Haverford College
Department of Computer Science

How to Make Your Programs Very Safe: An Overview of Practical Applications of Dependent Types

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

Literature Review

1.1 Background

A dependently typed programming language can have functions with types that depend on a value. 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 vect (list with length-in-type) can be expressed as:

Figure 1.1: Using a Vect data type.

```
Vect :: Int -> Type
-- A list has an integer denoting length,
-- and the type of its elements.
[1,2,3] :: Vect 3 Int
```

Now that we've introduced the length of the vect 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 vect 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 ::  Vect n a -> a -> Vect (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.

In this literature review, I will explore existing literature around practical real-world applications of dependent types. I'll take a look at three examples where a domain specific language can be built if a language can support full dependent types. I'll then show how dependent types can be applied to make systems programming and building distributed systems safer. I'll also take a look at how a dependently typed language can implement units of measurement, preventing a set of potentially costly and fatal human errors. The hope is to demonstrate that dependent types, long confined to theoretical mathematics, have tremendous promise in helping programmers build reliable and safe programs.

1.2 Dependent Types: A History

Dependent types in programming languages have their roots in intuitionist type theory or Martin Lof Type Theory [Martin-Lf, 1982]. This type theory serves as a foundation for *constructive mathematics* [McKubre-Jordens,]. Per Martin Lof was interested in a type theory that could be used as a programming language, where all well-typed programs must terminate [Nordstrom et al., 1990]. Sets in Martin-Lof Type Theory are viewed as problems and elements potential solutions to a problem. Since all programs terminate in Martin-Lof Type Theory and a set contains all potential solutions to a problem, the requirements for total correctness are satisfied [Nordstrom et al., 1990].

A dependently typed programming language based on foundations in Martin-Lof Type Theory called NuPrl was first released in 1984 [Constable et al., 1985]. NuPrl is used as a "proof assistant" that helps mathematicians and programmers formalize proofs [prl,]. Dependently typed proof assistants like NuPrl found a home at the intersection between programming language enthusiasts interested in total program correctness and constructive mathematicians interested in systems where mathematical formulisms could be systematically encoded. Other proof assistants with support for dependent types followed: Coq (1989) [Girard et al., 1989], ALF (1990) [Magnusson, 1995], Cayenne (1998) [Augustsson, 1998], Agda (1999) [Coquand and Coquand, 1999].

While there are now robust theorem provers that incorporate dependent types, work now is primarily concerned with bringing them into mainstream programming and software development. Idris (2011) was designed with general purpose programming in mind [Brady, 2017]. F* (2011) was introduced by Microsoft as a dependently typed language specifically designed around solving problems in secure distributed programming [Swamy et al., 2011]. In addition to the development of new programming languages with dependent type systems built into the language by design, work exists to mainstream dependent types into more prominent programming languages. The most active and promising mainstreaming work is on Haskell [Eisenberg, 2016, Gundry, 2013].

1.3 Implementing Domain-Specific Languages with Dependent Types

1.3.1 Cryptol: A DSL for Cryptography

Dependent type systems have potential applications in easily implementing domain specific languages (DSLs). Cryptol, for example, is a domain-specific language designed around cryptography ([Inc, 2002]). Problems inherent in implementing a Cryptol compiler or interpreter can be solved through dependent types ([Oury and Swiestra, 2008]). Cryptol is a functional programming language with advanced support for pattern matching. Since cryptography commonly requires dealing with low-level bit manipulation, it follows that Cryptol is designed around facilitating these operations and making them safe. A function that does this sort of low-level manipulation is the `swab` function, which takes in a 32-bit word and swaps the first two bytes ([Inc, 2002]).:

```
swab ::  Word 32 -> Word 32swab [a b c d] = [b a c d]
```

Ideally, a word would be represented by a vector of 32-bits. We would be able to declare a pattern match with swab that looks similar to the declaration presented by Oury and Swiestra above. How then does the compiler understand that this pattern match declaration means we expect the input vector to be divided into 4 separate vectors of 8 bits? This is where dependent types serve a practical purpose. By specifying types that split the length of the vector up into a multiple of two scalars, we can effectively implement this clever pattern match, allowing for powerful pattern matching required by the Cryptol language ([Oury and Swiestra, 2008]).

More coming in final draft of literature review

1.3.2 PBM: Generating Parsers with Data Description Languages

Work also exists to use dependent types in creating embedded data description languages, which are languages where a programmer can describe the structure of data and quickly generate a working parser [Oury and Swiestra, 2008]. For example, consider the portable bitmap (pbm) file format, which consists simply of “P4”, followed by the dimensions of the image in pixels

Figure 1.2: Universe declaration in Idris [Oury and Swiestra, 2008], Idris implementation by [Bailey, 2016].

```
data Bit : Type where
    O : Bit
    I : Bit

data U: Type where
    BIT : U
    CHAR : U
    NAT : U
    VECT : Nat -> U -> U

el : U -> Type
    el BIT = Bit
    el CHAR = Char
    el NAT = Nat
    el (VECT n u) = Vect n (el u)
```

as n, m integers separated by a space. After a newline, hde image is described as a string of $n \times m$ bits where 1 is black and 0 is white [pmb,]. If a parser were generated from a data description language, we expect the parser to either return well-typed data (a vector of bits and the dimensions of the image) or to signal that the data is not well-formed. In other words, if we want to generate parsers through embedded data description languages, we could specify the file format as a value. The type of the generated parser then, would depend on the file format as a value, making this an appropriate area to apply dependent types [Oury and Swiestra, 2008].

We can start by defining our *universe* (see Figure 1.2), which contains all the types that our parser will be manipulating in some way. We also define a function *el*, which will take any value of type U and convert it to an appropriate type. The combination of this data type declaration and this *el* function is a definition of the relevant universe for this problem domain [Oury and Swiestra, 2008].

From here, we can define a *Format* data type that enables us to describe the format of our data. When sequencing formats, we want two binary operators that either read or skip. To skip means to skip over the first parameter and generate a type for the file format from the second parameter. To read means to build a type from both parameters before moving on. We will need to define both these operations, a base operation that gives us a type, a terminal, and rejection if the input data is badly formed. We can declare such a type as follows:

In the code of Figure 1.3 one thing noteworthy is the $(**)$ operator, which is Idris syntactic sugar for a dependent pair. A dependent pair (a : A ** P) means that the type variable a is of type A and can also occur in the type P [de Muijnck-Hughes, 2015]. For example, consider Figure 1.4. The example will only type check correctly iff the natural number present in the first element of the pair is the same as the length of the list.

Having specified a data type that lets us declare formats, we can then move on to creating a specification. In Figure 1.5, a format for the PBM spec is provided. We are now able to write a parser that takes in a Format as a data type, and then is able to parse files to generate

Figure 1.3: Format data type in Idris [Oury and Swiestra, 2008], Idris implementation by [Bailey, 2016].

```
data Format : Type where
Bad : Format
End : Format
Base : U -> Format
Plus : Format -> Format -> Format
Skip : Format -> Format -> Format
Read : (f : Format) -> (Fmt f -> Format) -> Format

Fmt : Format -> Type
Fmt Bad = Void
Fmt End = Unit
Fmt (Base u) = el u
Fmt (Plus f1 f2) = Either (Fmt f1) (Fmt f2)
Fmt (Read f1 f2) = (x : Fmt f1 ** Fmt (f2 x))
Fmt (Skip _ f) = Fmt f

(>>) : Format -> Format -> Format
f1 >> f2 = Skip f1 f2

(>>=) : (f : Format) -> (Fmt f -> Format) -> Format
x >>= f = Read x f
```

Figure 1.4: Example for Dependent Pairs taken from the Idris documentation.

```
vec : (n : Nat ** Vect n Int)
vec = (2 ** [3, 4])
```

Figure 1.5: Format declaration of PBM [Oury and Swiestra, 2008], Idris implementation by [Bailey, 2016].

```
export
pbm : Format
pbm = char 'P' >>
      char '4' >>
      char ' ' >>
      Base NAT >>= \n =>
      char ' ' >>
      Base NAT >>= \m =>
      char '\n' >>
      Base (VECT n (VECT m BIT)) >>= \bs =>
      End
```

well-typed data. See Figure 1.6. This straightforward parsing code is aided by the types that we declared before, skipping, reading, and terminating where required by our file format specification. We can use Idris' REPL to see how the parser deals with our PBM specification in Figure 1.7. Here, we see that the type signature of the function created by giving the parse function our pbm specification is a function that takes in a list of bits and returns a matrix of bits with sizes bound by the natural numbers that we first specified in the file format.

In this section, we show that dependent types allow us to create embedded data description languages inside of a dependently typed language. We can then generate well-typed, reliable parsers without having to rewrite a lot of code. Thus, using dependently typed languages to write parsers with embedded data description languages both demonstrates promise in brevity and also safety.

1.3.3 Safer Databases: Relational Algebras

Additional work exists on relational algebras in dependent types [Oury and Swiestra, 2008], [Eisenberg, 2016]. A relational algebra would allow a programmer to interface with a database where all queries are guaranteed at compile time to be type safe. This eliminates a class of potential database errors and improves performance since runtime typechecking does not need to occur.

1.4 Systems Programming with Dependent Types

General purpose dependently typed programming languages such as Idris (and in the future, Haskell), allow programmers to integrate dependent types into lower level work than a theorem

Figure 1.6: Parser declaration [Oury and Swiestra, 2008], Idris implementation by [Bailey, 2016].

```
parse : (f : Format) -> List Bit -> Maybe (Fmt f, List Bit)
parse Bad bs = Nothing
parse End bs       = Just ((), bs)
parse (Base u) bs  = read u bs
parse (Plus f1 f2) bs with (parse f1 bs)
| Just (x, cs)   = Just (Left x, cs)
| Nothing with (parse f2 bs)
    | Just (y, ds) = Just (Right y, ds)
    | Nothing      = Nothing
parse (Skip f1 f2) bs with (parse f1 bs)
| Nothing        = Nothing
| Just (_, cs)   = parse f2 cs
parse (Read f1 f2) bs with (parse f1 bs)
| Nothing        = Nothing
| Just (x, cs) with (parse (f2 x) cs)
    | Nothing      = Nothing
    | Just (y, ds) = Just ((x ** y), ds)
```

Figure 1.7: Putting the PBM spec into Idris' REPL

```
*Parser> :t parse pbm
parse pbm :
    List Bit -> Maybe
        ( (x : Nat ** x1 : Nat ** x2 : Vect x (Vect x1 Bit) ** ())
        , List Bit)
```

Figure 1.8: Example of units of measurement in F#

```
[<measure>] type inch
[<measure>] type kg
[<measure>] type m
[<measure>] type s}
```

proving language would allow. This section of the literature review will take a look at Brady's *Idris: Systems Programming Meets Full Dependent Types* [Brady, 2011].

1.5 Well-Typed Unit Measurements with 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 ([Kennedy, 2009]). 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" [Hotz, 1999].

Currently, in F#, units of measure are defined with the `[<measure>]` attribute followed by the `type` reserved keyword (see Figure 1.8) [Carter et al., 2016]. We can define simple, non-derived units and also derived units.

Derived units are built out of more elemental definitions of units of measurement. See the definition of Newtons provided below:

```
[<measure>] type N = kg * m/s
```

The benefits of a type system that support units of measurement should become apparent from this quick tour of F#'s support for such a system. These types support decidable equality by definition. Two typed variables can only be equal because they have the same unit of measurement or derived unit of measurement and the same value. This means that errors of conversion between units will now be caught by our compiler, like the one below:

```
let distanceTraveled = 1.5<inch>
let velocityOfSpaceCraft :  float<m/s> = distanceTraveled/60<s>
```

While units of measurement are implemented as a feature in the F# language, which is not dependently typed, a dependently typed programming language would allow for a units of measurement system to be implemented [Gundry, 2013]. Gundry invites us to consider a system for describing units in terms of a constructor that allows us to both enumerate elementary units and also express derived units in terms of one another.

For now, unit only supports three elementary units (metres, seconds, kilograms), but one can imagine a full library implementing the entire SI Units system. Each elementary unit is implemented as a single 1 in the call to the Unit constructor with all entries as zero. Thus, we can express derived units in a call to the Unit constructor where negative integers would

Figure 1.9: Basic SI unit declarations in adapted from Dependent Haskell to Idris [Gundry, 2013]

```
data Unit : Int -> Int -> Int -> Type

Dimensionless : Type
Dimensionless = Unit 0 0 0

Metres : Type
Metres = Unit 1 0 0

Seconds: Type
Seconds = Unit 0 1 0

Kilograms: Type
Kilograms = Unit 0 0 1
```

represent elementary units present in the denominator. Newtons, a derived unit of $\frac{kg \times m}{s^2}$ can be expressed as it is above.

We can define quantities as a type containing a `Unit` and an integer. This then allows us to write simple constructors for the quantity type. We can then define well-typed multiplication and addition operations giving us similar guarantees to that which is given by units of measurement in F#.

Figure 1.10: Quantity as a type (dependent Haskell with Inch) [Gundry, 2013]

```
newtype Quantity u a = Q { value :: a}
metres :: a -> Quantity Metres a
seconds :: a -> Quantity Seconds a
kilograms :: a -> Quantity Kilograms a
(dimensionless, metres, seconds, kilograms) = (Q,Q,Q,Q)

plus :: Num a => Quantity u a -> Quantity u a -> Quantity u a
plus (Q x) (Q y) = Q (x + y)
```

As defined above, this enforces well-typed addition, requiring that two additions be of the same type. Additional work exists to incorporate prefixes (milli, centi, etc.) in such a way that adding units with different prefixes can be easily done [Gundry, 2013]. What we've shown here is that while units of measurement can be first-class features in a programming language like F#, a dependently typed language allows us to build certain functionality easily into the language without changing the language specification whatsoever.

1.6 Programming Distributed Systems

To be included in the final literature review. A review of "Secure Distributed Programming with Value-Dependent Types". [Swamy et al., 2011]

1.7 Proposal for Future Work

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 ([Chilingirian et al., 2016]).

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 ([Taylor, 2014]). If an open-sourced, verifiably correct counting program were devised, we could greatly protect the integrity of Australian elections.

1.8 Conclusion

While many literature reviews begin by looking examining a problem and looking for existing solutions, this literature review takes an opposite approach. The broader problem we are trying to answer is one that crosses various domains and engineering fields. To put it quite simply, *programs crash.* Dependent typed languages, long a toy for theoretical computer scientists and constructivist mathematicians, are increasingly becoming realistic tools to write code with necessary guarantees of correctness. In other words, dependent types are a solution in search of a problem.

In this literature review, I offered a brief summary as to what dependent types are and what languages exist where dependent type functionality is available. I then moved on to describe different applications of dependently typed programming that exist in literature. I started by looking at Cryptol, a DSL for cryptography, and showed how dependent types allow for implementing complex pattern-matching that the language requires [Oury and Swiestra, 2008]. I then moved on to discuss embedded data description languages, showing how one can describe how data is structured and generate a parser out of such a description [Oury and Swiestra, 2008]. I also examined the potential of dependent types to build a typesafe database, eliminating runtime typechecking and thus reducing error and increasing performance [Oury and Swiestra, 2008, Eisenberg, 2016].

Outside of domain specific languages, I also showed the application of dependent types to systems programming [Brady, 2011], building distributed systems [Swamy et al., 2011] and units of measurement [Gundry, 2013]. In this wide-ranging review, I've demonstrated that as dependent

types become brought into the mainstream, they have the potential to empower programmers to build safe, robust programs in ways that have not been possible before.

Bibliography

[prl,] The nature of the prl project. `http://www.nuprl.org/Intro/intro.html"`. Accessed: 2017-10-28.

[pmb,] The pbm format. `http://netpbm.sourceforge.net/doc/pbm.html`. Accessed: 2017-10-25.

[Augustsson, 1998] Augustsson, L. (1998). Cayenne—a language with dependent types. *SIGPLAN Not.*, 34(1):239–250.

[Bailey, 2016] Bailey, E. (2016). the-power-of-pi, cryptol.lidr. `https://github.com/yurrriq/the-power-of-pi/blob/master/src/Data/Cryptol.lidr`.

[Brady, 2017] Brady, E. (2017). *Type-Driven Development with Idris*. Manning Publications.

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

[Carter et al., 2016] Carter, P., Latham, L., and Wenzel, M. (2016). Units of measure. `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/units-of-measure`.

[Chilingirian et al., 2016] Chilingirian, B., Perumal, Z., Rivest, R. L., Bowland, G., Conway, A., Stark, P. B., Blom, M., Culnane, C., and Teague, V. (2016). Auditing australian senate ballots.

[Constable et al., 1985] Constable, R., Allen, S., Bromley, H., Cleaveland, W., Cremer, J., Harper, R., Howe, D., Knoblock, T., Mendler, N., Panangaden, P., Sasaki, J., and Smith, S. (1985). Implementing mathematics with the nuprl proof development system.

[Coquand and Coquand, 1999] Coquand, C. and Coquand, T. (1999). Structured type theory.

[de Muijnck-Hughes, 2015] de Muijnck-Hughes, J. (2015). Types and functions. `https://github.com/idris-lang/Idris-dev/blob/master/docs/tutorial/typesfuns.rst`.

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

[Girard et al., 1989] Girard, J., Lafont, Y., and Taylor, P. (1989). *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

[Gundry, 2013] Gundry, A. M. (2013). *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde.

[Hotz, 1999] Hotz, R. L. (1999). Mars probe lost due to simple math error. *Los Angeles Times*.

[Inc, 2002] Inc, G. (2002). *Cryptol Reference Manual*.

[Kennedy, 2009] Kennedy, A. (2009). Types for units-of-measure. *Microsoft Research, Cambridge, UK*.

[Magnusson, 1995] Magnusson, L. (1995). The implementation of alf - a proof editor based on martin-lf's monomorphic type theory with explicit substitution.

[Martin-Lf, 1982] Martin-Lf, P. (1982). Constructive mathematics and computer programming. In Cohen, L. J., o, J., Pfeiffer, H., and Podewski, K.-P., editors, *Logic, Methodology and Philosophy of Science VI*, volume 104 of *Studies in Logic and the Foundations of Mathematics*, pages 153 – 175. Elsevier.

[McKubre-Jordens,] McKubre-Jordens, M. Constructive mathematics.

[Nordstrom et al., 1990] Nordstrom, B., Petersson, K., and Smith, J. M. (1990). *Programming in Martin-Lof's Type Theory: An Introduction*. Oxford University Press.

[Oury and Swiestra, 2008] Oury, N. and Swiestra, W. (2008). The power of pi. *ACM SigPlan Notices*.

[Swamy et al., 2011] Swamy, N., Chen, J., Fournet, C., Strub, P.-Y., Bhargavan, K., and Yang, J. (2011). Secure distributed programming with value-dependent types.

[Taylor, 2014] Taylor, J. (2014). Senate calls for release of aec vote count source code. *ZDNet*.