on different proving tactics of Coq and Isabelle makes it hard to use their results in our proofs.

2

Theory

To realise the goal of defining and exploring a provably correct finite map, we must first learn more about integral concepts relating to it. This chapter introduces the most important concepts at a high level. These being the Agda programming language, correct-by-construction techniques, and finite maps.

2.1 Agda

Agda (Agda Developers, 2024) is an actively developed functional programming language. Notable features of Agda are its dependent type system and termination checking. To run Agda code it is necessary to first translate it into another language, e.g., Haskell by using the MAlonzo compiler (Benke, 2007), and then compile the resulting source code. Dependent typing allows for types to depend on values, and using this in clever ways allows the developer to express invariants directly in the data structure. Such a type could, for example, be the type of all integers strictly larger than some specified integer or a list that is sorted by definition. The syntax of Agda is similar to that of Haskell, but it has some differences that are valuable to know.

- Agda identifiers can contain operators and arbitrary unicode. This allows for usage of mathematical notation, e.g., $\equiv$ for equivalence or $\in$ for membership.
- All functions are required to be total. An example of such a function is the commonly used `head` function that returns the first element in a list. In Haskell, one can simply pattern match on the head of the list and return it; but if the list is empty, this strategy does not execute successfully. In Agda the `head` function must either result in a `Maybe` answer, giving `nothing` if the value does not exist, or it can use the dependent type system to express in the type declaration that the list cannot be empty.
- When developing Agda programs it is common to only utilise the type- and termination checkers, meaning that code is not necessarily executed at any point of development. If the developer wants to type check their program but has yet to finish a long proof, it is possible to use a so called "hole" in place of the remaining proof. Holes are denoted with ? or `{!!}`. Anything in a hole, e.g., `{!m!}`, is not type checked, but it is possible to use Agda to look at m's type and the surrounding context.

- It is possible to declare an argument as implicit if the type checker can figure it out itself. This is done by declaring a type or variable in single braces. For example:

```
n≡n⟹1+n≡1+n : {n : ℕ} → n ≡ n → suc n ≡ suc n
n≡n⟹1+n≡1+n refl = refl
```

  In this case, it is enough to provide a proof of equality since n can be derived from that proof.

- Instance arguments can be declared using double braces, e.g., {{x : Type }}, where Type is some record, data type, or variable. Instance arguments can be thought of as instances of type-class declarations. An example from Agda's documentation:

```
_==_ : {A : Set} → {{eqA : Eq A}} → A → A → Bool
```

  In the example Eq A is some appropriate type that allows for elements of A to be compared.

- A record is a type that is used to group values together. A field in a record declaration contains type declarations associated with the record. Within a record one can declare a constructor for the creation of record values. In a record value a field will instead be associated with a value of the declared type. For example, the dependent pair can be defined using a record:

```
record Σ (S : Set) (T : S → Set) : Set where
  constructor _,_
  field fst : S
        snd : T fst
```

  The Σ type will be used throughout the code and represents the type where the type of the second element depends on the first. This type is most commonly seen in our implementation using the _×_ function:

```
_×_ : (S T : Set) → Set
S × T = Σ S λ _ → T
infixr 5 _×_
```

- The --erasure option will be used in later chapters. With this option enabled, one can specify arguments, types, or functions with @0 or @erased causing them to be erased, or ignored, during compilation so that they are absent at run-time. This can, for example, remove an expensive proof from an otherwise efficient operation.

- Agda allows for local modules, meaning that module can be declared almost at any point in a file. This is commonly used to define module-specific parameters required by the data types, functions, and records in the module. An example of this can be seen in Section 2.3 later in this chapter. There, the BST type exists within the BSTM module which is parameterised by the relation R.

One of the key features of Agda is its ability to act as a proof assistant. This works because of the Curry-Howard correspondence (Howard et al., 1980), which expresses an equivalence between purely functional computer programs and mathematical proofs. In particular, a type corresponds to a proposition and a term to a proof. In the context of a language like Agda, this would mean that the type of a function represents a proposition, and its implementation the proof. For example, given a data type representing equivalence, we can show that $1 + 1 \equiv 2$:

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
a + zero = a
a + (suc b) = suc (a + b)

data _≡_ {A : Set} (n : A) : A → Set where
  refl : n ≡ n

1+1≡2 : (suc zero) + (suc zero) ≡ suc (suc zero)
1+1≡2 = refl
```

Further, in the proof that $1 + 1 \equiv 2$, Agda immediately understands that the type and proposition, (suc zero) + (suc zero) ≡ suc (suc zero) is equivalent, as it uses the definition of the _+_ function first to simplify the expression. This allows us to immediately use the constructor of _≡_, refl, as the implementation and proof. More involved proofs may require some assistance from the developer, for example:

```
_⊔_ : ℕ → ℕ → ℕ
zero ⊔ n      = n
suc a ⊔ zero = suc a
suc a ⊔ suc b = suc (a ⊔ b)

a⊔a≡a : (a : ℕ) → a ⊔ a ≡ a
a⊔a≡a zero = refl
a⊔a≡a (suc a) rewrite a⊔a≡a a = refl
```

The _⊔_ operation expresses the maximum of two numbers, and the proof shows that the maximum of a number compared with itself is, in fact, the same number. The issue is that when normalising, the final line in _⊔_, suc $(a \sqcup a)$, cannot simplify to suc a. The rewrite call helps Agda by showing that $a \sqcup a \equiv a$, which in turn causes the proof to look for suc $a \equiv$ suc a instead of suc $(a \sqcup a)$.

2.2 A Theory of Finite Maps

As previously mentioned collections, also known as finite maps, associative lists, or key-value maps, are well-known and often used data structures within software

Table 2.1: Functions from Collins and Syme (1995).

`Empty`	The finite map with no elements in its domain.
`Update` $f\ (x, y)$	Map x to y in the finite map f. There is no restriction on whether x exists in f, if $x \in f$ then the mapping will be updated so that x maps to y in f.
`Apply` $f\ x$	If $x \in f$ then `Apply` $f\ x$ returns some value v such that x maps to v in f.
`Domain` $f\ x$	Return a boolean value of `True` if $x \in f$, if not, return `False`.

development. Arguably one of the more important use cases for collections is the modeling of typing contexts and environments in programming languages. In this section, we wish to dive deeper into the theory of such data structures.

A finite map may have different purposes, but essential to most, if not for all applications, is the ability to associate keys with values. A theory of finite maps contains two parts: the functions that act upon a finite map and the axioms that define their behaviour. One such theory is described by Collins and Syme (1995). The functions described in their theory are available in Table 2.1.

For anyone familiar with computer science or programming, these functions are familiar, albeit some with different names. `Update` would be referred to as `insert`, `Apply` as `lookup`, and `Domain` as `member`. Collins and Syme explain that this simple set of functions is enough to be used to prove any property of finite maps. The set of functions is not provided on its own; each function must have a corresponding set of axioms to assert that the behaviour is correct. An axiom, in essence, is simply a proof of a certain behaviour of some function, but they are referred to as axioms as Collins and Syme reasons that they must all exist together, and if any axiom is removed the remaining set can no longer define a complete theory.

$$\vdash \forall f\ a\ b.\ \texttt{Apply}\ (\texttt{Update}\ f\ (a, b))\ a = b \tag{2.1}$$

$$\begin{aligned}&\vdash \forall x\ a.\ (a \neq c) \to \\ &\quad \forall f\ b\ d.\ (\texttt{Update}\ (\texttt{Update}\ f\ (a, b)))(c, d) = \texttt{Update}\ (\texttt{Update}\ f\ (c, d))(a, b)\end{aligned} \tag{2.2}$$

$$\vdash \forall f\ a\ b\ c.\ \texttt{Update}(\texttt{Update}\ f(a, b))(a, c) = \texttt{Update}\ f\ (a, c) \tag{2.3}$$

$$\vdash \forall a. \neg(\texttt{Domain Empty}\ a) \tag{2.4}$$

$$\vdash \forall f\ a\ b\ x.\ \texttt{Domain}\ (\texttt{Update}\ f\ (a, b))x = (x = a) \lor \texttt{Domain}\ f\ x \tag{2.5}$$

$$\begin{aligned}&\vdash \forall P. \\ &\quad P\ \texttt{Empty} \land \\ &\quad (\forall f.\ P\ f) \to (\forall x. \neg(\texttt{Domain}\ f\ x) \to \forall y.\ P\ (\texttt{Update}\ f\ (x, y))) \\ &\quad \to \forall f.\ P\ f\end{aligned} \tag{2.6}$$

Figure 2.1: The axioms of Collins and Syme (1995).

The first six axioms shown in figure 2.1 express the intended behaviour of the func-

tions, with axiom 2.6 being an inductive principle. Specifically, it is a strong induction principle.

Collins and Syme (1995) have proven all of their axioms in the HOL proof assistant, thereby showing that their theory is correct. They also show that the theory is consistent, as they are able to derive it using a function-based model. Finally, they show that the theory is complete, by utilising the same function-based model and showing that a bijection exists between the map described by the theory and the model.

2.3 Correct-by-Construction

Correct-by-construction is a technique used to assert correctness within the data structures themselves. This technique is especially useful for dependently typed languages, as the types can depend on instances of types. This allows correct-by-construction data structures to, at times, immediately be used to prove the correctness of operations, making it a matter of type-checking (Pierce, 2002).

The paper 'How to keep your neighbours in order' (McBride, 2014) describes a general technique to assert the correctness of ordering in data structures where an ordering invariant is necessary. The technique describes how to leverage Agda to automatically infer proofs of correctness while also hiding these proofs themselves, allowing for readable code and ease of use for the developer. McBride describes two different instances of the technique: for a universe of the PolyP language extension by Jansson and Jeuring (1997) and binary search trees.

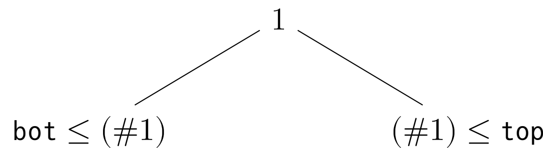
Appropriately, we can show a type of trees from the paper that is ordered by construction as an example. The following code originates from McBride's paper, is then further modified by our supervisor Andreas Abel, and finally modified again and presented here by us:

```
Rel : Set → Set₁
Rel A = A × A → Set

data Order (A : Set) : Set where
  top : Order A
  # : A → Order A
  bot : Order A

ext : ∀ {A} → Rel A → Rel (Order A)
ext R (_ , top) = ⊤
ext R (# x , # y) = R (x , y)
ext R (bot , _) = ⊤
ext R _ = ⊥

module BSTM {A : Set} (R : Rel A) where

  data BST (l u : Order A) : Set where
```

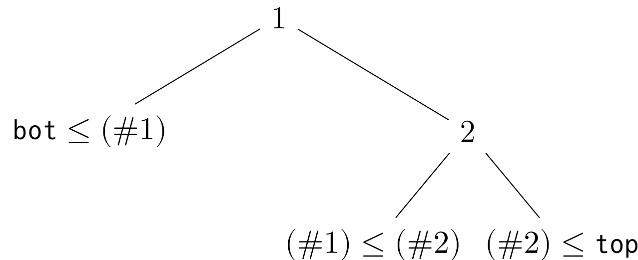
$$\text{leaf} : \{\{l{<}u : \text{ext}\ R\ (l\ ,\ u)\}\} \rightarrow \text{BST}\ l\ u$$
$$\text{node} : (k : A) \rightarrow \text{BST}\ l\ (\#\ k) \rightarrow \text{BST}\ (\#\ k)\ u \rightarrow \text{BST}\ l\ u$$

Along with the tree definition, BST, there are some auxiliary definitions, the Order data type and the ext function. The ext function expresses the ordering of the Order data type by utilising the relation on A. The constructors top and bot are shown to represent the top and bottom of an ordering. The data type itself is parameterised using the ordering, l and u in the BST type. The crucial part is in the left and right tree in the node constructor of BST, where we can see the # constructor of Order in action. The proof information of the ordering is stored in the leaves.

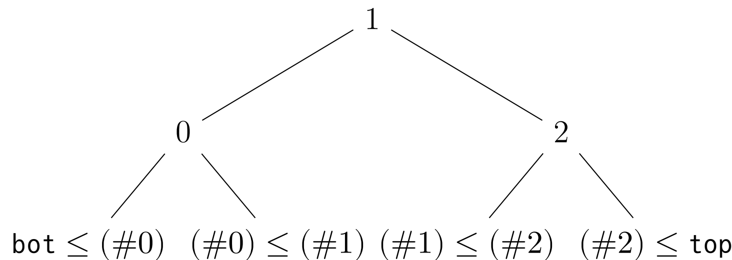
So how does this express an ordering? The simple and elegant answer is that an initial node is constructed such that both leaves contain evidence that the element is larger than `bot` and smaller than `top`:



Insertion of a new element into this tree, e.g., 2, results in the following tree:



See where this is going? Inserting one more element, 0, will make us certain:



The ordering proofs stored in the leaves express exactly the ordering of the tree! The technique described is used for McBride's BST, the AVL tree of the standard library, and our AVL-tree implementation.

3

An Extension of A Theory of Finite Maps

The theory presented by Collins and Syme (1995) provides a solid foundation for any implementation of collections. Collins and Syme uses the word “axiom” when defining the behaviour of finite map operations, we will instead use the term “law”. The reason for this is that the laws defined by us are not created just for specification of a theory but also with Agda development in mind. Additional functions and data types require that new laws be added and existing laws be modified. In this chapter, we present and reason about the additional functions and laws presented in our theory. In essence, our theory is grounded in Collins and Syme’s theory and is extended by two functions and two data types along with their laws.

The functions were gathered by an analysis of existing libraries as well as personal experience in programming with collections. Added functions are `insertWith`, `delete`, `unionWith`, and a lookup function related to membership types. A reason for deciding on the `unionWith` function is the possibility to reuse some of the helper functions; this is covered in depth in Chapter 6.

Instead of providing the `Domain` function, we decided to use membership types instead. There are two data types defined, $_ \in _$ and $_ \mapsto _ \in _$. The two types are provided to express different, but similar, properties. These data types are expressed by following two types:

$$_ \in _ : Key \rightarrow Map \rightarrow Set$$
$$_ \mapsto _ \in _ \; : Key \rightarrow Value \rightarrow Map \rightarrow Set$$

The type $k \in m$ expresses that a key k exists in a collection m, and $k \mapsto v \in m$ represents a more exact relationship where k is associated with v in the collection m. There exists an equivalence between $k \mapsto v \in m$ and `lookup` $k\ m \equiv v$, an interested reader may look at the laws in Figure 3.1 where it is shown.

The decision of using data types is to facilitate the verification of membership properties, because the data type expresses an absolute relationship. Compare this to `Domain` and `Apply` where one would still need to prove that `Domain` $k\ m \equiv$ `True` or `Apply` $k\ m \equiv v$ for the key k, collection m, and value v. In such cases, we believe that using the data types that express membership is easier to reason with compared to first showing that `Domain` $k\ m \equiv$ `True` or `Apply` $k\ m \equiv v$.

$$\vdash \forall k\ v \to \neg(k \mapsto v \in \emptyset) \tag{3.1}$$
$$\vdash \forall k \to k \notin \emptyset \tag{3.2}$$
$$\vdash \forall k \to \texttt{lookup}\ \emptyset\ k \equiv \texttt{nothing} \tag{3.3}$$
$$\vdash \forall m\ k\ \{v\} \to \texttt{lookup}\ m\ k \equiv \texttt{just}\ v \to k \mapsto v \in m \tag{3.4}$$
$$\vdash \forall m\ k\ v \to k \mapsto v \in m \to \texttt{lookup}\ m\ k \equiv \texttt{just}\ v \tag{3.5}$$
$$\vdash m\ k \to k \notin m \to \texttt{lookup}\ m\ k \equiv \texttt{nothing} \tag{3.6}$$
$$\vdash \forall m\ k \to \texttt{lookup}\ m\ k \equiv \texttt{nothing} \to k \notin m \tag{3.7}$$
$$\vdash \forall m\ k\ v \to k \mapsto v \in m \to k \in m \tag{3.8}$$
$$\vdash \forall m\ k \to k \in m \to (\exists v \to k \mapsto v \in m) \tag{3.9}$$

Figure 3.1: Laws defining the behaviour of `lookup`, $_ \mapsto _ \in _$, $_ \in _$, as well as the relation between them.

$$\vdash \forall k\ m\ f \to \texttt{lookup}\ k\ (\texttt{insertWith}\ k\ f\ m) \equiv f\ (\texttt{lookup}\ m\ k) \tag{3.10}$$
$$\begin{aligned}\vdash \forall k\ k'\ f\ f'\ m \to k \not\equiv k \\ \to \texttt{insertWith}\ k\ f\ (\texttt{insertWith}\ k'\ f'\ m) \\ \doteq \texttt{insertWith}\ k'\ f'\ (\texttt{insertWith}\ k\ f\ m)\end{aligned} \tag{3.11}$$
$$\vdash \forall m\ k\ x\ v\ f \to\ x \mapsto v \in (\texttt{insertWith}\ k\ f\ m) \to (x \equiv k) \uplus x \mapsto v \in m \tag{3.12}$$
$$\vdash \forall k\ v\ m \to k \mapsto v \in (\texttt{insert}\ k\ v\ m) \tag{3.13}$$
$$\begin{aligned}\vdash \forall k\ k'\ v\ v'\ m \to\ k \mapsto v \in m \to k \not\equiv k' \\ \to k \mapsto v \in (\texttt{insert}\ k'\ v'\ m)\end{aligned} \tag{3.14}$$
$$\vdash \forall k\ v\ v'\ m \to k \mapsto v \in (\texttt{insert}\ k\ v'\ m) \to v \equiv v' \tag{3.15}$$

Figure 3.2: Insert related laws.

$$\begin{aligned}\vdash P \\ \to P\ \emptyset \times (\forall m \to P\ m \to \forall k\ v\ \to k \notin m\ \to P\ (\texttt{insert}\ k\ v\ m)) \\ \to (\forall m \to Pm)\end{aligned} \tag{3.16}$$
$$\begin{aligned}\vdash P \\ \to P\ \emptyset \times (\forall m \to P\ m \to \forall k\ v \to P\ (\texttt{insert}\ k\ v\ m)) \\ \to (\forall m \to P\ m)\end{aligned} \tag{3.17}$$

Figure 3.3: The strong and weak induction principles for finite maps, respectively.

3.1 Modification of Collins and Symes' Axioms

All functions except for `Domain` in Collins and Syme (1995) exist in our theory, but, as mentioned previously, the names differ. The functions used by Collins and Syme are visible in Table 2.1 in the previous chapter. The names are changed to more widely recognised names, for example `insert` instead of `Update` and `lookup` instead of `Apply`. Additionally, we utilise Agda's unicode support to express the empty set as $\emptyset$ and the membership data types using appropriate symbols, i.e., $\in$ and $\mapsto$.

Collins and Symes' set of laws is extended with seven (7) new laws, visible in Figure 3.4. These laws cover the expected behaviour of `lookup`, `insert`, and `insertWith` as well as some interactions between `insertWith` and `lookup`. The reasoning behind introducing new laws is partly due to the new data types that express membership and also because they assist in proof writing and reasoning. The complete set of laws for the basic set of operations is available in Figures 3.1, 3.2, and 3.3.

$$\vdash \forall k \ \rightarrow \ \texttt{lookup}\ k\ \emptyset \equiv \texttt{nothing}$$
$$\vdash \forall m\ k\ v \rightarrow \texttt{lookup}\ k\ m \equiv v \rightarrow k \mapsto v \in m$$
$$\vdash \forall m\ k\ v \rightarrow k \mapsto v \in m \rightarrow \texttt{lookup}\ k\ m \equiv v$$
$$\vdash \forall k\ m\ f \rightarrow \texttt{lookup}\ k\ (\texttt{insertWith}\ k\ f\ m) \equiv f\ (\texttt{lookup}\ m\ k)$$
$$\vdash \forall k\ v\ m \rightarrow k \mapsto v \in (\texttt{insert}\ k\ v\ m)$$
$$\vdash \forall k\ v\ v'\ m \rightarrow k \mapsto v \in (\texttt{insert}\ k\ v'\ m) \rightarrow v \equiv v'$$
$$\vdash \forall k\ k'\ v\ v'\ m \rightarrow \ k \mapsto v \in m \rightarrow k \not\equiv k' \rightarrow k \mapsto v \in (\texttt{insert}\ k'\ v' m)$$

Figure 3.4: Additional insertion and lookup laws introduced by us.

3.2 Equivalence Between Finite Maps

The notion of equality is important for the next section of this chapter. Previously, we have seen the $\equiv$ notation, but this is not suitable when working with all implementations of finite maps. The definition of $\equiv$ in Agda defines intensional equality, meaning that the structures themselves must also be equal. For example, consider the following two tree representations of a map:

As we can see the domain of the two trees are the same and the same elements map to the same values, but as the structures are not equivalent they are not equal by $\equiv$ in Agda. For every element to exist in both maps and map to the same values

is instead defined to be equality between maps, and it is the definition that we use. The equality between maps is represented by the symbol $\doteq$ in our theory and is defined as follows:

$$m_1 \doteq m_2 = m_1 \subseteq m_2 \times m_2 \subseteq m_1.$$

As part of the definition we use $\subseteq$, which is defined for $t_1 \subseteq t_2$ when all elements in the domain of t_1 exist in t_2 and are mapped to the same values:

$$t_1 \subseteq t_2 = \forall k\ v.\ k \mapsto v \in t_1 \rightarrow k \mapsto v \in t_2.$$

3.3 Additional Functions and their Laws

$$\vdash \forall k\ m \rightarrow k \notin m \rightarrow \texttt{delete}\ k\ m \doteq m \tag{3.18}$$
$$\vdash \forall k\ m \rightarrow k \notin \texttt{delete}\ k\ m \tag{3.19}$$
$$\vdash \forall k_1\ k_2\ v\ m \rightarrow k_1 \mapsto v \in m \rightarrow k_1 \not\doteq k_2 \rightarrow k_1 \mapsto v \in \texttt{delete}\ k_2\ m \tag{3.20}$$
$$\vdash \forall k_1\ k_2\ v\ m \rightarrow k_1 \mapsto v \in \texttt{delete}\ k_2\ m \rightarrow k_1 \mapsto v \in m \tag{3.21}$$
$$\vdash \forall k\ v\ m \rightarrow k \mapsto v \notin \texttt{delete}\ k\ m \tag{3.22}$$
$$\vdash \forall k_1\ k_2\ m \rightarrow \texttt{delete}\ k_1\ (\texttt{delete}\ k_2\ m) \doteq \texttt{delete}\ k_2\ (\texttt{delete}\ k_1\ m) \tag{3.23}$$

Figure 3.5: Laws related to `delete`.

As mentioned in the introduction of this chapter, we extend the theory with two functions, namely `delete` and `unionWith`. The behaviour of these functions are expressed by the laws in Figures 3.5 and 3.6. In order to make sense of the behaviour defined by the laws, we may also express it in simpler terms.

`delete` has the sole purpose of deleting an element, and its associated value, from a collection. A deleted key in a collection `m` must not map to any value in `m`, **and**, it must not exist in `m`. The deletion laws, shown in Figure 3.5, express these properties as well as commutativity.

The purpose of `unionWith` is to “merge” two collections using a function to determine what happens in case of a collision. The behaviour of keys in a merged map is defined by laws 3.26 and 3.27. Together, they state that every key in the input maps must exist in the resulting map and that every key in the union must exist in the first input map, or the second input map, or in both. Laws 3.26 and 3.27 must both exist in order to assert that no additional keys are added or removed from a union of two maps.

However, defining a law for the value associated with a key is more difficult. This is because any union implementation must be biased in some way since the comparison and conflict resolution function will be performed in some order depending on the union call. We propose law 3.28 for our implementation, but this law may require modification for other implementations with different biases. In addition to the laws expressing key and value behaviour, there is a law of congruence 3.31.

$$\vdash \forall m\ f \rightarrow \texttt{unionWith}\ f\ m\ \emptyset \doteq\ m \tag{3.24}$$

$$\vdash \forall m\ f \rightarrow \texttt{unionWith}\ f\ \emptyset\ m \doteq m \tag{3.25}$$

$$\begin{aligned}&\vdash \forall m_1\ m_2\ f\ k\\ &\qquad \rightarrow k \in \texttt{unionWith}\ f\ m_1\ m_2\\ &\qquad \rightarrow k \in m_1 \uplus k \in m_2 \uplus (k \in m_1 \times k \in m_2)\end{aligned} \tag{3.26}$$

$$\begin{aligned}&\vdash \forall m_1\ m_2\ f\ k\\ &\qquad \rightarrow k \in m_1 \uplus k \in m_2\\ &\qquad \rightarrow k \in \texttt{unionWith}\ f\ m_1\ m_2\end{aligned} \tag{3.27}$$

$$\begin{aligned}&\vdash \forall k\ v_1\ v_2\ m_1\ m_2\ f\\ &\qquad \rightarrow k \mapsto v_1 \in m_1\\ &\qquad \rightarrow k \mapsto v_2 \in m_2\\ &\qquad \rightarrow k \mapsto f\ v_1\ v_2 \in \texttt{unionWith}\ f\ m_1\ m_2\end{aligned} \tag{3.28}$$

$$\begin{aligned}&\vdash \forall k\ v\ m_1\ m_2\ f\\ &\qquad \rightarrow k \mapsto v \in m_1\\ &\qquad \rightarrow k \notin m_2\\ &\qquad \rightarrow k \mapsto v\ \in \texttt{unionWith}\ f\ m_1\ m_2\end{aligned} \tag{3.29}$$

$$\begin{aligned}&\vdash \forall k\ v\ m_1\ m_2\ f\\ &\qquad \rightarrow k \notin m_1\\ &\qquad \rightarrow k \mapsto v \in m_2\\ &\qquad \rightarrow k \mapsto v\ \in \texttt{unionWith}\ f\ m_1\ m_2\end{aligned} \tag{3.30}$$

$$\begin{aligned}&\vdash \forall f\ m_1\ m_2\ m_3\\ &\qquad \rightarrow m_1 \doteq m_2\\ &\qquad \rightarrow \texttt{unionWith}\ f\ m_1\ m_3 \doteq \texttt{unionWith}\ f\ m_2\ m_3\end{aligned} \tag{3.31}$$

Figure 3.6: Laws related to `unionWith`.

4

Implementation

This chapter is dedicated to presenting our finite map implementation in Agda. It contains three parts; an abstract representation, the underlying implementation, and finally the implementation of the abstraction.

4.1 Abstraction

The extended theory presented in Chapter 3 is implemented in Agda as the following three (3) interfaces:

- BasicMap, replicating the extended theory of Collins and Syme described in Section 3.1.
- DeletableMap, extending a Basicmap by adding delete and the laws in Figure 3.5.
- MergableMap, which extends a Basicmap by adding union and the corresponding laws in Figure 3.6.

The splitting of the theory into three separate interfaces allows map implementations without a union or deletion function to reason about the correctness of insertion and lookup in line with Collins and Syme (1995). The definitions for the interfaces can be found in Appendix A.

4.2 AVL Tree Implementation

There are two primary differences between our implementation and the standard library. First, the key-value type was simplified by using a tuple of the key and value, Key × Value, instead of the dependent key-value pairs favoured by the The Agda Community. Secondly, the ordering proofs stored in the leaves of the tree were erased using the `@erased` notation. This was done in the hopes of improving the runtime performance of the different map operations since the erasure would remove the relatively slow operations on the ordering proofs such as mklim (the implementation of which can be found in Appendix A). An evaluation of the performance implications of this can be found in Section 5.3. It was regrettably impossible to erase the height or balancing proofs as a number of the map operations case splits on the height and

balancing proofs, making type erasure impossible in the current version of Agda. These modifications resulted in the map datatype in Listing 4.1

Listing 4.1: Agda definition of AVLMapIndexed.

```
data AVLMapIndexed (@0 V : Set v) (@0 l u : Key⁺) : @0 ℕ → Set (k ⊔ v ⊔ ℓ₁) where
  leaf : {{@erased l<u : l <⁺ u}} → AVLMapIndexed V l u 0
  node : ∀ {hl hr h}
          → ((k , v) : Key × V)
          → (lm : AVLMapIndexed V l [ k ] hl)
          → (rm : AVLMapIndexed V [ k ] u hr)
          → (bal : hl ~ hr ⊔ h)
          → AVLMapIndexed V l u (suc h)
```

The map operations insertWith, lookup, and delete, as well as the required helper functions, were directly lifted from the standard library implementation (The Agda Community, 2023) and modified to erase unnecessary proofs. The implementation of these are visible in Section A.5. The unionWith function on the other hand, was based on the algorithm introduced in Adams (1993) and will be discussed in more detail in Section 4.4.

Listing 4.2: Standard library definition of Any. Reproduced from The Agda Community, 2023

```
data Any {V : Value v} (P : Pred (K& V) p) {l u}
     : ∀ {n} → Tree V l u n → Set (p ⊔ a ⊔ v ⊔ ℓ₂) where
  here : ∀ {hˡ hʳ h} {kv : K& V} → P kv →
          {lk : Tree V l [ kv .key ] hˡ}
          {ku : Tree V [ kv .key ] u hʳ}
          {bal : hˡ ~ hʳ ⊔ h} →
          Any P (node kv lk ku bal)
  left  : ∀ {hˡ hʳ h} {kv : K& V}
          {lk : Tree V l [ kv .key ] hˡ} →
          Any P lk →
          {ku : Tree V [ kv .key ] u hʳ}
          {bal : hˡ ~ hʳ ⊔ h} →
          Any P (node kv lk ku bal)
  right : ∀ {hˡ hʳ h} {kv : K& V}
          {lk : Tree V l [ kv .key ] hˡ}
          {ku : Tree V [ kv .key ] u hʳ} →
          Any P ku →
          {bal : hˡ ~ hʳ ⊔ h} →
          Any P (node kv lk ku bal)
```

4.2.1 Any

The `Any` datatype is traditionally used to express that some predicate P holds for at least one value in a collection. In our case, it will be used to express that some key and value pair in the map is equal to the key and value pair stipulated in the predicate. In other words, we will define $k \mapsto v \in m$ as `Any` $(\lambda(k' \ , v') \to k \equiv k' \wedge v \equiv v')\ m$.

Our Any implementation was heavily inspired by the Agda Standard library (The Agda Community, 2023), Figure 4.2, with three notable differences. Firstly, erasure in the map datatype requires the erasure of the corresponding proofs in the Any datatype. Secondly, and more interestingly, we added two additional ordering proofs to the constructors left and right. This was done to simplify certain proofs where a comparison of two values and an Any proof might disagree on the order of elements. An example of such a proof is illustrated in Listing 4.3. The last two cases in the comparison where the membership proof and the comparison disagree on the ordering k and k' are impossible to resolve without the ordering proof $k < k'$ since nothing is known about the values prior to the comparison.

Listing 4.3: An example proof using compare.

```
example : ∀ k {v l u h} (m : AVLMapIndexed V l u h)
  → k ↦ v ∈ m
  → lookup m k ≡ just v
example k (node (k' , v) lm rm b) (left {{ k<k' }} prf) with compare k k'
... | le k<k' = example k lm prf -- compare agrees with the fact that k < k'
... | eq refl = {!!} -- compare belives that k ≡ k'
... | ge k>k' = {!!} -- compare believes that k > k' but we know that k < k'
```

The second modification required the predicate to only operate on the value, as the Key in the predicate needs to be fixed to a certain value for the ordering proofs to be definable. This can be thought of as a predicate on keys and values of the type $\lambda(k, v) \to k \equiv k' \times P\ v$ where P is some predicate on values and k' is some key. Lastly, we erased the maps from the Any datatype as they could conceivably slow down any operations on or using Any. This modification means that, at compile time, a Any object consists only of a path to the desired element in the map without any stored information about the map itself. These modifications resulted in the Any datatype in Listing 4.4.

Listing 4.4: Agda definition of Any

```
data Any (P : Pred V ℓp) {l u : Key⁺} (kp : Key) :
  ∀ {@0 h : ℕ} → @0 AVLMapIndexed V l u h → Set (k ⊔ ℓ₁ ⊔ v ⊔ ℓp) where
  here : ∀ {h hl hr} {v : V}
         {{@erased l<k : l <⁺ [ kp ]}} {{@erased k≤u : [ kp ] <⁺ u}}
         {@0 lm : AVLMapIndexed V l [ kp ] hl}
         {@0 rm : AVLMapIndexed V [ kp ] u hr}
         {@0 bal : hl ~ hr ⊔ h}
         → P v
         → Any P kp (node (kp , v) lm rm bal)

  left : ∀ {h hl hr} {(k' , v) : Key × V}
         {@0 lm : AVLMapIndexed V l [ k' ] hl}
         {{@0 k<k' : [ kp ] <⁺ [ k' ]}}
         {@0 rm : AVLMapIndexed V [ k' ] u hr}
         {@0 bal : hl ~ hr ⊔ h}
         → Any P kp lm
         → Any P kp (node (k' , v) lm rm bal)

  right : ∀ {h hl hr} {(k' , v) : Key × V}
         {@0 lm : AVLMapIndexed V l [ k' ] hl}
         {@0 rm : AVLMapIndexed V [ k' ] u hr}
         {{@0 k'<k : [ k' ] <⁺ [ kp ]}}
         → {@0 bal : hl ~ hr ⊔ h}
         → Any P kp rm
         → Any P kp (node (k' , v) lm rm bal)
```

Listing 4.5: Standard library type definition of All. Reproduced from The Agda Community (2023).

```
data All {V : Value v} (P : Pred (K& V) p) {l u}
     : ∀ {n} → Tree V l u n → Set (p ⊔ a ⊔ v ⊔ ℓ₂) where
  leaf  : {p : l <⁺ u} → All P (leaf p)
  node : ∀ {hˡ hʳ h} {kv : K& V}
         {lk : Tree V l [ kv .key ] hˡ}
         {ku : Tree V [ kv .key ] u hʳ} →
         {bal : hˡ ~ hʳ ⊔ h} →
         P kv → All P lk → All P ku → All P (node kv lk ku bal)
```

4.2.2 All

The `All` data structure is used to denote that a predicate is universally applicable across all elements of a collection. An illustrative implementation of this data structure can be seen in Listing 4.5. Conceptually, the `All` can be seen as a decorator, augmenting the original map with new information. One common use case for the `All` data structure is to use it as a typing environment by using a type-coercion function as the predicate. It is for this reason that we needed to extend BasicMap with an All instance as we whished to utilise it to define the environment in a proof-of-concept simply typed lambda calculus interpreter. The `All` data type is generalised by implementing it as a record, CorrectAll in Listing 4.6, extending the BasicMap, allowing the developer to leverage the correctness of the map operations.

Listing 4.6: The CorrectAll abstraction extending a BasicMap.

```
record CorrectAll (Map : Set (ℓ ⊔ ℓ' ⊔ ℓ₁)) (bMap : BMap {ℓ₁ = ℓ₁} Map) :
  Set (suc (ℓ ⊔ ℓ' ⊔ ℓ₁ ⊔ ℓa)) where


field
  All : Pred (K × V) ℓa → Map → Set (ℓ ⊔ ℓ' ⊔ ℓ₁ ⊔ ℓa)
  allInsert : {P : Pred (K × V) ℓa} {k : K} {v : V} {m : Map}
    → P (k , v) → All P m → All P (insert k v m)
  allLookup : {P : Pred (K × V) ℓa} {k : K} {v : V} {m : Map}
    → k ↦ v ∈ m → All P m → P (k , v)
```

The concrete implementation was lifted from the standard library and extended with an insertion and lookup function. Furthermore, erasure was used to erase the ordering proofs in the leaves, resulting in the definition in Listing 4.7. The insertion and lookup functions can be found in Appendix A, and their helper functions in our GitHub repository.

4.3 Interface Implementation

The removal of height and balancing proofs is required to implement the abstraction for our AVL map, as the map implementation has the type

$$\mathsf{Key}^{+} \to \mathsf{Key}^{+} \to \mathbb{N} \to \mathsf{Set}\ \ k \sqcup \ell' \sqcup \ell_1 .$$

The abstraction, on the other hand, expects a map of type Set $k \sqcup \ell' \sqcup \ell_1$. The "removal" is accomplished by "wrapping" the map in a data type that hides the height and ordering, as shown in Listing 4.8.

We are also required to implement a version of Any that hides the ordering as a direct consequence of the introduction of AVLMap. This Any instance is implemented in the same way as AVLMap, meaning that it hides unnecessary information, and it will be referred to as AnyM going forward. By utilising AVLMap and AnyM it is possible

Listing 4.7: Agda definition of All.

```
data All (P : Pred (Key × V) ℓa) {@0 l u : Key⁺}
  : ∀ {@0 h : ℕ} → AVLMapIndexed V l u h → Set (k ⊔ ℓ1 ⊔ v ⊔ ℓa) where
  leaf : {{ @0 l<u : l <⁺ u }} → All P leaf
  node : ∀ {hl hr h}
           → {p : Key × V}
           (let (k , v) = p)
           {lt : AVLMapIndexed V l [ k ] hl}
           {rt : AVLMapIndexed V [ k ] u hr}
           {bal : hl ~ hr ⊔ h}
           → P (k , v)
           → All P lt
           → All P rt
           → All P (node (k , v) lt rt bal)
```

Listing 4.8: The AVLMap definition, hiding the ordering and height proofs.

```
data AVLMap (V : Set ℓ') : Set (k ⊔ ℓ' ⊔ ℓ1) where
  map : ∀ {h} → AVLMapIndexed V ⊥⁺ ⊤⁺ h → AVLMap V
```

to instantiate the three interfaces that make up the abstraction. The full definitions of these interfaces, as well as the proofs, can be found in Appendix A, while the implementations can be found in our GitHub repository.

4.4 Union

The Agda standard library contains one possible unionWith implementation, shown in Figure 4.9. This is a naïve implementation defined using foldr that simply inserts every key-value pair in one of the finite maps into another.

Listing 4.9: Standard library unionWith.

```
unionWith : (∀ {k} → Val k → Maybe (Wal k) → Wal k) →
            -- left → right → result.
            Tree V → Tree W → Tree W
unionWith f t1 t2 = foldr (λ {k} v → insertWith k (f v)) t2 t1
```

The "naïvety" of this implementation can be explained in two parts: first, due to the lack of optimisations, and second, because it is very difficult to reason about. One simple and possible optimisation could be pattern matching on the empty map to immediately return $\texttt{t}_1$ or $\texttt{t}_2$. This would eliminate the case where you insert the

entirety of one map into an empty map. For the second part, the current implementation is based solely on the `Tree` abstraction. This abstraction in the standard library hides the underlying implementation, meaning that it is not possible to utilise height, ordering, nor the structure itself, as seen in Section 4.3. The hiding makes it almost impossible to reason about membership properties for this implementation of union. One demonstrative example of this can be seen in Listing 4.10. The problem arises from the fact that the type of insertion in the lambda must match *t2*, which is impossible since the insertion will alter the height of m.

Listing 4.10: Example showcasing the problem with the fold-based union. Attempting to give the argument m in the hole will produce the error:
`proj₁ (insertWith k (f v) m) ⊕ h2 != h2.`

```
inT1Union : ∀ {l u : Key⁺} {h1 h2 : ℕ }
  → (k : Key)
  → {v : V}
  → (f : V → Maybe V → V)
  → {{ l<k : l <⁺ [ k ] }} {{ k<u : [ k ] <⁺ u }}
  → (t1 : AVLMapIndexed V l u h1)
  → (t2 : AVLMapIndexed V l u h2)
  → k ↦ v ∈ t1
  → k ↦ v ∈
     (foldr (λ (k' , v') m → proj₂ $ insertWith k (f v) {!m!} ) t2 t1)
```

With this in mind, we have implemented a `unionWith` function for AVLMapIndexed that can be used in the abstraction, that contains the optimisations lacking in the standard library, and allows for reasoning outside of the abstraction.

Algorithm 1 The unionWith algorithm presented by Blelloch et al. (2016).

function UNIONWITH(f , t_1 , t_2)
  if t_1 is $\emptyset$ **then return** t_2
  if t_2 is $\emptyset$ **then return** t_1
  $(l_2,\ (k_2,\ v_2),\ r_2) \leftarrow t_2$
  $(l_1,\ v_1,\ r_1) \leftarrow \text{SPLITAT}(k_2,\ t_1)$
  $t_L \leftarrow \text{UNIONWITH}(f,\ l_1,\ l_2)$
  $t_R \leftarrow \text{UNIONWITH}(f,\ r_1,\ r_2)$
  return $\text{JOIN}(t_L,\ (k_2,\ f\ v_1\ v_2),\ t_R)$

An efficient divide-and-conquer algorithm for `unionWith` is presented in Blelloch et al. (2016), and is shown in Algorithm 1. The original algorithm was proposed by Adams (1993) for weight-balanced trees but formalised for AVL trees by Blelloch et al.

A join-based `unionWith` is also potentially more efficient than a fold-based approach. The complexity of a fold-based unionWith is simple, for t_1 and t_2 with sizes n and

m respectively, a fold-based unionWith must insert all n elements into a tree of size m. The complexity of insertion is $O(\log m)$ (Knuth, 1998), and as n elements are inserted, the complexity is simply $O(n \log m)$. The time complexity of a join-based implementation first requires knowledge of the two helping functions, `join` and `splitAt`. The proof for the complexity of the join-based algorithm is a bit more complex, so we leave the proof to Blelloch et al. (2016). They show that the complexity is $O(n \log(\frac{n}{m} + 1))$ for several join-based functions, `unionWith` among them.

Listing 4.11: Type declarations of splitAt and unionWith.

```
splitAt : ∀ {@0 l u} {h : ℕ}
  → (k : Key)
  → {{@erased l<k : l <⁺ [ k ]}} → {{@erased k<u : [ k ] <⁺ u}}
  → (m : AVLMapIndexed V l u h)
  → Split k l u h
unionWith : {h1 h2 : ℕ} → {@0 l u : Key⁺}
  → (V → Maybe V → V)
  → (t1 : AVLMapIndexed V l u h1)
  → (t2 : AVLMapIndexed V l u h2)
  → UnionReturn t1 t2
```

We have implemented a join-based unionWith, the full implementation of which is viewable in our GitHub repository. The type declarations of unionWith and splitAt can be seen in Figure 4.11. The unionWith function is implemented to operate exactly according to the algorithm, but with a few caveats due to the data type of our tree. Our tree is correct-by-construction by ordering and height, meaning that all trees produced by any functions must maintain ordering and height properties.

The problem of ordering can be solved easily, any order instance can be extended by an element that is smaller, or larger, than the current limits. This fact can be used when working with unionWith; if the trees do not have the same ordering already, the minimum of the two trees can be set as minimum for both, and the maximum can be extended in the same way.

Height is more problematic. In functions that operate on a single tree, height is often not an issue, as there is only one height to work with. For example, in insert, a single element is inserted into a tree of height h, and the tree can achieve a maximum height of $1+h$ and a minimum of h. The unionWith function, that operates on two arbitrary trees require other height restrictions. The arguments of unionWith itself is as one would expect, it takes a function of type $Value \to Value \to Value$ and two trees, t_1 and t_2, with arbitrary heights. The algorithm depends on two additional functions, splitAt and join, that have height constraints visible in the type declaration of gJoin, Listing 4.13, and the return type of splitAt, Listing 4.12. The problem arises from splitAt which returns two trees with the constraints that they can achieve at most the same height as the input tree. With no lower bound on the trees returned from

Listing 4.12: Return types of splitAt and unionWith.

```
record Split (x : Key) (@0 l u : Key⁺) (h : ℕ) : Set (k ⊔ v ⊔ ℓ₁) where
  constructor split
  field
    value : Maybe V
    leftH : ℕ
    @0 leftP : leftH ≤ h
    leftT : AVLMapIndexed V l [ x ] leftH
    rightH : ℕ
    @0 rightP : rightH ≤ h
    rightT : AVLMapIndexed V [ x ] u rightH

record UnionReturn {@0 l u : Key⁺} {h1 h2 : ℕ}
  (@0 t₁ : AVLMapIndexed V l u h1) (@0 t₂ : AVLMapIndexed V l u h2)
  : Set (k ⊔ v ⊔ ℓ₁) where
  constructor retval
  field
    hof : ℕ
    tree : AVLMapIndexed V l u hof
    @0 prf : hof ≤ (h1 + h2)
```

Listing 4.13: The type declaration of general join, gJoin. The name join is reserved for another joining function in the code base. The implementation of gJoin can be found in the git repository for the project.

```
gJoin : {hl hr : ℕ} {@0 l u : Key⁺}
  → ((k , v) : Key × V)
  → AVLMapIndexed V l [ k ] hl
  → AVLMapIndexed V [ k ] u hr
  → ∃ λ i → AVLMapIndexed V l u (i ⊕ max hl hr)
```

splitAt it is impossible to prove any lower bound in unionWith as it uses both trees returned from split.

The upper bound of unionWith is less than or equal to $\text{HEIGHT}(t_1) + \text{HEIGHT}(t_2)$. This can be proven by induction (Blelloch et al., 2016). We believe that the bound is smaller due to the height properties of gJoin. However, we have not been able to prove this. The bound is explored further in Section 6.2.

5

Result

With an implementation in place, we will in this chapter present the results of proving the laws of our theory, as well as the performance evaluation of the implementation. In addition to proofs and evaluation, we have created a small proof of concept program used to display the usability and flexibility of our finite map.

5.1 Proofs

We have proven 29 of the 35 correctness proofs for the implementation of the AVL map abstraction. More specifically, all the insertion, deletion, lookup, and relational proofs, as well as the union proofs corresponding to Laws 3.24 and 3.25 could be proven. Listing 5.1 displays one of these proofs, which is required by DeletableMap. The definition of delete can be seen in Section A.5 in Appendix A. The strategy of using a set of common helper functions employed by delete-safe' is used in most of the proofs to exploit reoccurring parts of the proof structure.

This leaves 6 of the more complex union proofs as well as the insertion-based induction principle. The union proofs could not be proven as proving the height bounds was of focus initially, delaying any practical implementation. The induction principle, on the other hand, could not be proven as we are unable to show that a finite number of insertions will reconstruct any map, this is discussed in greater detail and two possible approaches are presented in Section 6.2.

5.2 Proof of Concept: Lambda Calculus Interpreter

We implemented two versions of a simply typed lambda calculus interpreter, using the grammar and typing rules shown in Appendix B, Figures B.1 and B.2, to evaluate the usability of the map abstraction. The first version of the interpreter uses our map and `All` abstractions for the typing context and environment, the full implementation of which can be seen in Appendix B. The second version, on the other hand, uses the standard library AVL map and `All` in a nearly identical fashion.

When comparing the ease of use between the two implementations, we found them almost identical. The only real difference was the fact that the standard library did not offer insertion and lookup functions for their `All` data structure. This would

Listing 5.1: Proof for Law 3.22 with helper function types.

```
inᴿ-joinᴿ⁻ : ∀ {l u : Key⁺} {hl hr h : ℕ}
    {k : Key}
    {v : V}
    {p : Key × V}
    (lt : AVLMapIndexed V l [ proj₁ p ] hl)
    (rt⁻ : ∃ (λ i → AVLMapIndexed V [ proj₁ p ] u pred[ i ⊕ hr ]))
    (bal : hl ~ hr ⊔ h)
    → @erased [ proj₁ p ] <⁺ [ k ]
    → k ↦ v ∈ (proj₂ (joinʳ⁻ p lt rt⁻ bal))
    → k ↦ v ∈ (proj₂ rt⁻)

inᴸ-joinᴸ⁻ : ∀ {l u : Key⁺} {hl hr h : ℕ}
    {k : Key}
    {v : V}
    {p : Key × V}
    (lt⁻ : ∃ (λ i → AVLMapIndexed V l [ proj₁ p ] pred[ i ⊕ hl ]))
    (rt : AVLMapIndexed V [ proj₁ p ] u hr)
    (bal : hl ~ hr ⊔ h)
    → @erased [ k ] <⁺ [ proj₁ p ]
    → k ↦ v ∈ (proj₂ (joinˡ⁻ p lt⁻ rt bal))
    → k ↦ v ∈ (proj₂ lt⁻)

notInJoin : ∀ {l u : Key⁺} {hl hr h : ℕ}
   → (k : Key)
   → {v : V}
   → (lm : AVLMapIndexed V l [ k ] hl)
   → (rm : AVLMapIndexed V [ k ] u hr)
   → (bal : hl ~ hr ⊔ h)
   → ¬ (k ↦ v ∈ proj₂ (join lm rm bal))

del-safe' : ∀ {l u : Key⁺} {h : ℕ} (k : Key) {v : V} (m : AVLMapIndexed V l u h)
              {{ @erased l<k : l <⁺ [ k ] }} {{ @erased k<u : [ k ] <⁺ u }}
              → ¬ (k ↦ v ∈ proj₂ (delete k m))
del-safe' k leaf ()
del-safe' k (node p lm rm bal) prf with compare k (proj₁ p)
... | tri< k<p _ _ = let
    prf' = inᴸ-joinᴸ⁻ (delete k {{ p≤u = [ k<p ]ᴿ }} lm) rm bal [ k<p ]ᴿ prf
   in del-safe' k lm {{ k<u = [ k<p ]ᴿ }} prf'
... | tri≈ _ refl _ = notInJoin k lm rm bal prf
... | tri> _ _ p<k = let
    prf' = inᴿ-joinᴿ⁻ lm (delete k {{ [ p<k ]ᴿ }} rm) bal [ p<k ]ᴿ prf
   in del-safe' k rm {{ [ p<k ]ᴿ }} prf'
```

require more work from an end-user perspective if we were to treat the two maps as two different libraries.

However, it should be mentioned that our interpreter can use any implementation of BasicMap for its typing context and environment. This results in a great degree of flexibility since the interpreter itself does not depend on any specific underlying implementation, allowing a developer to change the map implementation to a more efficient or simpler version without having to rewrite any of their code.

5.3 Performance Comparison

The performance of our AVL map implementation was evaluated by compiling it into Haskell using Agda's GHC backend (Benke, 2007) and then benchmarking it against the standard library version using Criterion (O'Sullivan, 2014/2024). The benchmarks were performed using an 8-core AMD Ryzen 7 7700 processor with 32 GB of DDR5 RAM running at 6000 MHz. The map implementations were compared in terms of execution time for insertion and union operations. The insertion operations were compared by measuring the execution speed when inserting 1000, 10 000, 100 000, 1 000 000, 10 000 000 elements in ascending order into an empty map, the results of which can be seen in Figure 5.1.

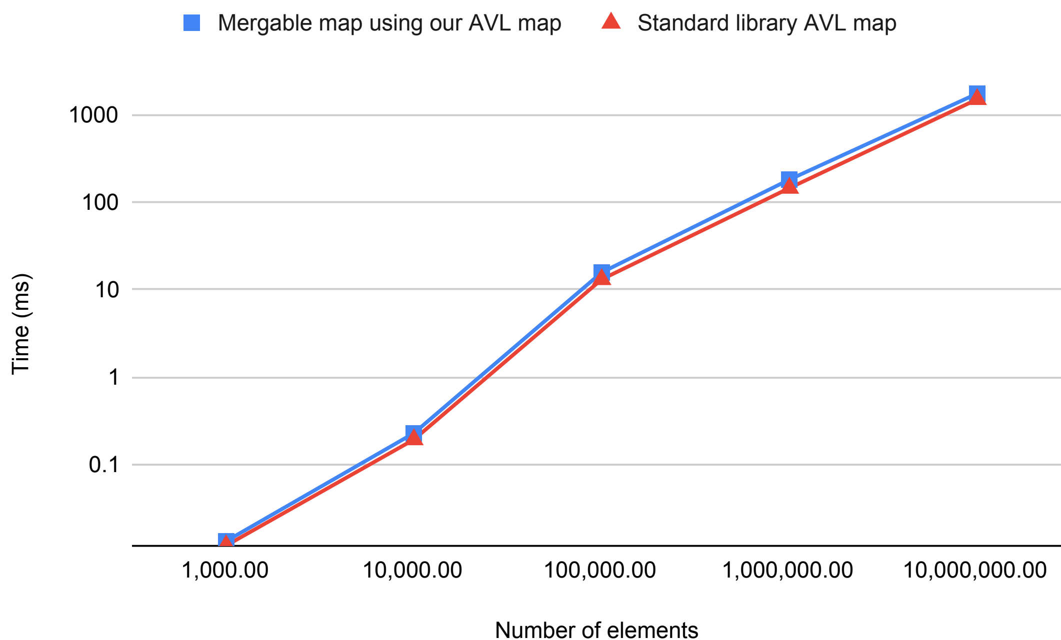
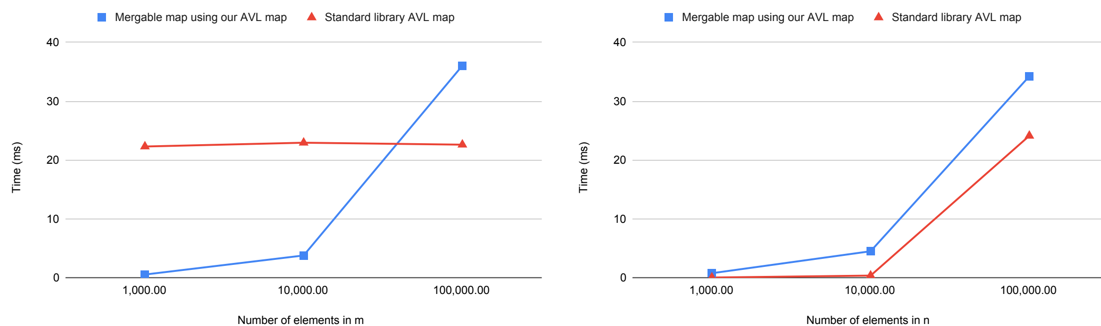


Figure 5.1: Elapsed time for inserting values in the range $0..n$ into an empty map, with logarithmic x and y axes.

Union, on the other hand, was compared in two different ways. First, by merging two overlapping maps where one of the maps contains 100 000 elements and the other contains 1000, 10 000, or 100 000 elements. The results of this can be seen in

Figure 5.2. Second, by merging an empty map with a map containing 1000, 10,000, or 100,0000 elements. The results of which are summarised in Table 5.1 and fully presented in Figure C.2. Each of these comparisons was performed using the same set of key-value pairs to ensure fairness.



(a) n is fixed with 100 000 elements. **(b)** m is fixed with 100 000 elements.

Figure 5.2: Times for merging two overlapping maps, n and m, using union. One of the maps contains 100,000 elements while the other one contains 1000, 10,000, or 100,000 elements.

Operation	Our implementation	Standard library
union $\emptyset$ $\emptyset$	13.4 ns	7.27 ns
union n $\emptyset$	13.11 ns	6.505 ns
union $\emptyset$ n	13.28 ns	22.83 ms

Table 5.1: Times for merging an empty map with a map, n, containing 100,000 elements.

We can conclude based on the results in Figure 5.1 and 5.2 as well as those in Table 5.1 that our AVL map implementation performs worse in all non-empty cases with the exception of performing union when the left tree is disproportionally larger than the right tree, which can be seen in Figure 5.2.

5.4 Memory Profiling

In addition to performance testing, we have performed basic memory profiling. GHC has great support for profiling, but as we are first compiling the Agda code to Haskell, and from Haskell to an executable, we were regretfully not able to model the profiling as precisely as we wished. However, it was possible to compile the program with the `-rtsopts` flag, enabling us to perform rudimentary profiling using the GHC run-time options. Although these results are not as accurate as we desired, we still believe that the results are interesting and relevant.

To force evaluation and consistent results, we used the same programs for which performance evaluation was performed, with the added change that each benchmark

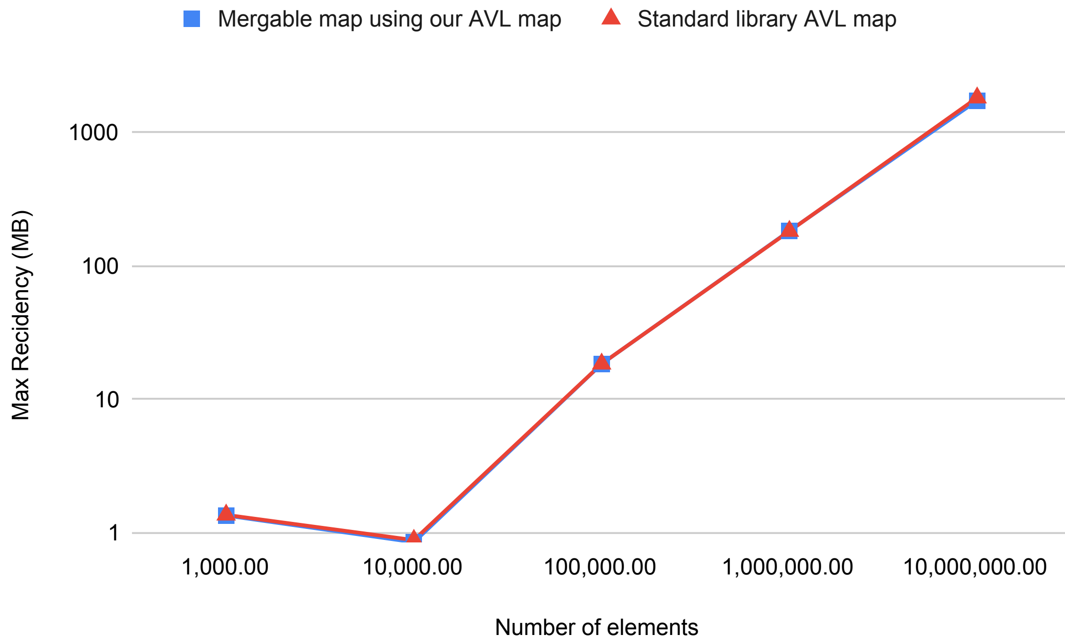


Figure 5.3: Maximum residency for inserting values in the range $0..n$ into an empty map, with logarithmic x and y axes.

was performed separately to produce relevant memory usage statistics. Both maximum residency, visible in Figure 5.3, and total memory usage for insert indicates that erasure does not have a material impact on memory usage. On the other hand, the maximum residency for union, visible in Figures 5.4 and 5.5, indicates that the chosen algorithm has a greater impact on memory usage than erasure. Graphs showing the total memory usage for these benchmarks as well as the linear version of the insertion benchmark, can be found in Appendix D.

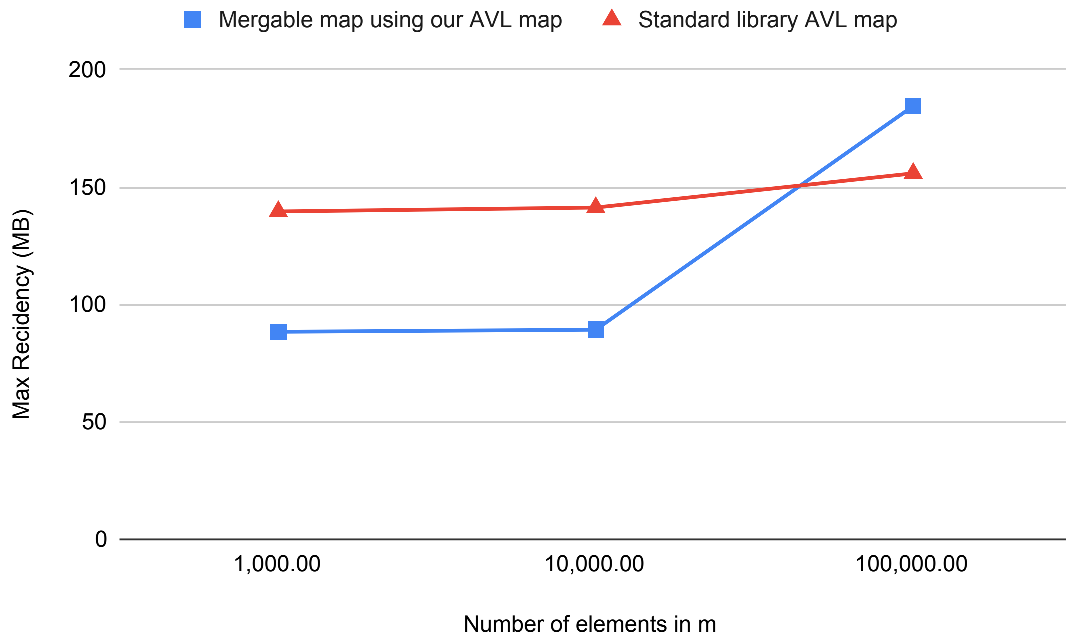


Figure 5.4: Maximum residency for for union where n is fixed with 100 000 elements.

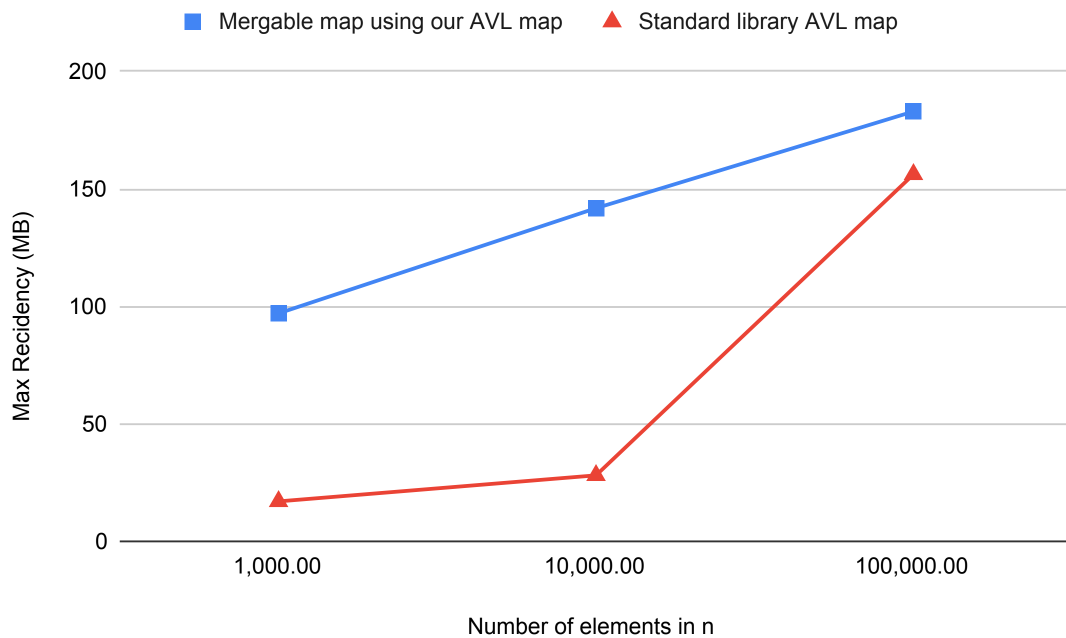


Figure 5.5: Maximum residency for for union where m is fixed with 100 000 elements.

6

Discussion

In this chapter, the extended theory, our implementation, and the results of performance and memory evaluation will be discussed. In addition, we will discuss correct-by-construction for finite maps and union properties of AVL and size-balanced trees.

6.1 Correct-by-Construction Key-Value Properties

When implementing a finite map, there are aspects that are well suited to be correct by construction as described by the introduction of this thesis. The implementation described in Chapter 4 implement these correct-by-construction aspects. An initial goal of this thesis was to explore the introduction of more correct-by-construction properties in finite map implementations. This could, for example, be to introduce key-value membership properties within the data structure of a tree. In the initial phase of research we entertained this thought but quickly realised that it is rather difficult to achieve. The primary reason for not integrating this into our implementation is two-fold:

- Proofs generated for a single tree does not necessarily hold for any other, i.e., a proof of membership for a specific element in one tree is potentially worthless upon insert, deletion, or any other operation that modifies the tree in some way.
- If correctness proofs are bundled with the collection, they **all** need to be confirmed relevant and updated on every modification that can potentially change elements in some way, as per the first point.

With that in mind, it was deemed that correct-by-construction key-value properties were not maintainable for this thesis. Membership correctness has instead been proven for most operations in our implementation.

A perhaps obvious approach that has not been explored more than theoretically by us is to keep a proof of membership in some strictly ordered data structure or accumulator. An accumulator would require a hash function producing hashes of keys that are all relatively prime so that dividing would give appropriate answers. The accumulator approach is a fun idea but not exactly feasible due to the difficulties of proving and defining the hash function. As for ordered structures, the obvious answer would be to use a set. Good set representations are, for example, tree structures because of their fast lookup, insertion, and deletion. This idea never left

the theoretical stage for good reason, as it requires maintaining two trees of the same size and with almost the same content. At that point, one might as well use the original structure to produce the proofs.

6.2 Union and Join

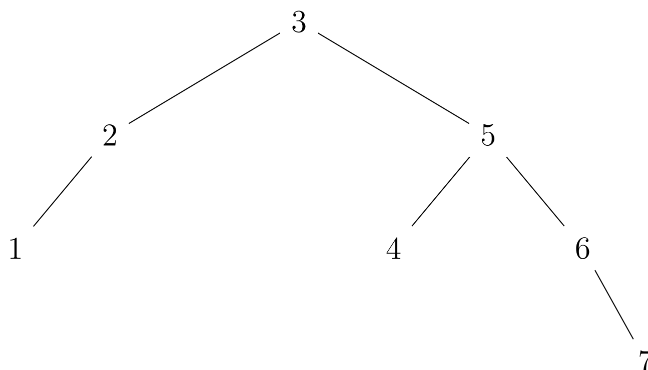
The union implementation of this thesis is lacking in one aspect, the height bound. The height bound is practically irrelevant at run-time, but it is interesting for reasoning with since it expresses a sort of “size” of AVL trees. Because of this relationship one wants to assert that the heights do not increase, or decrease, more than is possible. A more exact bound obviously expresses this more exactly. Due to the characteristics of the helper functions, it is not possible to provide a lower bound, and the upper bound is extraordinarily high at $\textsc{height}(t_1) + \textsc{height}(t_2)$!

We believe that the actual bound is lower. To reason why we first need to know the bounds of the helper functions, for gJoin we know that the bound is narrow, with $i + \textsc{max}(\textsc{height}(t_1), \textsc{height}(t_2))$ where $i \in \{0, 1\}$. splitAt, on the other hand, works by traversing a tree t to the point where the specified element, hereby the “pivot”, should be located in the tree, and returns two trees, one consisting of all elements in t less than the pivot and the other consisting of all elements in t larger than the pivot. This gives us two cases; either the pivot can exist within t, or it is outside the valid range of elements. In the outside case, both left and right, t itself will be returned, resulting in one tree with height $\textsc{height}(t)$ and the other with height 0. If the pivot exists inside t then a maximum height of $\textsc{height}(t)$ is possible for either tree. The combined minimum bound is obviously 0.

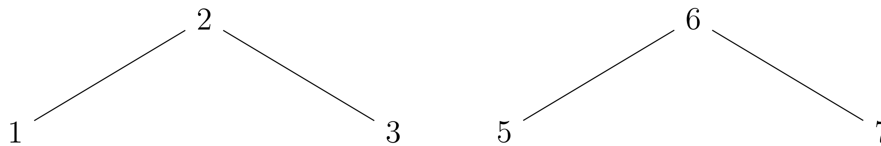
A height of interest when implementing split is the “minimum upper bound”. The minimum upper bound is defined by us to be part of the bound of the largest returned tree. This is interesting for the implementation as it gives the developer more to work with. The complete upper bound of the trees returned from split could then be defined as:

$$\textsc{height}(t) - 2 \leq \textsc{max}(\textsc{height}(t_l), \textsc{height}(t_r)) \leq \textsc{height}(t).$$

Where t_l and t_r are the output trees of splitAt. For example, splitting the following tree at 4:



The result of the split is the value associated with 4 and the two trees:



This bound is also supported by the join bound, since the largest tree input to join in splitAt is the lowest of the two children of t.

When reasoning about the bound of unionWith we state the condition that one tree is smaller than or equal to the other. Let unionWith be executed with the three arguments f, t_1 with height h_1, and t_2 with height h_2. Here we state that $h_1 \leq h_2$, i.e., t_1 is smaller than or equal to t_2.

We propose that the lower bound is $\textsc{max}(h_1, h_2)$. The reasoning for this is that since t_2 is divided at its root node, k, producing two balanced trees that are recursively unioned with the results of splitAt k t_1 it is not possible for the height to decrease. This is because any addition to the heights from t_2 and its children can not decrease in height according to the height bound of join, and as the two trees are balanced at the pivot the height will always increase to the original height of the tree. This lower bound remains in the case where $h_1 \equiv h_2$ by the same reasoning.

For the upper bound, we propose a limit of $1 + \textsc{max}(h_1, h_2)$. We have not been able to prove this, but we strongly believe that it is correct due to the properties of splitAt and join, and the fact that no counter-example has been found. Regardless of whether our proposed bound is correct or not, it is certain that the proven bound of $\textsc{max}(\textsc{height}(t_1), \textsc{height}(t_2))$ is far above what is reasonable for union of trees higher than 0 and 1.

6.2.1 The Benefits of Join

Blelloch et al. (2016) demonstrates the extreme flexibility of the join function by using it to define insert, delete, union, intersect, and difference. We provide a general join, meaning that it allows one to join any two AVL trees of varying heights. Our usage of the join function is limited to the union functionality, which means that it is used in splitAt and unionWith. However, similar to Blelloch et al. (2016), this generalised join function could be used to define the complete set of operations in further work on provably correct collections. This would greatly simplify the proof-writing process since a small number of helper functions could be used in a large number of different proofs. An example of such a function would be a function that states that if $k \mapsto v \in t_1$ then $k \mapsto v \in$ gJoin t_1 t_2. Such a function would be useful for proving a number of different insertion, deletion, and union properties.

6.2.2 Union in Size-Indexed Trees

Another approach that has not been fully explored is the possibility of using some size-indexed tree, e.g., weight-balanced (Adams, 1993), instead of AVL trees. The size of a tree is simply the number of elements, meaning $\textsc{size}(\texttt{leaf}) \equiv 0$ and

$\text{SIZE}(\texttt{node _ l r}) \equiv 1 + \text{SIZE}(\texttt{l}) + \text{SIZE}(\texttt{r})$. Weight-balanced trees are balanced by some factor or ration of the size of the tree. Trees indexed by size can be beneficial for union as the invariant for `join` and `splitAt` are more exact.

For example, a comparison of the height and size types of join, $i + \text{MAX}(h_1, h_2)$, for $i \in \{0, 1\}$, and $1 + s_2 + s_2$, shows that the height relation is less specific. It is not possible to assert the exact height at a glance, whereas the size will always increase by exactly 1 if the value added is a pivot between the two trees, which is a requirement by the function. Similarly, for splitAt the size must be $i + s \equiv s_l + s_r$, for $i \in \{0, 1\}$, given an input tree of size $1 + s$ and output trees with sizes s_l and s_r. This can be made even more exact if splitAt is divided into two functions, one for the case where the desired element exists in the map and another for the case where it does not. The unionWith bound for size-indexed trees is exactly:

$$\text{MAX}(\text{SIZE}(t_1), \text{SIZE}(t_2)) \leq \text{SIZE}(t_{out}) \leq \text{SIZE}(t_l) + \text{SIZE}(t_2).$$

The bounds for unionWith and splitAt are proven for size-indexed trees in the file `SizeTree.agda` in our GitHub repository, but join is postulated since its size invariant is obviously correct.

As shown, it is simpler to reason with splitAt, join, and unionWith using size-indexed trees, but the work needed to implement other basic operations has not been explored. One possible approach would be to convert AVL trees to weight-balanced trees when working with union, but such a conversion requires counting all elements, which could potentially make it inefficient.

Listing 6.1: A recurrence describing the minimum number of nodes in an AVL-tree (Adelson-Velskii & Landis, 1962)

```
minSize : ℕ → ℕ
minSize 0 = 0
minSize 1 = 1
minSize (suc (suc h)) = 1 + minSize h + minSize (suc h)
```

In addition to inefficiency, we see that a height-balanced tree using the AVL invariant can contain a varying number of nodes, the minimum of which can be expressed by the recurrence in Listing 6.1 and the maximum is described by $2^h - 1$.

The recurrence describes a tree that adheres to the AVL-invariant, but where the children of each tree always differ in height by exactly one. For the maximum we consider that the number of nodes in each layer doubles from the next to the other, e.g.,

$$1 + 2 + 4 + 8 + \cdots + 2^{h-1} = \sum_{i=0}^{h-1} 2^i.$$

As our implementation considers trees of size 1 to have height 1 we must start from 0 and go to $h - 1$. This geometric series can be expressed as $2^h - 1$.

The minimum and maximum number of nodes in a balanced AVL tree suggests that a weight-balanced tree may allow for trees that do not hold according to the AVL-invariant. This implies that it is only possible in some cases to easily transform a weight-balanced tree into a height-balanced tree.

6.3 Inductive Principle

Collins and Syme show that the insertion-based induction principle is provable, but we were unable to prove this for AVL trees. The main issue is the fact that we are unable to use structural induction, as the definition of the principle is structure agnostic, which forces us to induce on size of the tree. This causes a number of issues; the most significant being that we need to be able to remove some element from a map such that inserting the element into the resulting map produces the original map. This need arises from the step function with the type:

$$\forall k\ v\ m_2.\ P\ m_2 \to k \notin m_2 \to P\ (\texttt{insert}(k, v)\ m_2).$$

The step function represents the inductive step and is used when $\text{SIZE}(m_1) > 0$. In this case, we want to show that $P\ m_1$ holds, but we can only show that $P\ (\texttt{insert}(k, v)\ m_2)$ holds for some $k, v,$ and m_2.

We have found two possible ways of accomplishing this, but have not been able to implement them in Agda, as they rely on functions or proofs that we were unable to complete. The first approach uses Postulate 1, defining a new function with the purpose of finding an element that can be reinserted to produce the original tree.

Postulate 1. *Assume that there exist some function* `separate` *that given a map* m*, containing at least one element, produces* $((k\ ,\ v)\ ,\ m')$ *such that* `insert` $(k\ ,\ v)\ m' \equiv m$ *and* $\forall k\ v\ m\ m'.$ `separate` $m \equiv ((k\ ,\ v)\ ,\ m') \to k \notin m$*. The induction principle is provable if* `separate` *is definable.*

Proof. We want to show that

$$P\ \emptyset \times (\forall k\ v\ m.\ P\ m \to k \notin m \to P\ (\texttt{insert}(k\ ,\ v)\ m)) \to (\forall m \to P\ m)$$

We will perform induction on the size of the m.

Base case (size(m) = 0): $P\ m$ hold since we know that $P\ \emptyset$ holds.

Inductive step (size(m) > 0): Assume that $P\ m'$ holds, where `separate` $m \equiv ((k\ ,\ v)\ ,\ m')$. Using the step function, we get P (`insert` $(k\ ,\ v)\ m'$) which is equal to $P\ m$ according to the postulate. □

The main issue with Postulate 1 is the definition of `separate`, which needs to cleverly remove an element so that insertion reverses the removal. Some early testing of this approach seems to indicate that `separate` is definable for AVL trees, but more work is required to fully verify the viability of this approach.

The second approach uses Postulate 2, which uses insertion and deletion to achieve the same result as Postulate 1.

Postulate 2. *We are able to prove the induction principle by assuming that* $\forall\ k\ v\ m.\ k \mapsto v \in m \rightarrow$ `insert` $(k\ ,\ v)$ (`delete` $k\ m) \equiv m$.

Proof. We want to show that

$$P\ \emptyset \times (\forall k\ v\ m.\ P\ m \rightarrow k \notin m \rightarrow P\ (\texttt{insert}\ (k\ ,\ v)\ m)) \rightarrow (\forall m \rightarrow P\ m)$$

We will perform induction on the size of the m.
Base case (size(m) = 0): $P\ m$ holds since we know that $P\ \emptyset$ holds.

Inductive step (size(m) > 0): Assume that P (`delete` $k\ m$), where (k, v) is the pivot element in m, holds. Using Law 3.19, we can show that $k \notin$ `delete` $k\ m$. Using the given step function we get P (`insert` (k, v) (`delete` $k\ m$)), which is equal to $P\ m$ according to postulate 2. ☐

The second approach is more appealing as it does not define any new function but instead uses two existing functions that have been extensively verified. The main issue lies in proving the equivalence between the original map and the modified one. One might be able to simplify the proof by specifying it for when p is deleted from `node` $p\ lm\ rm\ bal$. This would simplify the proof, since we would not have to search the entire tree structure for the key-value pair that should be removed and reinserted.

6.3.1 Equality and Induction

This section has thus far used _≡_ when reasoning about the equality of maps, which is not exactly ideal as its definition of equality requires the trees to have identical structures. The focus on _≡_ stems from the fact that the current formulation of the induction principle requires us to rewrite insert $k\ v\ m'$ into m in the final step of the proof.

Another possible approach is to use the _≐_ relation, discussed in Chapter 3, when reasoning about the equality of the trees. This approach would be beneficial since we would no longer have to consider the structure of the trees when establishing that they are equal, which would greatly simplify the implementation. This would be rather simple to implement as all we would have to do is to require that the predicate, P, respects the relation _≐_. This would essentially provide us with a function of the type:

$$\forall\ m\ m' \rightarrow m \doteq m' \rightarrow P\ m \rightarrow P\ m'.$$

Such a function could be used to "rewrite" the result of the step function into the desired type at the end of the computation, given that they fulfil extensional equality as per _≐_. This approach could allow us to use existing insertion and deletion proofs when implementing Postulate 2.

However, there are still issues with this approach. First, it would limit the number of predicates that can be used in the induction principle as all of them will not necessarily respect _≐_. This would also place a greater burden on the developer as they would have to prove that their predicate respects _≐_. Second, and more

importantly, the height parameterisation of the current AVL map implementation makes it rather difficult to prove that a combination of operations, e.g., insert and delete, is equal to the original map even when using _≐_. This stems from the fact that we cannot predict whether or not an object will change the height of the map. This seems to indicate that height parameterisation might not be the best approach for this kind of functions and that size parameterisation could possible simplify or outright solve these problems.

6.4 Size and Height Parameterised AVL-Trees

A reoccurring issue for both the induction principle and union is the fact that the height of a map is not enough to reason about the number of elements. This naturally raises the question: why not parameterise on the number of elements instead of the height of the map? This modification could simplify the implementation of the induction principle. Additionally, this would also simplify the union proofs, as discussed in Section 6.2.2. However, there are a couple of drawbacks to this approach.

Firstly, parameterising on both the height and number of elements would not make much sense since we would be conveying the same information twice. We would therefore be required to express the balancing of the trees in terms of their size. This could make the balancing proofs more complicated, as well as any of the operations that modify the balancing of the map, such as insertion, deletion, and union. This would additionally complicate the correctness proofs for these operations, since there would be a greater number of possible cases when rotating the trees, or more precisely there would be the same number of possible rotations, but a greater number of cases that result in these rotations.

Secondly, if we instead parameterise on both the height and the size, then we might have to introduce a connection between the size and height. More specifically, Agda might not be able to understand the connection between height and size since the height does not increase at the same rate as the size. This might cause problems when proving certain properties about the map operations or when defining the operations themselves.

6.5 Performance Evaluation

The performance results presented in Section 5.3 show that our implementation has a consistent overhead compared to the standard library implementation. The insertion results in Figure 5.1 clearly show this, as our insertion function perfectly mirrors the standard library, with the exception that we erase the ordering proof updates. One would therefore assume that our insertion function should be at least as fast if not faster than the standard library version, since we should be performing less work in each insertion because of the erasure. This seems to indicate that there is something intrinsic in our implementation that is slowing us down. One possible explanation could be that the instance argument ordering proofs are somehow slowing down the

execution time; this would be very strange as these proofs are erased and should therefore have no impact on the performance. Another reason could be the added abstraction, recall that any call to a tree operation must be performed through the abstraction.

The union results in Figure 5.2 and in Table 5.1 are more reasonable, since our union implementation, specifically splitAt, has to produce a large number of different proofs to be able to prove the correctness of the height of the resulting tree. It is also clearly visible in the figures that our unionWith is more optimal for some specific inputs, namely when the first input tree is smaller than the second input tree. Possible optimisations for this in production can be to first compare the height of the trees and swap them accordingly when calling unionWith. This optimisation could also be performed for the standard library.

6.5.1 Effects of Erasure on Performance

The `@erasure` and `@0` notation in Agda signify run-time irrelevance, meaning that anything marked with that will be ignored during compilation. In our library, we extensively denote the code with this run-time irrelevance notation. We have not noticed any significant performance gains from the introduction of erasure. It is not possible for us to tell whether the consistent overhead of our implementation negates any potential gain from run-time irrelevance. It could be interesting to compare our code without erasure but as this requires manually deleting all erasure notations from the code base we decided against it. In addition to the extra labour late in the development process the results of the performance comparison show that ours performs very similarly to the standard library that does not use erasure.

Agda2hs (Cockx et al., 2022) is an alternative to MAlonzo that aims at producing readable Haskell code. The Agda2hs program is lacking some features necessary for our code, specifically type-level lambdas and module parameterisation. Worth noting is that these features are not possible to implement in Haskell. At the time of performance evaluation, it was deemed out of scope to modify the code base to be able to compile to Haskell using Agda2hs.

6.5.2 Effects of Erasure on Memory

The memory profiling performed using GHC's run-time options shows no correlation between memory usage and erasure. Which is strange as one would expect there to be a noticeable difference. We believe that this could be caused by Haskell's call-by-need evaluation strategy, since Haskell might not need to evaluate any of the values that are run-time irrelevant in our map but exist in the standard library. If that is the case then the erasure notation is perhaps unnecessary in most of our cases. The most useful aspect of the notation would be if there are very inefficient proofs, which our implementation currently lacks for execution of the operations.

The memory usage results combined with the performance results would indicate that erasure might not be beneficial in a call-by-need language, at least in terms of performance or memory usage.

6.5.3 On Correctness and Performance

As is explained and shown the performance of our map implementation is very similar to the standard library when using the same algorithms. This leads us to the question: are there other ways that one can improve the performance, and at what cost of correctness? We believe that there are three main paths one can take to try to answer this:

1. Simplify the implementation as much as possible such that there are no properties that need to be maintained in the implementation itself. The correctness would in this case need to be proven completely outside of the implementation, much alike the laws described and proven by us.

2. Use an implementation that leverages correct-by-construction such that properties are correct by design. The difficulty with this case is that correct-by-construction may require in-line proofs in the implementation, which can slow it down. In addition to proofs in the implementation, one would also need to make sure that the types are correct, which may be difficult, as shown for union in Section 6.2.

3. Have two implementations, one that is easy to reason with, for example point two in this list, and another that is very efficient, for example point one. Given that one implementation is easier to reason with one could prove that and then show an equivalence between the two implementations, and mark the slow as run-time irrelevant.

As one could perhaps guess, we have been striving towards an implementation like point two above. There are still improvements that could be made, for example, our implementation shows a consistent overhead on insertion, and performing union is in general slower despite theoretically better complexity.

The question remains whether to prioritise correctness or performance. There is no clear answer to this question unfortunately, as it depends on the purpose of the program. Our implementation was done purely of theoretical interest, beginning with exploring how reasonable it was to use correct-by-construction key-value membership properties and ending with attempting to improve the standard library. We are generally satisfied with the performance and memory usage of our implementation, despite hoping for better results given improved union implementation and the amount of code marked as run-time irrelevant.

Real world scenarios of course require different priorities, for example, security concerns or cost analysis of taking the extra time to formally verify your software instead of developing “good enough” software, that works but is not necessarily verified.

7

Conclusion

The work presented in this thesis is guided by the following questions, in addition to providing a provably correct and usable collection for Agda:

1. Which commonly used functions are lacking proofs and is it possible to add and prove these?
2. How much of an underlying map implementation can be abstracted while retaining the usability and flexibility of the map?
3. Can the implementation be written such that parts of the code used only for verification are erased during compilation?

Initially, it was believed that most, if not all, of the functions available for collections in Agda lacked proofs. This was not the case. However, the laws provided in this thesis extend the existing proofs and add additional laws for deletion and union that previously had none. The majority of the laws are proven, the only exception being the inductive principle and some proofs for union. These exceptions stem from the AVL implementation since reasoning with the height does not express the correct properties for the inductive principle, and similarly, for union, the bounds of the height were not deemed satisfactory, so time was spent on finding, and proving new bounds.

The answer to the second question is that a lot of underlying details can be abstracted. This of course comes with consequences too; to allow for the greatest flexibility of implementation, the inductive principle was defined using the insert operation, which led to us not being able to prove it correct for our implementation. But in general, the abstraction is flexible and usable, as seen in the simply typed lambda calculus interpreter example in Section 5.2.

Finally, we believe that it is possible to erase a lot of information from the structure, but this was not shown to improve the performance of our implementation. By analysis of the generated Haskell code, it is unclear what impact run-time irrelevance has on our code. Due to constraints stemming from parameterised modules and type-level lambda abstraction, we were not able to use a different program (Cockx et al., 2022), that is known to completely remove parts marked as run-time irrelevant, to generate readable Haskell.

To conclude, we have successfully implemented a collection in Agda that is provably correct, and fully proven for all operations except `unionWith`.

7.1 Future Work

Future work is presented in Chapter 6, and is briefly summarised here. As we only add proofs for two new operations, there is more work available for map operations, namely implementations suitable for proving and reasoning with as well as providing laws that express appropriate behaviour.

The height bound of union in AVL trees is something we have attempted to define, but this has been unsuccessful. We believe that the bound is tighter than suggested by Blelloch et al., but we were unable to show it. The believed upper bound is $1 + \textsc{max}(\textsc{height}(t_1), \textsc{height}(t_2))$ for the input trees t_1 and t_2, and while we were unable to prove it, we have not been able to find counterexamples. We hope that future work will either prove or disprove our claim and provide an appropriate bound.

As described in Chapter 6, weight-balanced trees make it easier to reason with union properties. An implementation using weight-balanced trees that are proven correct is something we have not explored but is a reasonable alternative considering the struggles of union described in this report. An implementation that uses size for reasoning could additionally be beneficial for proving the inductive principle.

Correct-by-construction aspects of collections can also be further explored, for example, by bundling membership proofs with operations that insert or delete elements. Our implementation contains proofs that operate similarly, but are not as tightly integrated as it could be. Other areas of future research could be how the proofs are constructed, e.g., other set representations. Future work on proof construction and usage can show what is efficient, most maintainable, or most appropriate for reasoning.

The use of Agda2hs (Cockx et al., 2022) instead of the MAlonzo compiler (Benke, 2007) is something we would have liked to have done, but the implementation is currently dependent on features not supported by Agda2hs. Future developers could accommodate the implementation or write their own, to test the impact of run-time irrelevance and to obtain Haskell code that is easier to analyse.

Bibliography

Adams, S. (1993). Efficient sets - A balancing act. *J. Funct. Program.*, *3*(4), 553–561. https://doi.org/10.1017/S0956796800000885

Adelson-Velskii, G. M., & Landis, E. M. (1962). An algorithm for the organization of information. *Soviet Mathematics Doklady*, *3*(1259-1262), 4.

Agda Developers. (2024). *Agda* (Version 2.6.4.3). https://agda.readthedocs.io/

Ben-Ari, M. (2001). The bug that destroyed a rocket. *ACM SIGCSE Bulletin*, *33*(2), 58–59.

Benke, M. (2007). Alonzo—a compiler for agda. *Talk at Agda Implementors Meeting*, *6*(4).

Blelloch, G. E., Ferizovic, D., & Sun, Y. (2016). Just join for parallel ordered sets. In C. Scheideler & S. Gilbert (Eds.), *Proceedings of the 28th ACM symposium on parallelism in algorithms and architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016* (pp. 253–264). ACM. https://doi.org/10.1145/2935764.2935768

Cockx, J., Melkonian, O., Escot, L., Chapman, J., & Norell, U. (2022). Reasonable Agda is correct Haskell: Writing verified Haskell using agda2hs. In N. Polikarpova (Ed.), *Haskell '22: 15th ACM SIGPLAN international Haskell symposium, Ljubljana, Slovenia, September 15 - 16, 2022* (pp. 108–122). ACM. https://doi.org/10.1145/3546189.3549920

Collins, G., & Syme, D. (1995). A theory of finite maps. In E. T. Schubert, P. J. Windley & J. Alves-Foss (Eds.), *Higher order logic theorem proving and its applications, 8th international workshop, Aspen Grove, UT, USA, September 11-14, 1995, proceedings* (pp. 122–137, Vol. 971). Springer. https://doi.org/10.1007/3-540-60275-5_61

Coquand, T., & Huet, G. P. (1988). The calculus of constructions. *Inf. Comput.*, *76*(2/3), 95–120. https://doi.org/10.1016/0890-5401(88)90005-3

Howard, W. A., et al. (1980). The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, *44*, 479–490.

Jansson, P., & Jeuring, J. (1997). Polyp - A polytypic programming language. In P. Lee, F. Henglein & N. D. Jones (Eds.),

Conference record of popl'97: The 24th ACM SIGPLAN-SIGACT symposium on principles of programming languages, papers presented at the symposium, Paris, France, 15-17 January 1997 (pp. 470–482). ACM Press. https://doi.org/10.1145/263699.263763

Knuth, D. E. (1998). *The art of computer programming, Volume III, 2nd Edition.* Addison-Wesley. https://www.worldcat.org/oclc/312994415

McBride, C. T. (2014). How to keep your neighbours in order. In J. Jeuring & M. M. T. Chakravarty (Eds.), *Proceedings of the 19th ACM SIGPLAN international conference on functional programming, Gothenburg, Sweden, September 1-3, 2014* (pp. 297–309). ACM. https://doi.org/10.1145/2628136.2628163

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, *3*(4), 184–195. https://doi.org/10.1145/367177.367199

Melkonian, O. (2024). *Formal prelude*. GitHub. https://github.com/omelkonian/formal-prelude

Nipkow, T., Paulson, L. C., & Wenzel, M. (2002). *Isabelle/hol - A proof assistant for higher-order logic* (Vol. 2283). Springer. https://doi.org/10.1007/3-540-45949-9

O'Sullivan, B. (2024). *Criterion: A Haskell microbenchmarking library* (Version 0.2.2.0). Retrieved 2024, from http://www.serpentine.com/criterion/

Pierce, B. C. (2002). Going further: Dependent types. In *Types and programming languages* (pp. 462–466). MIT Press.

The Agda Community. (2023, December). *Agda Standard Library* (Version 2.0). https://github.com/agda/agda-stdlib

The Coq Team. (1989). *The Coq Proof Assistant*. Retrieved 2023-12-01, from https://coq.inria.fr

Wong, W. E., Debroy, V., & Restrepo, A. (2009). The role of software in recent catastrophic accidents. *IEEE reliability society 2009 annual technology report*, *59*(3).

A

Definitions of Abstractions, an Ordering Function, and Some Function Implementations

This appendix contains the definitions of the following abstractions and functions: records BMap, DMap, and MMap, functions mklim, insertWith, insert, delete, and lookup.

A.1 Definition of the BMap Record

```
module _ {ℓ₁ : Level} {K : Set ℓ} {V : Set ℓ'} where
  record BMap (Map : Set (ℓ ⊔ ℓ' ⊔ ℓ₁)) : Set (lsuc (ℓ ⊔ ℓ' ⊔ ℓ₁)) where
    constructor mkMap
    field
      ∅ : Map
      singleton : K → V → Map
      _∈_ : K → Map → Set (ℓ ⊔ ℓ' ⊔ ℓ₁)
      _↦_∈_ : K → V → Map → Set (ℓ ⊔ ℓ' ⊔ ℓ₁)
      lookup : Map → K → Maybe V
      lookup∈ : ∀ {k m} → k ∈ m → V
      insertWith : K → (Maybe V → V) → Map → Map


insert : K → V → Map → Map
insert k v m = insertWith k (λ _ → v) m

_↦_∉_ : K → V → Map → Set (ℓ ⊔ ℓ' ⊔ ℓ₁)
k ↦ v ∉ m = ¬ (k ↦ v ∈ m)

_∉_ : K → Map → Set (ℓ ⊔ ℓ' ⊔ ℓ₁)
k ∉ m = ¬ (k ∈ m)

_⊆_ : Map → Map → Set (ℓ ⊔ ℓ' ⊔ ℓ₁)
n ⊆ m = ∀ k v → k ↦ v ∈ n → k ↦ v ∈ m
```

```
    _≐_ : Map → Map → Set (ℓ ⊔ ℓ' ⊔ ℓ₁)
    n ≐ m = (n ⊆ m) × (m ⊆ n)

    field
      refl⊆ : Reflexive _⊆_
      trans⊆ : Transitive _⊆_
      refl≐ : Reflexive _≐_
      sym≐ : Symmetric _≐_
      trans≐ : Transitive _≐_


↦∈To∈ : ∀ {k v m} → k ↦ v ∈ m → k ∈ m
∈To∈↦ : ∀ {k m} → k ∈ m → ∃ (λ v → k ↦ v ∈ m)

∈↦-∅ : ∀ k v → k ↦ v ∉ ∅
∈-∅ : ∀ k → k ∉ ∅

∈-singleton : ∀ k k' v v' → k ↦ v ∈ singleton k' v' → k ≡ k' × v ≡ v'

ips : (P : Map → Set (ℓ ⊔ ℓ'))
    → P ∅ × (∀ m → P m → ∀ k v → k ∉ m
                  → P (insertWith k (λ _ → v) m))
    → (∀ m → P m)

-------------------------------------------------------------------------------
-- Insertion and lookup properties
-------------------------------------------------------------------------------
lookup-∅ : ∀ k → lookup ∅ k ≡ nothing

∈⟹lookup : ∀ m k {v} → lookup m k ≡ just v → k ↦ v ∈ m
lookup⟹∈ : ∀ m k v → k ↦ v ∈ m → lookup m k ≡ just v

∉⟹nothing : ∀ m k → k ∉ m → lookup m k ≡ nothing
nothing⟹∉ : ∀ m k → lookup m k ≡ nothing → k ∉ m

lookup≡lookup∈ : ∀ k m → (k∈m : k ∈ m) → just (lookup∈ k∈m) ≡ lookup m k

mapsTo : ∀ {m k} → (k∈m : k ∈ m) → k ↦ lookup∈ k∈m ∈ m

lookup-insert : ∀ k m f
                → lookup (insertWith k f m) k ≡ just (f (lookup m k))

ins-comm : ∀ k k' v v' m
           → k ≢ k'
           → insert k v (insert k' v' m)
              ≐ insert k' v' (insert k v m)

∈-ins : ∀ m k x v f
        → x ↦ v ∈ (insertWith k f m)
        → (x ≡ k) ⊎ x ↦ v ∈ m
```

```
insert∈ : ∀ k v m → k ↦ v ∈ (insert k v m)

∈insert : ∀ k {v} {v'} m → k ↦ v ∈ (insert k v' m) → v ≡ v'

insert-safe : ∀ {k k' v v' m}
  → k ↦ v ∈ m → k ≢ k' → k ↦ v ∈ (insert k' v' m)

ip : (P : Map → Set (ℓ ⊔ ℓ'))
       → P ∅ × (∀ m → P m → ∀ k v → P (insertWith k (λ _ → v) m))
       → (∀ m → P m)
ip P (b , s) mp = ips P (b , λ m x k v _ → s m x k v ) mp
```

A.2 Definition of the DMap Record

```
record DMap (Map : Set (ℓ ⊔ ℓ' ⊔ ℓ₁)) : Set (lsuc (ℓ ⊔ ℓ' ⊔ ℓ₁)) where
  constructor mkDMap

  field
    bMap : BMap Map
    delete : K → Map → Map


------------------------------------------------------------------------------
-- Deletion properties
------------------------------------------------------------------------------
del-∉ : ∀ {k m} → k ∉ m → delete k m ≐ m
del-∈ : ∀ {k m} → k ∉ delete k m
del-safe : ∀ {k k' v m} → k' ↦ v ∈ m → k ≢ k' → k' ↦ v ∈ delete k m
del-noAdd : ∀ {k k' v m} → k ↦ v ∈ delete k' m → k ↦ v ∈ m
del-removeK : ∀ {k v m} → k ↦ v ∉ delete k m
del-comm : ∀ k k' m → delete k (delete k' m) ≐ delete k' (delete k m)
```

A.3 Definition of the MMap record

```
record MMap (Map : Set (ℓ ⊔ ℓ' ⊔ ℓ₁)) : Set (lsuc (ℓ ⊔ ℓ' ⊔ ℓ₁)) where
  constructor mkMMap

  field
    bMap : BMap Map
    unionWith : (V → Maybe V → V) → Map → Map → Map


------------------------------------------------------------------------------
  -- Union Properties
```

```
---------------------------------------------------------------------------
∪-∅ᴸ : ∀ m f → unionWith f m ∅ ≐ m
∪-∅ᴿ : ∀ m f → unionWith f ∅ m ≐ m
∪-∅ : ∀ m f → unionWith f m ∅ ≐ unionWith f ∅ m
∪-∈ : ∀ m1 m2 f k
          → k ∈ unionWith f m1 m2
          → k ∈ m1 ⊎ k ∈ m2 ⊎ (k ∈ m1 × k ∈ m2)
∪-∈' : ∀ m1 m2 f k
          → k ∈ m1 ⊎ k ∈ m2
          → k ∈ unionWith f m1 m2
∪-safe : ∀ k v₁ v₂ m₁ m₂ f
   → k ↦ v₁ ∈ m₁
   → k ↦ v₂ ∈ m₂
   → k ↦ f v₁ (just v₂) ∈ unionWith f m₁ m₂
∪-safe-left : ∀ k v m₁ m₂ f
   → k ↦ v ∈ m₁
   → k ∉ m₂
   → k ↦ f v nothing ∈ unionWith f m₁ m₂
∪-safe-right : ∀ k v m₁ m₂ f
   → k ∉ m₁
   → k ↦ v ∈ m₂
   → k ↦ v ∈ unionWith f m₁ m₂
```

A.4 The Implementation of the mklim Function

```
@erased mklim : ∀ {l u h}
          → AVLMapIndexed V l u h
          → l <⁺ u
mklim (leaf {{p}}) = p
mklim {l} {u} (node p lt rt bal) = trans⁺ l (mklim lt) (mklim rt)
```

A.5 Implementation of insertWith, insert, delete, and lookup

The left and right join functions join a key-vale pair with two sub-trees, performing a single or double rotation if necessary.

```
insertWith : {@0 l u : Key⁺} {@0 h : ℕ} (k : Key) (f : Maybe V → V)
             {{@erased l≤u : l <⁺ [ k ]}} {{@erased p≤u : [ k ] <⁺ u}}
             → AVLMapIndexed V l u h
             → ∃ λ i → AVLMapIndexed V l u (i ⊕ h)
insertWith k f leaf = 1# , node (k , f nothing) leaf leaf ~0
insertWith k f (node p lt rt bal) with compare k (proj₁ p)
```

```
... | tri< k<p _ _ = joinˡ⁺ p (insertWith k f {{p≤u = [ k<p ]ᴿ}} lt) rt bal
... | tri≈ _ refl _ = 0# , node (k , f (just (proj₂ p))) lt rt bal
... | tri> _ _ p<k = joinʳ⁺ p lt (insertWith k f {{[ p<k ]ᴿ}} rt) bal

insert : {@0 l u : Key⁺} {@0 h : ℕ} (kv : Key × V)
         {{@erased l≤p : l <⁺ [ (proj₁ kv) ]}}
         {{@erased p≤u : [ (proj₁ kv) ] <⁺ u}}
         → AVLMapIndexed V l u h
         → ∃ λ i → AVLMapIndexed V l u (i ⊕ h)
insert (k , v) m = insertWith k (λ _ → v) m


delete : ∀ {@0 l u : Key⁺} {@0 h : ℕ} (k : Key)
         {{@erased l≤p : l <⁺ [ k ]}} {{@erased p≤u : [ k ] <⁺ u}}
         → AVLMapIndexed V l u h
         → ∃ λ i → AVLMapIndexed V l u pred[ i ⊕ h ]
delete k leaf = 0# , leaf
delete k (node p lt rt bal) with compare k (proj₁ p)
... | tri< k<p _ _ = joinˡ⁻ p (delete k {{p≤u = [ k<p ]ᴿ}} lt) rt bal
... | tri≈ _ refl _ = join lt rt bal
... | tri> _ _ p<k = joinʳ⁻ p lt (delete k {{[ p<k ]ᴿ}} rt) bal


lookup : ∀ {@0 l u : Key⁺} {@0 h : ℕ}
         → AVLMapIndexed V l u h
         → Key
         → Maybe V
lookup leaf k = nothing
lookup (node p lt rt bal) k with compare k (proj₁ p)
... | tri< k<p _ _ = lookup lt k
... | tri≈ _ refl _ = just (proj₂ p)
... | tri> _ _ p<k = lookup rt k
```

B

Simply Typed Lambda Calculus Interpreter

B.1 Definitions and Typing Rules

$$
\begin{aligned}
\tau &::= \mathbb{N} \mid () \mid \tau \rightarrow \tau \\
e &::= n \mid () \mid x \mid e_1 + e_2 \mid \lambda x.e \mid e_1\ e_2 \\
v &::= n \mid () \mid \lambda x.e
\end{aligned}
\tag{B.1}
$$

Figure B.1: Types, Expressions, and Values for the simply typed lambda calculus.

$$
\begin{array}{ll}
\text{Nat}\dfrac{}{\Gamma \vdash n : \mathbb{N}} & \text{Unit}\dfrac{}{\Gamma \vdash () : ()} \\[2em]
\text{Var}\dfrac{x \mapsto \tau \in \Gamma}{\Gamma \vdash x : \tau} & \text{Add}\dfrac{\begin{array}{c}\Gamma \vdash e_1 : \mathbb{N} \\ \Gamma \vdash e_2 : \mathbb{N}\end{array}}{\Gamma \vdash e_1 + e_2 : \mathbb{N}} \\[2em]
\text{Abs}\dfrac{(x, \tau_1) :: \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2} & \text{App}\dfrac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \\ \Gamma \vdash e_2 : \tau_1\end{array}}{\Gamma \vdash e_1 e_2 : \tau_2}
\end{array}
\tag{B.2}
$$

Figure B.2: Typing rules for the simply typed lambda calculus.

B.2 Agda Implementation

```
data Unit : Set where
  unit : Unit

Var : Set
Var = ℕ

data Type : Set where
  nat : Type
  unit : Type
  _=>_ : Type → Type → Type

[[_]] : Type → Set
[[ nat ]] = ℕ
[[ unit ]] = Unit
[[ τ => τ' ]] = [[ τ ]] → [[ τ' ]]

private
  variable
    Ctx : Set

module Interpreter (bMap : BMap {K = ℕ} {Type} Ctx)
    (cAll : CorrectAll {ℓa = 0ℓ} {ℕ} {Type} Ctx bMap) where
  open Map.BasicMap.BMap bMap public
  open Map.CorrectAll.CorrectAll cAll public

  Env : Ctx → Set
  Env c = All (λ (_ , τ) → [[ τ ]]) c

  data _⊢_ : Ctx → Type → Set where
    T-Int  : ∀ {Γ : Ctx}
             → ℕ
           -------------
             → Γ ⊢ nat

    T-Add  : ∀ {Γ : Ctx}
             → Γ ⊢ nat
             → Γ ⊢ nat
           -------------
             → Γ ⊢ nat

    T-Unit : ∀ {Γ : Ctx}
           --------------
             → Γ ⊢ unit

    T-Var  : ∀ {Γ : Ctx} {τ : Type} {x : Var}
             → x ↦ τ ∈ Γ
```

```
                  ---------------------------
                    → Γ ⊢ τ

        T-Abs  : ∀ {Γ : Ctx} {x : Var} {τ₁ τ₂ : Type}
                    → (insert x τ₁ Γ) ⊢ τ₂
                  --------------------------------------
                    → Γ ⊢ (τ₁ => τ₂)

        T-App  : ∀ {Γ : Ctx} {τ₁ τ₂ : Type}
                    → Γ ⊢ (τ₁ => τ₂)
                    → Γ ⊢ τ₁
                  ---------------------------
                    → Γ ⊢ τ₂

translate : ∀ {Γ : Ctx} {τ : Type} → Env Γ → Γ ⊢ τ → [[ τ ]]
translate _ (T-Int n) = n
translate env (T-Add e₁ e₂) = translate env e₁ + translate env e₂
translate env T-Unit = unit
translate env (T-Var {x = x'} prf) = allLookup {k = x'} prf env
translate env (T-Abs {x = x} e) e' = translate (allInsert {k = x} e' env) e
translate env (T-App e₁ e₂) = translate env e₁ (translate env e₂)
```

C

Evaluation Figures

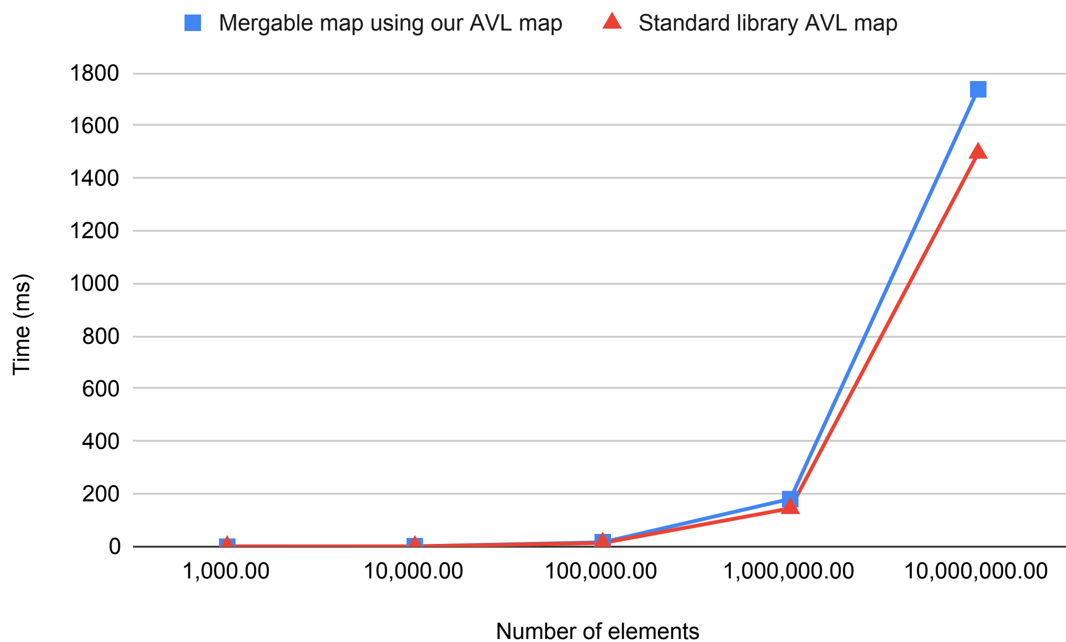
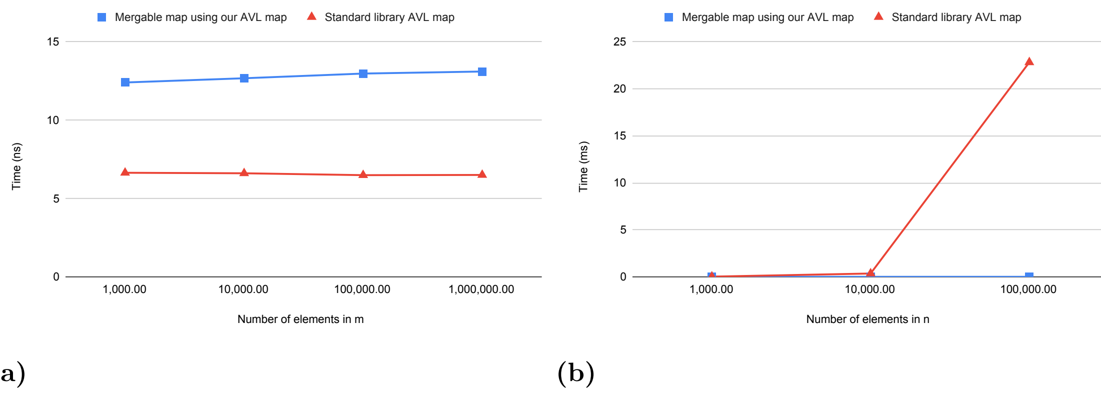


Figure C.1: Elapsed time for inserting values in the range $0..n$ into an empty map.



(a) **(b)**

Figure C.2: Times for merging an empty map with a map containg 1000, 10,000, or 100,000 elements using union.

D

Memory Profiling Results

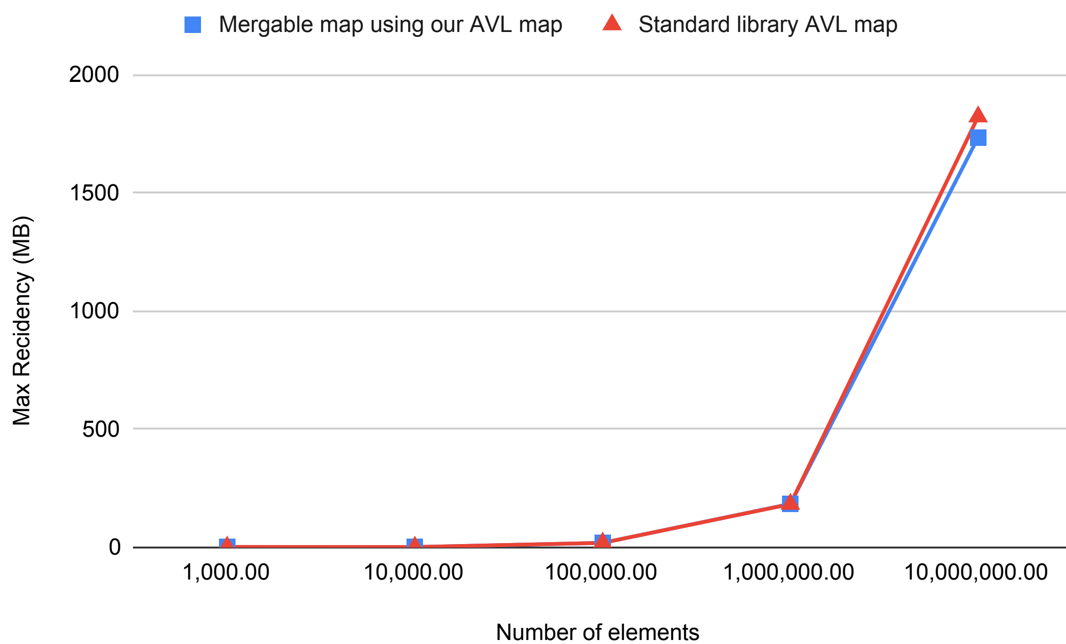


Figure D.1: Maximum residency for inserting values in the range $0..n$ into an empty map.

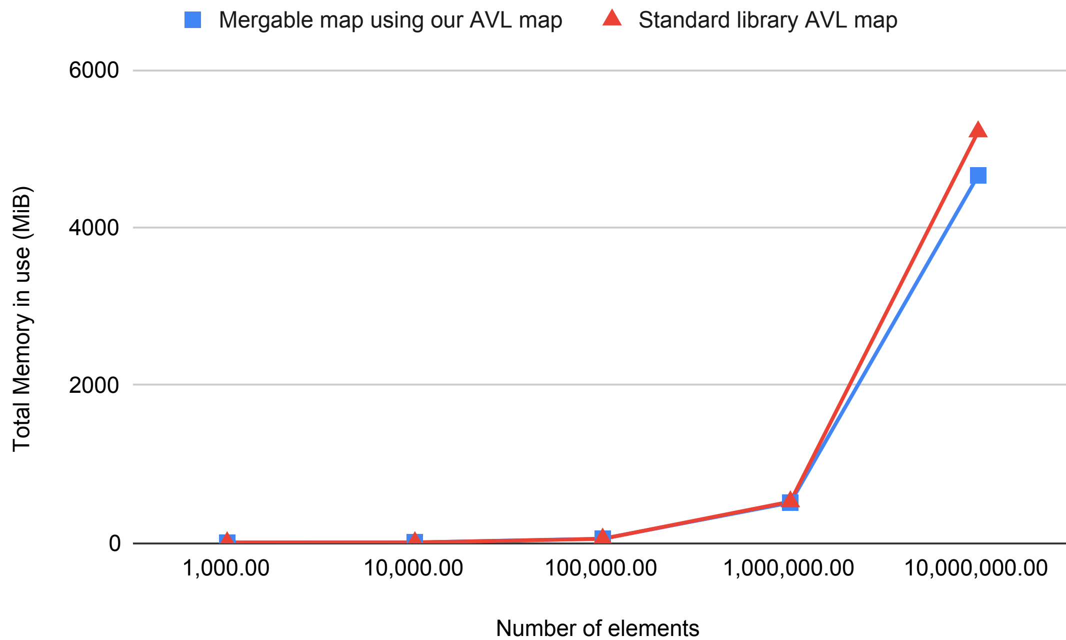


Figure D.2: Total memory usage for inserting values in the range $0..n$ into an empty map.

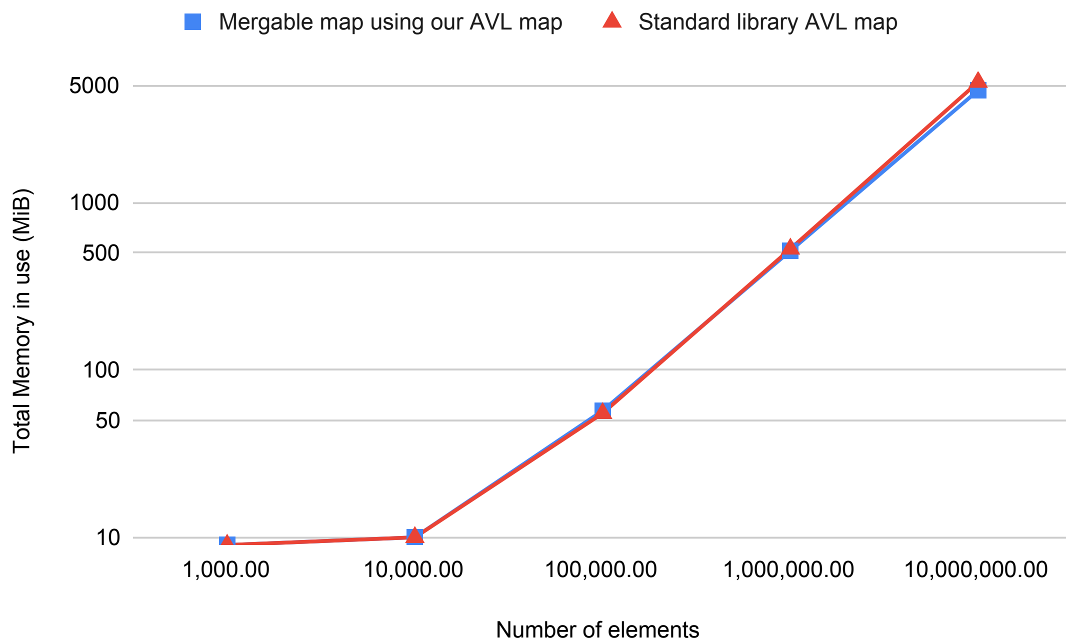


Figure D.3: Total memory usage for inserting values in the range $0..n$ into an empty map, with logarithmic x and y axis.

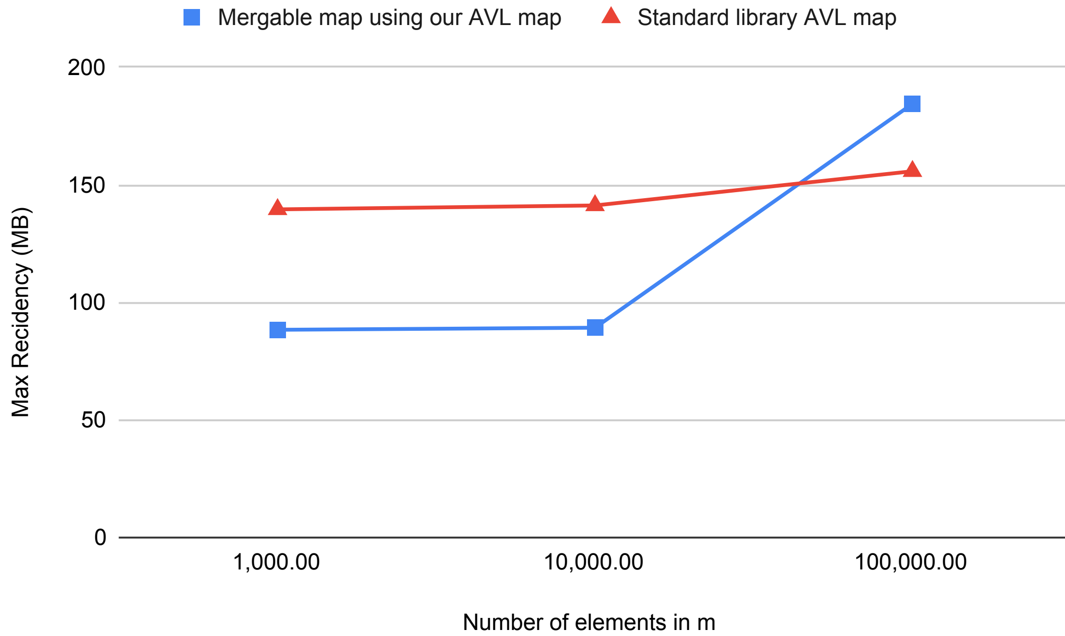


Figure D.4: Total memory usage for for union where n is fixed with 100 000 elements.

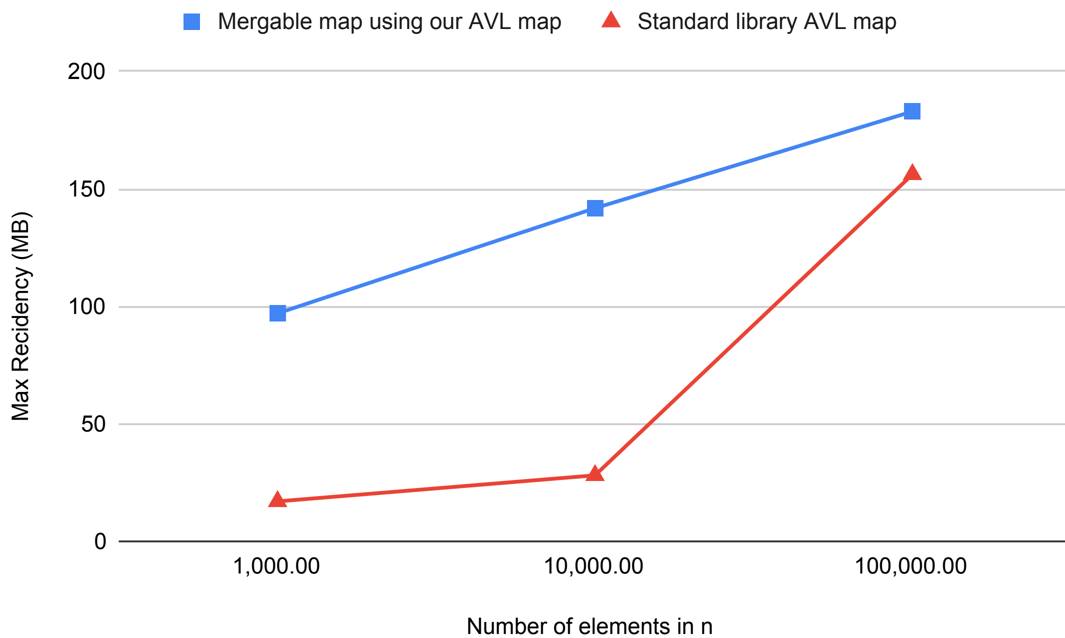


Figure D.5: Total memory usage for for union where m is fixed with 100 000 elements.