

# Dependent Types and Theorem Proving: Introduction to Dependent Types

Wojciech Kołowski

March 2021

- 1 Greetings
- 2 General idea
- 3 First-class types
  - What does “first-class” mean?

# General info

- The lectures will be held weekly on Fridays.
- Don't worry if you miss a lecture – the slides are pretty massive and the talks are going to be recorded.
- Each lecture ends with some exercises which will help you familiarize yourself with  $F^*$  and better understand the ideas covered in the talk.
- But you don't need to do them if you don't want to.
- This talks repo: <https://github.com/wkolowski/Dependent-Types-and-Theorem-Proving>

# Plan of lectures

- Lecture 1: Programming with dependent types.
- Lecture 2: Proving theorems with dependent types.
- Lecture 3: Differences between programming and proving.
- Lecture 4: Examples of bigger programs and longer proofs.
- Lecture 5: A deeper dive into  $F^*$ .

# Learning outcomes

- You won't be scared of all those obscure, scary and mysterious names and notations.
- You will get basic familiarity with the ideas behind dependent types.
- You will begin to see logic and mathematics in a very different light, much closer to your day job (at least if you are a programmer working in F#).
- If you do the exercises, you will gain a basic proficiency in F\*.

# Introducing F\*

- F\* (pronounced “eff star”) is a general-purpose purely functional programming language.
- Member of the ML family, syntactically most similar to F#.
- Aimed at program verification.
- Dependent types.
- Refinement types.
- Effect system.
- Not a .NET language.
- Neither compiled nor interpreted – it’s a proof assistant, i.e. just a typechecker.
- To run a program, it has to be extracted to some other language, like F#, OCaml, C or WASM, and then compiled.

Don't worry, be happy, ask questions

I KNOW YOU DIDN'T UNDERSTAND THE PREVIOUS SLIDE,  
BUT BY THE END OF THESE TALKS, YOU WILL!

# Useful F\* links

- **You can run F\* inside your browser** (and have a nice tutorial guide you):  
<http://www.fstar-lang.org/tutorial/>
- GitHub: <https://github.com/FStarLang/FStar>
- Homepage: <http://www.fstar-lang.org/>
- Download: <http://www.fstar-lang.org/#download>
- Papers (not approachable for ordinary mortals):  
<http://www.fstar-lang.org/#papers>
- Talks/presentations (more approachable):  
<http://www.fstar-lang.org/#talks> (some of these are quite approachable if you're interested)



# Prerequisites

- To understand what we will be talking about, you should have a working knowledge of  $F\#$  and the basic concepts of functional programming, namely:
- Functions as first-class citizens, including higher-order functions.
- Algebraic data types, including sum types and product types.
- Pattern matching and recursion.
- Even if you know these, you may be unfamiliar with the particular names – for example, “sum types” is a name used in academia and Haskell, but in  $F\#$  they are better known as “tagged unions”.

# Code snippet no 1

- We will now see some code that shows how these things look in F\*.
- See the file `Lecture1/Prerequisites.fst`.

# Values and types

- To understand dependent types, first we have to understand **dependency**.
- And to understand dependency, we need to be aware of the distinction between **values** and **types**.
- By **values**, we mean the bread-and-butter of programming: numbers, strings, arrays, lists, functions, etc.
- It should be pretty obvious to you that in most languages, **types** are not of the same status as numbers or functions.

# Dependencies

- Dependency is easy to understand. In fact, if you know basic  $F\#$ , then you already know most of it, because in  $F\#$ :
- **Values can depend on values:** we can think that the sum  $n + m$  is a number that depends on the numbers  $n$  and  $m$ . This dependency can be expressed as a function: `fun (n m : int) -> n + m`.
- **Values can depend on types:** for example, the identity function `fun (x : 'a) -> x` depends on the type `'a`.
- **Types can depend on types:** for example, the  $F\#$  type `Set<'a>` depends on the type `'a`.

# Naming the dependencies

- I bet you spotted the pattern in the previous slide, but it's a good idea to also have a name for the feature provided by each kind of dependency.
- Values can depend on values: (first-class) **functions**.
- Values can depend on types: **polymorphism** (i.e. “generics”).
- Types can depend on types: **type operators**.

# Dependent types

- There's yet another kind of dependency, which is not present in  $F\#$ , but is present in  $F^*$  and is the topic of this lecture.
- **Types can depend on values: dependent types.**
- But what are dependent types good for? You have been living your whole life without them, after all!
- Before we go on to explain dependent types, let's see some examples.

# Matrix multiplication

- We can only multiply matrices whose dimensions match, i.e. we can multiply an  $n \times m$  matrix by a  $m \times k$  and get an  $n \times k$  matrix as a result.
- How to model this in our favourite programming language without dependent types?
- The best we can do is to have a **type of matrices** `Matrix` and then matrix multiplication has type `matmult : Matrix -> Matrix -> Matrix`.
- What happens when we call it with matrices of the wrong dimensions?

- `matmult`  $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$  is well-typed, but will throw an

`IllegalArgumentException` or some other kind of runtime error, or maybe it will crash even less gracefully.

# Matrix multiplication with dependent types

- In a language with dependent types we can create a **type of  $n \times m$  matrices** `Matrix n m` and give multiplication the type  
`matmult : (n : ℕ) -> (m : ℕ) -> (k : ℕ) ->`  
`Matrix n m -> Matrix m k -> Matrix n k`
- Now `matmult` is a function which takes five arguments: the three matrix dimensions and the two matrices themselves.
- After giving it the dimensions of the first matrix from the previous slide, `matmult 2 2` has type `(k : ℕ) -> Matrix 2 2 -> Matrix 2 3 -> Matrix 2 3`.

- It is clear that `matmult 2 2 k`  $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$  **is not**

**well-typed** for any  $k$ , because the last argument is of type `Matrix 3 3`, but an argument of type `Matrix 2 k` was expected.



# A nice paper

- Dependent types can also be used to keep track of units of measure.
- This is possible in F# too, but it's a built-in feature of the compiler, whereas the dependently typed solution is much more principled.
- It is also composable – we can keep track of both matrix dimensions and units.
- There's a nice paper about this: **Type systems for programs respecting dimensions** available at <https://fredriknf.com/papers/dimensions2021.pdf>

# Array access

- When accessing the  $i$ -th element of an array,  $i$  must be smaller than the length of the array.
- How to model this in our favourite programming language without dependent types?
- We have Array  $A$ , **the type of arrays of  $A$ s**, and we can access its elements with a function `get : Array A -> int -> A`.
- What happens, when  $i$  is greater than the length of the array? Or, what happens when  $i$  is negative?
- `get [ | 'a'; 'b'; 'c' ] 5` is well-typed, but will throw an `IndexOutOfBoundsException` or result in a segmentation fault.

# Array access with dependent types 1/2

- In a language with dependent types we can have `Array A n`, **the type of arrays of As whose length is n**.
- Then we have a few possibilities to model the type of `get`.
- `get : (n : ℕ) -> Array A n -> (i : ℕ) -> i < n -> A`.
- In this variant, the fourth argument of `get` is a proof that the index isn't out of bounds (we will cover proofs in the next lecture).
- We can't prove `5 < 3`, so we don't have any proof to feed into `get 3 [| 'a'; 'b'; 'c' |] 5 : 5 < 3 -> Char`.

## Array access with dependent types 2/2

- $\text{get} : (n : \mathbb{N}) \rightarrow \text{Array } A \ n \rightarrow (i : \mathbb{N}\{i < n\}) \rightarrow A.$
- In this variant we use refinement types (which we will cover later today) to automatically guarantee that  $i$  isn't out of bounds.
- $\text{get } 3 \ [ | \ 'a'; \ 'b'; \ 'c' \ | ] \ 5$  is not well-typed, because the typechecker can't prove  $5 < 3$ , and thus 5 is not of type  $\mathbb{N}\{5 < 3\}.$

# Why should we care?

- Large software systems written in dynamically typed languages are hard to refactor, because all type checks occur at runtime.
- Statically typed languages make the situation better, because they move these checks to compile time, making them much more likely to be useful.
- But in simple functional languages like F# there's still plenty of room for dynamic runtime checks – array bounds checking, division by zero, and a lot of user-defined checks which throw exceptions in case of failure.
- With dependent types, these can be eliminated, which results in **even more refactorability and maintainability**.
- This is also a matter of performance – **no runtime checks means faster programs**.

# The running summary 1

- **Dependent types are types that can depend on values.**

# We're getting serious

- The above slides present nice fairy tales. . .
- . . . but how do dependent types actually work?
- And how to use them in F\*?
- And what can ordinary programmers use them for besides number crunching with matrices and arrays?

What does “first-class” mean?

# Juggling dependencies

- Given a functional language like F#, how to enable types to depend on values?
- Of course we want to retain the other kinds of dependencies (values on values, values on types, types on types).
- It turns out it's best **throw away all kinds of dependencies besides the basic one** (values on values)...
- ...and then **turn types into values!**



What does “first-class” mean?

# Values and types

- So now values encompass both old, ordinary values (integers, tuples, functions, etc.) and new values (types).
- This way we get all four kinds of dependencies:
- Ordinary values can depend on ordinary values.
- Ordinary values can depend on type values.
- Type values can depend on type values.
- Type values can depend on ordinary values.

What does “first-class” mean?

# First-class types

- In other words, we want to blur the distinction between values and types.
- In yet other words, we want to **make types first-class citizens of our programming language**.
- But what does “first-class” mean?

What does “first-class” mean?

# What does “first-class” mean?

- In C, we can define functions that take an `int` and return an `int`.
- But we can't define a function that takes a function from `ints` to `ints` and returns an `int`.
- This means that functions from `ints` to `ints` are not treated the same as `ints`.
- As far as C is concerned, we can say that `ints` are first-class, but functions are not first-class (thus, they are “second-class”).
- But C has function pointers, so you may be skeptical when I claim it doesn't have first-class functions.

What does “first-class” mean?

## Some heuristics

- The concept of “first-class” is neither precisely defined nor exact. Rather, it’s more of a functional programming folklore that obeys the “I know it when I see it” principle.
- However there are some heuristics that can help you.
- Something is first-class when it can be:
  - bound/assigned to variables.
  - stored in data structures.
  - passed to functions as an argument.
  - returned from functions.
  - constructed at runtime.
  - nameless, i.e. it can exist without giving it any name.
- So, which of these is criteria is not fulfilled by C’s function pointers?

What does “first-class” mean?

# A first-class quiz

- Let's have a little quiz to check if you get it.
- Are the below language features first-class in F# or not? Try to think about it for up to a minute and write down your answer.
- Functions?
- Recursive functions?
- Arrays?
- Types?
- Modules?
- Records?

What does “first-class” mean?

# A type-based definition of “first-class”

- Heuristics from previous slides are nice. . .
- but I prefer to think about first-class-ness in a different way, which is better from the functional programming point of view.
- A concept  $C$  is first-class in some programming language if in that language there is, loosely speaking, a type of all  $C$ s.
- This means that a language has first-class functions if for any two types  $A$  and  $B$  there is a type  $A \rightarrow B$  of all functions from  $A$  to  $B$ .

What does “first-class” mean?

## A first-class quiz again

- The same quiz again, but this time try to use the type-based definition of “first-class”.
- Are the below language features first-class in F# or not? Try to think about it for up to a minute and write down your answer.
- Functions?
- Recursive functions?
- Arrays?
- Types?
- Modules?
- Records?

What does “first-class” mean?

## A first-class quiz answers

- **Functions?** For any  $'a$  and  $'b$  there's a type of functions  $'a \rightarrow 'b$ .
- **Recursive functions?** There's no separate type of recursive functions, even though there's a syntactic distinction between `let` and `let rec`!
- **Arrays?** For any type  $'a$  there's a type of arrays, namely `array<'a>`.
- **Types?** There are types in  $F\#$  and there are type variables  $a'$ ,  $b'$ ,  $c'$  etc., but we can't assign them any type!
- **Modules?** Modules are not first-class, they have a whole separate world of their own.
- **Records?** This one is mixed depending on how you understand it. On the one hand, for any kind of record you can imagine, there's a corresponding type. But on the other hand, there is no type of all record types.



What does “first-class” mean?

# Computing with first-class types

- Previously we learned that “making  $X$  first-class” means adding a type of all  $X$ s.
- For types, this means that we need to have a **type of types**.
- And that’s it – we don’t need anything else.
- Because types are first-class in  $F^*$ , we can assign them to variables, pass them to functions as arguments and return them from functions, and even compute types by recursion.
- It might a bit difficult to wrap your head around this idea, so let’s see how it looks in  $F^*$ .
- The code snippet can be found in `Lecture1/FirstClassTypes.fst`

What does “first-class” mean?

## The running summary 2

- Dependent types are types that can depend on values.
-