<Razvan Kusztos>
<Girton>
<rek43>

Part II Project Proposal

Verified functional data structures and algorithms in Agda

<10th October 2016>

Project Originator: Razvan Kusztos

Resources Required: See attached Project Resource Form

Project Supervisor: Dr. Timothy Griffin

Signature:

Director of Studies: Chris Hadley

Signature:

Overseers: Markus Kuhn and Peter Sewell

Signatures:

Introduction

Functional data structures are persistent data structures. That is, whenever a change occurs to the data structure, the programmer should expect to have both the previous and the new version available. This is different from the usual imperative approach, where updates are destructive and yield ephemeral data structures. Although for long it was a consensus that the functional data structures are not as efficient as their imperative counterparts, Okasaki's book [1] goes in depth to optimizing structures for this paradigm. It is coded in a Standard ML and provides informal proofs.

Another approach to providing proofs to such data structures is by using languages, such as Coq, HOL, Isabelle or Agda [5]. During this project I aim to use Agda as it is the one to help me learn about dependent types.

Agda is a functional, dependently typed programming language [5]. In this setting, types can depend on values, allowing for more expressive type declaration. Agda's environment allows the users the ability to write both the code and the proofs that have as objects entities in the code in the same language. An illustrative book on this topic is [6]

Defining data structures and functions in a dependently typed programming language opens up new possibillities for better expressivity. Invariants can now be part of the typing system. A great example of this that is already part of the literature is the implementation or Red-Black trees, which can maintain all five invariants in the typing declaration. This ensures that any update operation on a Red-Black tree will continue to maintain these invariants.rbtrees

Agda was originally developed as a programming language, quite similar in syntax to Haskell, but kept as simple as possible. The dependently typed environment allows, via the Curry-Howard correspondence to create human-readable proofs. Therefore, in the previous example, we can also demonstrate in the same programming language that the operations that are implemented not only keep the invariants, but also yield the right results and have the expected mathematical properties.

Although not designed with this goal in mind, if the typing information is sufficient, Agda can act not only as an interactive prover, but the typing information can actually help the coding process by using tools such as automatic case split and expressive goal information. Moreover, support of Unicode characters in the compiler front-end allows for even better readability.

The Agda Library is not extensively documented and is a work in progress. It would be great if via this project I can not only demonstrate the power of this language, but also contribute with more efficient data structures for the standard library.

Resources Required

I will undertake this project on my laptop.

Starting Point

At the beginning of this project I have some experience with functional programming languages (SML from the Foundations of Computer Science course in Part IA) and Scala (from a Udacity course). Basics of structural induction, which will almost certainly be a central part are gained from the Semantics of Programming Languages course. I will have to familiarize myself with Agda and most recent advents in terms of functional data structures, as well as relevant chapters from the Denotational Semantics and Type Theory courses in Part II. No prior experience with Agda, dependent types and proofs in this environment.

Substance and Structure of the Project

The main structure of the project is gradually building up to more and more specialized data structures that could be reused in Agda libraries and proving properties about their correctness and mathematical properties

Okasaki's book describes a mechanism for optimizing functional data structures that were thought to be a lot inferior to the imperative counter-parts. A general purpose data structure that stems from okasaki's principles and implemented by Hinze and Patterson [2] is the Finger Tree. This will be implemented, along with other basic data structures to illustrate the power of dependent typing in action. Finger Trees will then be used to implement more powerful data structures, such as Random Access Lists or Heaps and prove properties of them to be correct. Then, benchmarks will be ran to show that they are indeed more efficient than what the standard library already provides. The project will aim to build more complex data structures over general purpose ones, showing a software engineering approach that ensures correctness by using dependent types.

Finger Trees have been implemented in Coq [2] although the paper does not describe any recursive proofs, which are the most important. There is not any implementation yet that provides size information on the Finger Trees. I hope that, via this dissertation I can provide a complete dependent implementation of Finger Trees. Once they are proved correct, I can build up towards any data structure that their monoidal structure allows. Some examples in the literature are heaps, sorted sequences, concatenable dequeues etc.

The objective of this project is to ultimately write code that is proven to be bug free and efficient such that it can at some point be made available to the developers of the Agda

standard library. In case the Finger Trees dependent implementation appears to be unfeasible, there is a fallback plan on starting to implement Okasaki's numerical representation inspired data structures, defined in Chapter 9 of [1]. This will have the same structure as before, building a layering of proved data structures.

The project will start with an introduction section which will provide all the necessary background for understanding dependently typed programming, the data structures that are chosen and what work has currently been done in this field, as well as an introduction to Agda and the libraries that will be used.

The first chapter will be devoted to showing how dependent typing in Agda can be used to ensure correctness by following through the simple classic example of vectors, which has been studied before in the past, introducing the concepts that will be used in the Implementation phase. This will be shown in contrast with the classic functional list implementation. Proofs of associatativy of concatenation and distributivity of length over concatenation will also illustrate how proofs and implementation can be incorporated in the same environment with Agda.

The implementation phase will introduce Finger Trees and also provide their complete implementation, as well as all the associated concepts. On top of this, I will implement a selection of more specialized data structures. It will be interesting to see how proving properties about the specialized data structures require recursively proofs about the base data structures. There are no previous implementations of data structures such as maps or priority queue that can be reused and also guarantee the amortized cost complexity of the structures I propose.

Evaluation will consist of output of the Agda type checker, showing that the code is indeed correct, as well as output of a test program. Numerical results can be derived from the runtime of random tests of increasing lengths, giving insights into the complexity. A qualitative discussion of using dependent types for programming in general can also illustrated with examples from the previous chapters. Furthermore, a dependently typed version of the data structure will be compared against a non-depedently typed version. Although the higher level of abstraction is guaratneed to have some performance drawback, it would be interesting to see by exactly how much the runtime is slowed down. Since there are simmilar data structures implemented in Coq, a comparisons between Agda and Coq could also be part of the evaluation phase. The proof that the new data structures are indeed more efficient than what the Agda standard library provides would also be good as an argument of trying to publish this code for the extension of the standard library.

Variants

Depending on the success of transposing as much of the information as possible in the type of the data structures, it might turn out the other data structures are more illustrative to the process, such as perhaps maps and sets, in which case those will be the chosen data

structures to evolve Finger Trees to. As already mentioned, if the complete dependent implementation of Finger Trees become too cumbersome, the Okasaki's numerical representation inspired data structures are a good and similar in spirit alternative. I would like to stick to the Finger Trees as much as possible because they are already established as a swiss army knife of functional data structures. Some work could be done on trying to specialize on of these structures as much as possible to see how much expressivity can be derived in the chosen paradigm. One example of this is found in [1] exercises, where a sublist is defined by a type that contains information about the underlying list. This could be extended to sets and subsets. The evaluation could also be based on a previously implemented library on computing complexities in a functional environment. [4]

Success Criterion

For the project to be deemed a success the following items must be successfully completed.

1. Agda and Dependent typing should have a convincing justification.
2. The basic, general-purpose data structure should be implemented, illustrating the advantages de dependent typing brings, as well as an explanation of why they are more efficient than standard approaches.
3. This data structure will be supported by proofs.
4. More complex data structures can be built on top. This will enforce correctness throughout.
5. Providing proofs of various mathematical properties of the data structure, whether they exhibit any algebraic structure properties, whether invariants hold, such as insertion of an element does indeed insert the element into the data structure.
6. Further proofs of the invariants that are held by the typing system, proving that there hasn't been made any mistake while writing the type annotations.
7. The data structures should be packaged in a reusable library that should be made available online.
8. Considerations of complexity should be provided.
9. The dissertation must be planned and written.

Timetable and Milestones

Weeks 1 to 5

These first weeks will be used for learning and reading time mostly. I aim to start writing the introduction material that will cover agda and dependent typing introductions, marking my progress as I familiarize myself with the language. By the end of this period, I plan to find out whether a dependently typed implementation of Finger Trees is indeed feasible.

Weeks 6 and 7

Meeting with the supervisor to show the progress, and also set the exact workflow through the data structures

Weeks 8 to 10

Finish implementation of the base data structures and proofs + write-up in the dissertation

Weeks 11 and 12

Finish implementation of the complex data structures and proofs.

Weeks 13 to 19 (including Easter vacation)

Write-up in the dissertation or the previous work. Try to find in literature and example where I can specialize the data structure so much that would truly show the power of the chosen setup (if time). Everything done in this period should be written down.

Weeks 20 to 26

Evaluation phase, careful discussion with examples from literature, Run-time evaluation Type-checking agda output. Interacitve example of coding in a dependently typed programming language.

Weeks 27 to 31

Brush up, write bibliography, add any information that was missed, add information that was discovered after the writing up and has not been included yet. Restructure to make

the dissertation more readable

Weeks 32 to 33

Further improvements related to the material. Meeting with supervisor.

Week 34

Milestone: Submission of Dissertation.

References

[1] Chris Okasaki, *Purely Functional Data Structures*, School of Computer Science, Carnegie Mellon University, September 1996, Pittsburgh, PA 15213

[2] Ralf Hinze, Ross Patterson *Finger trees: A Simple General-Purpose Data Structure*,

[3] LINUS EK OLA HOLMSTRM STEVAN ANDJELKOVIC *Formalizing Arne Andersson trees and Left-leaning Red-Black trees in Agda* Institutionen fr Data- och informationsteknik CHALMERS TEKNISKA HGSKOLA

[4] Nils Anders Danielsson *Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures* Chalmers University of Technology

[5] Ulf Norell and James Chapman. *Dependently Typed Programming in Agda*

[6] Aaron Stump. *Verified Functional Programming in Agda* The University of Iowa, Iowa City, Iowa, USA