How to Make Your Programs Very Safe

A Review of Practical Applications of Dependent Types

Kevin Jiah-Chih Liao*
Haverford College
370 Lancaster Avenue
Haverford, Pennsylvania 19041
me@liaokev.in

ACM Reference format:
Kevin Jiah-Chih Liao. 2018. How to Make Your Programs Very Safe. In *Proceedings of ACM Principles of Programming Languages, Los Angeles, California, USA, January 2018 (POPL '18)*, 3 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

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. For example, consider a function *replicate*, that takes in any Peano natural number *n* and returns a length-indexed vector of *n* Peano natural numbers of the value *n*. We can express this in Idris, a dependently typed functional programming language, as follows in Figure 1.

Figure 1. Implementation of Replicate in Idris

```
-- replic 3 will give us (3 :: 3 :: 3 :: Nil)
replic : (n : Nat) -> Vect n Nat
replic n = replHelp n n where
  replHelp : (n : Nat) -> Nat -> Vect n Nat
  replHelp Z _ = Nil
  replHelp (S n) val = (val :: replHelp n val)
```

Thus, the co-domain of our function is dependent upon some value that is provided at runtime. Calling `replic 5` will limit the co-domain of the function to all length-indexed vectors of length 5, etc. Having written this type, a programmer now has compile-time guarantees that the logic for a function was expressed correctly and behaviour will correspond closer to what is expected.

While research in dependent types has traditionally been in the domains of theoretical mathematics, where researchers use dependently typed theorem provers to write mathematical formalisms, dependent types have applications outside of such a theoretical domain. In my work, I seek to find patterns in the existing body of literature around practical applications of dependent types. My goal is to enumerate signs that a programmer should look for that the solution to a problem could benefit from the use of dependent types.

2 Work Covered

This review is a tour of potential applications of dependent types. In this abstract, I have chosen a brief example, units of measurement, in order to demonstrate their potency. However, in addition to units of measurement, dependent types have far ranging applications. A poster of my research would contain a couple of the applications which I am exploring:

1. Complicated pattern matching in the Cryptol DSL for cryptographic applications. [9]
2. Generating file format parsers from a data description language. [9]
3. A relational database and algebra. [4, 9]
4. An alternative to monadic transformations with an algebraic effects DSL [2]
5. Programming distributed systems with F* and dependent types. [10]
6. A low-level domain-specific language demonstrating the applicability of dependent types to systems programming. [1]

3 Units of Measurement

This section is adapted from Adam Gundry's Ph.D Dissertation [5], 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 [5]. Units of measurement are already implemented in Microsoft's F# Programming Language [7]. 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" [6].

*ACM ID: 2142219. Undergraduate. Prepared under the advisement of Richard Eisenberg at Bryn Mawr College.

POPL '18, January 2018, Los Angeles, California, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Figure 2. Displays a unit error that would be caught at compile-time with units of measurement.

```
distanceTraveled : Quantity Kilograms
distanceTraveled = inches 20

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

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 for an example of a program that should error. This example should error in the case we will be describing here, where quantities are indexed by their *unit* (metres, kilograms, etc.) and not their *dimension* (length, weight, etc.), however, if the latter is done, a compiler could do a conversion implicitly. This is explored in Maranushi and Eisenberg's 2014 paper "Experience Report: Type-Checking Polymorphic Units for Astrophysics Research in Haskell [8].

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 without change to the language specification [5]. 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 [5].

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. An example of such a declaration is provided in Figure 4, where the definition for `Newtons` is given.

We can define quantities as a type containing a `Unit` and a `Double`. This then allows us to write simple constructors for the quantity type. We can then define well-typed multiplication and addition operations giving us the ability to express derived units in terms of the elementary units that we've defined.

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.

Figure 3. Basic SI unit declarations in adapted from Dependent Haskell to Idris [5]

```
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)
```

The types for `times` and `over` representing division and multiplication are shown in Figure 4. 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 4.

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

Figure 4. Definition of division and multiplication of dependently typed units of measurement. Ported to Idris from [5]

```
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))
```

to derive units for examples that F# is unable to [3, 5]. 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`).

4 Conclusions

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. As I continue my research, I will expand on these ideas and make the scenarios more concrete and identifiable to a programmer, which I hope to present at POPL.

References

[1] Edwin C. Brady. 2011. Idris: Systems Programming Meets Full Dependent Types. *ACM Workshop on Programming Languages Meets Program Verification* (2011).

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

[3] Philip Carter, Luke Latham, and Maira Wenzel. 2016. Units of Measure. (2016).

[4] Richard Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practice.* Ph.D. Dissertation. University of Pennsylvania.

[5] Adam Michael Gundry. 2013. *Type Inference, Haskell and Dependent Types.* Ph.D. Dissertation. University of Strathclyde.

[6] Robert Lee Hotz. 1999. Mars Probe Lost Due to Simple Math Error. *Los Angeles Times* (1999).

[7] Andrew Kennedy. 2009. Types for Units-of-Measure. *Proceedings of the Third Summer Conference of the Central European Functional Programming School* (2009).

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

[9] Nicolas Oury and Wouter Swierstra. 2008. The Power of Pi. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08).* ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/1411204.1411213

[10] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure Distributed Programming with Value-dependent Types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11).* ACM, New York, NY, USA, 266–278. https://doi.org/10.1145/2034773.2034811