# CS Seminar

## An Introduction to Dependently Typed Functional Programming in Idris

Mark Farrell

Tuesday, April 21, 12:30 PM
Slonim Conference Room (Room 430)
6050 University Ave., Halifax

**Abstract** In this seminar, we will introduce students and faculty members to dependently typed functional programming in Idris. Dependently typed functional programming languages can often allow software specifications to be expressed precisely with types in the language - and hence can help to ensure that software is correct when implemented according to specification. Traditionally, dependently typed functional programming languages have largely only been used in higher education, where it is often acceptable to express a solution to a problem precisely, at a consequent loss in program efficiency. Idris, however, aims to enable programming practitioners to have increased long-term confidence in the correctness of their software, and to be able to better reason about the behaviour of their software in a team environment, without necessarily compromising the efficiency of their software.

We will begin this seminar by covering basic information about the Idris programming language & introducing our audience to the idea of reasoning about program behaviour with types and constraints. Afterwards, we will then provide key examples of value-dependent data types in Idris (i.e. Vectors, Graphs & Polyhedra) as well common type classes and their laws (i.e. Semigroups, Monoids, Functors, Applicatives, Monads, Foldables & Traversables). Finally, we will provide proofs in Idris that certain value-dependent data types are valid instances of these type classes.

By the end of this seminar, we hope to movitate our audience to use and contribute to Idris. Though, we will be content if our audience has gained an appreciation of how dependently typed functional programming can assist them in ensuring the correctness of their software in an industrial setting - regardless of whether or not they choose to adopt Idris in their daily work. Some prior exposure to functional programming, in a programming language like Haskell, could be an asset in understanding the content of this seminar, though none is required.

**Brief Biography** Mark Farrell is currently a second-year student in the Faculty of Mathematics at the University of Waterloo, interning as a Research Assistant at the Faculty of Computer Science, Dalhousie University, Halifax NS,

Canada. He's interested in making small contributions to the Idris programming language in his spare time. In the past, he has interned as a Software Developer at Defence Research & Development Canada. He has also conducted volunteer work for the Center for Research & Education on Aging at Lawrence Berkeley National Laboratory, attempting to develop software that extractions relations from journal articles in order to build a model of the chemical processes involved in neuroendocrine aging programmatically. Before entering university, he was primarily interested in reverse engineering network communication protocols.

# 1 Introduction

Let's begin by talking about the Idris programming language itself, i.e. its core syntax and semantics as well as some high-level features. Most Idris programs, when elaborated, consist only of five forms: variable definitions, anonymous functions, function applications, the value-dependent function type and the type of types [2]. Although, we've explicitly included support for certain primitive types, namely fixed-width integers, floating point numbers & foreign pointers on top of this core language [1]; we've added support for these primitive types in order to to achieve reasonable performance when performing numerical computations & doing systems programming in Idris. In addition to our core language and support for primitive types, we've added support for algebraic data types to Idris; this allows us to compose types by taking the sums & products of simpler types and hence allows us to pattern match on those composite types when defining functions to perform operations on them [3].

Now, let's look at a motivational example of dependently typed functional programming in Idris: we will create a value-dependent type for vectors, i.e. lists whose length is expressed in the type, and show how expressing length in the type can help us ensure the correctness of our software. Typically, in a functional programming language without value-dependent types, we'd define a recursive data type for lists whose length we do not express in its type:

```
data List : Type -> Type where
  Nil : List a
  (::) : a -> List a -> List a
```

It is possible in this case, for instance, to take the head of an empty list; this would be incorrect, since an empty list does not have a head. However, if we express the length our list in its type, we increase our confidence that our software is correct before running it: in this case, we can make sure our head function only accepts non-empty lists as arguments.

Let's define our value-dependent data type for lists whose length are expressed in the type, conventially called vectors:

```
data Vect : Nat -> Type -> Type where
  Nil : Vect Z a
  (::) : a -> Vect n a -> Vect (S n) a
```

You might have also seen the definition for Peano representation of the natural numbers before, useful for doing type arithmetic:

```
data Nat = Z | S Nat
```

In the Peano representation of the natural numbers, a natural number is either zero or the successor of another natural number, e.g. one is the the successor of zero, two is the successor of one and so on and so forth. This representation of the natural numbers lets us, in this case, express the length of lists in its type, and express the resultant lengths of lists when we add or remove items from

them. Hence we can be sure that certain operations that we're performing on vectors - our value-dependent data type - are correct before our software runs.

## 2 Installation

Before we move forward and talk about more interesting examples of dependently typed functional programming in Idris, let's first talk about how to install it. Currently, it is recommended that you build Idris from source. It is necessary to have installed the Haskell Platform as well as Git, GNU Make & Bash before installing Idris. After installing these dependencies, clone the Idris repository, create a cabal sandbox, install cabal dependencies and then build Idris with GNU Make:

```
o=git://github.com/idris-lang/Idris-dev
git clone $o
cd Idris-dev
cabal sandbox init
cabal update
cabal install --only-dependencies
make
```

Try launching the Idris interpreter:

```
.cabal-sandbox/bin/idris
Idris>
```

Now you have installed Idris.

# References


[1] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

[2] Stephanie Weirich. Designing dependently-typed programming languages. `https://www.cs.uoregon.edu/research/summerschool/summer14/curriculum.html`, June 2014.

[3] Haskell Wiki. Algebraic data types. `https://wiki.haskell.org/Algebraic_data_type`, June 2014.