Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Dependently Typed Programming

Andres Löh

Department of Information and Computing Sciences
Utrecht University

September 15, 2010

Hello world

```
module HelloWorld where
open import IO
main = run (putStrLn "Hello world!")
```

Hello world

```
module HelloWorld where
open import IO
main = run (putStrLn "Hello world!")
```

Remarks:

- Syntax heavily inspired by Haskell.
- Module header required.
- No prelude. Everything has to be defined or imported.
- More advanced module system (open import).
- Function definitions look similar to Haskell.

Datatypes

```
data Bool : Set where
  false : Bool
  true  : Bool
```

Datatypes

```
data Bool : Set where
  false : Bool
  true  : Bool
```

Remarks:

- GADT syntax. List constructors with their type signatures.
- Instead of kinds, we have a type of types called Set.
- Identifier names are quite liberal. In particular, constructor names do not have to start with an uppercase letter.
- A single colon (:) is used for the "is-of-type" relation.

Functions

$$
\begin{aligned}
\text{not} &: \text{Bool} \rightarrow \text{Bool} \\
\text{not false} &= \text{true} \\
\text{not true} &= \text{false}
\end{aligned}
$$

Functions

$$
\begin{aligned}
&\text{not} : \text{Bool} \rightarrow \text{Bool} \\
&\text{not false} = \text{true} \\
&\text{not true} = \text{false}
\end{aligned}
$$

Remarks:

- Functions can be defined via pattern matching.
- Unicode symbols such as $\rightarrow$ are allowed, also for user-defined identifiers.
- Type signatures for abstractions are required.

Functions

$$\begin{aligned} \text{not} &: \text{Bool} \to \text{Bool} \\ \text{not false} &= \text{true} \\ \text{not true} &= \text{false} \end{aligned}$$

Remarks:

- Functions can be defined via pattern matching.
- Unicode symbols such as $\to$ are allowed, also for user-defined identifiers.
- Type signatures for abstractions are required.

$$\neg = \text{not}$$

Infix and mixfix

```
infixr 6 _∧_
_∧_ : Bool → Bool → Bool
false ∧ _ = false
true ∧ x = x
```

Infix and mixfix

```
infixr 6 _∧_
_∧_ : Bool → Bool → Bool
false ∧ _ = false
true ∧ x = x
```

Remarks:

- Underscores mark argument positions.
- Infix declarations as in Haskell.
- Not only binary operators are allowed.

Infix and mixfix

```
infixr 6 _∧_
_∧_ : Bool → Bool → Bool
false ∧ _ = false
true ∧ x = x
```

Remarks:

- Underscores mark argument positions.
- Infix declarations as in Haskell.
- Not only binary operators are allowed.

```
if_then_else_ : {A : Set} → Bool → A → A → A
if_then_else_ true t e = t
if_then_else_ false t e = e
```

Polymorphism

The type of if_then_else_ is polymorphic:

```
if_then_else_ : {A : Set} → Bool → A → A → A
```

Polymorphism

The type of if_then_else_ is polymorphic:

```
if_then_else_ : {A : Set} → Bool → A → A → A
```

Remarks:

- Recall System F: polymorphism is abstraction over type arguments.
- Here, A is a type argument, hence A : Set. Again, no syntactic restrictions on names.
- An argument in curly braces is **implicit**. Agda will try to infer it wherever the function is called.

Polymorphism – contd.

```
id : {A : Set} → A → A
id x = x

const : {A B : Set} → A → B → A
const x y = x

infixr 9 _∘_

_∘_ : {A B C : Set} → (B → C) → (A → B) → (A → C
(f ∘ g) x = f (g x)
```

Pairs

```
infixr 2 _×_
infixr 4 _,_

data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

Pairs

```
infixr 2 _×_
infixr 4 _,_
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

Remarks:

- The type arguments A and B are called **parameters** of the datatype.
- The parameters become implicit arguments of the constructor, i.e.,
  _,_ : {A B : Set} → A → B → A × B.
- Due to the infix declarations, we can use nested pairs for arbitrary tuples: false , false , true : Bool × Bool × Bool.

Pitfall: identifier names and spacing

- In Agda, names may consist of nearly arbitrary combinations of letters and symbols.
- Often, explicit spaces are required to make Agda realize that one token ends and the next begins.
- Compare

```
false,false,true
false, false, true
false , false , true
```

The first is one identifier, the second is an application of the function "false," to itself and the Boolean "true", the third is a triple of three Booleans.

Maybe

```
data Maybe (A : Set) : Set where
  just    : A → Maybe A
  nothing : Maybe A
```

The "fold" on Maybe:

```
maybe : {A B : Set} → B → (A → B) → Maybe A → B
maybe n j nothing = n
maybe n j (just x) = j x
```

Totality

Let's try to define Haskell's fromJust:

```
fromJust : {A : Set} → Maybe A → A
fromJust (just x) = x
```

Totality

Let's try to define Haskell's fromJust:

```
fromJust : {A : Set} → Maybe A → A
fromJust (just x) = x
```

- This definition will cause an "Incomplete pattern matching" error.
- Agda definitions must be **total**, i.e., defined on all possible inputs and terminating.

Interactive development

Agda's Emacs mode supports interactive development. We can type a question mark and then load the file into Emacs:

```
maybe : {A B : Set} → B → (A → B) → Maybe A → B
maybe n j m = ?
```

Interactive development

Agda's Emacs mode supports interactive development. We can type a question mark and then load the file into Emacs:

```
maybe : {A B : Set} → B → (A → B) → Maybe A → B
maybe n j m = { }0
```

Emacs turns the question mark into a numbered **goal**, and you can obtain information about the goal such as its type and the local context.

Case splitting and refining

If we want to pattern match on m, we can do it interactively, by placing m in the goal and selecting the "Case" command.

```
maybe : {A B : Set} → B → (A → B) → Maybe A → B
maybe n j m = {m}0
```

Case splitting and refining

If we want to pattern match on m, we can do it interactively, by placing m in the goal and selecting the "Case" command.

```
maybe : {A B : Set} → B → (A → B) → Maybe A → B
maybe n j (just y) = { }0
maybe n j nothing  = { }1
```

Case splitting and refining

If we want to pattern match on m, we can do it interactively, by placing m in the goal and selecting the "Case" command.

```
maybe : {A B : Set} → B → (A → B) → Maybe A → B
maybe n j (just y) = { }0
maybe n j nothing  = { n }1
```

Goals can be filled with type-correct expressions.

Case splitting and refining

If we want to pattern match on m, we can do it interactively, by placing m in the goal and selecting the "Case" command.

```
maybe : {A B : Set} → B → (A → B) → Maybe A → B
maybe n j (just y) = { }0
maybe n j nothing = n
```

Goals can be filled with type-correct expressions.

Case splitting and refining

If we want to pattern match on m, we can do it interactively, by placing m in the goal and selecting the "Case" command.

```
maybe : {A B : Set} → B → (A → B) → Maybe A → B
maybe n j (just y) = {j}0
maybe n j nothing = n
```

Goals can be filled with type-correct expressions.

Goals can also be refined with a correct approximation.

Case splitting and refining

If we want to pattern match on m, we can do it interactively, by placing m in the goal and selecting the "Case" command.

```
maybe : {A B : Set} → B → (A → B) → Maybe A → B
maybe n j (just y) = j { }0
maybe n j nothing = n
```

Goals can be filled with type-correct expressions.

Goals can also be refined with a correct approximation.

Case splitting and refining

If we want to pattern match on m, we can do it interactively, by placing m in the goal and selecting the "Case" command.

```
maybe : {A B : Set} → B → (A → B) → Maybe A → B
maybe n j (just y) = j {y}0
maybe n j nothing = n
```

Goals can be filled with type-correct expressions.

Goals can also be refined with a correct approximation.

Case splitting and refining

If we want to pattern match on m, we can do it interactively, by placing m in the goal and selecting the "Case" command.

```
maybe : {A B : Set} → B → (A → B) → Maybe A → B
maybe n j (just y) = j y
maybe n j nothing = n
```

Goals can be filled with type-correct expressions.

Goals can also be refined with a correct approximation.

Lists

```
infixr 5 _::_

data List (A : Set) : Set where
  []    : List A
  _::_  : A → List A → List A
```

Lists

```
infixr 5 _::_

data List (A : Set) : Set where
  []    : List A
  _::_  : A → List A → List A
```

```
map : {A B : Set} → (A → B) → List A → List B
map f []          = []
map f (x :: xs)   = f x :: map f xs

infixr 5 _++_

_++_ : {A : Set} → List A → List A → List A
[] ++ ys          = ys
(x :: xs) ++ ys   = x :: (xs ++ ys)
```

Fold on lists

```
foldr : {A B : Set} → (A → B → B) → B → List A →
foldr op e []          = e
foldr op e (x :: xs)   = op x (foldr op e xs)

and : List Bool → Bool
and = foldr _∧_ false
```

Fold on lists

```
foldr : {A B : Set} → (A → B → B) → B → List A →
foldr op e []          = e
foldr op e (x :: xs)   = op x (foldr op e xs)

and : List Bool → Bool
and = foldr _∧_ false
```

Once again, we cannot define partial functions such as head and tail.

Fold on lists

```
foldr : {A B : Set} → (A → B → B) → B → List A →
foldr op e []          = e
foldr op e (x :: xs)   = op x (foldr op e xs)

and : List Bool → Bool
and = foldr _∧_ false
```

Once again, we cannot define partial functions such as head and tail.

What about termination?

Termination

What if we define a nonterminating function?

```
repeat : {A : Set} → A → List A
repeat x = x :: repeat x
```

Termination

What if we define a nonterminating function?

```
repeat : {A : Set} → A → List A
repeat x = x :: repeat x
```

Agda's termination checker fails and Emacs highlights the offending function.

Termination – contd.

How does the termination checker accept a recursive definition such as map?

```
map : {A B : Set} → (A → B) → List A → List B
map f []        = []
map f (x :: xs) = f x :: map f xs
```

Termination – contd.

How does the termination checker accept a recursive definition such as map?

```
map : {A B : Set} → (A → B) → List A → List B
map f []        = []
map f (x :: xs) = f x :: map f xs
```

The recursion here is **structural**: one of the arguments in the recursive call is a subterm of the argument in the original call (here xs compared to x :: xs).

Natural numbers

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

Natural numbers

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

We can use more convenient notation for natural numbers if we declare:

```
{-# BUILTIN NATURAL ℕ #-}
{-# BUILTIN ZERO zero #-}
{-# BUILTIN SUC suc #-}
```

Natural numbers

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

We can use more convenient notation for natural numbers if we declare:

```
{-# BUILTIN NATURAL ℕ #-}
{-# BUILTIN ZERO zero #-}
{-# BUILTIN SUC suc #-}
```

Now, for instance, 3 is an abbreviation for suc (suc (suc zero)).

lookup

```
lookup : {A : Set} → ℕ → List A → Maybe A
lookup n       []        = nothing
lookup zero    (x :: xs) = just x
lookup (suc n) (x :: xs) = lookup n xs
```

filter

```
filter : {A : Set} → (A → Bool) → List A → List A
filter p []          = []
filter p (x :: xs)   = if p x then x :: filter p xs
                               else      filter p xs
```

filter

```
filter : {A : Set} → (A → Bool) → List A → List A
filter p []          = []
filter p (x :: xs)   = if p x then x :: filter p xs
                               else      filter p xs
```

Can we abbreviate the repeated term?

Local definitions

```
filter : {A : Set} → (A → Bool) → List A → List A
filter p []        = []
filter p (x :: xs) = let rec = filter p xs
                     in  if p x then x :: rec
                                else      rec
```

```
filter : {A : Set} → (A → Bool) → List A → List A
filter p []        = []
filter p (x :: xs) = if  p x then x :: rec
                             else      rec
  where rec = filter p xs
```

No case, but with

```
filter : {A : Set} → (A → Bool) → List A → List A
filter p []       = []
filter p (x :: xs) with p x
... | true  = x :: filter p xs
... | false =      filter p xs
```

No case, but with

```
filter : {A : Set} → (A → Bool) → List A → List A
filter p []       = []
filter p (x :: xs) with p x
... | true  = x :: filter p xs
... | false =      filter p xs
```

Remarks:

- The with construct is tied to the declaration, it cannot appear in the middle of an expression like case.
- Neither let nor where scope over multiple branches of a with construct.
- Instead of writing ..., we can repeat (or later refine) the previous left hand side.

Types are like values

Type synonyms are just definitions:

$$\begin{aligned}
&\mathsf{Nat} : \mathsf{Set} \\
&\mathsf{Nat} = \mathbb{N} \\
&\mathsf{NatList} : \mathsf{Set} \\
&\mathsf{NatList} = \mathsf{List\ Nat} \\
&\mathsf{ListList} : \mathsf{Set} \to \mathsf{Set} \\
&\mathsf{ListList\ A} = \mathsf{List\ (List\ A)}
\end{aligned}$$

Vectors I

A vector of length n holds n values of the same type.

Vectors I

A vector of length n holds n values of the same type.

The "empty" type is a datatype without constructors:

```
data ⊥ : Set where
```

Vectors I

A vector of length n holds n values of the same type.

The "empty" type is a datatype without constructors:

```
data ⊥ : Set where
```

We can now compute the Vector type:

```
Vector : Set → ℕ → Set
Vector A zero    = ⊥
Vector A (suc n) = A × Vector A n
```

Vectors I

A vector of length n holds n values of the same type.

The "empty" type is a datatype without constructors:

```
data ⊥ : Set where
```

We can now compute the Vector type:

```
Vector : Set → ℕ → Set
Vector A zero    = ⊥
Vector A (suc n) = A × Vector A n
```

The type Vector is parameterized over a natural number – it is a true **dependent type**.

Safe head and tail I

```
head : {A : Set} {n : ℕ} → Vector A (suc n) → A
head (x , xs) = x

tail : {A : Set} {n : ℕ} → Vector A (suc n) → Vector A n
tail (x , xs) = xs
```

Safe head and tail I

```
head : {A : Set} {n : ℕ} → Vector A (suc n) → A
head (x , xs) = x

tail : {A : Set} {n : ℕ} → Vector A (suc n) → Vector A n
tail (x , xs) = xs
```

Remarks:

- We give the name n to the number we abstract over and use it later! So head and tail are **dependent functions**.
- Not only type-level arguments can be implicit! The natural number here can be inferred from the context.
- Multiple named arguments can be juxtaposed in a type signature, without function arrows in between. Such a construct is called a **telescope**.

A safe lookup function?

Can we write a function

$$\text{lookup} : \{A : \text{Set}\}\,\{n : \mathbb{N}\} \to \mathbb{N} \to \text{Vector}\ A\ n \to A$$

now?

A safe lookup function?

Can we write a function

$$\text{lookup} : \{A : \text{Set}\}\,\{n : \mathbb{N}\} \to \mathbb{N} \to \text{Vector}\ A\ n \to A$$

now?

No, we need a way to restrict the index to be a number between 0 and $n - 1$.

Finite types I

The type Finite n should be a type with n distinct elements.

Finite types I

The type Finite n should be a type with n distinct elements.

We prepare by defining the "unit type" ⊤ and the disjoint union of two types _⊎_ (like Either in Haskell):

```
data ⊤ : Set where
  tt : ⊤

infixr 1 _⊎_

data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

Finite types I

The type Finite n should be a type with n distinct elements.

We prepare by defining the "unit type" ⊤ and the disjoint union of two types _⊎_ (like Either in Haskell):

```
data ⊤ : Set where
  tt : ⊤

infixr 1 _⊎_

data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

Now:

```
Finite : ℕ → Set
Finite zero    = ⊥
Finite (suc n) = ⊤ ⊎ Finite n
```

Defining safe lookup I

$$\begin{aligned}\mathsf{lookup} : \{A : \mathsf{Set}\}\,\{n : \mathbb{N}\} &\to \\ \mathsf{Finite}\ n \to \mathsf{Vector}\ A\ n &\to A\end{aligned}$$

How do we match on a value of type Finite or Vector without knowing n?

Defining safe lookup I

```
lookup : {A : Set} {n : ℕ} →
         Finite n → Vector A n → A
lookup {n = zero}  i xs = { }0
lookup {n = suc _} i xs = { }1
lookup {n = suc _} i xs = { }2
```

How do we match on a value of type Finite or Vector without knowing n?

Let us match on the implicit argument n first.

Defining safe lookup I

```
lookup : {A : Set} {n : ℕ} →
         Finite n → Vector A n → A
lookup {n = zero}  i       xs       = { }0
lookup {n = suc _} (inj₁ i) (x , xs) = x
lookup {n = suc _} (inj₂ j) (x , xs) = lookup j xs
```

How do we match on a value of type Finite or Vector without knowing n?

Let us match on the implicit argument n first.

We can now complete the suc case.

Defining safe lookup I

```
lookup : {A : Set} {n : ℕ} →
         Finite n → Vector A n → A
lookup {n = zero}  ()        ()
lookup {n = suc _} (inj₁ i) (x , xs) = x
lookup {n = suc _} (inj₂ j) (x , xs) = lookup j xs
```

How do we match on a value of type Finite or Vector without knowing n?

Let us match on the implicit argument n first.

We can now complete the suc case.

We explain to Agda that the zero case is impossible by using the **absurd pattern** with no right hand side.

Vectors II

Often, dependent types can be more nicely expressed using **inductive families**:

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

Vectors II

Often, dependent types can be more nicely expressed using **inductive families**:

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

- Agda supports overloading of constructors – we can reuse the constructor names of List.
- The definition looks like a GADT in Haskell, but the restricted argument (the **index**) is a natural number, not a type.
- The index changes per constructor, so we cannot give it a global name for the whole family and move it to the right of the colon.

Safe head and tail II

```
head : {A : Set} {n : ℕ} → Vec A (suc n) → A
head (x :: xs) = x

tail : {A : Set} {n : ℕ} → Vec A (suc n) → Vec A n
tail (x :: xs) = xs
```

Finite types II

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

Note that there are no constructors that target Fin zero.

Finite types II

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

Note that there are no constructors that target Fin zero.

Yet another way of defining Fin would be

```
Fin : ℕ → Set
Fin n = Vec ⊤ n
```

Defining safe lookup II

```
lookup : {A : Set} {n : ℕ} → Fin n → Vec A n → A
lookup zero    (x :: xs) = x
lookup (suc i) (x :: xs) = lookup i xs
```

Defining safe lookup II

```
lookup : {A : Set} {n : ℕ} → Fin n → Vec A n → A
lookup zero    (x :: xs) = x
lookup (suc i) (x :: xs) = lookup i xs
```

Remarks:

- We can match on the Fin directly now, because it is a datatype – no need to match on the implicit natural number.
- As a consequence, Agda can figure out automatically that the case where n is zero is impossible – no need to use the absurd pattern.

Appending vectors

```
infixl 6 _+_
_+_ : ℕ → ℕ → ℕ
zero  + n = n
suc m + n = suc (m + n)
_++_ : {A : Set} {m n : ℕ} → Vec A m → Vec A n → Vec
[] ++ ys = ys
(x :: xs) ++ ys = x :: xs ++ ys
```

Appending vectors

```
infixl 6 _+_
_+_ : ℕ → ℕ → ℕ
zero  + n = n
suc m + n = suc (m + n)
_++_ : {A : Set} {m n : ℕ} → Vec A m → Vec A n → Vec
[] ++ ys = ys
(x :: xs) ++ ys = x :: xs ++ ys
```

Arbitrary computations can appear in types. This is one of the reasons why Agda tries to insist on totality.

When are two types equal?

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

In the first line of _++_, how does Agda see that Vec A n is the same type as Vec A (zero + n)?

When are two types equal?

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

In the first line of _++_, how does Agda see that Vec A n is the same type as Vec A (zero + n)?

Agda applies definitions symbolically when testing for type equality.

For the same reason, the second line typechecks.

Definitional equality is limited

This definition of vector reverse fails to typecheck:

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

The type error states that Agda does not know that $n + 1$ is suc n! (It knows that $1 + n$ is suc n.)

A workaround

```
snoc : {A : Set} {n : ℕ} →
       Vec A n → A → Vec A (suc n)
snoc []       y = y :: []
snoc (x :: xs) y = x :: snoc xs y

reverse : {A : Set} {n : ℕ} → Vec A n → Vec A n
reverse [] = []
reverse (x :: xs) = snoc (reverse xs) x
```

Proving equalities

```
infix 4 _≡_
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Proving equalities

```
infix 4 _≡_
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Remarks:

- A value of type x ≡ y serves as a proof that x and y are equal.
- The only way to construct such a proof is by reflexivity.
- Pattern matching exposes the equality to the type checker.

Properties of equality

```
sym : {A : Set} {x y : A} → x ≡ y → y ≡ x
sym refl = refl

trans : {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans refl refl = refl

cong : {A B : Set} (f : A → B) {x y : A} →
       x ≡ y → f x ≡ f y
cong f refl = refl

subst : {A : Set} (P : A → Set) {x y : A} → x ≡ y → P x
subst P refl p = p
```

Proving that $\mathsf{n} + 1$ is $1 + \mathsf{n}$

```
n+1≡1+n : (n : ℕ) → n + 1 ≡ 1 + n
n+1≡1+n zero    = refl
n+1≡1+n (suc n) = cong suc (n+1≡1+n n)
```

Defining reverse again

```
reverse : {A : Set} {n : ℕ} → Vec A n → Vec A n
reverse []          = []
reverse (x :: xs)   = subst (Vec _) (n+1≡1+n _)
                      (reverse xs ++ x :: [])
```

Defining reverse again

```
reverse : {A : Set} {n : ℕ} → Vec A n → Vec A n
reverse []          = []
reverse (x :: xs)   = subst (Vec _) (n+1≡1+n _)
                      (reverse xs ++ x :: [])
```

Note that we can use _ on the right hand side to tell Agda to infer arguments that are known from the context.