Haverford College
Department of Computer Science

How to Teach Someone to Love Vegetables: 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

Abstract

Programs crash. Static type systems serve to make the life of a programmer easier by providing static compile-time guarantees of certain program behaviour, guaranteeing that a function will map from a domain to a co-domain. A dependently typed programming language gives a programmer even additional expressive power, allowing types to be expressed relative to values given at runtime. Thus, the co-domain of our function is dependent upon some value that is provided at runtime.

While research in dependent types has traditionally been in the domains of theoretical mathematics, where researchers use dependently typed theorem provers to write mathematical formulisms, dependent types have applications outside of such a theoretical domain. In my work, I seek to summarize the existing body of work around practical applications of dependent types. This review is a tour of a potential application of dependent types. My review explores the following additional examples:

1. Complicated pattern matching in the Cryptol DSL for cryptographic applications. (Oury & Swierstra, 2008)
2. Generating file format parsers from a data description language. (Oury & Swierstra, 2008)
3. A relational database and algebra. (Oury & Swierstra, 2008; Eisenberg, 2016)
4. An alternative to monadic transformations with an algebraic effects DSL (Brady, 2013)
5. Programming distributed systems with F* and dependent types. (Swamy *et al.*, 2011)
6. A low-level domain-specific language demonstrating the applicability of dependent types to systems programming. (Brady, 2011)

Through a review of the practical applications of dependent types in existing and implemented functional programming languages, I demonstrate that while dependent types are often thought of as far-off and theoretical, they are currently available in advanced functional programming languages and serve practical purposes.

I set out to look for patterns that signal to a programmer that dependent types would be particularly useful for their work. So far, I've found that the problems that I'm exploring seem to be focused around two main themes: the serializing and deserializing of data, and building domain-specific languages.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

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 `replicate` that takes in an integer of value `b` and some element of type `a`. It then *replicates* the element `b` times, producing of length `b` containing copies of the element of type `a`. The Haskell type signature[1] for such a function could look like in Figure 1.1.

```
replicate :: Int -> a -> [a]
```

Figure 1.1: Non-dependently typed type signature for replicate

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 length-indexed vector can be expressed as Figure 1.2.

Now that we've introduced the length of the vect type as part of its type signature, we can write a much tighter type signature for our replicate function. Essentially, any replicate function will

[1]This literature review assumes prior knowledge with Haskell/Idris-style type signatures, since all the examples I've provided are in this style. An introduction is available in the excellent *Real World Haskell*, available here: http://book.realworldhaskell.org/read/types-and-functions.html

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

Figure 1.2: Using a length-indexed vector (Vect) data type.

```
replicate : (len : Int) -> (x : elem) -> Vect len elem
```

Figure 1.3: Dependently typed type signature for replicate

take in an integer with some value `len`, some element `x` of type `elem`, and the function will produce some vector of length `len` holding elements of type `elem`. This type signature is shown in Figure 1.3.

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 an integer of value 3 and some boolean is inputted, the co-domain of our function is the set of all length-indexed vectors with length 3 and type boolean. 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 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 History

Dependent types in programming languages have their roots in intuitionistic type theory or Martin-Löf Type Theory (Nordstrom *et al.*, 1990; Martin-Löf, 1972). This type theory serves as a foundation for *constructive mathematics* (Martin-Löf, 1982). Per Martin-Löf was interested in a type theory that could be used as a programming language, where all well-typed programs must terminate (Martin-Löf, 1972). His type theory is based on the principles of constructive mathematics and is interesting because of the Curry-Howard isomorphism. The Curry-Howard isomorphism is the notion that there is a direct correspondence between mathematical proofs in constructive mathematics and programs. 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 it is a proof proving the corresponding mathematical proposition

(Martin-Löf, 1982; Martin-Löf, 1972). To illustrate this, consider the examples in Figure 1.1. [2].

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

Table 1.1: Comparison between Haskell type signatures and mathematical proofs to illustrate Curry-Howard Isomorphism

From the figure, we see examples of 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. Proving the isomorphism is beyond the scope of this literature review but these are examples that should demonstrate that such an isomorphism exists such that a strong relationship between proving mathematical propositions and programming languages exists.

Mathematicians took an interest in creating a programming language based on the Curry-Howard isomorphism and Martin-Löf type theory, since the Curry-Howard isomorphism meant that a valid 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 (Constable *et al.*, 1986). NuPrl is used as a *proof assistant* that helps mathematicians and programmers formalize proofs (Constable *et al.*, 1986). 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 formalisms could be systematically encoded. Other proof assistants with support for dependent types followed: Coq (1989) (Coquand & Huet, 1988), ALF (1990) (Magnusson, 1994), Agda (1999) (Norell, 2007).

While there are now robust theorem provers that incorporate dependent types, recent work has emerged 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).

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

Chapter 2

Review of Existing Literature

In the previous section, I offered a brief summary of the motivations and history behind dependent typed programming. Having done so, I now offer an introduction to different practical applications of dependent types that allow a programmer to write safer programs through tighter type specifications. These examples are structured in order of the difficulty I personally found in understanding them, and I would greatly encourage readers to read in that order.

2.1 Well-Typed Unit Measurements with Dependent Types

This section is adapted from Adam Gundry's Ph.D Dissertation (Gundry, 2013), among other resources for well-typed unit measurements with dependent types.

My first example is to demonstrate how dependent types can eliminate bugs that can arise from improper unit conversions (Gundry, 2013). 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 the $125-million Mars Climate Orbiter when "spacecraft engineers failed to convert from English to Metric units of measurement" (Hotz, 1999).

```
distanceTraveled : Quantity Kilograms
distanceTraveled = inches 20

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

Figure 2.1: Displays a unit error that would be caught at compile-time with units of measurement.

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 2.1 for an example of a program that should error [1]

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 (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

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)
```

Figure 2.2: Basic SI unit declarations in adapted from Dependent Haskell to Idris (Gundry, 2013)

For now, our data declaration for `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#.

[1]This example should error in the case we've described here, where quantities are indexed by their *unit* and not their *dimension*, however, if the latter is done, a compiler could do a conversion implicitly. This is explored in (Maranushi & Eisenberg, 2014).

As defined above, this enforces well-typed addition, requiring that two values be of the same type. we can also define operations that allow us to express fractional units. The types for `times` and `over` representing division and multiplication are shown in Figure 2.3. In this code, we see that multiplying or dividing two units together at the value-level does an addition operation on all the elementary units at the type level. 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 2.3.

```
times : Quantity (Unit m s g) -> Quantity (Unit m' s' g')
        -> Quantity (Unit (m + m') (s + s') (g + g'))
times (Q x) (Q y) = Q (x * y)

inverse : Quantity (Unit m s g) -> Quantity (Unit (-m) (-s) (-g))
inverse (Q x) = Q (1 / x)

over : Quantity (Unit m s g) -> Quantity (Unit m' s' g')
        -> Quantity (Unit (m -m') (s - s') (g - g'))
over x y = times x (inverse y)

Newtons : Type
Newtons = Unit 1 -2 1

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

Figure 2.3: Definition of division and multiplication of dependently typed units of measurement. Ported to Idris from (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. This system is merely a simplistic proof-of-concept designed to demonstrate to the user that dependently-typed languages can implement units-of-measurement. By allowing integer values to appear as part of a type signature, we've written a rudimentary system that allows a user to write well-typed units. In addition, the user is able to define derived units (see our definition of Newton).

Since derived units have become type-level arithmetic, we know that we will be able to consistently derive units where required in contrast to other methods in non-dependently-typed programming languages. Our implementation is able to derive units for examples that F# is unable to (Gundry, 2013; Carter *et al.*, 2016). At the same time, error messages remain very difficult to debug. Rather than showing the unit names for conversions that fail, Idris will show the type declaration (e.g. Instead of showing `Metres` in an error message, Idris will show `Unit 1 0 0`).

2.2 Safer Databases: Relational Algebras

This section is a summary of one of the sections from Oury Swierstra's "Power of Pi" (Oury & Swierstra, 2008)

Our second example is a type-safe relational algebra Databases are one of the pillars of modern software. Databases are employed in everything from social media software (Bronson *et al.*, 2013) to flight scheduling and booking (Unterbrunner *et al.*, 2009). 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 database table schema, which would allow a programmer to easily compose type-safe queries with compile-time verification of correctness (Oury & Swierstra, 2008). Many SQL interfaces do runtime type checking, meaning if we can avoid these type checks, there may be performance gains.

Name : String	ID : Vect 6 Char	ClassYear : Vect 4 Char
Kevin Liao	123456	2018
David Smith	234567	2018

Table 2.1: Table of students.

ClassName : String	ID : Vect 8 Char
Intro to CS	CMSCH105
Writing Sem	WRPRH101

Table 2.2: Table of classes.

Name : String	ID : Vect 6 Char	ClassYear : Vect 4 Char	ClassName : String	ID : Vect 8 Char
Kevin Liao	123456	2018	Intro to CS	CMSCH105
Kevin Liao	123456	2018	Writing Sem	WRPRH101
David Smith	234567	2018	Intro to CS	CMSCH105
David Smith	234567	2018	Writing Sem	WRPRH101

Table 2.3: Comparison between Haskell type signatures and mathematical proofs to illustrate Curry-Howard Isomorphism

In this example, we are going to build up a simple dependently-typed relational algebra in Idris. This relational algebra is going to allow us to read from a database and also do a cartesian product. Let's say, for example, that I am a registrar at a college have a table of students and a table of classes, and I want to enroll each student in all the classes in the class table. In other words I would want to compute the cartesian product of these tables, making sure that they contain disjoint schema. Such an operation is shown in the following tables. Table 2.1 shows a table of students, Table 2.2 is a table of classes and Table 2.3 is the cartesian product of the two tables.

We can start by defining types to represent a database schema. A database table refers to rows of data that correspond to a declared schema. A schema is a list of attributes that each element

```
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 2.4: Declaration of schema. Idris adaption of code from Power of Pi (Oury & Swierstra, 2008)

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 2.4, 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.

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 2.5.

Finally, we define a type `RA` which denotes expressions in our relational algebra. For the sake of brevity, we've limited it to `Read` and `Product`. We then have a function that does an IO operation on a database, taking in a query of type Relational Algebra, to return us a database table of consisting of the results of our query.

`Product` computes the Cartesian product of two tables, producing a larger table where each row in the first table maps to every row in the second table. This operation only works if the two schemas are disjoint. While it is trivial to write a function `disjoint` that checks to see if two

```
-- 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)

-- Tells us if two schemas are disjoint.
disjoint : Schema -> Schema -> Bool

-- A simple proposition that stops any false proposition from compiling
So Bool -> Type
So True = Unit
So False = Void

-- Only modeling Read and Product RA expressions.
data RA : Schema -> Type where
    Read : Handle s -> RA s
    Product : (So $ disjoint s s') => RA s -> RA s' -> RA (append s 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))
```

Figure 2.5: Declaration of connect. Idris adaption of code from Power of Pi (Oury & Swierstra, 2008)

schemas are disjoint, we want to enforce that product is only a valid RA expression on disjoint schemas. Thus, we want the type of `Product` to enforce that its two schemas are disjoint. In addition, we have a simple proposition `So` that ensures that whatever proposition is supplied to it is valid. In the case of building a cartesian product, we are looking to make sure that the two schemas are disjoint.

Let's say that I now have to do the registrar cartesian product that I described in the beginning of this section. What I'd first do is make two calls to the `connect` function. I will send the schema of my students table in one call and the schema of the classes call in another. If both connections succeed, I will be given two values of type `Handle Student`, `Handle Class` respectively, which give me the certainty that the table contains data conforming to such a schema. I can then feed these two `Handle` values into a `Read` expression, which will give me values of table rows of type `RA Student` and `RA Class`. I can then take the cartesian product. My `Product` expression first checks to make sure that the `Student` schema and `Class` schema are disjoint, which they are, before appending the `Class` schema to the `Student` schema to make the final schema that describes the rows of data held by the results of our cartesian product of two tables.

The crux of how dependent types are applicable in this example lies in the `connect` function. In this example, we can see how the type of the result from our connection to the database table depends on the value of the schema that we pass in. We pass a schema as value, validate that the relevant table contains data conforming to this exact schema. Subsequent expressions in our relational algebra are predicated on database connections, validated with dependent types. In a way, `connect` elevates schema values to the type level and subsequent relational algebra expressions like `Read` or `Product` can then carry out modifications of this schema at the type level.

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 database queries not 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.

We circle back to a question that we've been asking earlier: in what ways does this overhead give us an advantage over an ordinary database driver? Earlier, we were discussing how an ordinary database driver will usually send a query over to the database as a string and receive data as a string. What this means is that a database driver will have to validate whether data conforms to the schema specified by a user on every read operation.

2.3 PBM: Generating Parsers with Data Description Languages

This section is a summary of one of the sections from Oury Swierstra's "Power of Pi" (Oury & Swierstra, 2008)

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 & Swierstra, 2008). For example, consider the portable bitmap (pbm) file format, which consists simply of the characters "P4", followed by the dimensions of the image in pixels as n, m integers separated by a space. After a newline, the image is described as a string of $n \times m$ bits where 1 is black and 0 is white (Kay & Levine, 1994). A sample of a file that conforms to this specification is shown below where $n = 3, m = 4$.

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 & Swierstra, 2008).

```
P4 3 4
0 1 0
1 1 1
1 1 1
0 1 0
```

Figure 2.6: Sample PBM image

We can start by defining our *universe* (see Figure 2.7)[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 maps 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 & Swierstra, 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 shown in Figure 2.8.

In this code, notice how `Fmt` lifts values of type `Format` to the type level. Data that is badly formatted will be of type `Void`, a type showing that it has zero inhabitants. `End` maps to the `Unit` type, which is a data type denoting one inhabitant. `Base` takes in a type at the value from our type universe and uses `el` to bring it to the type level.

[2]Original implementation in (Oury & Swierstra, 2008) was in Agda. An Idris implementation was sourced from https://github.com/yurrriq/the-power-of-pi/blob/master/src/Data/Cryptol.lidr. All Idris code in this section was adapted from this repository.

```
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)
```

Figure 2.7: Universe declaration in Idris (Oury & Swierstra, 2008)

The definition of Read is a little curious. The use of the (∗∗) operator deserves some comment. This is the Idris syntactic sugar for a dependent pair. A simple example for what dependent pairs are is shown in Figure 2.9. The type for the item on the right depends on the value of the item on the left. In this case, we are specifying that the list on the right must contain n items, where n is the value of the natural number on the left. A dependent pair (a : A ** P) means that the variable a is of type A and can also occur in the type P (Brady, 2017). In the case of Read, what we are specifying is that the type of the value on the right may depend on the type of the value on the left.

```
data Format : Type where
Bad : Format
End : Format
Base : U -> 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 (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 2.8: Format data type in Idris (Oury & Swierstra, 2008)

Having specified a data type that lets us declare formats, we can then move on to creating a specification. In Figure 2.10, 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 2.11. As visible in Figure 2.10, our PBM specification simply looks like a description of our file format. We skip over the header P4, and then read in a natural number m and n, before taking in the matrix of bits as a nested length-indexed vector of bits.

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

Figure 2.9: Example for Dependent Pairs taken from the Idris documentation.

```
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
```

Figure 2.10: Format declaration of PBM (Oury & Swierstra, 2008)

```
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 (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 2.11: Parser declaration (Oury & Swierstra, 2008)

The parser in Figure 2.11 recurses through our definition of the file format and input data as a list of bits and ensures that at each step of the way, the data in the list of bits conforms to the file format specification. Our binary Skip operator simply skips over the item on the left and calls parse on the right. Read first does a pattern match on the result of a recursive call to parse on the left side. If parse succeeds, it runs parse on the right side and builds a dependent pair relationship between the type of the left and the right.

We can use Idris' REPL to see how the parser deals with our PBM specification in Figure 2.12. 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.

What has this given us? The application of dependent types in this example is more salient than our previous examples. This entire example hinges on building up a type for parsed data through dependent pairs. We are able to establish the relationship between the values present in our file format description and the subsequent type of the parsed data. We know that our dependently typed parser will give us well-formed data that fits the constraints of our file description or nothing at all. A parser without dependent types could not enforce such constraints in its static type system. Our data parser will either give us Nothing, or well-typed, well-formed code. An ordinary parser will only be able to give us data in the form of an

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

Figure 2.12: Putting the PBM spec into Idris' REPL

untyped blob. In other words, we've gotten both safety and convenience out of the application of dependent types.

Once the heavy-lifting of writing `parse`, `Fmt`, and `Format` has been done, a programmer has to specify a universe and the subsequent definition of a file format is not difficult.

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.

2.4 Cryptol: A DSL for Cryptography

This section is a summary of one of the sections from Oury Swierstra's "Power of Pi" (Oury & Swierstra, 2008)

Dependent type systems have potential applications in easily implementing domain specific languages (DSLs). Cryptol, for example, is a domain-specific language designed around cryptography (Lewis, 2007). Problems inherent in implementing a Cryptol compiler or interpreter can be solved through dependent types (Oury & Swierstra, 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 (Lewis, 2007).:

```
swab :: Word 32 -> Word 32
swab [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 & Swierstra, 2008). The final result looks like Figure 2.13.

```
-- We specify in our type that we are splitting the
-- input word into 4 bytes.
swab : Word 32 -> Word 32
swab xs with (splitView 8 4 xs)
swab _ | Split [a, b, c, d] = concat [b, a, c, d]

data SplitView : {n : Nat} -> (m : Nat)
                -> Vect (m * n) a -> Type where
Split : (xss : Vect m (Vect n a)) -> SplitView m (concat xss)

splitConcatLemma : (xs : Vect (m * n) a)
                    -> concat (split n m xs) = xs

split : (n : Nat) -> (m : Nat) -> Vect (m * n) a -> Vect m (Vect n a)


splitView : (n, m : Nat) ->
            (xs : Vect (m * n) a) -> SplitView m xs
splitView n m xs =
  let prf  = sym (splitConcatLemma xs {m = m} {n = n})
      view = Split (split n m xs) {n = n} in
  rewrite prf in view
```

Figure 2.13: Final swab function (Oury & Swierstra, 2008)

Idris and other dependently typed programming languages provide a `with` keyword that lets you pattern match on the results of applying the inputs of a function to another function. In this example, we are pattern matching on the results of feeding in the input word to a function called `splitview` that splits our word into 8 bytes. There are some constraints on such a function. We expect that `splitView 9 3` will be an illegal call because `splitView` takes in a 32-bit word yet this splitView is splitting a word of size 27 (9×3). We'll spend the remainder of this section exploring what goes into implementing `splitView`.

`splitView`'s implementation is also shown in Figure 2.13. It is predicated on the existence of some function `split`, that splits up an input vector into appropriate-length pieces based on input. splitView is necessary because we cannot simply write `swab` as a call to `split`. If our function 'swab' were simply a call to split, there would be a problem. Since our swab function is a concatenation of the results of split, we have no way of guaranteeing to the compiler that the result length of that would be the same length as the input list. In other words, we cannot guarantee `concat (split n m xs) = xs`, where `xs` refers to the input 32-bit word.

Such a proof is provided by `splitConcatLemma`. The specifics of this inductive proof are provided in (Oury & Swierstra, 2008). The example is provided to illustrate that the machinations to make difficult pattern matching possible with dependent types are non-trivial. However, once these proofs are implemented, someone writing future compilers can very easily draw on them simply by calling the `splitView` function to allow for the definition of complex pattern-matching. I'll reflect on why this is important in my conclusion.

2.5 Programming with Algebraic Effects

This section is a summary of Edwin Brady's "Programming and Reasoning with Algebraic Effects and Dependent Types" (Brady, 2013)

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 (O'Sullivan *et al.*, 2008)[3]. 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 (Brady, 2013). We can start by defining an `EFFECT` type, as seen in Figure 2.14. This `EFFECT` data type enumerates the different kinds of side effects that an Effects DSL could need to handle. 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 2.14.

The code in Figure 2.14 provides an example definition for `State`.

`EFFECT` enumerates different side effects that we'd expect an effects DSL to handle for us. We want to define an `Effect` type which we can express our side effects in terms of, and also, a type class `Handler` that allows us to run expressions in a certain computational context. The definition of these two items are given in Figure **??**. Notice how `Effect` is parameterized by an input type, output type, and type of the computation. Handler allows us to specify some computational context and then run relevant effects. These definitions will become more clear as we implement a `State` effect.

The example that we will be dealing with later uses the `STATE` effect, so we can look at its implementation. We want `STATE` to have a `get` and put operation, of which, we expect `get` to be able to be called an unlimited amount of times without any impact on the type of state. We would like `put` to potentially mutate the type of `STATE`. We therefore, define our two effects in the data class, where Get does not modify the resource held in `State` whatsoever, so its effect looks exactly paramaters are all the same type. `Put`'s mutation takes in a value of type `b` and changes the type held by `State`. As a put operation, we expect nothing to be returned, so the type of the computation is `()`.

Functions with different side effects are invoked from an execution context `Env`. The specifics of how execution contexts are defined are not important, however, we can note that different effects require different execution contexts. `STATE`, for example, is strictly locally confined and thus its execution context does not matter. Meanwhile, `STDIO` does IO operations, and therefore

[3]This part of the literature assumes that you are familiar with Monads, which is how the Haskell programming language handles side effects. For more information, I highly recommend reading O'Sullivan et. al's Real World Haskell or other widely available tutorials on Monads in Haskell.

```
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] ()


data State : Effect where
    Get : State a a a
    Put : b -> State a b ()

get : Eff m [STATE x] x
get = mkEffect Get

put : x -> Eff m [STATE x] ()
put val = mkEffect (Put val)
```

Figure 2.14: Definition of effect type

its execution context must be inside of the IO Monad. We can specify a type class `Handler`. All effects should be instances of this type class. `Handler` enforces that the environment we invoke a function from is the correct one. In other words, if I attempted to invoke a `runpure` function that had an `STDIO` side effect, the program will not compile.

```
Effect : Type
Effect = (res : Type) -> (res' : Type) -> (t : Type) -> Type

class Handler (e : Effect) (m : Type -> Type) where
    handle : res -> (eff : e res res' t) -> (res' -> t -> m a) -> m add

instance Handler State m where
    handle st Get     k = k st st
    handle st (Put n) k = k n ()

instance Handler StdIO IO where
    handle () (PutStr s) k = do putStrs; k () ()
    handle () GetStr     k = do x <- getLine; k () x
```

Figure 2.15: Declaring handlers for side effects. Taken from Brady's work. (Brady, 2013)

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.

This tagging program is a simple function that recurses through a tree. The function `gets` from and `puts` to a state that is kept alive for the duration of the program. Since this side effect is 'created' at function call-time and 'destroyed' on termination, it makes sense to say that this function can be invoked as a pure function. Thus, we have a function with internal side effects

```
-- 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')

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)
```

Figure 2.16: Tagging a binary tree with integers. Taken from Brady's work. (Brady, 2013)

that have been clearly specified. We can see here how specifying side effects as a list of effects rather than with monad transformers means that we can easily add or remove side effects as required.

Isolating side effects is a form of best practice and doing so is not easy. While monads provide a way to handle side effects, programs multiple side effects often require the use of monad transformers, making a programmer more likely to take coarse-grained shortcuts which avoid many of the benefits of monads in the first place. Building an `Effects` DSL allows the programmer to specify effects of a program that executes inside a single execution context. This allows for multiple monadic side effects to be effectively juggled by a programmer.

Chapter 3

Conclusion

3.1 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).

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.

3.2 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 & Swierstra, 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 & Swierstra, 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 & Swierstra, 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.

In all of these applications, I found something incredibly interesting that I'd like to reflect on. I don't deny that the process of understanding all of these papers at the undergraduate-level has not been an easy undertaking. This is important because the underlying discussion question about this literature review has been making dependent types accessible to a practical programmer. However, while writing out inductive proofs in Idris or building a relational algebra are difficult tasks, the result is surprisingly easy for a programmer to use.

1. A programmer looking to implement complicated pattern-matching for a domain-specific language like Cryptol can easily implement that complex pattern-matching, as well as many other complicated pattern-matching techniques once splitView and its associated lemmas are implemented.
2. A programmer wanting to use well-typed units of measurement can simply import a library of SI units without having to worry about proving lemmas that demonstrate two derived units are the same.
3. A programmer wanting to use a type-safe relational database simply has to express schemas and then write expressions in our relational algebra detailing the input and output schema.
4. A programmer wanting to build a parser can simply describe the data using a provided data description language.
5. A programmer who wants to use the algebraic effects DSL can just describe the side effects present in their function without worrying about the complex underlying dependently-typed machinations.

In other words, at the current state, I find dependent types to still be too complicated for the average programmer to understand. What people who understand dependent types can do however, is work on dependently typed libraries where the building blocks of what a programmer might use are dependently typed. A programmer who uses a dependently-typed relational algebra needn't understand the complex underlying machinations of building such a relational algebra. They merely need to understand that writing schemas and expressions in the relational algebra give way to type safety. In other words, the problem of getting every programmer to love and write programs with dependent types is certainly non-trivial, but providing dependently-typed libraries can be easily done. You can't make a kid love vegetables, but nothing's stopping you from slipping a little spinach into their dinner.

References

Brady, Edwin. 2017. *Type-Driven Development with Idris*. Manning Publications Company.

Brady, Edwin C. 2011. Idris: Systems Programming Meets Full Dependent Types. *ACM Workshop on Programming Languages Meets Program Verification.*

Brady, Edwin C. 2013. Programming and Reasoning with Algebraic Effects and Dependent Types. *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming.*

Bronson, Nathan, Amsden, Zach, Cabrera, George, Chakka, Prasad, Dimov, Peter, Ding, Hui, Ferris, Jack, Giardullo, Anthony, Kulkarni, Sachin, Li, Harry, Marchukov, Mark, Petrov, Dmitri, Puzar, Lovro, Song, Yee Jiun, & Venkataramani, Venkat. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. *Pages 49–60 of: The 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX.

Carter, Philip, Latham, Luke, & Wenzel, Maira. 2016. *Units of Measure.*

Chilingirian, Berj, Perumal, Zara, Rivest, Ronald L., Bowland, Grahame, Conway, Andrew, Stark, Philip B., Blom, Michelle, Culnane, Chris, & Teague, Vanessa. 2016. Auditing Australian Senate Ballots.

Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T., & Smith, S. F. 1986. *Implementing Mathematics with the Nuprl Proof Development System.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Coquand, Thierry, & Huet, Gérard. 1988. The calculus of constructions. *Information and computation*, **76**(2-3), 95–120.

Eisenberg, Richard. 2016. *Dependent Types in Haskell: Theory and Practice.* Ph.D. thesis, University of Pennsylvania.

Gundry, Adam Michael. 2013. *Type Inference, Haskell and Dependent Types.* Ph.D. thesis, University of Strathclyde.

Hotz, Robert Lee. 1999. Mars Probe Lost Due to Simple Math Error. *Los Angeles Times.*

Kay, David C., & Levine, John R. 1994. *Graphics File Formats.* 2nd edn. New York, NY, USA: McGraw-Hill, Inc.

Kennedy, Andrew. 2009. Types for Units-of-Measure. *Proceedings of the Third Summer Conference of the Central European Functional Programming School.*

Lewis, Jeff. 2007. Cryptol: Specification, Implementation and Verification of High-grade Cryptographic Applications. *Pages 41–41 of: Proceedings of the 2007 ACM Workshop on Formal Methods in Security Engineering*. FMSE '07. New York, NY, USA: ACM.

Magnusson, Lena. 1994. *The Implementation of ALF: A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. Ph.D. thesis, University of Gothenburg.

Maranushi, Takayuki, & Eisenberg, Richard A. 2014. Experience Report: Type-Checking Polymorphic Units for Astrophysics Research in Haskell. *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, 31–38.

Martin-Löf, Per. 1972. An Intuitionistic Theory of Types. *Unpublished Preprint*.

Martin-Löf, Per. 1982. Constructive Mathematics and Computer Programming. *Pages 153 – 175 of:* Cohen, L. Jonathan, Łoś, Jerzy, Pfeiffer, Helmut, & Podewski, Klaus-Peter (eds), *Logic, Methodology and Philosophy of Science VI*. Studies in Logic and the Foundations of Mathematics, vol. 104, no. Supplement C. Elsevier.

Nordstrom, Bengt, Petersson, Kent, & Smith, Jan M. 1990. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press.

Norell, Ulf. 2007 (September). *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

O'Sullivan, Bryan, Goerzen, John, & Stewart, Donald Bruce. 2008. *Real World Haskell*. " O'Reilly Media, Inc.".

Oury, Nicolas, & Swierstra, Wouter. 2008. The Power of Pi. *Pages 39–50 of: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP '08. New York, NY, USA: ACM.

Swamy, Nikhil, Chen, Juan, Fournet, Cédric, Strub, Pierre-Yves, Bhargavan, Karthikeyan, & Yang, Jean. 2011. Secure Distributed Programming with Value-dependent Types. *Pages 266–278 of: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. New York, NY, USA: ACM.

Taylor, Josh. 2014. Senate Calls for Release of AEC Vote Count Source Code. *ZDNet*.

Unterbrunner, P., Giannikis, G., Alonso, G., Fauser, D., & Kossmann, D. 2009. Predictable Performance for Unpredictable Workloads. *Proc. VLDB Endow.*, **2**(1), 706–717.