Dependent Types and Theorem Proving: Introduction to Dependent Types

Wojciech Kołowski

March 2021

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 - basics of F*

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

Why should we care about dependent types? 1/3

- Programs written in dynamically typed languages perform a lot of runtime checks.
- Beyond a certain size **dynamically typed software is hard to extend, refactor and maintain because errors manifest very late** in the development process, i.e. at runtime.
- Statically typed languages make the situation better, because they move typechecking to compile time, which means a lot of errors get caught much sooner.
- **Static typing is good**.

Why should we care about dependent types? 2/3

- But in simple functional languages like F# there's still plenty of runtime checks – division by zero, taking the head of empty list and a lot of user-defined checks which throw exceptions in case of failure.

- With dependent types, all runtime checks can be turned into static checks – **all errors are type errors**.

- This results in more extensible, refactorable and maintainable software (and also better performance – less stuff to do at runtime).

- We can not only get rid of runtime checks, dependent types can also replace most unit tests and property tests.

- **Dependent types bring static typing to its limits**.

Why should we care about dependent types? 3/3

- And when I say all errors are typing errors, I really mean it – with dependent types, we can express all properties, formulate all specifications and describe all mathematical objects.

- **Dependent types reveal a deep connection between functional programming and logic**.

- Despite their great power, dependent types are easy to understand and significantly simplify the language design.

- Have you ever heard about fancy Haskell stuff like multi-param typeclasses, GADTs, higher-rank types, higher-kinded types, existential types and so on?

- No? No problem – **with dependent types, we get all of that (and much more) for free**.

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 : $\mathbb{N}$) -> (m : $\mathbb{N}$) -> (k : $\mathbb{N}$) ->
  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 : $\mathbb{N}$) -> 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 `A`s 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

- `get : (n : ℕ) -> Array A n -> (i : ℕ{i < n }) ->`
  `A.`
- In this variant we use refinement types (which we will cover later today) to automatically guarantee that `i` isn't out of bounds.
- `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 `ℕ{5 < 3}`.

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?

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!

The running summary 1

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

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**!

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.

First-class types

- How do we turn types into values?
- In programming languages' parlance, this process is called **making types first-class citizens of the language**.
- But what does "first-class" mean, anyway?

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 `int`s to `int`s and returns an `int`.
- This means that functions from `int`s to `int`s are not treated the same as `int`s.
- As far as C is concerned, we can say that **integers 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.

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 fullfilled by C's function pointers?

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?**
- Functions?
- Recursive functions?
- Arrays?
- Modules?
- Records?
- Types?

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.
- For a given programming, **a concept X is first-class if there is a type of all Xs**, loosely speaking.
- 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.

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?**
- Functions?
- Recursive functions?
- Arrays?
- Modules?
- Records?
- Types?

A first-class quiz answers

- Functions? For any 'a and 'b there's a type of functions 'a -> '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>`.
- Modules? Modules don't have types, they have signatures. But signatures are not types, so modules are not first-class.
- 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.
- Types? There are types in F# and there are type variables a', b', c' etc., but we can't assign them any type!

Computing with first-class types

- Previously we learned that "X is first-class" means that there is a type of all Xs.
- For types, this means that we need to have a **type of types**.
- And that's it – we don't need anything else.
- Note: the phrase "types of types" sounds (and looks) bad, so we will call it **the universe of types**, or in short, just **the universe**.
- 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.

Code snippet no 2 - first-class types in F*

- It might a bit difficult to wrap your head around the idea of first-class types, so let's see how it plays out in F*.
- The code snippet can be found in
  `Lecture1/FirstClassTypes.fst`

The running summary 2

- Dependent types are types that can depend on values.
- **In dependently typed languages:**
- **There is a universe – a type whose elements are themselves types**.

So far so good

- So far so good, but we still don't know how dependent types work.
- We saw some in the examples, but they were left unexplained.
- We also saw some more in the last code snippet, but those were the crudest and most primitive dependent types in existence.
- **Now that we have learned about first-class types and the universe of types, we can learn dependent types proper**.

Dependent types by analogy

- We will introduce dependent types by analogy.
- **Each of the various kinds of dependent types out there is just a generalization of an ordinary non-dependent type that is well-known to functional programmers**:
- Dependent function types are a generalization of function types.
- Dependent pair types are a generalization of products.
- Dependent record types are a generalization of records.
- Inductive types are a generalization of algebraic data types.

Type families

- In the coming slides, we will often refer to **type families**.
- **A family of types indexed by type** a is just a function a ->
  Type.
- There can be many indices, like in a -> b -> Type.
- We have already seen examples in the last code snippet:
- vec : Type -> nat -> Type is a family of types whose
  members vec a n are lists of length n and elements of type a
- matrix : Type -> nat -> nat -> Type is a family of
  types whose members matrix a n m are $n \times m$ matrices with
  entries of type a.

Non-dependent functions

- Recall how ordinary function types work in F#.
- If a : Type is a type and b : Type is a type, then there is a type a -> b : Type of functions that take an element of a and return an element of b.
- We create functions of type a -> b by writing fun (x : a) -> e where e is an expression of type b in which x may occur.
- If we have a function f : a -> b and x : a, then we we can apply f to x, written f x, to get an element of type b.

Dependent functions

- Now, watch the analogy unfold...

- If a : Type is a type and b : a -> Type **is a family of types**, then there is a type (x : a) -> b x **of dependent functions** which take an element of a **named** x and return an element of b x.

- We create functions of type (x : a) -> b x by writing fun (x : a) -> e where e is an expression of type b x in which x may occur.

- If we have a function f : (x : a) -> b x and x : a, then we can apply f to x, written f x, to get an element of type b x.

- Hint: it's probably easiest to pronounce (x : a) -> b x as "for all x of type a, b of x". Thus is revealed the connection to logic, which we will see in the next lecture.

More dependent functions

- Of course, we can iterate the dependent function type to get a type of functions whose output type dependent on the value of many inputs.

- `(x : a) -> b x`

- `(x : a) -> ((y : b x) -> c x y)`

- Dependent function type associates to the right, just like ordinary function type, so we can drop the parentheses. We can also drop all but the last arrow.

- `(x : a) (y : b x) (z : c x y) -> d x y z`

- `(x : a) (y : b x) (z : c x y) (w :  d x y z) -> e x y z w`

- etc.

Code snippet no 3 - dependent functions in F*

- Let's see how to use dependent functions in F*.
- See the code snippet `Lecture1/DependentFunctions.fst`

The running summary 3

- Dependent types are types that can depend on values.
- In dependently typed languages:
- There is a universe – a type whose elements are themselves types.
- **There is a type of dependent functions which are just like ordinary functions, but their output TYPE can depend on the VALUE of their input**.

Non-dependent pairs

- Recall how ordinary pairs work in F#.
- If a : Type is a type and b : Type is a type, then there is a type a * b : Type of pairs.
- To create a pair, we write (x, y) where x is of type a and y is of type b.
- To use a pair p : a * b, we use projections – we have fst p : a and snd p : b.
- We can also pattern match on pairs.

Dependent pairs

- Now, watch the analogy unfold...
- If a : Type is a type and b : a -> Type **is a family of types**, then there is **a type** (x : a) & b x : Type **of dependent pairs**.
- To create a dependent pair, we write (| x, y |) where x is of type a **and** y **is of type** b x.
- To use a pair p : (x : a) & b x, we use projections – we have fst p : a and snd p : b (fst p) (**note that the type of the second projection depends on the value of the first projection**).
- We can also pattern match on dependent pairs.

More dependent pairs

- **We can iterate the dependent pair type**, while dropping unneeded parentheses – analogously to what we did for dependent functions.

- (x : a) & b x

- (x : a) & (y : b x) & c x y

- (x : a) & (y : b x) & (z : c x y) & d x y z

- **But using iterated dependent pairs is very inconvenient!**

- To access component of a dependent quadruple p we would have to write fst p, fst (snd p), fst (snd (snd p)) and snd (snd (snd p)).

Dependent record types

- There's a better way than iterating dependent pair types: dependent record types.

- A record is basically a labeled tuple.

- **A dependent record is basically a labeled dependent tuple**.

- This means that the TYPES of later fields in a dependent record can depend on the VALUES of earlier fields.

Code snippet no 4 - dependent records in F*

- Let's see how dependent records work in F*.

- See the code snippet `Lecture1/DependentRecords.fst`

The running summary 4

- Dependent types are types that can depend on values.
- In dependently typed languages:
- There is a universe – a type whose elements are themselves types.
- There is a type of dependent functions which are just like ordinary functions, but their output TYPE can depend on the VALUE of their input.
- **Dependent record types are just like ordinary records, but the TYPES of later fields can depend on the VALUE of earlier fields**.

Inductive types are discriminated unions

- Recall how ordinary inductive types work in F# (where they are called discriminated unions).
- To define an inductive type I : Type, we list its constructors.
- The constructors are ordinary functions which take some arguments (which may be of type I, i.e. the one that is being defined) and return an element of I.
- To create an element of I, we use one of the constructors and provide it with the arguments it requires.
- To use an element of I, we pattern match on it and for each case we provide an expression which will be computed if that case matches.

Inductive families 1/2

- Now watch the analogy unfold...
- To define an **inductive family** I : a -> Type, we list its constructors. Here a is some type that is already defined.
- The constructors are **dependent functions** which take some arguments (which **may be of type** I y for some y : a) and **return an element of the type** I x, for some x : a.
- To create an element of I x, we use one of the constructors and provide it with the arguments it requires.
- To use an element of I x, we pattern match on it and for each case we provide an expression which will be computed if that case matches.

Inductive families 2/2

- This time it's a bit harder to spot the analogy, so let's elaborate on it.

- Instead of a single type I : Type, we define a **family of types** I : a -> Type all at once.

- In this context, values of type a are called **indices** of the family I.

- We define a **separate type for each possible index**.

- To create a value that belongs to **some type** I x in the family, a constructor may require an argument that belongs to I y, **a different type** in the family.

Code snippet no 5 - inductive families in F*

- Let's see how inductive families work in F*.
- See the code snippet `Lecture1/InductiveFamilies.fst`

The running summary 5

- Dependent types are types that can depend on values.
- In dependently typed languages:
- There is a universe – a type whose elements are themselves types.
- There is a type of dependent functions which are just like ordinary functions, but their output TYPE can depend on the VALUE of their input.
- Dependent record types are just like ordinary records, but the TYPES of later fields can depend on the VALUE of earlier fields.
- Inductive families are just like ordinary inductive types, but the TYPES in the family can depend on the VALUE of the index.

Pi type and multiplication

- The dependent function type **is also known as the Pi type**.
- **This name comes from a notation**: $(x : a) \rightarrow b\ x$ is sometimes written as $\prod\limits_{x:\ a} b(x)$.
- **This notation comes from an analogy with multiplication**. In math $\prod\limits_{k=0}^{n} a_k$ means $a_0 \cdot a_1 \cdot ... \cdot a_n$.
- **We can think about dependent function types in this way too**. For example, the type $(x : bool) \rightarrow p\ x$ is equivalent to $p\ true * p\ false$.
- The result of multiplication is called a product, hence the dependent function type **is also known as the dependent product type**.
- As it turns out, the dependent function type **generalizes both the ordinary function type and the product type**, but in different ways.

Sigma type and addition

- The dependent pair type **is also known as the Sigma type**.
- **This name comes from a notation**: `(x : a) & b x` is sometimes written as $\sum_{x:\, a} b(x)$.
- **This notation comes from an analogy with addition**. In math $\sum_{k=0}^{n} a_k$ means $a_0 + a_1 + ... + a_n$.
- **We can think about dependent pair types in this way too**. For example, the type `(x : bool) & p x` is equivalent to `p true + p false` (where `+` just means a simple tagged union).
- The result of addition is called a sum, hence the dependent pair type **is also known as the dependent sum type**.
- As it turns out, the dependent pair type **generalizes both the product type and the sum type**, but in different ways.

Inductive types and polynomials 1/2

- An inductive type is **EITHER** constructor 1 applied to
  arguments x1 **and** x2 . . . **and** xℕ **OR** constructor 2 applied to
  arguments . . . **OR** constructor M applied to arguments . . .
- In math, OR means **addition**, whereas AND means
  multiplication.
- So, an inductive type boils down to a **Sum of Products**.
- These products are made of two kinds of arguments: recursive
  arguments (whose type is the inductive type that is being
  defined) and non-recursive ones.
- If you think about it long enough, **inductive types
  correspond to polynomials**.

Inductive types and polynomials 2/2

- This could be hard to swallow, so let's see examples.
- Lists satisfy the equation $\text{List}(A) = 1 + A \times \text{List}(A)$.
- Here 1 corresponds to the nil constructor, whereas the A and $\text{List}(A)$ on the right correspond to the arguments of the cons constructor.
- This corresponds to the polynomial $F(X) = 1 + A \times X$.
- $\text{List}(A)$ is the least fixed point of this polynomial, i.e. the smallest type X that satisfies $F(X) = X$.
- Here "fixed point" corresponds to the fact that we create lists using constructors (nil and cons), whereas "least" corresponds to the fact that all lists are made of finitely many constructors.