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:

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

Figure 1.1: Using a Vect data type.

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

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 intuitionistic type theory or Martin-Löf Type Theory [23, 20]. This type theory serves as a foundation for *constructive mathematics* [22]. Per Martin-Löf was interested in a type theory that could be used as a programming language, where all well-typed programs must terminate [20]. His type theory is based on the principles of constructive mathematics, mainly Curry-Howard isomorphism. Curry-Howard isomorphism is the notion that there is a direct correspondence between mathematical proofs and type theory. In other words, a type signature is synonymous with a mathematical proposition, and if a valid program satisfying the constraints of such a type signature exists, then a proof proving the corresponding mathematical proposition [21, 20]. To illustrate this, consider the examples in Figure 1.2 [1].

From the figure, we see the Curry-Howard isomorphism that forms the fundamentals of Martin-Löf's type theory. We see that in all four cases, if there is a function that inhabits the type signature, the mathematical proposition presented is valid and if a function does not exist, the proposition is invalid.

Mathematicians took an interest in creating a programming language based on Curry-Howard isomorphism and Martin-Löf type theory, since Curry-Howard isomorphism meant that a valid

[1] These examples were provided by Prof. Richard Eisenberg in a discussion.

Figure 1.2: Comparison between Haskell type signatures and mathematical proofs to illustrate Curry-Howard Isomorphism

Haskell Type Signature	Math Proposition	Haskell type signature inhabited?	Proof exists?
$\forall a.a \rightarrow a$	$p \rightarrow p$	True	True
$\forall ab.(a, b) \rightarrow a$	$(p \wedge q) \rightarrow p$	True	True
$\forall ab.a \rightarrow b$	$p \rightarrow q$	False	False
$\forall ab.a \rightarrow (a, b)$	$p \rightarrow (p \wedge q)$	False	False

function that inhabits a type signature could be equivalent to a proof. A dependently typed programming language based on foundations in Martin-Löf Type Theory called NuPrl was first released in 1984 [10]. NuPrl is used as a *proof assistant* that helps mathematicians and programmers formalize proofs [1]. 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) [14], ALF (1990) [19], Cayenne (1998) [3], Agda (1999) [11].

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 [5]. F* (2011) was introduced by Microsoft as a dependently typed language specifically designed around solving problems in secure distributed programming [25]. 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 [13, 15].

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 ([17]). Problems inherent in implementing a Cryptol compiler or interpreter can be solved through dependent types ([24]). 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 ([17]).:

```
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 ([24]).

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 [24]. For example, consider the portable bitmap (pbm) file format, which consists simply of "P4", followed by the dimensions of the image in pixels 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 [2]. 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 [24].

We can start by defining our *universe* (see Figure 1.3), 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 [24].

Figure 1.3: Universe declaration in Idris [24], Idris implementation by [4].

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

data U : Type where
    STRING : U
    BOOL : U
    CHAR : U
    NAT : U
    VECT : Nat -> U -> U

el : U -> Type
    el STRING     = String
    el BOOL       = Bool
    el CHAR       = Char
    el NAT        = Nat
    el (VECT n u) = Vect n (el u)
```

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:

Figure 1.4: Format data type in Idris [24], Idris implementation by [4].

```
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
```

In the code of Figure 1.4 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 [12]. For example, consider Figure 1.5. 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 1.6, 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 well-typed data. See Figure 1.7. 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 1.8. 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.

Figure 1.5: Example for Dependent Pairs taken from the Idris documentation.

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

Figure 1.6: Format declaration of PBM [24], Idris implementation by [4].

```
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
```

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.

Figure 1.7: Parser declaration [24], Idris implementation by [4].

```
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 1.8: 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)
```

1.3.3 Safer Databases: Relational Algebras

Databases are one of the pillars of modern software. Databases are employed in everything from social media software [8] to flight scheduling and booking [27]. Different programming languages have varying interfaces for querying an SQL database. Oftentimes, a query and response interface simply asks a user to send and receive a SQL query and as a string and to process a response given as a string. While this offers flexibility, it means that there is no compile-time guarantee of correctness and that poorly written SQL queries result in run-time crashes, rather than compile-time debugging. We would like to be able to easily compose types from a set of values (database schema), which would allow a programmer to easily compose type-safe queries with compile-time verification of correctness [24]. Many SQL interfaces do runtime type checking, meaning if we can avoid these type checks, there may be performance gains.

We can start by defining types to represent a database schema. A database table is a row of elements that correspond to a declared schema. A schema is a list of attributes that each element should have. An attribute is simply a column name and the type of what the column contains. Thus, we can declare an attributes and schemas as follows in Figure 1.9.

Where U refers to the universe that we built in an earlier example. Given that we built a schema, we are now able to define a schema to hold students. Take, for example, a schema that stores a student's name, student id, and class year. Our next job is to be able to express a database table with a row of instances of a schema. This is reflected in our declaration of the `Row` type, which lets us join rows together, ending in an EmptyRow. An example `kevin` is provided.

Figure 1.9: Declaration of schema. Idris adaption of code from Power of Pi [24]

```
Attribute : Type}
(String, U)}

Schema : Type}
List Attribute}

Students : Schema}
Students = (   ("name", STRING)}
            :: ("id", VECT 6 CHAR)}
            :: ("classyear", VECT 4 CHAR)}
            :: Nil)}

data Row : Schema -> Type where
    EmptyRow : Row Nil
    ConsRow : el u -> Row s -> Row ((name, u) :: s)

kevin : Row Students
kevin = ConsRow "Kevin Jiah-Chih Liao"
    (ConsRow ('1'::'2'::'3'::'4'::'5'::'6'::Nil)
    (ConsRow ('2'::'0'::'1'::'8'::Nil) EmptyRow))
```

Figure 1.10: Declaration of connect. Idris adaption of code from Power of Pi [24]

```
-- Connect takes in a server name, a table name,
-- and a schema, and executes IO operations on that table.
-- This type ensures that upon connection, if successful,
-- the table on the server must have the same schema.
connect : String -> String -> (s : Schema) -> IO (Handle s)


-- Only modeling a single 'read' operation.
data RA : Schema -> Type where
    Read : Handle s -> RA s

-- Takes in a query written with our relational algebra and then sends
   back a table that corresponds to our schema.
toSQL : RA s -> IO (List (Row s))
```

Now that we've defined the schema and tables, it's time to show how we connect and query the table. Usually this means that we have a function `connect` that takes in a servername and tablename as strings, and a SQL query as a string, before returning a string in an IO monad. Now that we've defined a schema type, we can build a well-typed database connection that validates that the database table we are connecting to is of the same schema as the schema we are requesting. If connection succeeds, that means that all subsequent requests with the `Handle schema` type returned by the connect function must be safe, because we know that the schemas we are reading or writing from the table must correspond to the schema in our code. This connect function is defined in Figure 1.10.

Finally, we define a type "Relational Algebra", which contains the opperations that we would perform in a database query. For the sake of brevity, we've limited it to `Read`. We then have a function that does an IO operation on a database, taking in a query of type Relational Algebra,

Figure 1.11: Displays a unit error that would be caught at compile-time with units of measurement.

```
distanceTraveled : Quantity Inches
distanceTraveled = inches 20

distanceLeft : Quantity Metres
distanceLeft = (metres 1000) - distanceTraveled
```

to return us a database table of consisting of the results of our query.

Thus, in this example, we've shown the potential for dependent types to build a type safe database, where if the schemas match on connection, all subsequent queries should be type safe. While there was some overhead in setting up the types, once the database software is written, declaring a schema with elements in our universe was pretty easy and I anticipate database queries not being significantly more difficult to compose and write. Thus, this is an exciting application of dependent types where programmers can potentially benefit from high-level use of dependently typed software once the more complex moving parts are written.

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 proving language would allow. This section of the literature review will take a look at Brady's *Idris: Systems Programming Meets Full Dependent Types* [6].

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 ([18]). 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" [16].

An implementation of units of measurement should have types that 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 if coding style guidelines enforce that all numeric values must be well-typed with units of measurement, there will be compile-time guarantees that errors of conversion between units of measurement will not occur. See Figure 1.11 for an example of a program that should error.

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 [15]. 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.

Figure 1.12: Basic SI unit declarations in adapted from Dependent Haskell to Idris [15]

```
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

data Quantity u = Q Double

metres : Double -> Quantity Metres
metres v = (Q v)

seconds : Double -> Quantity Seconds
seconds v = (Q v)

kilograms : Double -> Quantity Kilograms
kilograms v = (Q v)

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

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 represent elementary units present in the denominator.

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#.

As defined above, this enforces well-typed addition, requiring that two additions be of the same type. we can also define operations that allow us to express fractional units. For example, a Newton of force is defined as a $kg \times ms^{-2}$. Therefore, if we are able to compose types through multiplication and division, we can express a Newton with our units system. See Figure 1.13.

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.

Figure 1.13: Definition of division and multiplication of dependently typed units of measurement. Ported to Idris from [15]

```
Newtons : Type
Newtons = Unit 1 -2 1

newtons : Double -> Quantity Newtons
newtons val = over
    (times (kilograms val) (metres 1))
    (times (seconds 1) (seconds 1))
```

1.6 Programming Distributed Systems

To be included in the final literature review. A review of "Secure Distributed Programming with Value-Dependent Types". [25]

1.7 Programming with Algebraic Effects

In functional programming, we want to isolate side effects as much as possible to keep our code clear. In Haskell and many other programming languages, side effects like IO, State, Random Number Generation, etc. are handled by Monads <CITATION NEEDED>. If we want to use several Monads at once (our code requires simultaneous handling of different side effects), we are often required to use monad transformers. While this approach works for programs that require one or two transformations between monads, as we bring in more and more side effects, the number of transformation monads we need to write increases quite quickly.

Work exists to sidestep the problem of handling increasingly complex monadic transformations by encoding algebraic effects as a domain-specific language in a dependently typed programming language [7]. We can start by defining an `EFFECT` type, as seen in Figure 1.14. In order for a function to use our effects DSL, it will have to be of type `Eff`, where Eff is a data declaration where the 'execution context' m (optionally a Monad) is specified, a list of side effects, and the program's return type. For example, a function that of the execution context `IO` that throws side effects, does work on STDIO, and maintains an integer state will look something like the function `example` in Figure 1.14.

Figure 1.14: Definition of effect type

```
data EFFECT : Type where
    STATE     : Type -> EFFECT
    EXCEPTION : Type -> EFFECT
    FILEIO    : Type -> EFFECT
    STDIO     : EFFECT
    RND       : EFFECT

data Eff : (m : Type -> Type) -> (es : List EFFECT) ->
    (a : Type) -> Type

example : Eff IO [EXCEPTION String, STDIO, STATE Int] ()
```

We can now apply this small Effects DSL we have defined to work on some simple programs where we need to maintain a side effect of some sort. I will provide an example of a program where we tag each node of a binary tree with a unique ID.

Figure 1.15: Tagging a binary tree with integers. Taken from Brady's work. [7]

```
-- Simple type def of binary tree in Idris
data Tree a = Leaf
            | Node (Tree a) a (Tree a)

-- Takes in a tree and produces a tagged tree with
-- State containing an integer passed inside of
-- the function.
tag : Tree a -> Eff m [STATE Int] (Tree (Int, a))
tag Leaf = return Leaf
tag (Node l x r) = do
    l' <- tag l
    lbl <- get; put (lbl + 1)
    r' <- tag r
    return (Node l' (lbl, x) r')

get : Eff m [STATE x] x

put : x -> Eff m [STATE x] ()

EffM :  (m   : Type -> Type) ->
        (es  : List EFFECT) ->
        (es' : List EFFECT) ->
        (a   : Type) -> Type

runPure : Env id es -> EffM id es es' a -> a

tagFrom : Int -> Tree a -> Tree (Int, a)
tagFrom x t = runPure [x] (tag t)
```

1.8 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 ([9]).

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

1.9 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 [24]. 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 [24]. I also examined the potential of dependent types to build a typesafe database, eliminating runtime typechecking and thus reducing error and increasing performance [24, 13].

Outside of domain specific languages, I also showed the application of dependent types to systems programming [6], building distributed systems [25] and units of measurement [15]. 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

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

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

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

[4] Bailey, E. the-power-of-pi, cryptol.lidr, 2016.

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

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

[7] Brady, E. C. Programming and reasoning with algebraic effects and dependent types. *ICFP '13 Proceedings of the 18th ACM SIGPLAN international conference on Functional programming* (2013).

[8] Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H., Marchukov, M., Petrov, D., Puzar, L., Song, Y. J., and Venkataramani, V. TAO: Facebook's distributed data store for the social graph. 49–60.

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

[10] 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. Implementing mathematics with the nuprl proof development system.

[11] Coquand, C., and Coquand, T. Structured type theory.

[12] de Muijnck-Hughes, J. Types and functions, 2015.

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

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

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

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

[17] Inc, G. *Cryptol Reference Manual.* 2002.

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

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

[20] Martin-Löf, P. An intuitionistic theory of types. *Unpublished Preprint* (1972).

[21] Martin-Löf, P. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, L. J. Cohen, J. Łoś, H. Pfeiffer, and K.-P. Podewski, Eds., vol. 104 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1982, pp. 153 – 175.

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

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

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

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

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

[27] Unterbrunner, P., Giannikis, G., Alonso, G., Fauser, D., and Kossmann, D. Predictable performance for unpredictable workloads. *Proc. VLDB Endow. 2*, 1 (Aug. 2009), 706–717.