Producing a verified implementation of sequences in agda2hs

Shashank Anand

Abstract

The abstract should be short and give the overall idea: what is the background, the research questions, what is contribution, and what are the main conclusions. It should be readable as a stand-alone text (preferably no references to the paper or outside literature).

1 Introduction

Haskell[4] is a strongly typed purely functional 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. This is demonstrated for example in Chapter 16 of the book Programming in Haskell (2nd edition) by Graham Hutton[11], 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.

In a dependently typed programming language such as Agda[1], we can do better than that and actually write a formal proof of correctness in the language itself[13][14]. This proof is checked automatically by the typechecker, and re-checked every time the code is changed. So 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[9], these languages remain hard to use. So these guarantees are currently only available to expert users.

The agda2hs project[2] 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, and then translate the result to Haskell so it can be used and understood by Haskell programmers.[1] However, agda2hs does not completely identify the common subset of Haskell and Agda, and the implications of the missing subset on the expressive ability of Agda is unknown.

Sequence[6] is a Haskell library that represents lists internally as finger trees[10]. Sequences are more efficient than traditional lists for a variety of operations such as concatenation, splitting, index access, etc.

The objective of the paper is to discuss the implementation of a verified implementation of the Sequence library using the common subset identified by agda2hs. In section 2, the method used to produce and verify the implementation is discussed. Section 3 discusses the

[1]Project description as provided in the project Practical Verification of Functional Libraries on Project Forum.

proof strategies. Section 4 ?. Section 5 details the ethical aspects of the research. Section 6 (7?) concludes by discussing relevant improvements and future work.

2 Background

2.1 Data.Sequence

Data.Sequence is a Haskell library that provides methods to create and operate on lists that are internally represented as finger trees[10]. Sequences support operations on lists that are more efficient than on traditional lists. Most notably they are :

- Constant-time access to both the front and the rear end of the list.
- Logarithmic-time concatenation
- Logarithmic-time splitting, take, drop.
- Logarithmic-time random access

However, the finger tree implementation leads to reduced efficiency of some operations. Most notably, prepending an element to a finger tree has worst case time complexity $\mathcal{O}(\log n)$ Sequences are also typically slower than lists for operations with the same $\mathcal{O}()$ complexity.

2.2 Dependent Types

Dependent types are types that depend on a value of a type[8]. Dependent types are widely used to prove properties/encode proofs in dependently typed programming languages.
Two important dependent types used in this paper are IsTrue and $\equiv$.

IsTrue is defined as follows.

```
data IsTrue : Bool -> Set where
  instance itsTrue : IsTrue true
```

IsTrue p can be used in the type of a function to ensure that the function only accepts inputs for which the predicate p holds. As IsTrue false is an empty type, the function is not defined for inputs for which the predicate does not hold.

$\equiv$ is defined as follows. It is provided as part of the module Agda.Builin.Equality[3].

```
data _≡_ {A : Set} : (x y : A) -> Set where
  refl : (x : A) -> x ≡ x
```

A $\equiv$ B can be used to assert that A is strongly equivalent to B. This dependent type is used in this paper to prove properties on Sequences. This can be achieved by defining and implementing functions that return a value of type A $\equiv$ B

2.3 Totality of Functions in Agda

Total functions are functions that are defined for all values over the domain of the input, never throw an exception, and always terminate and return a value[12]. In agda, all functions are required to be total[7].
For example, take the following definition of head of a list.

```
head : {a : Set} -> List a -> a
head (x :: xs) = x
```

This definition is invalid because the case head [ ] is missing. However, it is impossible to get the head of an empty list. Therefore, this function can only be defined if the given list is non-empty. Thus, it is necessary to limit the domain of the function by supplying a proof that the list is non-empty as an implicit argument[5]. This can be done using the IsTrue dependent type described in 2.2.

```
head : {a : Set} -> (xs : List a) -> {IsTrue (isNonEmpty xs)} -> a
head (x :: xs) = x
```

As IsTrue false is an empty type, the function only accepts non-empty lists, thus making a non-total function total.

3 Methodology or Problem Description

Choose one that fits your research best:

3.1 Methodology

Typically in general research articles, the second section contains a description of the research methodology, explaining what you, the researcher, is doing to answer the research question(s), and why you have chosen this method. For purely analytical work this is a description of the data collection or experimental setup on how to test the hypothesis, with a motivation. In any case this section includes references to necessary background information.

3.2 Formal Problem Description

For some types of work in computer science the methodology is standard: analyze the problem (e.g., make assumptions and derive properties), present a new algorithm and its theoretical background, proving its correctness, and evaluate unproven aspects in simulation. Then an explanation of the methodology is often omitted, and the setup of the evaluation is part of a later section on the evaluation of the ideas.[2] In this case, explain relevant concepts, theory and models in this section (with references) and relate them to your research question. Also this section then typically contains a more precise, formal description of the problem.

Do not forget to give this section another name, for example after the problem you are solving.

4 Your contribution

In computer science typically the third section contains an exposition of the main ideas, for example the development of a theory, the analysis of the problem (some proofs), a new algorithm, and potentially some theoretical analysis of the properties of the algorithm.

Do not forget to give this section another name, for example after the method or idea you are presenting.

Some more detailed suggestions for typical types of contributions in computer science are described in the following subsections.

[2]This already shows that there is no single outline to be given for all papers.

Experimental work

In this case, this section will mostly contain a description of the methods/algorithms you will be comparing. Although not all methods need to be described in detail (providing appropriate references are available), make sure that you reveal sufficient details to a reader not familiar with these methods to: a) obtain a high-level understanding of the method and differences between them, and b) understand your explanation of the results.

Improvement of an idea

In this case, you would need to explain in detail how the improvement works. If it is based on some observation that can be proven, this is a good place to provide that proof (e.g., of the correctness of your approach).

5 Experimental Setup and Results

As discussed earlier, in many sciences the methodology is explained in section 2 and this section only discusses the results. However, in computer science, most often the details of the evaluation setup are described here first (simulation environment, etc.). Very important here is that any skilled reader would be able to reproduce this setup and then obtain the same results.

Then, results are reported in an accessible manner through figures (preferably with captions that allow them to be understood without going through the whole text), observations are made that clearly follow from the presented results. Conclusions are drawn that follow logically from the previous material. Sometimes the conclusions are in fact hypotheses, which in turn may give rise to new experiments to be validated.

You may want to give this section another name.

6 Responsible Research

Reflect on the ethical aspects of your research and discuss the reproducibility of your methods.

7 Discussion

Results can be compared to known results and placed in a broader context. Provide a reflection on what has been concluded and how this was done. Then give a further possible explanation of results.

You may give this section another name, or merge it with the one before or the one hereafter.

8 Conclusions and Future Work

Summarize the research question(s) and the answers to the research question(s). Make statements. Highlight interesting elements.

Discuss open issues, possible improvements, and new questions that arise from this work; formulate recommendations for further research.

ideally, this section can stand on its own: it should be readable without having read the earlier sections.

A The obvious

A.1 Reference use

- use a system for generating the bibliographic information automatically from your database, e.g., use BibTex and/or Mendeley, EndNote, Papers, or ...
- all ideas, fragments, figures and data that have been quoted from other work have correct references
- literal quotations have quotation marks and page numbers
- paraphrases are not too close to the original
- the references and bibliography meet the requirements
- every reference in the text corresponds to an item in the bibliography and vice versa

A.2 Structure

Paragraphs

- are well-constructed
- are not too long: each paragraph discusses one topic
- start with clear topic sentences
- are divided into a clear paragraph structure
- there is a clear line of argumentation from research question to conclusions
- scientific literature is reviewed critically

A.3 Style

- correct use of English: understandable, no spelling errors, acceptable grammar, no lexical mistakes
- the style used is objective
- clarity: sentences are not too complicated (not too long), there is no ambiguity
- attractiveness: sentence length is varied, active voice and passive voice are mixed

A.4 Tables and figures

- all have a number and a caption
- all are referred to at least once in the text
- if copied, they contain a reference
- can be interpreted on their own (e.g. by means of a legend)

References

[1] Agda. https://github.com/agda/agda.

[2] Agda2hs. https://github.com/agda/agda2hs.

[3] Built-ins. https://agda.readthedocs.io/en/latest/language/built-ins.html#equality.

[4] Haskell language. https://www.haskell.org/.

[5] Implicit arguments in agda. https://agda.readthedocs.io/en/v2.6.1.3/language/implicit-arguments.html.

[6] Sequence. https://hackage.haskell.org/package/containers-0.6.4.1/docs/Data-Sequence.html.

[7] What is agda?

[8] Dependent type. https://en.wikipedia.org/wiki/Dependent_type, Apr 2021.

[9] Frederic Besson, Sandrine Blazy, and Pierre Wilke. Compcerts: A memory-aware verified c compiler using pointer as integer semantics. *Interactive Theorem Proving Lecture Notes in Computer Science*, pages 81–97, 2017.

[10] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(02):197, 2005.

[11] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2016.

[12] Eric Normand. What is a total function?, Sep 2019.

[13] Christopher Schwaab and Jeremy G. Siek. Modular type-safety proofs in agda. *Proceedings of the 7th workshop on Programming languages meets program verification - PLPV '13*, 2013.

[14] Paul van der Walt and Wouter Swierstra. Engineering proof by reflection in agda. *Implementation and Application of Functional Languages*, pages 157–173, 2013.