Haverford College
Department of Computer Science

"I am the Senate.": An Overview of Practical Applications of Dependent Types with Potential Election Integrity Guarantees

Kevin Jiah-Chih Liao

Submitted in part fulfilment of the requirements for the degree of
Bachelors of Sciences in Computer Science of Haverford College, May 2018

Contents

List of Tables

List of Figures

Chapter 1

Thesis Proposal

1.1 Motivation and Objectives

A function, at its core, is a map from a domain to a co-domain. In other words, we expect there to be a certain set of elements in the universe for which our function can give us a corresponding output. A way to remove certain bugs in programs is to ensure that a function in a program is indeed mapping from the correct set of potential inputs to the set of potential outputs.

One can consider static type systems as a way to narrow down the set of potential inputs to the set of possible outputs. For example, a function that takes in a string and outputs an integer gives certain compile-time guarantees to its programmer. If compilation succeeds, the domain of this function will be strictly limited to an element in the set of all possible strings in the universe and the output will be limited to an element of the set of all possible integers.

However, consider, for example, a function that appends an item to a list. Under a regular type system, we would say that this function takes in a list of elements of type a, an element of type a, and returns elements of type a. A Haskell type signature for this function would look like this:

```
append ::  [a] -> a -> [a].
```

Let's imagine that we have a list data type signature that contains information not only about the type of the elements that the list contains, but also about the length of the list. That is to say, the type signature of a list can be expressed as:

```
List :: Int -> Type

-- A list has an integer denoting length, and the type of its elements.

[1,2,3] :: List 3 Int
```

Now that we've introduced the length of the list type as part of its type signature, we can write a much more strict and bug-free type signature for our append function. Essentially, any append function would take any list with length n and type a. It also takes in an element of type a to append. It outputs a list of length $n + 1$ and type a. This type signature looks like:

```
append :: List n a -> a -> List (n+1) a
```

What's peculiar about this is that the co-domain of this function is not particularly fixed. In fact, it depends on the value of its input. For example, if a list of length 3 and type Int is inputted, the co-domain of our function is the set of all lists with length 4 and type integer. This is an example application of *dependent types*. What we've done is created a function where the co-domain varies as the input value varies. The guarantee of type safety provided by this type signature is substantial.

The goal of dependent types is to write programs with extreme guarantees of compile-time safety. We can use the types of the parameters of a function to place tighter limits on the set that consists of its co-domain, with the co-domain varying depending on the values of the input parameters.

Currently, programming languages designed around dependent types have been implemented, for example: Idris ([Bra11]), Agda ([BDN09], [Nor09]), Coq ([Chl13]), and F*([SHK$^+$16]).There is also significant existing work to bring dependent types into Haskell, an existing more mainstream programming language ([Eis16], [Gun13]). Gundry and Eisenberg both provide interesting examples of how to apply dependent types in practical programming.

Amongst other examples, Eisenberg describes a relational database where queries can be guaranteed to be type safe. This eliminates a class of potential database errors and improves performance since runtime typechecking does not need to occur. A dependently typed relational database would have far-reaching ramifications in fields ranging from web development to defense software and demonstrates the incredible potential present in applying dependent types.

A practical application for dependent types from Gundry's thesis is to eliminate bugs that can arise from improper unit conversions. Units of measurement are already implemented in Microsoft's F# Programming Language ([Ken09]). If numbers carry a type denoting their unit of measurement with them, we can ensure at compile time that improper unit conversions are not going to occur at runtime. These bugs can be catastrophic, as made evident by NASA's loss of a $125-million "Mars Climate Orbiter" when "spacecraft engineers failed to convert from English to Metric units of measurement" [Hot99]. Bringing full dependent types to the Haskell programming language would allow a similar system to be implemented for Haskell.

The goal of my literature review is to explore existing literature around the potential practical applications for dependent types. A spring semester research component of my thesis would involve using a dependently typed language to solve a problem where type safety and guarantees of correctness are critical. While I'm still uncertain about the direction to proceed, I'm interested in looking at elections and e-voting and whether or not we can provide guarantees of correctness to vote counting software written in dependently typed languages. I take a particular interest in the Australian Senate voting verification process because verification of vote count is an NP-complete problem ([CPR+16]).

If we were able to verify that vote counting software is correct at compile-time, we would sidestep the need to run verification code that is trying to solve an np-complete problem. Currently, the Australian government uses proprietary code to count Australian senate ballots and has refused to release the source code after a Freedom of Information Act request ([Tay14]). If an open-sourced, verifiably correct counting program were devised, we could greatly protect the integrity of Australian elections.

Bibliography

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda - a functional language with dependent types. *Theorem Proving in Higher Order Logics*, 2009.

I cite this to show how current work exists to implement dependent types in the Agda programming language. It may contain applications of dependent types that I find useful.

[Bra11] Edwin C. Brady. Idris: Systems programming meets full dependent types. *ACM Workshop on Programming Languages Meets Program Verification*, 2011.

I cite this to demonstrate that dependent types exist in the Idris programming language. In addition, an intersection between dependent types and systems programming may be worth exploring.

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types.* 2013.

I cite this to show that dependent types exist in the Agda programming language.

[CPR+16] Berj Chilingirian, Zara Perumal, Ronald L. Rivest, Grahame Bowland, Andrew Conway, Philip B. Stark, Michelle Blom, Chris Culnane, and Vanessa Teague. Auditing australian senate ballots. 2016.

I cite this to show that auditing Australian Senate Ballots is currently an NP-complete problem. Through type-safe programming, I aim to sidestep this lengthy process. In this paper, Bayesian audits are proposed instead of a full audit in order to get a verifiable result with some degree of mathematical precision.

[Eis16] Richard Eisenberg. *Dependent Types in Haskell: Theory and Practice.* PhD thesis, University of Pennsylvania, 2016.

Richard Eisenberg's work, as mentioned in the thesis proposal, provides an account of how dependent types would look in Haskell. He also provides examples of practical applications of dependent types, such as a type-safe database. This work forms one of the foundations of my research.

[Gun13] Adam Michael Gundry. *Type Inference, Haskell and Dependent Types.* PhD thesis, University of Strathclyde, 2013.

As mentioned in the thesis proposal, Gundry provides an earlier proposal for how dependent types could look in Haskell. Like Eisenberg, Gundry provides examples of dependent type applications such as removing unit conversion errors, which I describe briefly in my proposal.

[Hot99] Robert Lee Hotz. Mars probe lost due to simple math error. *Los Angeles Times*, 1999.

I am only citing this to show an example of what would happen in a worst-case unit conversion error.

[Ken09] Andrew Kennedy. Types for units-of-measure. *Microsoft Research, Cambridge, UK*, 2009.

I cite this while discussing current support for units of measure in F#. One potential path for my thesis is to explore units of measurement with dependent types. Such a thesis would involve drawing from how F# implements units of measurement.

[Nor09] Ulf Norell. Dependently typed programming in agda. *International Workshop on Types in Language Design and Implementation*, 2009.

I cite this to show dependent types exist in the Agda programming language.

[SHK$^+$16] Nikhil Swamy, Ctlin Hricu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cdric Fourneta nd Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Bguelin. Dependent types and multi-monadic effects in f*. *Principles of Programming Languages, POPL*, 2016.

I cite this to show that dependent types exist in the F* programming language.

[Tay14] Josh Taylor. Senate calls for release of aec vote count source code. *ZDNet*, 2014.

Cited to show that the current code for the Australian Senate vote counting program is proprietary.