

# Verified Programming in Agda

Ian Orton



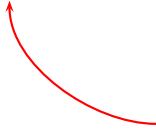
Research Students' Lecture Series, 2017

# Agda

- ▶ Dependently typed functional programming language
- ▶ Originally developed by Ulf Norell in 2007
- ▶ Both a programming language and a proof assistant
- ▶ Total - all functions terminate!

# Agda

- ▶ Dependently typed functional programming language
- ▶ Originally developed by Ulf Norell in 2007
- ▶ Both a programming language and a proof assistant
- ▶ Total - all functions terminate!



Not Turing complete

# Type systems

- ▶ Assign types to expressions  
e.g. `int f(string input) {...}`
- ▶ Check at compile time that types match  
e.g. `f("abc")`    `f(42)`
- ▶ Usually involves more work for the programmer
- ▶ But lets us find bugs at compile time

# Inductive types

- ▶ Main way of adding new types in a functional language
- ▶ Describe how terms are built up from a series of constructors
- ▶ E.g. lists are either:
  - ▶ The empty list
  - ▶ The first element and the tail of the list

# Lists

Lists in OCaml can be defined like so:

```
type 'a list = Nil | Cons of 'a * 'a list
```

So, for example, the empty list is:

```
let emp = Nil
```

And the list [0,1,2,3] is:

```
let upto3 =  
  Cons (0, Cons (1, Cons (2, Cons (3, Nil))))
```

# Lists

Lists in Agda can be defined like so:

```
infixr 4 _::_  
data List(A : Set) : Set where  
  []      : List A  
  _::__   : A → List A → List A
```

So, for example, the empty list is:

```
empty : List ℕ  
empty = []
```

And the list  $[0,1,2,3]$  is:

```
upto3 : List ℕ  
upto3 = 0 :: 1 :: 2 :: 3 :: []
```

# Pattern matching

You use inductive types by **pattern matching**. Essentially case analysis on the outermost constructor of an argument.



# Pattern matching

You use inductive types by **pattern matching**. Essentially case analysis on the outermost constructor of an argument.

For example, we can define the append function by pattern matching on the first argument:

```
append :  $\forall \{A\} \rightarrow \text{List } A \rightarrow \text{List } A \rightarrow \text{List } A$   
append [] ys = ys  
append (x :: xs) ys = x :: append xs ys
```

# Getting the head of a list

Let's try writing a function to get the head of a list:

`first` :  $\forall \{A\} \rightarrow \text{List } A \rightarrow A$

`first xs = { }0`

# Getting the head of a list

Let's try writing a function to get the head of a list:

`first` :  $\forall \{A\} \rightarrow \text{List } A \rightarrow A$

`first` `[]` = `{ }0`

`first` `(x :: xs)` = `{ }1`

# Getting the head of a list

Let's try writing a function to get the head of a list:

`first` :  $\forall \{A\} \rightarrow \text{List } A \rightarrow A$

`first` `[]` = `{ }0`

`first` (`x :: _`) = `x`

# Getting the head of a list

Let's try writing a function to get the head of a list:

```
data Option(A : Set) : Set where
  Some : A → Option A
  None : Option A
```

```
first : ∀ {A} → List A → Option A
first [] = None
first (x :: _) = Some x
```

# A problem

Sometimes the type system can get in our way:

...

```
let queue = insert input queue in
```

```
let min   = first queue      in
```

```
let x     = min + 5          in
```

...

# A problem

Sometimes the type system can get in our way:

```
...  
let queue = insert input queue in  
let min   = first queue      in  
let x     = min + 5          in  
...
```

Option  $\mathbb{N} \leq \mathbb{N}$  of type Set when checking that the expression min has type  $\mathbb{N}$

# Dependent types

- ▶ Types which depend on a value
- ▶ Allow us to enforce stronger constraints at compile time
- ▶ For example: lists of length  $n$



# Dependent types - Vectors

Recall the definition of lists in Agda:

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

# Dependent types - Vectors

Recall the definition of lists in Agda:

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

We can define lists of length  $n$  like so:

```
data Vec(A : Set) : ℕ → Set where
  []      : Vec A 0
  _:::_   : ∀ {n} → A → Vec A n → Vec A (1 + n)
```

# Vectors

Nothing changes about the examples from before except the type:

`empty` : `Vec` `ℕ` `0`

`empty` = `[]`

`upto3` : `Vec` `ℕ` `4`

`upto3` = `0 :: 1 :: 2 :: 3 :: []`

# Vectors

Similarly for append, nothing changes except the type:

`append` :  $\forall \{A\ m\ n\} \rightarrow$

`Vec A m`  $\rightarrow$  `Vec A n`  $\rightarrow$  `Vec A (m + n)`

`append [] ys` = `ys`

`append (x :: xs) ys` = `x :: append xs ys`

# Getting the head of a list - revisited

Using vectors we can define first like so:

$$\begin{aligned} \text{first} &: \forall \{A\ n\} \rightarrow \text{Vec } A \ (1 + n) \rightarrow A \\ \text{first } xs &= \{ \} 0 \end{aligned}$$

# Getting the head of a list - revisited

Using vectors we can define first like so:

$$\begin{aligned} \text{first} &: \forall \{A\ n\} \rightarrow \text{Vec } A \ (1 + n) \rightarrow A \\ \text{first } (x :: xs) &= \{ \} 0 \end{aligned}$$

# Getting the head of a list - revisited

Using vectors we can define first like so:

$$\begin{aligned} \text{first} &: \forall \{A\ n\} \rightarrow \text{Vec } A \ (1 + n) \rightarrow A \\ \text{first } (x :: \_) &= x \end{aligned}$$

# A problem - revisited

Consider the code from before:

```
...  
let queue = insert input queue in  
let min   = first queue      in  
let x     = min + 5           in  
...
```




# A problem - revisited

Consider the code from before:

*Vec*  $\mathbb{N}$  *n*

```
...  
let queue = insert input queue in  
let min   = first queue   in  
let x     = min + 5        in  
...
```



# A problem - revisited

Consider the code from before:

$$\forall \{n\} \rightarrow \mathbb{N} \rightarrow \\ \text{Vec } \mathbb{N} \ n \rightarrow \text{Vec } \mathbb{N} \ (1 + n)$$

*Vec*  $\mathbb{N} \ n$

...

let queue = insert input queue in  
let min = first queue in  
let x = min + 5 in

...

# A problem - revisited

Consider the code from before:

$\forall \{n\} \rightarrow \mathbb{N} \rightarrow$

$Vec \mathbb{N} n \rightarrow Vec \mathbb{N} (1 + n)$

$Vec \mathbb{N} n$

...

let queue = insert input queue in

let min = first queue in

let x = min + 5 in

...

$Vec \mathbb{N} (1 + n)$

# Generalising the problem

The previous example was an instance of a more general problem: **out of bounds** errors. Using vectors we can solve this problem more generally.

# Generalising the problem

The previous example was an instance of a more general problem: **out of bounds** errors. Using vectors we can solve this problem more generally.

We would like to define the following:

$$\_!\_ : \forall \{A\ n\} \rightarrow \text{Vec } A\ n \rightarrow \mathbb{N} < n \rightarrow A$$

How can we define the type  $\mathbb{N} <$ ?

# The type $\mathbb{N} <$

What is a number less than  $n$ ? Can we give an inductive definition?

# The type $\mathbb{N} <$

What is a number less than  $n$ ? Can we give an inductive definition?

```
data N< : N → Set where
  zero  : ∀ {n} → N< (1 + n)
  1+    : ∀ {n} → N< n → N< (1 + n)

three   : N< 7
three   = 1+ (1+ (1+ zero))

three'  : N< 10
three'  = 1+ (1+ (1+ zero))
```

# Lookup

Now we can define the lookup function `_!_`:

$$\begin{aligned} \_!\_ &: \forall \{A\ n\} \rightarrow \text{Vec } A\ n \rightarrow \mathbb{N} < n \rightarrow A \\ (x :: \_) \ ! \ \text{zero} &= x \\ (\_ :: xs) \ ! \ (1 + m) &= xs \ ! \ m \end{aligned}$$

Out of bounds errors are now caught at compile time:



# Lookup

Now we can define the lookup function `_!_`:

$$\begin{aligned} \_!\_ &: \forall \{A\ n\} \rightarrow \text{Vec } A\ n \rightarrow \mathbb{N} < n \rightarrow A \\ (x :: \_) &!\ \text{zero} = x \\ (\_ :: xs) &!\ (1+ m) = xs\ !\ m \end{aligned}$$

Out of bounds errors are now caught at compile time:

```
one = upto3 ! (1+ zero)
out-of-bounds =
  upto3 ! (1+ (1+ (1+ (1+ zero))))
```

# Lookup

Now we can define the lookup function `_!_`:

$$\begin{aligned} \_!\_ &: \forall \{A\ n\} \rightarrow \text{Vec } A\ n \rightarrow \mathbb{N} < n \rightarrow A \\ (x :: \_) &!\ \text{zero} = x \\ (\_ :: xs) &!\ (1 + m) = xs\ !\ m \end{aligned}$$

Out of bounds errors are now caught at compile time:

```
one = upto3 ! (1+ zero)
out-of-bounds =
  upto3 ! (1+ 1+ 1+ 1+ zero))
```

# Proofs

As well as refining types, dependent types can be used to encode proofs. For example, proofs that one number is less than another.

In Agda, proofs are actual data that we can compute with. This is known as **proof relevance**.

# Insert

Consider the definition of insert:

```
insert :  $\forall \{n\} \rightarrow \mathbb{N} \rightarrow \text{Vec } \mathbb{N} \ n \rightarrow \text{Vec } \mathbb{N} \ (1 + n)$   
insert x [] = x :: []  
insert x (y :: ys) with x ≤? y  
... | yes _ = x :: y :: ys  
... | no _ = y :: insert x ys
```

# Insert

Consider the definition of insert:

`insert` :  $\forall \{n\} \rightarrow \mathbb{N} \rightarrow \text{Vec } \mathbb{N} \ n \rightarrow \text{Vec } \mathbb{N} \ (1 + n)$

`insert`  $x \ [] = x :: []$

`insert`  $x \ (y :: ys)$  `with`  $x \leq? y$

... | `yes`  $\_ = x :: y :: ys$

... | `no`  $\_ = y :: \text{insert } x \ ys$

`_≤_` :  $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Set}$

`_≤?_` :  $(x \ y : \mathbb{N}) \rightarrow \text{Dec } (x \leq y)$

# Less than or equal

When is  $m \leq n$ ?

What would a proof of this look like?

# Less than or equal

When is  $m \leq n$ ?

What would a proof of this look like?

In Agda:

```
data _≤_ : ℕ → ℕ → Set where
  0≤n : ∀ {n} → 0 ≤ n
  1+≤ : ∀ {m n} →
    m ≤ n → (1 + m) ≤ (1 + n)
```

# $\mathbb{N}<$ - revisited

Another way to define the type  $\mathbb{N}<$ :

```
record N<'(n : N) : Set where
  field
    m      : N
    m<n    : m < n
```



# Sortedness

When is a list sorted?

What would a proof of this look like?

a

# Sortedness

When is a list sorted?

What would a proof of this look like?

a

```
data Sorted : List N → Set where
  emp      : Sorted []
  con      : ∀ {x : N} {xs : List N}
    → x ≤-all xs → Sorted xs → Sorted (x :: xs)
```

# Sortedness

When is a list sorted?

What would a proof of this look like?

a

```
data Sorted : List N → Set where
  emp   : Sorted []
  con   : ∀{x : N}{xs : List N}
    → x ≤-all xs → Sorted xs → Sorted (x :: xs)
```

```
data ≤-all_ (x : N) : List N → Set where
  ≤[] : x ≤-all []
  ≤:: : ∀{y : N}{ys : List N}
    → x ≤ y → x ≤-all ys → x ≤-all (y :: ys)
```

# Insert preserves sortedness

Now we can prove the following:

`insert-preserves-sortedness :`

`∀ (x : N)(ys : List N)  
 → Sorted ys → Sorted (insert x ys)`

# Insert preserves sortedness

Not quite as simple as previous examples:

```
insert-preserves-sortedness :  
  ∀ (x : N)(ys : List N)  
    → Sorted ys → Sorted (insert x ys)  
insert-preserves-sortedness x [] emp = con ≤[] emp  
insert-preserves-sortedness x (y :: ys) (con ysys ys-sorted) with x ≤? y  
... | yes xsy = con (≤:: xsy (lemma ys ysys)) (con ysys ys-sorted) where  
  lemma : (ys : List N) → y ≤-all ys → x ≤-all ys  
  lemma [] ≤[] = ≤[]  
  lemma (y' :: ys) (≤:: ysy' ysys) = ≤:: (≤-trans xsy ysy') (lemma ys ysys)  
... | no ¬xsy = con (lemma ys ysys) (insert-preserves-sortedness x ys ys-sorted) where  
  ysx : y ≤ x  
  ysx with ≤-total x y  
  ysx | inj1 xsy = contradiction xsy ¬xsy  
  ysx | inj2 ysx = ysx  
  lemma : (ys : List N) → y ≤-all ys → y ≤-all insert x ys  
  lemma [] ≤[] = ≤:: ysx ≤[]  
  lemma (y' :: ys) (≤:: ysy' ysys) with x ≤? y'  
  ... | yes xsy' = ≤:: ysx (≤:: ysy' ysys)  
  ... | no ¬xsy' = ≤:: ysy' (lemma ys ysys)
```

# No silver bullet

# No silver bullet

We could have defined insert like so:

`insert' : N → List N → List N`

`insert' x ys = ys`

# No silver bullet

We could have defined insert like so:

```
insert' : N → List N → List N  
insert' x ys = ys
```

And we could have proved:

```
insert'-preserves-sortedness :  
  ∀ (x : N)(ys : List N)  
    → Sorted ys → Sorted (insert' x ys)  
insert'-preserves-sortedness x ys sorted = sorted
```



# Thanks for listening!

In conclusion, Agda's type system:

- ▶ Allows you to track static information e.g. length
- ▶ Allows you to constrain and refine your types
- ▶ Allows you to prove properties of your code
- ▶ Is not a silver bullet

## Any Questions?