# Practical Verification of the Haskell Ranged-sets Library

**Ioana Savu**[1] , **Jesper Cockx**[1] , **Lucas Escot**[1]
[1]TU Delft

## Abstract

*agda2hs* is a project that aims to combine the best parts of Haskell and Agda by providing a common subset between them. It allows programmers to implement libraries in Agda, verify their correctness and then translate the result to Haskell so they can be used by Haskell programmers. In this paper, a verified Agda implementation of the Ranged-sets Haskell library is provided, using *agda2hs*. In order to produce a verified implementation of this library, we proved its preconditions, invariants and properties.

## 1 Introduction

Haskell [1] is a purely functional and strongly typed programming language. One of the big advantages of purity is that it makes it very easy to reason about the correctness of algorithms and data structures. An example of this can be found in Chapter 16 of the book Programming in Haskell (2nd edition) by Graham Hutton [2], where he uses equational reasoning, case analysis, and induction to prove properties of Haskell functions. However, since these proofs are done 'on paper', there is always a risk that the proof contains a mistake, or that the code changes to a new version but the proof is not updated.

Agda [3] is a total language that has a more expressive typing system: it fully supports dependent types. These properties come with a great advantage: one can write a formal proof of correctness in the language itself. Types can be used to express properties and therefore proofs of these properties, this isomorphism being known as the Curry-Howard correspondence [4]. These proofs are checked automatically by the type-checker, and re-checked every time the code is changed. Therefore, it provides an unusually high degree of confidence in the correctness of the algorithm. Unfortunately, despite several big successes in dependently typed programming such as the CompCert verified C compiler [5], these languages remain hard to use. Therefore, these guarantees are currently only available to expert users. [1]

Software testing can ensure reliability, high performance and even security of a product. However, it has been often disregarded by programmers with experience in formal verification since it can only show the presence of errors, not the absence of them, as Dijkstra pointed out [6]. Formal verification comes with two great advantages. Firstly, we no longer rely on generating test cases for the test suites, also known as exhaustive testing, where a property can falsely hold because there was no generated input that would make it fail. In Agda, one can actually prove mathematically the property, ensuring that it always holds. The second advantage is that type-checking Agda proofs is much faster than running multiple test suites. As an example, when running a QuickCheck [7] property, not only that it needs to generate the input, but it runs the functions that need to be tested against all generated values.

The *agda2hs* project [8] is a recent effort to combine the best parts of Haskell and Agda. In particular, it identifies a common subset of the two languages, and provides a faithful translation of this subset from Agda to Haskell. This allows library developers to implement libraries in Agda and verify their correctness using formal verification, and then translate the result to Haskell so it can be used and understood by Haskell programmers.

This project investigates whether a verified implementation of the Ranged-sets Haskell library [9] can be reproduced in Agda, using *agda2hs*:

- We explain how we can port Haskell functions, known to be a partial programming language, to Agda, which is a total language (Section 2).
- We identify the library's preconditions, invariants and properties and explain how to formally prove them (Section 3).
- We explain the limitations of proving the properties of the Ranged-sets library and the techniques used to do so (Section 4).
- We compare the original Haskell Ranged-sets library to the *agda2hs* translation of the Agda implementation of the library (Section 5).

## 2 Preliminaries

This section contains the background information required in order to better understand the method used for porting the

[1]Project Description according to Project Forum

library to Agda. It starts with introducing the Ranged-sets library in 2.1. Next, in 2.2 we explain how we deal and convert Haskell partial functions in Agda, where they do not exist.

### 2.1 About Ranged-sets

The Ranged-sets library allows programming with sets of values that are described by lists of ranges. A value is a member of the set if it lies within one of the ranges. The ranges in a set are ordered and non-overlapping, so the standard set operations can be implemented by merge algorithms in `O(n)` time [9], where `n` is the number of ranges in that set.

The Ranged-sets library is composed of three main modules: Boundaries, Ranges and RangedSet. The first module describes the `Boundary` type, which divides an ordered type into values above and below the boundary. The Ranges module defines the `Range` type which is described by a lower and an upper boundary. Finally, the RangedSet module contains the `RSet` type, which is the actual ranged set, defined by an ordered list of non-overlapping ranges.

### 2.2 From partial to total functions

Everything that we call computation, can be reproduced using a Turing machine [10]. A Turing machine has three possible outcomes: it can accept a given input, it can reject it, or it can loop. Furthermore, we call a system Turing-complete, if it can simulate any Turing machine. In terms of programming languages, most of them are Turing-complete, although a truly Turing-complete machine requires infinite memory, property which is physically impossible when talking about computers. Haskell is a Turing-complete programming language, also known as a partial language, while Agda is total language: it accepts only programs that are provably terminating.

You may wonder how we can reproduce a Haskell partial function that can loop forever or throw run-time errors due to incomplete pattern matching or direct calls to the `error` function, in Agda, a total language that cannot simulate these outcomes, since it does not allow incomplete patterns nor has an `error` function. The answer is pretty simple: we just limit the input space by adding preconditions. By doing so, we make sure that the input values are accepted by the total program and their corresponding output consists of another value. An example of limiting the input space can be seen in Fig. 1.

```
head : (xs : List a) → ⦃ NonEmpty xs ⦄ → a
head (x :: _) = x
```

Figure 1: Limiting the input space of a function in order to make it total.

### 2.3 Proving properties with dependent types

Agda is a dependently typed programming language. When a type's definition depends on a value, it is known as a dependent type. In other words, "we can encode the properties of values as types whose elements are proofs that the property is true" [11]. For this research, we mostly used two dependent types in order to prove decidable properties. They can be seen in Fig. 2. For any value `a`, `IsTrue a` is the type of proofs that `a = true`. Similarly, `IsFalse a` is the type of proofs that `a = false`.

```
data IsTrue : Bool → Set where
  instance itsTrue : IsTrue true

data IsFalse : Bool → Set where
  instance itsFalse : IsFalse false
```

Figure 2: `IsTrue` and `IsFalse` dependent types

## 3 Porting to Agda

This section describes the techniques used in order to port the Ranged-sets library to Agda. Firstly, we introduce the dependencies required in order to implement the library in Agda. Secondly, we show how to implement the library types and their corresponding functions in Agda. Moreover, we explain the techniques used to prove the preconditions, invariants and finally, properties of the Ranged-sets library. The entire implementation is available at: https://github.com/ioanasv/research-project.

Note that for some of the proofs that are described in this section, we made used of unproven postulates. Most of them were related to boolean logic and properties of the `Ord` type. They must hold for every instance of this type, otherwise its implementation would not be correct. However, there are also some postulates about the Ranged-sets types that were used for easing proving preconditions, invariants or properties and remain subject of future work. More on the usage of postulates can be found in 4.

### 3.1 Dependencies

In order to implement the Ranged-sets library to Agda, we need firstly to implement its dependencies in *agda2hs*. The Haskell modules that are required by the library but not present in *agda2hs* are the following: `Char`, `Real` (only the `Ratio` and `Integral` types), and the `sort` method from the `List` module. Converting them to Agda is straight-forward, since there are no partial functions needed from these modules.

### 3.2 The data types

In 2.1 we described the main modules of the Ranged-sets library: Boundaries, Ranges and RangedSet. Now, we shall explain their actual Agda translation.

**Boundaries**

Boundaries work with ordered types, i.e. instances of the `Ord` type class. There are four types of boundaries: `BoundaryBelow x`, `BoundaryAbove x`, `BoundaryBelowAll` and `BoundaryAboveAll`, where `x` is an ordered value. However, when taking about equality between `BoundaryBelow` and `BoundaryAbove`, we want `BoundaryBelow x = BoundaryAbove y` when there is no value `z` between `x` and `y` [9]. As you might imagine, this is not trivial for continuous types, such as the `Double` type. If `x = y` does not hold, then there are an infinite number of values between these two. To solve this problem, there is the

`DiscreteOrdered` type that defines the `adjacent` method, which returns `true` for `x` and `y` if there is no value between them. This method returns `false` for all `x` and `y` continuous.

The `DiscreteOrdered` type class is implemented as an record type, that has two fields: `adjacent` and `adjacentBelow`. In general, Haskell type classes are modelled in Agda as record types, this representation being known as "dictionary translation" [12]. Furthermore, since the `DiscreteOrdered` type class works with ordered types, we provide an instance argument, denoted by `{{ }}`, to the record type. This can be seen in Fig. 3. An instance argument in Agda is equivalent to a type class constraint in Haskell.

```
record DiscreteOrdered (a : Set) ⦃ _ : Ord a ⦄ : Set where
    field
        adjacent : a → a → Bool
        adjacentBelow : a → Maybe a
```

Figure 3: The `DiscreteOrdered` record type.

The `Boundary` type is implemented as a data type with four different constructors, each for one of the previously described types. This can be seen in Fig. 4. It has two implemented instances, `Eq` and `Ord` for equality and ordered boundaries, respectively. The only function defined for this data type is `above`, which takes as arguments a `Boundary` and a value of the same type, and returns `true` if the value is above the boundary.

```
data Boundary (a : Set) ⦃ b : Ord a ⦄ ⦃ _ : DiscreteOrdered a ⦃ b ⦄ ⦄ : Set where
    BoundaryAbove    : a → Boundary a
    BoundaryBelow    : a → Boundary a
    BoundaryAboveAll : Boundary a
    BoundaryBelowAll : Boundary a
```

Figure 4: The `Boundary` data type.

## Ranges

The `Range` type is implemented as a data type, its constructor begin a function that expects two values of type `Boundary`. This can be seen in Fig. 5. The `Range` data type has three instances: `Eq`, `Ord` and `Show`, for equality, ordering and pretty-printing, respectively.
The implementation of all the functions defined for Ranges is straight-forward, since the corresponding Haskell functions are already total. Their description can be found below.

- `rangeUpper` returns the upper boundary of a range.
- `rangeLower` returns the upper boundary of a range.
- `rangeIsEmpty` returns true if the range is empty (the upper boundary is greater or equal to the lower boundary).
- `emptyRange` returns the range defined by `BoundaryAboveAll` and `BoundaryBelowAll`.
- `rangeHas` returns `true` if the given value is in the range.
- `rangeListHas` returns true if any range in the list contains the given value.
- `fullRange` returns the range defined by `BoundaryBelowAll` and `BoundaryAboveAll`.
- `singletonRange` takes as argument a value `v` and returns the range formed by `BoundaryBelow v` and `BoundaryAbove v`.
- `rangeIsFull` returns `true` if the range is full.
- `rangeOverlap` returns `true` if two ranges overlap.
- `rangeEncloses` returns `true` if the second range is empty or included in the first range.
- `rangeUnion` returns the union of two ranges as a range list.
- `rangeIntersection` returns the intersection of two ranges.
- `rangeDifference` returns the difference between two ranges as a range list.
- `rangeSingletonValue` returns the value contained in the singleton range if the given range is indeed singleton.

```
data Range (a : Set) ⦃ o : Ord a ⦄ ⦃ dio : DiscreteOrdered a ⦄ : Set where
    Rg : Boundary a → Boundary a → Range a
```

Figure 5: The `Range` data type.

## RangedSet

The RangedSet modules defines the `RSet` type, implemented as a data type with a constructor that takes a list of ranges and a proof that they are sorted and non-overlapping, as can be seen in Fig. 6. The latter argument is constructed using the `IsTrue` dependent data type, as described in 2.3 and shall be discussed in detail in 3.4.

```
data RSet (a : Set) ⦃ o : Ord a ⦄ ⦃ dio : DiscreteOrdered a ⦄ : Set where
    RS : (rg : List (Range a)) → {IsTrue (validRangeList rg)} → RSet a
```

Figure 6: The `RSet` data type.

There are several functions defined for the RSet data type:

- `rSetRanges` returns the range list from which the `RSet` is constructed.
- `makeRangedSet` creates a `RSet` out of a range list.
- `unsafeRangedSet` creates a `RSet` out of a range list, without verifying that the list is valid.
- `validRangeList` determines if the ranges in the list are both in order and non-overlapping.
- `normaliseRangeList` rearranges and merges the ranges in the list so that they are in order and non-overlapping.
- `normalise` normalises a range list that is known to be already sorted (this is a private routine).
- `rSingleton` creates a `RSet` from a single element.
- `rSetUnfold` creates a `RSet` from an initial boundary and two functions.
- `rSetIsEmpty` returns `true` if the `RSet` has no members.
- `rSetIsFull` returns `true` if the `RSet` is full.

- `rSetHas` returns `true` if the given value is within the `RSet`.
- `rSetIsSubset` returns `true` if the first argument is a subset of the second argument, or it is equal.
- `rSetIsSubsetStrict` returns `true` if the first argument is a subset of the second argument.
- `rSetUnion` returns the union of two `RSets`.
- `rSetIntersection` returns the intersection of two `RSets`.
- `rSetDifference` returns the difference of two `RSets`.
- `rSetNegation` returns the negation of the `RSet`.
- `rSetEmpty` returns the `RSet` created from the empty range list.
- `rSetFull` returns the `RSet` created from the full `Range`.

## 3.3 Preconditions

A precondition asserts that a condition is fulfilled for a given value before it is actually used as input. The Boundaries and Ranges module do not have any preconditions that need to be met in order to define the functions from these modules. However, there are a few preconditions to consider before porting the functions from the RangedSet module to Agda.

### The `normalise` function

The `normalise` function takes as input a range list and outputs the corresponding normalised range list by checking if two consecutive ranges are *touching* (the upper boundary of the first range is larger or equal to the lower boundary of the second range). In that case, they will be merged into a single range. However, in order to produce the normalised form of a range list, the input must fulfill the following preconditions:

- The input range list must be **sorted**.
- The input range list must contain only **valid ranges**. This means that for all ranges, their upper boundary is greater or equal to the lower boundary.

We assert that the preconditions are met by using two instance arguments of the type `IsTrue`, as can be seen in Fig. 7.

```
normalise : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄ → (rg : List (Range a))
  → ⦃ IsTrue (sortedRangeList rg) ⦄ → ⦃ IsTrue (validRanges rg) ⦄ → List (Range a)
```

Figure 7: The instance arguments are used as preconditions for the `normalise` function.

### The `unsafeRangedSet` function

`unsafeRangedSet` is a function that takes as argument a list of ranges and returns a `RSet` created from that list. However, a `RSet` is defined as a list of ordered and non-overlapping ranges, and this condition needs to be checked for the provided list of ranges before creating a `RSet` out of them. We assert that **the range list is valid** by using an instance argument of the type `IsTrue`, as can be seen in Fig. 8. It is further used as an invariant when it is passed as an implicit argument to the constructor of the `RSet` data type.

```
unsafeRangedSet : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄
  → (rg : List (Range a))
  → ⦃ IsTrue (validRangeList ⦃ o ⦄ ⦃ dio ⦄ rg) ⦄ → RSet a
unsafeRangedSet rs ⦃ prf ⦄  = RS rs {prf}
```

Figure 8: The instance argument is used as a precondition for creating a `RSet` using the provided range list.

### The `rSetUnfold` function

`rSetUnfold` is a function used to create a ranged set by providing an initial lower bound, an upper function from a lower boundary to an upper boundary used to define ranges, and a successor function from a lower boundary to another lower boundary. The precondition that needs to be checked are the following:

- The upper function must **return a greater boundary** than the input one.
- The successor function must **return a greater boundary** than the input one.

If the preconditions would not be checked, the function could **loop forever**, and this behaviour is not defined for Agda functions, as previously discussed. Therefore, we assert this precondition by adding two instance arguments consisting of proofs that the functions are valid. This can be seen in Fig. 9.

```
rSetUnfold : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄ → (b : Boundary a)
  → (f : Boundary a → Boundary a) → (g : Boundary a → Maybe (Boundary a))
  → ⦃ IsTrue (validFunction2 b f) ⦄ → ⦃ IsTrue (validFunction b g) ⦄ → RSet a
```

Figure 9: The instance arguments assert that the two provided functions return a value greater than the argument.

## 3.4 Invariants

An invariant is a predicate that satisfied the following condition: if it is true before executing a sequence of operations, it must also remain true afterwards [13]. We identified one invariant for the Ranged-sets library, more specifically for the `RSet` data type: **any RSet must be constructed from a valid range list**, where valid means that the ranges are ordered and non-overlapping.
We encode this invariant as an implicit argument to the `RSet` constructor. This can be seen in Fig. 6. It consists of a proof that the provided range list is valid. Therefore, we made sure that for every function that takes as argument a `RSet`, it is a valid one.
Furthermore, whenever a new `RSet` is returned by a function, we provide a proof that its range list is valid. There are five functions that create and output a new RSet that we shall discuss: `unsafeRangedSet`, `makeRangedSet`, `rSetUnion`, `rSetIntersection` and `rSetNegation`. The first one was described in 3.3, where we required the proof of the range list begin valid as a precondition as well, thus we can use it as an invariant and pass it as argument when constructing the `RSet`. This can be seen in Fig. 8. The other functions and how to prove that the invariant holds shall be explained further.

### The `rSetUnion` function

The `rSetUnion` function takes two valid `RSets` and computes their union. The implementation of this function can be seen in Fig. 10. The `normalise` function was previously described in 3.3. `merge1` is a function that merges the two range lists.
This function requires a proof that the invariant holds. In other words, we need to prove that by merging and normalising two valid range lists, we obtain a valid range list.

```
rSetUnion : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄
  → (rs1 rs2 : RSet a) → RSet a
rSetUnion ⦃ o ⦄ ⦃ dio ⦄ r1@(RS ls1) r2@(RS ls2) =
  RS (normalise (merge1 ls1 ls2) ⦃ merge1Sorted r1 r2 ⦄ ⦃ merge1ValidRg r1 r2 ⦄)
   {unionHolds r1 r2}
```

Figure 10: The `rSetUnion` function that computes the union of two `RSets`.

The technique used in order to proof that the `rSetUnion` function returns a valid `RSet` is splitting it into smaller parts:

1. Prove that if the two range lists are valid, then the result of the function `merge1` for these two lists is a sorted list. This proof is also passed as an instance argument to the normalise function, since it requires this precondition, as discussed in 3.3. This can be seen in Fig. 11.
2. Prove that if the two range lists are valid, then the result of the function `merge1` for these two lists is a list that contains only 'valid' ranges (for all ranges in the list, the upper boundary is greater or equal than its lower boundary). This proof is also passed as an instance argument to the normalise function, since it requires this precondition, as discussed in 3.3. This can be seen in Fig. 12.
3. Prove that if a range list is sorted and all ranges are *valid*, the normalised list is valid, see Fig. 13.

Finally, we can conclude that the union of two valid `RSets` is a valid `RSet`, by combining these three proofs, as can be seen in Fig. 14.

```
merge1Sorted : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄ → (rs1 rs2 : RSet a)
  → IsTrue (sortedRangeList (merge1 (rSetRanges rs1) (rSetRanges rs2)))
```

Figure 11: Function signature for proving that `merge1` outputs a sorted range list.

```
merge1ValidRg : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄ → (rs1 rs2 : RSet a)
  → IsTrue (validRanges (merge1 (rSetRanges rs1) (rSetRanges rs2)))
```

Figure 12: Function signature for proving that `merge1` outputs valid ranges.

### The `rSetNegation` function

The `rSetNegation` function takes as argument a valid `RSet` and must output another valid `RSet`, which contains all elements (here ranges) from the domain which are not in the first `RSet`. The negation functions works as follows.

- Transform the range list from which the `RSet` is created in a list of boundaries, by splitting every range in a list in its lower and upper boundary (see Fig. 16).

```
validNormalised : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄
  → (ms : List (Range a))→ (prf : IsTrue (sortedRangeList ms))
  → (prf2 : IsTrue (validRanges ms))
  → IsTrue (validRangeList (normalise ms ⦃ prf ⦄ ⦃ prf2 ⦄))
```

Figure 13: The output of `normalise` is a valid range list.

```
unionHolds : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄ → (rs1 rs2 : RSet a)
  → IsTrue (validRangeList (normalise (merge1 (rSetRanges rs1) (rSetRanges rs2))
  ⦃ merge1Sorted rs1 rs2 ⦄ ⦃ merge1ValidRg rs1 rs2 ⦄))
unionHolds ⦃ o ⦄ ⦃ dio ⦄ rs1@(RS r1) rs2@(RS r2) = validNormalised
   (merge1 r1 r2) (merge1Sorted rs1 rs2) (merge1ValidRg rs1 rs2)
```

Figure 14: Proof that union of two `RSets` outputs a valid `RSet`.

- Check whether the boundary list starts with the `BoundaryBelowAll` value, and if so, remove it and return the rest. If this is not the case, the `BoundaryBelowAll` value is prepended to the list (see Fig. 17). At this point, the boundary list has an odd number of elements.
- Group the boundaries in pairs in order to create Range objects out of them. However, since the boundary list has an odd number of elements, there will remain one boundary that is not paired with anything. In that case, if the boundary is `BoundaryAboveAll`, no `Range` is created. Otherwise, create a `Range` from the given boundary and `BoundaryAboveAll` (see Fig. 18).

The implementation of the `rSetNegation` function can be seen in Fig. 15.

```
rSetNegation : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄ → RSet a → RSet a
rSetNegation ⦃ o ⦄ ⦃ dio ⦄ set@(RS ranges {prf}) =
   RS (ranges1 (setBounds1 (bounds1 ranges))) {negation set prf}
```

Figure 15: Negation of a `RSet`.

In order to prove that the invariant holds, the splitting technique was used and the following steps were followed:

1. Prove that if the range list was valid in the first place, then the boundary list obtained from this list is also valid.
2. Prove that if a boundary list is valid, `setBounds1` applied to the same list outputs a valid list. There are two cases that need to be covered for this step: either the first boundary in the list was removed, or `BoundaryBelowAll` was prepended. It is not hard to prove that for any valid list of boundaries, its tail is also valid. Furthermore, we can also prove that `BoundaryBelowAll` is less or equal to any other boundary, so the order is preserved if this value is prepended to the list. We proved this using equivalence, as can be seen in Fig. 19.
3. Prove that if a list of boundaries is valid and we apply the `ranges1` function to this list, the returned range list is also valid. This can be proved using induction.

Finally, we combine the previously three proofs in order to prove that the invariant holds. Firstly, we prove that if a range list is valid, then the boundary list obtained using `setBounds1` on its boundaries is also valid.

This is proven using transitivity between the previously described first two proofs. Using this property, we can finally prove that if a range list `rs` is valid, then `ranges1 (setBounds1 (bounds1 rs))` is also valid. The signature of the invariant proof can be seen in Fig. 20.

```
bounds1 : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄
  → List (Range a) → List (Boundary a)
bounds1 (r :: rs) = (rangeLower r) :: (rangeUpper r) :: (bounds1 rs)
bounds1 [] = []
```

Figure 16: Helper function used for `RSet` negation, creates a list of boundaries from a list of ranges.

```
setBounds1 : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄
  → List (Boundary a) → List (Boundary a)
setBounds1 (BoundaryBelowAll :: xs) = xs
setBounds1 xs = (BoundaryBelowAll :: xs)
```

Figure 17: Helper function used for `RSet` negation, creates a list the boundaries for the negated set.

```
ranges1 : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄
  → List (Boundary a) → List (Range a)
ranges1 (b1 :: b2 :: bs) = (Rg b1 b2) :: (ranges1 bs)
ranges1 (BoundaryAboveAll :: [])  = []
ranges1 (b :: []) = (Rg b BoundaryAboveAll) :: []
ranges1 _ = []
```

Figure 18: Helper function used for `RSet` negation, creates a list of ranges from a list of boundaries.

```
validSetBounds : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄
  → (bs : List (Boundary a))
  → validBoundaryList (setBounds1 bs) ≡ validBoundaryList bs
```

Figure 19: The `setBounds1` function does not change the output of the `validRangeList` function.

```
negation : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄ → (rs : RSet a)
  → (IsTrue (validRangeList (rSetRanges rs)))
  → (IsTrue (validRangeList (ranges1 (setBounds1 (bounds1 (rSetRanges rs))))))
```

Figure 20: Function signature for proving that the invariant holds for the `RSet` negation.

### The `rSetUnfold` function

As we previously discussed in 3.3, the `rSetUnfold` function is used to create a `RSet` using an initial lower boundary, an upper function that increments the boundary, and a successor function that increments the previous lower boundary in order to create another range. Its implementation can be seen in Fig. 21, where `ranges2` is a function that creates a list of ranges using the initial boundary, the upper and successor functions. However, this list needs to be normalised, since the result of the successor function applied to a boundary is not necessarily larger than the result of the upper function applied to the same boundary.

In 3.3, we introduced the `normalise` function and its preconditions: that the input range list is sorted and that it contains only valid ranges. By providing these proofs for `ranges2`, we can also easily prove the invariant. Therefore, we need to prove three properties: firstly, that `ranges2` produces a sorted range list, secondly, that it produces a range list consisting only of valid ranges and finally, that normalising a sorted range list results in a valid range list. The first two proofs follow easily from the functions being valid (their output value is larger than the input value). The latter proof is also used for proving the invariant of the `rSetUnion` function and its signature can be seen in Fig. 13. Finally, they are used together to prove the invariant for `rSetUnfold`, see Fig. 21.

```
rSetUnfold : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄ → (b : Boundary a)
  → (f : Boundary a → Boundary a) → (g : Boundary a → Maybe (Boundary a))
  → ⦃ IsTrue (validFunction2 b f) ⦄ → ⦃ IsTrue (validFunction b g) ⦄ → RSet a
rSetUnfold b f g ⦃ ff ⦄ ⦃ gg ⦄ = RS (normalise ranges2res ⦃ sorted ⦄ ⦃ validrg ⦄)
  {validNormalised ranges2res sorted validrg}
  where
    validrg = ranges2ValidRg b f g ff gg
    sorted = unfoldSorted b f g
    ranges2res = ranges2 b f g
```

Figure 21: Implementation of `rSetUnfold`.

### The `rSetIntersection` function

The `rSetIntersection` function takes two valid `RSets` and computes their intersection. Its implementation can be seen in Fig. 22, where `merge2` is a function that outputs the intersection of two ranges, one from each `RSet`, in order. We want to provide a proof that their intersection consists of a valid range set, see Fig. 23. The property that the invariant holds for `rSetIntersection` was proved using induction.

```
rSetIntersection : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄
  → RSet a → RSet a → RSet a
rSetIntersection ⦃ o ⦄ ⦃ dio ⦄ rs1@(RS ls1) rs2@(RS ls2) =
  RS (filter (λ x → rangeIsEmpty ⦃ o ⦄ ⦃ dio ⦄ x == false) (merge2 ls1 ls2))
  {intersection0 rs1 rs2}
```

Figure 22: Intersection of two `RSets`.

```
intersection0 : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄ → (rs1 rs2 : RSet a)
  → IsTrue (validRangeList (filter (λ x → (rangeIsEmpty x == false))
    (merge2 (rSetRanges rs1) (rSetRanges rs2))))
```

Figure 23: Function signature for proving that the invariant holds `RSet` intersection.

### The `makeRangedSet` function

The `makeRangedSet` function creates a new `RSet` out of a range list, without imposing any constraints on that list. It calls the `normaliseRangeList` method, that filters out any empty range, sorts the list and finally, calls the `normalise` method on the filtered and sorted list. Its implementation can be found in Fig. 24.

Since it calls `normalise`, we also need to provide proofs for its preconditions, as described in 3.3. Firstly, we need to prove that the sorted and filtered list is indeed sorted, and that its ranges are valid. Finally, we can make use of the proof that normalising a list which is sorted and has all ranges valid, results in a valid range list. This proof was also used for proving that the invariant holds for `rSetUnion` and `rSetUnfold` and

```
normaliseRangeList : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄
  → List (Range a) → List (Range a)
normaliseRangeList [] = []
normaliseRangeList rs = normalise (sort (filter
  (λ r → (rangeIsEmpty r) == false) rs)) ⦃ sortedList rs ⦄
    ⦃ validRangesList rs ⦄
```

Figure 24: Function used for creating a new `RSet` out of an arbitrary list.

can be found in Fig. 13.
We postulated the proofs of the sorted and filtered list begin indeed sorted (Fig. 25) and not containing invalid ranges (Fig. 26). An example of how to prove that calling the function `sort` on a list returns a sorted list can be found here: https://twanvl.nl/blog/agda/sorting. Moreover, by filtering out the empty ranges (a range is empty if its upper boundary $\geq$ lower boundary) from a list, we ensure that the list contains only valid ranges (a range is valid if its upper boundary $\leq$ lower boundary). If this postulate does not hold, then the `filter` function from the `List` module is not correct.

```
sortedList : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄ → (rs : List (Range a))
  -> IsTrue (sortedRangeList (sort (filter (λ r → (rangeIsEmpty r) == false) rs)))
```

Figure 25: If we apply sort to a range list, the resulting list is indeed sorted.

```
validRangesList : ⦃ o : Ord a ⦄ → ⦃ dio : DiscreteOrdered a ⦄ → (rs : List (Range a))
  -> IsTrue (validRanges (sort (filter (λ r → (rangeIsEmpty r) == false) rs)))
```

Figure 26: If we filter out the empty ranges from a range list, the result list has all ranges valid.

## 3.5 Properties

When talking about data types, a property is a characteristic that should hold for every value of that type. We can state properties for each type defined in the Ranged-sets library: `Boundary`, `Range` and `RSet`. The library defines several QuickCheck properties for the `Range` and `RSet` data types that shall be described next. We focused on translating these QuickCheck properties into Agda proofs. Moreover, we also state other properties that are worth to verify, but were not covered by QuickCheck.

**Boundary**
There were no QuickCheck properties defined for the `Boundary` data type, however we can defined a few properties that can help when further verifying the other modules.

- `BoundaryBelowAll` is less or equal to any other boundary.
- `BoundaryAboveAll` is greater or equal to any other boundary.

**Range**
The Ranges module comes with several QuickCheck properties defined in Haskell. We reproduced the following, as proofs in Agda:

- The union of two ranges has a value if and only if either range has it.
- The length of the union of two ranges is either 1 or 2.
- The intersection of two ranges has a value if and only if both ranges have it.
- The intersection of two ranges is non-empty if and only if they overlap.
- The singleton range contains its member.
- The singleton range contains only its member.
- A singleton range can have its value extracted.
- The empty range is not a singleton.
- The full range is not a singleton.

There were also some properties that were not covered by QuickCheck that we considered important, therefore we proved them:

- The union of two ranges commutes.
- The intersection of two ranges commutes.
- If a range is non-empty, the lower boundary is less than the upper boundary.

**RSet**
The RangedSet module comes with various QuickCheck properties that test the implementation of the `RSet` data type. We proved in Agda the following ones:

- A normalised range list is valid for `unsafeRangedSet`.
- The empty set has no members.
- The full set has every member.
- The intersection of a set with its negation is empty.
- A set is the non-strict subset of itself.
- A set is not the strict subset of itself.
- Intersection commutes.
- Union commutes.
- For any value $v$ and ranged sets $A$ and $B$, if $v \in (A \cup B)$ then $v \in A$ or $v \in B$.
- For any value $v$ and ranged sets $A$ and $B$, if $v \in (A \cap B)$ then $v \in A$ and $v \in B$.
- For any value $v$ and ranged set $A$, if $v \in A$ then $v$ is not in the negation of A.
- For any value $v$ and ranged sets $A$ and $B$, if $v \in A$ and $v \notin B$ then $v \in (A - B)$.

There are some other properties not covered by QuickCheck that are worth proving. Since a `RSet` is mathematically a set, in other words a collection of things which we call members, any property that holds in set theory should also hold for a `RSet`. The "A Set Theory Formalization" report [14] aims to prove the set theory axioms in Agda. Although the implementation of a `RSet` is different from the one of a usual set, we proved that some of its usual properties also hold for `RSets`:

- For any two ranged sets $A$ and $B$, if $A \subset B$ then $A \subseteq B$.
- For any two ranged sets $A$ and $B$, if $A \subset B$ then $B \not\subseteq A$ holds.

- For any value $v$ and ranged sets $A$ and $B$, if $v \in (A \cup B)$ then $v \in (B \cup A)$.
- For any value $v$ and ranged set $A$, $v \in (A \cup A)$ is equivalent to $v \in A$.

## 4 Limitations of proving in Agda

Porting the Ranged-sets library to Agda comes also with a notable disadvantage. Proving properties can quickly become unfeasible, since a mathematical proof needs to check all paths that the program can take. When a function has multiple possible patterns for the input, when it is composed of nested `if then else` statements, or when there are multiple nested functions used, proving quickly becomes tedious.
There are some QuickCheck properties that were not converted to Agda proofs during this project and therefore remain subject of future work. The most notable ones are the following:

- De Morgan's Law for Intersection: negating the intersection of two sets is equivalent to the union of the two sets' negations.
- De Morgan's Law for Union: negating the union of two sets is equivalent to the intersection of the two sets' negations.

However, there are some techniques that could be useful when proving properties about the Ranged-sets library. The approaches used in this paper for simplifying proofs are the following:

- reduce them to smaller proofs, technique which we previously called **splitting**;
- define the preconditions and prove them for every function that requires this;
- define the invariants and prove them by either splitting or by using the preconditions proofs;
- define the post-conditions (properties) and prove them by either splitting or by using the invariants proofs;
- define helper proofs as postulates.

We must note that using postulates comes with a drawback: they do not ensure that if the code modifies, the property still holds. That is because postulates are used for defining values as types without providing their actual implementation [15]. As mentioned in 3, our implementation uses some unproven postulates, mostly about boolean logic and properties of the `Ord` type. These can be found here: https://github.com/ioanasv/research-project/blob/master/src/RangedSetsProp/library.agda. However, there are also a few postulates about the Ranged-sets library types, that were used as helper proofs throughout the process of verification. Some of them can be considered out of the scope of the project, such as proving that the `sort` method actually produces a sorted list or that the `filter` method actually filters a list. Nevertheless, proving the used postulates is possible and remains a subject of future work.

## 5 Experimental Setup and Results

The Agda version of the Ranged-Sets library was implemented in Visual Studio Code [16]. The *agda* and *agda-mode* programs were installed using Cabal [17] [18]. *agda2hs* can be found here: https://github.com/agda/agda2hs. It can be installed and built using either the existing *make* file, or by using *Stack* [19].

The Ranged-sets library can be reproduced and verified in Agda. There are no practical limitations of total functions for the implementation of this library. Moreover, proving preconditions comes with a great improvement to the library. Even though the preconditions are mentioned in the documentation of the library, they are not checked, thus some functions may not behave as expected when these constraints are not met. Another important advantage of porting the Ranged-sets library to Agda is the possibility of guaranteeing that the invariants hold. By proving them, the output of the functions is ensured to be valid, as was explained in 3.4. Furthermore, we converted most of the QuickCheck properties from the Haskell Ranged-sets library to Agda proofs. Some of the QuickCheck properties provided by this library turned out to be unfeasible for this project and future research is needed in order to complete them, as explained in 4.

An quantitative overview of the Ranged-sets Haskell library, its Agda implementation and our proofs can be seen in Fig. 27. Note that for a fair comparison, we only counted the lines of code of the QuickCheck properties we proved in Agda. The biggest increase can be observed when translating the QuickCheck properties to Agda proofs, this being the most challenging part of our project.

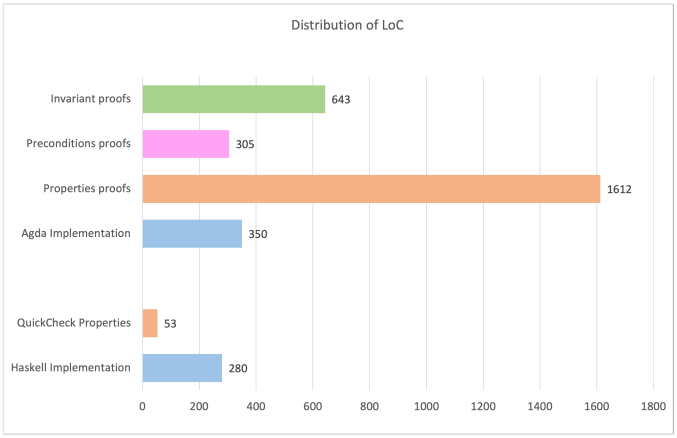


Figure 27: Distribution of lines of code over the Haskell library, its Agda implementation and proofs.

The Agda implementation of the Ranged-sets library can be ported back to Haskell, using *agda2hs*. The implicit arguments are removed from Agda functions when translating to Haskell, thus the proofs for the preconditions and invariants are no longer present nor required, therefore it has the same functionality as the original Haskell library. An example can be seen in Fig. 28, where the Agda implementation of the `RSet` requires an implicit argument that the provided range list is valid, as discussed in 3.4.

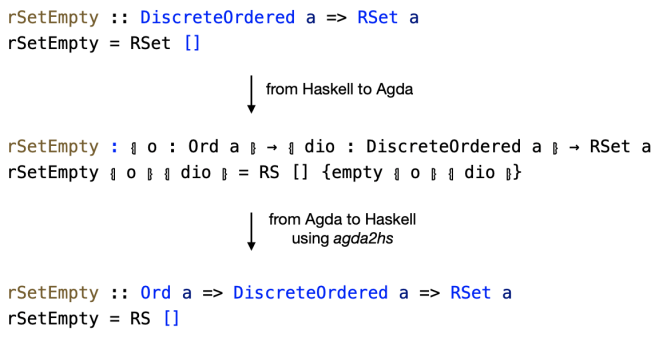


Figure 28: Translation from Haskell to Agda and back to Haskell using *agda2hs*.

# 6 Responsible Research

The integrity and reproducibility of experimental results is vital for research projects since it creates trust, ensures transparency and therefore, confidence. The aim of this research is to produce a verified translation of the Haskell Ranged-sets functional library to Agda.

To ensure reproducibility, in Section 3, we explain the general guidelines of the implementation and verification of the library. If the reader has further questions or uncertainties, the entire code is available at https://github.com/ioanasv/research-project. By making our code public, we ensure that it can be easily checked that the techniques we presented are adequate for translating and verifying this specific library. Moreover, in 28, we introduce the experimental setup: the IDE that was used, as well as the needed dependencies. This guarantees that the code can be run on any machine that supports Visual Studio Code, Agda and *agda2hs*.

Furthermore, in order to ensure integrity, we state in Section 4 the limitation of our research, as well as the fact that our implementation relies on some unproven postulates used to ease the verification process. We aimed for scientific honesty, thus the reader can trust and understand our work.

# 7 Related Work

## 7.1 hs-to-coq

Similar to *agda2hs*, hs-to-coq is a tool that translates total Haskell code to Coq in order to verify the correctness of Haskell programs [20]. Comparable to Agda, Coq is a proof assistant [21] and therefore, a great tool for writing proofs. The "Ready, Set, Verify! Applying hs-to-coq to real-world Haskell code" project used hs-to-coq to translate a part of the Haskell `containers` library to Gallina, the specification language of Coq, and verified its properties using the Coq proof assistant [22]. In order to state the specifications of the library, they used several techniques, such as: retrieve the invariants from the *validity* of a type, use the already defined QuickCheck properties, formalize the specifications given in the comments of the library, or prove the type class laws, if any. These approaches are very similar to our work, as is explained in 3. However, our work focuses only on proving preconditions, invariants, and properties (post-conditions).

## 7.2 Balancing weight-balanced trees

Hirai and Yamamoto [23] aimed to provide a purely functional and reliable version of the weight-balanced tree data structure. The reasoning behind their research is the fact that in many functional implementations of this data structure, after insertion and deletion, the balancing property does not hold anymore. They identified the valid rotation parameters under which the balancing property holds for insertion and deletion, using Coq. However, our work is different and broader than this: we also provide proofs about preconditions and properties, not only about invariants.

## 7.3 Liquid Haskell

Liquid Haskell [24] is a program that verifies Haskell programs against logical specifications using refinements types (types embedded with predicates that need to be satisfied by all values of those types). It uses a satisfiability modulo theories (SMT) solver that determines the satisfiability of first-logic formulas. In the case of Liquid Haskell, a formula is a predicate defined in a refinement type. "A tale of two provers: verifying monoidal string matching in liquid Haskell and Coq" [25] compares Liquid Haskell and Coq as theorem provers and concludes that both have advantages: while Liquid Haskell is SMT-automated, Coq has a large variety of tactics for proving.

# 8 Conclusions and Future Work

This paper presents a verified implementation of the Haskell Ranged-sets library in Agda, using *agda2hs*, a program that provides a common subset between these two programming languages. After adding the missing dependencies that the library requires, the Agda implementation was possible and successful. In order to verify it, we proved its preconditions, invariants and properties. Moreover, by adding and verifying preconditions and invariants, we produced a better and safer version of this library: even though they were mentioned in the documentation of the Haskell library, the preconditions and invariants were never checked.

Moreover, we converted most of the QuickCheck properties to Agda proofs, the main advantage being that we no longer rely on the input generated by QuickCheck to verify them. However, proofs in Agda can quickly become unfeasible even when proving properties about moderate size functions. Future research is needed for easing the process of proving, such as improvement in the Agda's reflection mechanism used for equivalence relations, or even identifying similar properties and the corresponding approaches to actually prove them.

# References

[1] *Haskell language*, https://www.haskell.org/.

[2] G. Hutton, *Programming in Haskell*. Cambridge University Press, 2016.

[3] *Agda*, https://github.com/agda/agda.

[4] *Curry-howard correspondence*, https://www.cs.cornell.edu/courses/cs3110/2021sp/textbook/adv/curry-howard.html.

[5] F. Besson, S. Blazy, and P. Wilke, "Compcerts: A memory-aware verified c compiler using pointer as integer semantics," *Interactive Theorem Proving Lecture Notes in Computer Science*, pp. 81–97, 2017. DOI: 10.1007/978-3-319-66107-0_6.

[6] P. Dybjer, Q. Haiyan, and M. Takeyama, "Combining testing and proving in dependent type theory," in *Theorem Proving in Higher Order Logics*, D. Basin and B. Wolff, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 188–203.

[7] *Quickcheck: Automatic testing of haskell programs*. [Online]. Available: https://hackage.haskell.org/package/QuickCheck.

[8] *Agda2hs*, https://github.com/agda/agda2hs.

[9] PaulJohnson, *Pauljohnson/ranged-sets*. [Online]. Available: https://github.com/PaulJohnson/Ranged-sets.

[10] B. J. Copeland, "The Church-Turing Thesis," in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., Summer 2020, Metaphysics Research Lab, Stanford University, 2020.

[11] U. Norell, "Dependently typed programming in agda," in *Proceedings of the 6th International Conference on Advanced Functional Programming*, ser. AFP'08, Heijen, The Netherlands: Springer-Verlag, 2008, pp. 230–266, ISBN: 3642046517.

[12] D. Devriese and F. Piessens, "On the bright side of type classes: Instance arguments in agda," vol. 46, Sep. 2011, pp. 143–155. DOI: 10.1145/2034574.2034796.

[13] *Invariant (mathematics)*, May 2021. [Online]. Available: https://en.wikipedia.org/wiki/Invariant_(mathematics).

[14] A. Calle-Saldarriaga, "A Set Theory Formalization," Mathematical Engineering, Universidad EAFIT, Tech. Rep., Jun. 2017.

[15] *Postulates*. [Online]. Available: https://agda.readthedocs.io/en/v2.6.1.3/language/postulates.html.

[16] *Visual studio code - code editing. redefined*, Apr. 2016. [Online]. Available: https://code.visualstudio.com/.

[17] *Installation - agda 2.6.2 documentation*. [Online]. Available: https://agda.readthedocs.io/en/latest/getting-started/installation.html.

[18] *Cabal user guide*. [Online]. Available: https://cabal.readthedocs.io/en/3.4/.

[19] S. contributors, *The haskell tool stack*. [Online]. Available: https://docs.haskellstack.org/en/stable/README/.

[20] A. Spector-Zabusky, J. Breitner, C. Rizkallah, and S. Weirich, "Total haskell is reasonable coq," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2018, Los Angeles, CA, USA: Association for Computing Machinery, 2018, pp. 14–27, ISBN: 9781450355865. DOI: 10.1145/3167092. [Online]. Available: https://doi.org/10.1145/3167092.

[21] T. C. development team, *The coq proof assistant reference manual*, 2016. [Online]. Available: http://coq.inria.fr.

[22] J. Breitner, A. Spector-Zabusky, Y. Li, C. Rizkallah, J. Wiegley, and S. Weirich, "Ready, set, verify! applying hs-to-coq to real-world haskell code (experience report)," *Proc. ACM Program. Lang.*, vol. 2, no. ICFP, Jul. 2018. DOI: 10.1145/3236784. [Online]. Available: https://doi.org/10.1145/3236784.

[23] Y. HIRAI and K. YAMAMOTO, "Balancing weight-balanced trees," *Journal of Functional Programming*, vol. 21, no. 3, pp. 287–307, 2011. DOI: 10.1017/S0956796811000104.

[24] N. Vazou, "Liquid haskell: Haskell as a theorem prover," Ph.D. dissertation, University of California, San Diego, 2016.

[25] N. Vazou, L. Lampropoulos, and J. Polakow, "A tale of two provers: Verifying monoidal string matching in liquid haskell and coq," in *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, ser. Haskell 2017, Oxford, UK: Association for Computing Machinery, 2017, pp. 63–74, ISBN: 9781450351829. DOI: 10.1145/3122955.3122963. [Online]. Available: https://doi.org/10.1145/3122955.3122963.