

Dependent Types and Theorem Proving: Introduction to Dependent Types

Wojciech Kołowski

9 November 2021

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:
- All about types: algebraic data types, sum types, product types, record types, pattern matching etc.
- All about functions: functions as first-class citizens, higher-order functions, recursive functions, currying etc.
- 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 “discriminated unions”.

Learning outcomes

- You will get basic familiarity with the ideas behind all dependently typed languages.
- You will learn about all the different kinds of dependent types and what they are good for.
- You will be able to continue learning about dependent types on your own and won't be put off by all those obscure, scary and mysterious names and notations.

- 1 Intro
- 2 Why
- 3 Examples
- 4 The Universe
- 5 Functions
- 6 Records
- 7 Inductives
- 8 Refinements
- 9 Life of Pi
- 10 Logic

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.

Useful links

- **Repo with all lecture materials:** <https://github.com/wkolowski/Dependent-Types-and-Theorem-Proving>
- **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)

Code snippet no. 1 - basics of F*

- We will now see some code that shows how these prerequisites look in F* (hint: basically the same as in F#).
- Click to see the code snippet: [Code/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 define `Matrix n m`, **the type of $n \times m$ matrices**, 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.

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 can define Array a , **the type of arrays that hold elements of type a** , 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

- In a language with dependent types we can define `Array a n`, **the type of arrays of length n that hold elements of type a** , and we give array access the type `get : (n : ℕ) -> Array a n -> (i : int{0 <= i < n}) -> a`.
- We use refinement types (which we will cover later) to statically guarantee that i isn't out of bounds.
- `get 3 [| 'a'; 'b'; 'c' |] 5` is not well-typed, because the typechecker can't prove $0 \leq 5 < 3$, and thus 5 is not of type `int{0 <= 5 < 3}`.

Values and types

- To understand dependent types, first we have to be aware of the distinction between **values** and **types** that is present in all the usual mainstream programming languages.
- 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`. This kind of dependency is called **generics** (or, in academia, **polymorphism**).
- **Types can depend on types:** for example, the $F\#$ type `Set<'a>` depends on the type `'a`. This kind of dependency is called **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.**
- We have already seen two examples: the type `Matrix n m` depends on the numbers `n` and `m`, and the type `Array a n` depends on the type `a` and the number `n`.
- 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).

Juggling dependencies

- It turns out it's best to **throw away all kinds of dependencies besides the basic one** (values on values, i.e. functions)...
- ...and then **turn types into values!**
- In other words: we want to make types first-class citizens of our language.
- Then we will be able to express all 4 kinds of dependencies using plain old functions.

What does “first-class” mean?

- **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.

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 language, **a concept X is first-class if there is a type of all X s**, loosely speaking.
- This means that, for example, 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 .

The Universe of Types

- 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.

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 $a \rightarrow \text{Type}$.
- There can be many indices, like in $a \rightarrow b \rightarrow \text{Type}$.
- We have already seen examples in the last code snippet:
- `Array : Type \rightarrow nat \rightarrow Type` is a family of types whose members `Array a n` are types of arrays of length n that hold elements of type a .
- `Matrix : nat \rightarrow nat \rightarrow Type` is a family of types whose members `Matrix n m` are $n \times m$ matrices.

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^* .
- Click to see the code snippet: [Code/Universe.fst](#)

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 functions are a generalization of functions.
 - Dependent pairs are a generalization of pairs.
 - Dependent records are a generalization of records.
 - Inductive types are a generalization of algebraic data types.

Non-dependent functions

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

Dependent functions

- Now, watch the analogy unfold...
- If $a : \text{Type}$ is a type and $b : a \rightarrow \text{Type}$ **is a family of types**, then there is a type $(x : a) \rightarrow b\ x$ **of dependent functions** which take an input of type a **named** x and return an output of type $b\ x$.
- We create functions of type $(x : a) \rightarrow b\ x$ 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 $f : (x : a) \rightarrow b\ x$ and an $x : a$, 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 $(x : a) \rightarrow b\ x$ as "for all x of type a , b of x ". Thus is revealed the connection to logic, which we will revisit later.

More dependent functions

- Of course, we can iterate the dependent function type to get a type of functions whose output type depends on many input values.
- $(x : a) \rightarrow b\ x$
- $(x : a) \rightarrow ((y : b\ x) \rightarrow c\ x\ y)$
- 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.
- $(x : a)\ (y : b\ x)\ (z : c\ x\ y) \rightarrow d\ x\ y\ z$
- $(x : a)\ (y : b\ x)\ (z : c\ x\ y)\ (w : d\ x\ y\ z) \rightarrow e\ x\ y\ z\ w$
- etc.

Code snippet no. 3 - dependent functions in F^*

- Let's see how to use dependent functions in F^* .
- Click to see the code snippet: [Code/DependentFunctions.fst](#)

Non-dependent pairs

- Recall how ordinary pairs work in $F\#$.
- If $a : \text{Type}$ is a type and $b : \text{Type}$ is a type, then there is a type $a * b : \text{Type}$ 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 $\text{fst } p : a$ and $\text{snd } p : b$.
- We can also pattern match on pairs.

Dependent pairs

- Now, watch the analogy unfold...
- If $a : \text{Type}$ is a type and $b : a \rightarrow \text{Type}$ **is a family of types**, then there is **a type** $(x : a) \ \& \ b \ x : \text{Type}$ **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 $\text{fst } p : a$ and $\text{snd } p : b \ (\text{fst } p)$ (**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 components of a dependent quadruple p we need to write $\text{fst } p$, $\text{fst } (\text{snd } p)$, $\text{fst } (\text{snd } (\text{snd } p))$ and $\text{snd } (\text{snd } (\text{snd } p))$.

Dependent record types

- There's a better way than iterating dependent pairs: dependent records.
- 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*.
- Click to see the code snippet: [Code/DependentRecords.fst](#)

Inductive types refresher

- Recall how ordinary inductive types work in F# (where they are called discriminated unions; in Haskell, they are known as algebraic data types).
- To define an inductive type $I : \text{Type}$, 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** $I : a \rightarrow \text{Type}$, 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 $I : \text{Type}$, we define a **family of types** $I : a \rightarrow \text{Type}$ 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^* .
- Click to see the code snippet: [Code/InductiveFamilies.fst](#)

Summary of dependent types

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

Some downsides of dependent types

- In dependently typed languages there is a lot of types.
- This is a blessing, because we can express all the complicated types and properties we need in order to guarantee correctness of our programs.
- But the richness of types also causes problems: it is often the case that there are many ways to define essentially the same type, which can give us a lot of headache.
- It also means we need to write a lot of boilerplate - for example, we need to define `map` separately for lists and vectors.

Refinement types to the rescue

- Beware: refinement types are not commonly considered to be dependent types!
- A refinement is just a different name for a function that returns a boolean.
- A refinement type is a type together with a refinement.
- In F^* syntax: if $p : a \rightarrow \text{bool}$, then $x : a\{p\ x\}$ is the type of values of type a for which p returns true.

Code snippet no. 6 - refinement types in F^*

- Let's see how refinement types work in F^* .
- Click to see the code snippet: [Code/Refinements.fst](#)

This is (almost) the end

- This is the end of the basic version of the slides.
- However, there are two bonus sections, which were initially supposed to be put into a continuation of this presentation (which was eventually cancelled).
- The next section describes the alternative names and notations used for dependent types in the literature.
- The last section is an (incomplete) introduction to logic. Towards the end, it explains the link between dependent types and logic.

Pi types 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 $\prod_{x: a} b(x)$.
- **This notation comes from an analogy with multiplication.** In math $\prod_{k=0}^n a_k$ means $a_0 \cdot a_1 \cdot \dots \cdot a_n$.
- **We can think about dependent function types in this way too.** For example, the type $(x : \text{bool}) \rightarrow p\ x$ is equivalent to $p\ \text{true} * p\ \text{false}$.
- 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 types 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 $a_0 + a_1 + \dots + a_n$.
- **We can think about dependent pair types in this way too.** For example, the type $(x : \text{bool}) \& p\ x$ is equivalent to $p\ \text{true} + p\ \text{false}$ (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 x_1 **and** $x_2 \dots$ **and** x_N **OR** constructor 2 applied to arguments \dots **OR** constructor M applied to arguments \dots
- 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.

Boolean “logic”

- Being a programmer, you are good friends with the booleans, aren't you?
- There are two booleans, `true` and `false`.
- We can combine booleans `b` and `c` with the usual boolean functions:
 - `not b` – “not `b`”
 - `b && c` – “`b` and `c`”
 - `b || c` – “`b` or `c`”

What is a logic

- Boolean logic is not an example of what logicians call a “logic”, in the sense that it is not a “logical system”, but merely a type with some unary and binary functions on it.
- A logic usually consists of:
- A definition of what **propositions** we’re dealing with.
- A **semantics**, which tells us what these propositions mean.
- A **proof system**, which tells us which propositions can be proven and disproven.
- A **soundness theorem** which states that propositions proven true using the proof system are semantically true.
- Optionally, there may also be a **completeness theorem** which states that all semantically true propositions can be proven.

Propositions

- A proposition asserts that something is the case, irrespectively of whether this really is the case or not.
- Math example: “4 is a prime number.”
- Software example: “For each input string x , if x is not malformed, my program produces as output an array of length at most 10.”
- Hardware example: “This circuit implements addition of 16 bit integers.”
- Real world example: “It’s raining or I like trains.”
- **Beware! Formal logic is not very good for reasoning about the real world!**

Propositional constants and connectives

- Propositions (usual letters: P, Q, R) are defined as follows:
- \top – the true proposition.
- \perp – the false proposition.
- P, Q, R, \dots – propositional variables.
- $\neg P$ – negation, read “not P ”.
- $P \vee Q$ – disjunction, read “ P or Q ”.
- $P \wedge Q$ – conjunction, read “ P and Q ”.
- $P \implies Q$ – implication, read “ P implies Q ” or “if P then Q ”.
- $P \iff Q$ – logical equivalence, read “ P if and only if Q ”.

Classical logic

- Classical logic is the most widely known/taught/used logical system in the world.
- In classical logic, **we think of propositions as being either true or false.**
- Therefore, classical logic is the logic in which truth values are the booleans.
- The truth value of a propositional variable is determined by a **valuation** $v : \text{Var} \rightarrow \text{Bool}$.
- If $v(P) = \text{true}$, then P is considered to be true.
- Otherwise it's considered false.

Semantics of classical logic 1/2

- Given a valuation $v : \text{Var} \rightarrow \text{Bool}$, the truth value of a proposition can be determined with a recursive function $\llbracket - \rrbracket : \text{Prop} \rightarrow \text{Bool}$.
- $\llbracket \top \rrbracket = \text{true}$
- $\llbracket \perp \rrbracket = \text{false}$
- $\llbracket P \rrbracket = v(P)$, where P is a variable.
- $\llbracket \neg P \rrbracket = \text{not } \llbracket P \rrbracket$
- $\llbracket P \vee Q \rrbracket = \llbracket P \rrbracket \text{ || } \llbracket Q \rrbracket$
- $\llbracket P \wedge Q \rrbracket = \llbracket P \rrbracket \text{ \&\& } \llbracket Q \rrbracket$
- $\llbracket P \implies Q \rrbracket = (\text{not } \llbracket P \rrbracket) \text{ || } \llbracket Q \rrbracket$
- $\llbracket P \iff Q \rrbracket = \llbracket P \rrbracket \text{ == } \llbracket Q \rrbracket$

Semantics of classical logic 2/2

- P is satisfiable when $\llbracket P \rrbracket = \text{true}$ for some valuation.
- P is falsifiable when $\llbracket P \rrbracket = \text{false}$ for some valuation.
- P is a tautology when $\llbracket P \rrbracket = \text{true}$ for all valuations.

Example

- Example: the proposition $P \implies Q$ is satisfiable (for $v(P) = \text{true}$, $v(Q) = \text{true}$).
- It is also falsifiable (for $v(P) = \text{true}$, $v(Q) = \text{false}$).
- Therefore, it is not a tautology.
- Example: the proposition $P \wedge Q \implies Q \wedge P$ is a tautology.
- We have $\llbracket P \wedge Q \implies Q \wedge P \rrbracket =$
(not ($v(P) \ \&\& \ v(Q)$)) $\mathit{||}$ ($v(P) \ \&\& \ v(Q)$).
- For any values of $v(P)$ and $v(Q)$ we always get true.

Classical logic is not that good

- We can check whether a proposition is a tautology by trying all possible valuations, but there are exponentially many of them.
- We can do better by defining a proof system with some axioms and inference rules, which would allow us to **prove** that a proposition is a tautology without trying all valuations.
- We won't do that because **classical logic is not the right logical system for proving programs correct**.
- We will use constructive logic instead.

Constructive logic 1/2

- In constructive logic, **propositions ARE NOT either true or false.**
- In constructive logic we usually think about propositions **in terms of their proofs.**
- In everyday language and also in mathematics as it is usually practiced, a “proof” means an argument by which one human demonstrates the truth of a statement to another human.
- In constructive logic, a proof is a formal object which **certifies that the given proposition has been proven**, in which case we say that the propositions holds.
- Meaning of propositions is determined by how we can prove them and how we can use their proofs to prove other propositions.

Constructive logic 2/2

- If we have a proof of P , we may think of it as “true” (although we shouldn’t think in terms of true and false).
- If we have a proof of $\neg P$, we may think that P is “false”.
- If we have neither proof, we don’t know anything about P .

Propositions are types, proofs are programs

- **We can represent propositions using types.**
- \top corresponds to the unit type.
- \perp corresponds to the Empty type (which can be defined as a disjoint union with zero constructors).
- $P \vee Q$ corresponds to a disjoint union or with constructors
`inl : P -> or P Q` and `inr : Q -> or P Q`.
- $P \wedge Q$ corresponds to the product type $P * Q$.
- $P \implies Q$ corresponds to the function type $P \rightarrow Q$.
- $\neg P$ corresponds to the function type $P \rightarrow \text{Empty}$.
- $P \iff Q$ corresponds to the type $(P \rightarrow Q) * (Q \rightarrow P)$.
- **You already know how to prove these propositions – just write the appropriate program!**

Dependent types, predicates, relations and quantifiers

- What about dependent types?
- Type families $p : a \rightarrow \text{Type}$ correspond to predicates, i.e. propositions that describe properties of objects of type a .
Type families with many indices, like $r : a \rightarrow b \rightarrow \text{Type}$, correspond to relations, i.e. propositions that describe relationships between a thing of type a and a thing of type b .
- The dependent function type $(x : a) \rightarrow p\ x$ corresponds to the universal quantifier $\forall x : a, p(x)$, which can be read as “each object x of type a satisfies the predicate p ”.
- The dependent pair type $(x : a) \& p\ x$ corresponds to the existential quantifier $\exists x : a, p(x)$, which can be read as “there exists an object x of type a which satisfies the predicate p ”.
- Inductive families are a very comfortable way of defining predicates and relations.