How to Make Your Programs Very Safe

A Review of Practical Applications of Dependent Types

Kevin Jiah-Chih Liao
Haverford College
Haverford, Pennsylvania
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 n and returns a vect of n Peano natural of value 1 to 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 : (n : Nat) -> Vect n Nat
replic n = replicateHelper n n where
  replicateHelper : (n : Nat) -> Nat -> Vect n
       Nat
  replicateHelper Z _ = Nil
  replicateHelper (S n) val = (val ::
      replicateHelper 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 Vects of length 5, etc. Thus, having written this type, a programmer now has compile-time guarantees of extreme correctness that would not be otherwise possible.

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.

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

2 Additional Work

This review is a tour of a potential application 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. My review explores the following additional examples:

1. Complicated pattern matching in the Cryptol DSL for cryptographic applications. [7]
2. Generating file format parsers from a data description language. [7]
3. A relational database and algebra. [3, 7]
4. An alternative to monadic transformations with an algebraic effects DSL [2]
5. Programming distributed systems with F and dependent types. [8]
6. A low-level domain-specific language demonstrating the applicability of dependent types to systems programming. [1]

3 Units of Measurement

A practical application for dependent types from is to eliminate bugs that can arise from improper unit conversions [4]. Units of measurement are already implemented in Microsoft's F# Programming Language ([6]). 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" [5].

Figure 2. 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
```

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.

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 [4]. 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 3. Basic SI unit declarations in adapted from Dependent Haskell to Idris [4]

```
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 4.

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

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

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

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.

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 accessible to the modern programmer. In addition to empowering constructive mathematicians with theorem provers, dependent types give a programmer ways to write extremely safe programs with certain compile-time guarantees of correctness.

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

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

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

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

[7] Nicolas Oury and Wouter Swiestra. 2008. The Power of Pi. *ACM SigPlan Notices* (2008).

[8] 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