# Introduction to Dependent types

Adam Krupicka
Faculty of Informatics
Masaryk University, Brno

28.6.2016

Abstract

This short paper aims to give an intuitive overview of Dependent types, their correspondence with First-order predicate logic, and their practical uses.

## 1 Introduction

In sections 3 and 4, some prior knowledge of the Curry-Howard correspondence and related topics is assumed. This can be found in e.g. [1].

## 2 Definitions

$\lambda\!\!\rightarrow$ refers to the Simply typed $\lambda$ calculus.

BHK refers to the Brouwer-Heyting-Kolmogorov interpretation of intuitionistic logic.

$\mathbb{N}$ the set of all natural numbers is understood to include the number zero.

# 3 Dependent types

A dependent type is a type which depends on other values. For example, the type of vectors of natural numbers of length $n$ depends on the concrete value of $n$.

## 3.1 λΠ

The simplest system of dependent types, usually referred to as λΠ, is outlined below. There are two basic building blocks used to construct dependent types.

Dependent functions A dependent function is a function from some value $a$ of type $A$ to the type $B(a)$. Formally, this is written as $\Pi(a : A).B(a)$. For example, $\Pi(n : \mathbb{N}).Vec\mathbb{N}(n)$ would be a function from some $n : \mathbb{N}$ to the type of vectors of natural numbers of length $n$, where we write $Vec\mathbb{N}(n)$ for $n$-tuples of natural numbers. If $B(a)$ is a constant function to some type $C$, then we get a regular function type $A \to C$, familiar from the Simply typed λ calculus. For example, $\Pi(n : \mathbb{N}).\mathbb{N}$ is equivalent to $\mathbb{N} \to \mathbb{N}$.

Dependent pairs A dependent pair is a pair where the value $a : A$ of the first element determines the type of the second element $B(a)$. This is written as $\Sigma(a : A).B(a)$. If $B(a)$ is a constant function to some type $C$, then we get a regular non-dependent pair of the type $A \times C$.

This is the basic outline of the simplest system of dependent types, usually referred to as λΠ. The operators ″Π″ and ″Σ″ are allowed to range only over values, as was the case in e.g. the type of vectors of some length $n$. We saw that already this system subsumed the type operators ″ →″ and ″×″ from $\lambda \to$.

## 3.2 λΠ2

When we further extend the range of the ″Π″ and ″Σ″ operators to allow ranging over types, we immediately obtain a richer system. This system now subsumes the ″Λ″ operator known from the Polymorphic λ calculus.

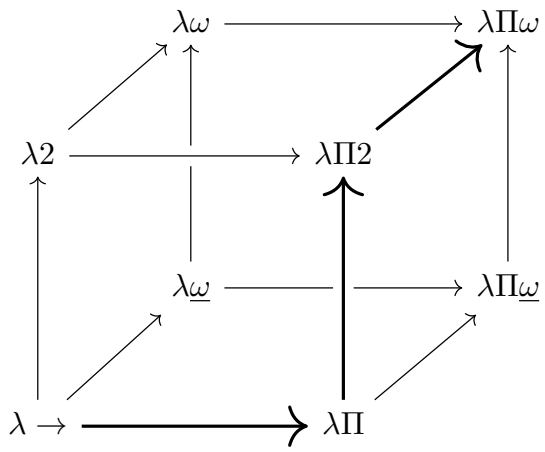


Figure 1: The λ cube. Path that dependent types lead us in thick.

For example, we can now generalize our example of vectors from the previous section to all types of elements, rather than just one fixed type:

$$\Pi(A : \mathcal{U}).\Pi(n : \mathbb{N}).Vec(A, n)$$

Here, $\mathcal{U}$ stands for the type of all the types in our system, excluding $\mathcal{U}$ itself.

## 3.3 λΠω

When we further allow the operators to range over higher types, we obtain the system λΠω. Here we define a hierarchy of so-called universes, where the type $\mathcal{U}$ from the previous section (whose type was unclear, or perhaps it did not have a type at all) becomes $\mathcal{U}_0$ of the type $\mathcal{U}_1$, which is itself of the type $\mathcal{U}_2$, and so on; these are the universes. This hierarchy lets us define higher level functions, e.g. a function of the type $\mathbb{N} \rightarrow \mathcal{U}_0$ (a function, which to each natural number assigns a type from the universe $\mathcal{U}_0$) would live inside the universe $\mathcal{U}_1$.

This system, with some extensions, serves as the basis for the proof assistant Coq [2]. This hierarchy is outlined in Figure 1.

# 4 The Curry-Howard correspondence

Curry was first to observe that types of combinators in Combinatory logic corresponded with axiom-schemas in certain Proof calculi. Later, Howard noticed that a proof calculus called Natural deduction could be directly interpreted as $\lambda\!\rightarrow$, with this correspondence:

| Natural deduction | $\lambda\!\rightarrow$ |
|---|---|
| propositions | types |
| proofs | type derivations |
| proof normalization | conversion into normal form |

Lambek showed that the correspondence can be extended to cartesian closed categories, thus making it three-fold.

Dependent types were first introduced in an attempt to find a corresponding $\lambda$ calculus for intuitionistic First-order order predicate logic. In this section, I will give a brief outline of the correspondence. For the curious reader, more information can be found in e.g. [3].

It should be noted that the logic I am drawing the correspondence with is a many-sorted first-order logic. Although it is possible to draw the correspondence with regular first-order logic, the dependent $\lambda$ calculus thus obtained would be rather restrictive, as it would allow us to construct types only from one atomic type.

Quantifiers A proof of the formula $\forall x : A.\varphi(x)$ is, under the BHK interpretation, a function transforming all inputs of the type $A$ into proofs of $\varphi(x)$. One can see that it is similar to the interpretation of the formula $A \rightarrow B$ (which is a function transforming proofs of $A$ into proofs of $B$). The difference is that in the proof of $\forall x : A.\varphi(x)$, the 'consequent' depends on the value of the 'antecedent'. The formulas of the form $\forall x : A.\varphi(x)$ correspond to the type of the dependent function.

The situation is analogous in the case of the formulas of the form $\exists x : A.\varphi(x)$. A proof of such a proposition is a pair of some $x$, and a proof of $\varphi(x)$. This corresponds to the dependent pair.

Predicates Predicates serve the role of type constructors — if $A$ is a n-ary predicate, then $A(t_1, ..., t_n)$ (for some terms $t_1, ..., t_n$) is a type. Nullary

predicates can be viewed as atomic types, e.g. the type $\mathbb{N}$ of natural numbers.

### 4.1 Equality and decidability

The language of First-order logic usually includes also equality. It is also necessary to decide the unifiability of some types when attempting to type check function application. Compared to the type system of $\lambda\!\rightarrow$, this task is less trivial in dependently typed systems. Consider, for example, the type of vectors of natural numbers of length 3: $Vec\mathbb{N}(3)$. When working with natural numbers, we might find it convenient to define addition — thus allowing us to write e.g. $Vec\mathbb{N}(2+1)$ to express the same type. This has the consequence of having to perform computation on the type level when type checking our programs. Thus it is convenient to limit ourselves to total functions on the type level, else the type checking of our program becomes undecidable.

## 5 Dependent types in practice

In this section, I have included a practical demonstration of dependent types. We will use a proof assistant/programming language based on intuitionistic type theory called Agda [4]. Agda is even more expressive than $\lambda\Pi\omega$, as it constitutes something akin to a so-called Pure Type System[1]. Pure Type Systems are a generalization over the systems of the $\lambda$ cube from Figure 1 [3].

### 5.1 Vectors

We have already seen that dependent types make it possible to define vectors of fixed length. To encode vectors in Agda, we will first have to define natural numbers.

```
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ
```

[1]Perhaps with some extensions, which are yet to be formally well described.

This is a data type declaration with two constructors. The first, nullary constructor simply represents zero. The second, unary constructor represent the successor function. For example to encode the number 4, we would write `succ(succ(succ(succ(zero))))`. The type `ℕ` is itself of the type `Set`. `Set` is a shorthand for $\texttt{Set}_0$, which is itself of the type $\texttt{Set}_1$, etc. This is akin to the universe hierarchy described in Subsection 3.3.

Equipped with Peano numbers, we can now define the type of vectors.

```
data Vec (A : Set) : ℕ → Set where
  []   : Vec A zero
  _::_ : {n : ℕ} → A →  Vec A n → Vec A (succ n)
```

A vector is parametrized by the type of it's elements `A` and is itself of the type `ℕ → Set`. Again we have two constructors; the first constructor simply represents an empty vector. The second constructor prepends a value to the head of a vector. For example, the vector $[1, 2, 3]$ would be represented as `1 :: (2 :: (3 :: []))`.

Compared to regular lists of unknown length, the immediate improvement is that now we can define our `head` and `tail` functions in a manner which enforces these to be only applied to non-empty vectors.

```
head : {n : ℕ} → {A : Set} → Vec A (succ n) → A
head (x :: _) = x

tail : {n : ℕ} → {A : Set} → Vec A (succ n) → Vec A n
tail (_ :: xs) = xs
```

Note that the length of the input vector is `succ n`. This enforces the programmer to always pass to the function an input vector of length at least 1, otherwise the type checker will not successfully verify the program.

We can define addition and multiplication on Peano numbers recursively.

```
_+_ : ℕ → ℕ → ℕ
zero + b = b
(succ a) + b = succ (a + b)

_*_ : ℕ → ℕ → ℕ
zero * b = zero
(succ a) * b = b + (a * b)
```

With these operations, we can now express more advanced functions on vectors.

```
append : {n m : ℕ} → {A : Set} → Vec A n → Vec A m → Vec A (n + m)
append [] ys = ys
append (x :: xs) ys = x :: append xs ys

concat : {n m : ℕ} → {A : Set} → Vec (Vec A m) n → Vec A (n * m)
concat [] = []
concat (xs :: xss) = append xs (concat xss)
```

`append` simply appends two vectors, whereas `concat` concatenates a vector of vectors into a single vector. Note that each vector is of the same, fixed length $m$ inside the outer vector of length $n$. This is a natural way to encode matrices. As an example of a matrix function, we might want to extract the diagonal of a matrix. However, the matrix must be square! For us, this is easy to express on the type level.

```
map : {n : ℕ} → {A B : Set} → (A → B) → Vec A n → Vec B n
map _ [] = []
map f (x :: xs) = (f x) :: (map f xs)

diagonal : {n : ℕ} → {A : Set} → Vec (Vec A n) n → Vec A n
diagonal [] = []
diagonal (xs :: xss) = head xs :: diagonal (map tail xss)
```

## 5.2 Natural deduction

In the previous subsection, we have seen a few examples of how dependent types can help us be more expressive about the types of our computations. In this subsection, we will look at another application of dependent types — theorem proving. Following the Curry–Howard correspondence, this should come as no surprise. Dependent types are very expressive, which is why they are fitting for the task. Now, let us prove some basic properties of Propositional logic.

Conjunction Our system will closely model the rules of Natural Deduction. We will define introduction as a data type constructor, and eliminations as functions.

```
data _∧_ (P : Set) (Q : Set) : Set where
  _,_ : P → Q → (P ∧ Q)

∧-elim₁ : {P Q : Set} → (P ∧ Q) → P
∧-elim₁ (p , q) = p

∧-elim₂ : {P Q : Set} → (P ∧ Q) → Q
∧-elim₂ (p , q) = q
```

We can prove some basic properties of conjunction, e.g. commutativity. We will implement a function, which — with the help of pattern matching — deconstructs the proof of $p \wedge q$ into proofs of $p$ and $q$, and then arranges them back into a conjunction in the required order. This corresponds with the standard understanding of implication in intuitionistic logic, namely that implication is a function which transforms proofs of the antecedent into proofs of the consequent.

```
∧-comm' : {P Q : Set} → (P ∧ Q) → (Q ∧ P)
∧-comm' (p , q) = (q , p)
```

Note that, strictly speaking, we have only proved one direction of the equivalence. In this simple case this makes no difference, as both directions are analogous, however in more complex proofs we would wish to prove both directions. We will therefore define equivalence on a meta-level (not inside our Natural Deduction system) and restate the commutativity proof.

```
_⇔_ : (P : Set) → (Q : Set) → Set
p ⇔ q = (p → q) ∧ (q → p)

∧-comm : {P Q : Set} → (P ∧ Q) ⇔ (Q ∧ P)
∧-comm = (∧-comm' , ∧-comm')
```

The commutativity proof now reflects our intuitive understanding of it. Both implications are analogous, therefore we can simply reuse our original proof in both directions.

Associativity can be proved in a very similar fashion. We can view the first two functions as lemmas, and the third function as a corollary.

```
∧-assoc¹ : {P Q R : Set} → (P ∧ (Q ∧ R)) → ((P ∧ Q) ∧ R)
∧-assoc¹ (p , (q , r)) = ((p , q) , r)

∧-assoc² : {P Q R : Set} → ((P ∧ Q) ∧ R) → (P ∧ (Q ∧ R))
∧-assoc² ((p , q) , r) = (p , (q , r))

∧-assoc : {P Q R : Set} → (P ∧ (Q ∧ R)) ⇔ ((P ∧ Q) ∧ R)
∧-assoc = (∧-assoc¹ , ∧-assoc²)
```

Disjunction We will now define disjunction. As in Natural Deduction, we will employ two introductions and one elimination.

```
data _∨_ (P : Set) (Q : Set) : Set where
  left : P → (P ∨ Q)
  right : Q → (P ∨ Q)

∨-elim : {P Q R : Set} → (P → R) → (Q → R) → (P ∨ Q) → R
∨-elim f _ (left p) = f p
∨-elim _ g (right q) = g q
```

Once again, our functions closely model the rules of Natural Deduction. We will not concern ourselves with proofs of commutativity and associativity, as they are similar to those of conjunction.

Instead, we will present proofs of theorems of both our logical connectives combined[2]. A proof of absorption can be formalized thusly:

```
abs₁ : {P Q : Set} → (P ∧ (P ∨ Q)) ⇔ P
abs₁ = (∧-elim₁ , λ p → (p , left p))

abs₂ : {P Q : Set} → (P ∨ (P ∧ Q)) ⇔ P
abs₂ = (∨-elim (λ x → x) ∧-elim₁ , left)
```

One direction of the theorems is always trivial; the other has to be constructed appropriately. For this, we utilize a lambda expression in the first theorem, and we have to eliminate a disjunction in the second theorem — in this case, the identity function comes in handy.

Finally, we shall prove the distributive laws. These are a bit more wordy, however they follow the same principles of simply reorganizing the formulas as we require.

[2]That is, identities valid in the variety of Boolean algebras.

```
distrib₁¹ : {P Q R : Set} → (P ∧ (Q ∨ R)) → ((P ∧ Q) ∨ (P ∧ R))
distrib₁¹ (p , (left q)) = left (p , q)
distrib₁¹ (p , (right r)) = right (p , r)

distrib₁² : {P Q R : Set} → ((P ∧ Q) ∨ (P ∧ R)) → (P ∧ (Q ∨ R))
distrib₁² (left (p , q)) = p , (left q)
distrib₁² (right (p , r)) = p , (right r)

distrib₁ : {P Q R : Set} → (P ∧ (Q ∨ R)) ⇔ ((P ∧ Q) ∨ (P ∧ R))
distrib₁ = (distrib₁¹ , distrib₁²)

distrib₂¹ : {P Q R : Set} → (P ∨ (Q ∧ R)) → ((P ∨ Q) ∧ (P ∨ R))
distrib₂¹ (left p) = (left p , left p)
distrib₂¹ (right (q , r)) = (right q , right r)

distrib₂² : {P Q R : Set} → ((P ∨ Q) ∧ (P ∨ R)) → (P ∨ (Q ∧ R))
distrib₂² ((left p) , _) = left p
distrib₂² (_ , (left p)) = left p
distrib₂² ((right q) , (right r)) = right (q , r)

distrib₂ : {P Q R : Set} → (P ∨ (Q ∧ R)) ⇔ ((P ∨ Q) ∧ (P ∨ R))
distrib₂ = (distrib₂¹ , distrib₂²)
```

Note that at several occasions we have avoided having to use disjunction elimination, and instead utilized pattern matching to handle the distinct possibilities of deconstructing a proof of $p \vee q$ and finding within a proof of either $p$, or $q$. As per the Curry-Howard correspondence, this essentially amounts to the same thing. We take advantage of using pattern matching on the meta-level, rather than having to use `∨-elim` directly.

# 6 Concluding remarks

We saw that dependent types are a very powerful formalism with firm grounds in Logic. Dependent types can not only help us be more expressive about our computations, but can also serve as a constructive basis for mathematics.

# References


[1] Darryl McAdams. "A Tutorial on the Curry-Howard Correspondence". In: (2013). URL: http://purelytheoretical.com/papers/ATCHC.pdf.

[2] Yves Bertot and Pierre Castéran. Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer Science & Business Media, 2013.

[3] Morten Heine Sørensen and Pawel Urzyczyn. Lectures on the Curry-Howard isomorphism. Vol. 149. Elsevier, 2006.

[4] Ulf Norell. Towards a practical programming language based on dependent type theory. Vol. 32. Citeseer, 2007.