Agda II: Dependent Types

Type Theory and Mechanized Reasoning

Lecture 4

CAS CS 400

Introduction

Administrivia

- Assignment 1 will be released tomorrow (it will be short)
- Its possible to register for the course (we have 4 people!)

Objectives

1. Look at (play with) dependent types. Our goal is not to understand dependent types, but see what happens when we can use them.

2. Draw a connection to induction, and start looking at what this means for using Agda as a proof assistant.

Recap

Recall: Named Parameters

```
f : (a : ℕ) → ℕ
f x = x
```

Recall: Named Parameters

```
f : (a : ℕ) → ℕ
f x = x
```

The fundamental feature of Agda is that we can

» name parameters
» **use those names elsewhere in the type.**

Recall: Named Parameters

named parameter

```
f : (a : ℕ) → ℕ
f x = x
```

The fundamental feature of Agda is that we can

» name parameters
» **use those names elsewhere in the type.**

Recall: Named Parameters

```
            named parameter
f : (a : ℕ) → ℕ
f x = x
```

The fundamental feature of Agda is that we can

» name parameters
» **use those names elsewhere in the type.**

The named stands for the value that will be passed into the function.

Recall: Polymorphism

```
id : {a : Set} → a → a
id x = x
```

Recall: Polymorphism

```
id : {a : Set} → a → a
id x = x
```

We get polymorphism from this feature.

Recall: Polymorphism

```
id : {a : Set} → a → a
id x = x
```

We get polymorphism from this feature.

The type **a** refers to the one that *will* be passed in when **id** is called.

Recall: Types are First-Class

```
indexType : Set
indexType = ℕ

mkType : ℕ → Set
mkType 0 = ℕ
mkType (suc _) = Bool
```

Types can be used anywhere we use values.

We get type synonyms from this feature.

Recall: Generalized Algebraic Data Types

```
data TypedBox : Set → Set where
  natBox : ℕ → TypedBox ℕ
  boolBox : Bool → TypedBox Bool
```

We get GADTs from this feature.

(no worries if you missed this on Monday)

Practice Problem

```
Let's look at the code...
```

Playing with Dependent Types

What's next?

The *dependent* part of dependent types is the fact that the output type can depend on the input **value**.

How can we use a value like a number or a list in a type?

First Element

```
head : {a : Set} → List a → Maybe a
head [] = nothing
head (x :: _) = just x
```

First Element

```
head : {a : Set} → List a → Maybe a
head [] = nothing
head (x :: _) = just x
```

`head` is tricky to implement in Agda because it's not naturally total.

First Element

```
head : {a : Set} → List a → Maybe a
head [] = nothing
head (x :: _) = just x
```

`head` is tricky to implement in Agda because it's not naturally total.

What do we do on an empty list?

Exception or Monad

```
head : {a : Set} → List a → Maybe a
head [] = nothing
head (x :: _) = just x
```

Usually you have two options:

» Throw an exception
» Work with Maybes or Results (as above)

Exception or Monad

```
head : {a : Set} → List a → Maybe a
head [] = nothing
head (x :: _) = just x
```

Usually you have two options:

» Throw an exception
» Work with Maybes or Results (as above)

What if the *types* forced us to use this function correctly?

demo

Non-Empty Lists

```
data NonEmpty : {a : Set} → List a → Set where
  hasFirst :
    {a : Set} →
    {x : a} →
    {xs : List a} →
    NonEmpty (x ∷ xs)
```

Non-Empty Lists

```
data NonEmpty : {a : Set} → List a → Set where
  hasFirst :
    {a : Set} →
    {x : a} →
    {xs : List a} →
    NonEmpty (x :: xs) named parameters appearing later
```

Non-Empty Lists

```
data NonEmpty : {a : Set} → List a → Set where
  hasFirst :
    {a : Set} →
    {x : a} →
    {xs : List a} →
    NonEmpty (x :: xs) named parameters appearing later
```

NonEmpty has one constructor.

Non-Empty Lists

```
data NonEmpty : {a : Set} → List a → Set where
  hasFirst :
    {a : Set} →
    {x : a} →
    {xs : List a} →
    NonEmpty (x :: xs) named parameters appearing later
```

NonEmpty has one constructor.

It is <u>impossible</u> to build something of type:

NonEmpty []

First Element (Again)

```
head : {a : Set} → (l : List a) → NonEmpty l → a
head (x :: _) hasFirst = x
```

Our new version requires evidence which *guarantees* that the input is nonempty.

We can never accidentally call **head** on a nonempty list.

Totality of head

```
head : {a : Set} → (l : List a) → NonEmpty l → a
head (x :: _) hasFirst = x
```

Totality of head

```
head : {a : Set} → (l : List a) → NonEmpty l → a
head (x :: _) hasFirst = x
```

How does Agda know this function is total?

Totality of head

```
head : {a : Set} → (l : List a) → NonEmpty l → a
head (x :: _) hasFirst = x
```

How does Agda know this function is total?

<u>Answer</u>: The **hasFirst** pattern enforces that **[]** is not a valid pattern for **l**. (strange)

Vectors

Vectors

Vectors are fixed-length lists.

They are a canonical example of a useful form of dependent types.

Let's do a demo.

Vectors vs. Lists

```
data Vec (a : Set) : ℕ → Set where
  [] : Vec a 0
  _∷_ : {n : ℕ} → a → Vec a n → Vec a (suc n)

data List (a : Set) : Set where
  [] : List a
  _∷_ : a -> List a -> List a
```

The only difference is the added dependency on a number.

Vectors vs. Lists

```
data Vec (a : Set) : ℕ → Set where
  [] : Vec a 0
  _::_ : {n : ℕ} → a → Vec a n → Vec a (suc n)

data List (a : Set) : Set where
  [] : List a
  _::_ : a -> List a -> List a
```

The only difference is the added dependency on a number.

Example: Adding Vectors

```
addVec : {n : ℕ} → Vec ℕ n → Vec ℕ n → Vec ℕ n
addVec [] [] = []
addVec (x :: xs) (y :: ys) = (x + y) :: addVec xs ys
```

Again, how does Agda know this function is total?

Answer: The patterns for **n** influence the patterns for *both* of the following inputs.

Practice Problem

Implement a **head** *function for vectors. What should the type of this function be?*

Vector Lookup

The Idea

Since vectors are a fixed-length, we should *never* have to deal with out-of-bounds errors.

Can we implement a type which represents the possible indices of a vector?

demo

Fin vs. Nat

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

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

Fin vs. Nat

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

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

`Like vectors, Fins are like Nats with additional number information in the types.`

Fin vs. Nat

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

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

Like vectors, Fins are like Nats with additional number information in the types.

Fin vs. Nat

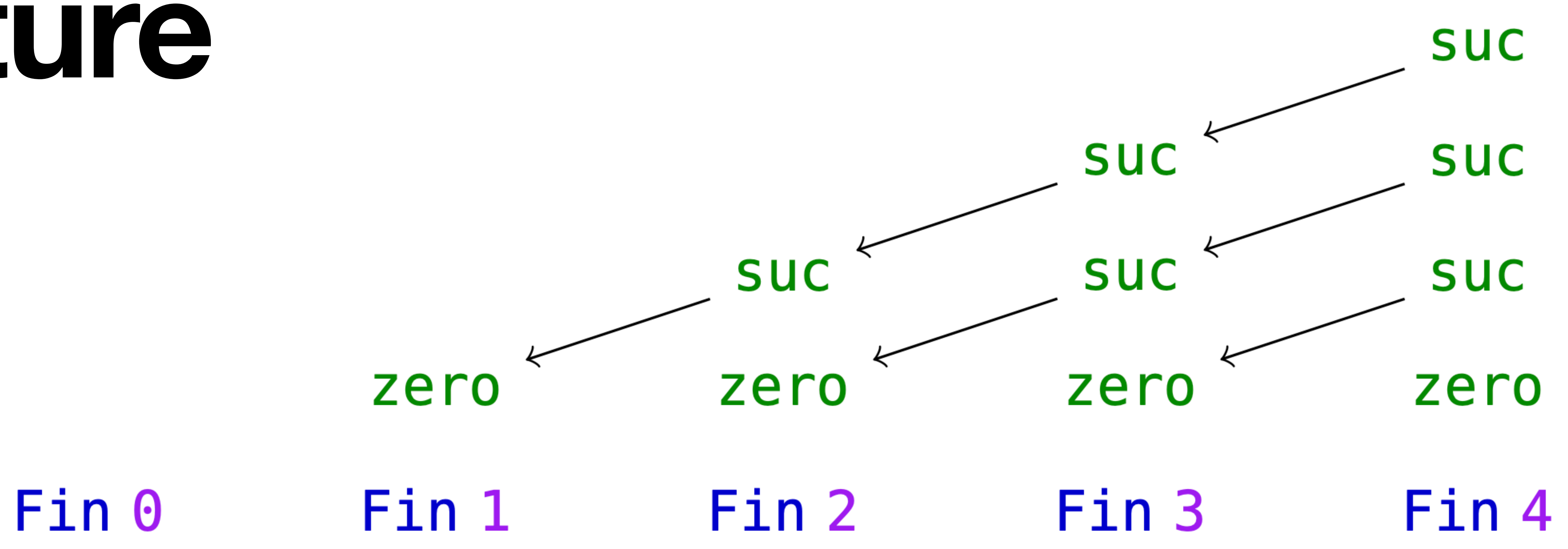
```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc : {n : ℕ} → Fin n → Fin (suc n)

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

`Like vectors, Fins are like Nats with additional number information in the types.`

`We can think of this information an` *`upper bound.`*

The Picture



The ℕ in the type tells you how many values there are:

$$\mathrm{Fin}\ n \approx \{x \in \mathbb{N} : x < n\}$$

demo

Vector Lookup

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

```
What a satisfying function definition...

The "edge cases" are "handled" by the types.
```

Practice Problem

Write a function ***dec–nat*** *which, given a ℕ* ***n,*** *constructs a* ***Vec n*** *of ℕ in decreasing order all the way to 0.*

For a challenge, try increasing order.

Induction

What is NonEmpty?

```
data NonEmpty : {a : Set} → List a → Set where
  hasFirst :
    {a : Set} →
    {x : a} →
    {xs : List a} →
    NonEmpty (x :: xs)
```

Non-Emptiness is a property of Lists.

In logic-speak, it's a predicate.

What is NonEmpty?

type of a predicate on lists

```
data NonEmpty : {a : Set} → List a → Set where
  hasFirst :
    {a : Set} →
    {x : a} →
    {xs : List a} →
    NonEmpty (x :: xs)
```

Non-Emptiness is a property of Lists.

In logic-speak, it's a predicate.

Induction on Natural Numbers

Induction on Natural Numbers

We said induction is a mathematical principle for proving that a property holds of *all* natural numbers.

Induction on Natural Numbers

We said induction is a mathematical principle for proving that a property holds of *all* natural numbers.

In Agda property of ℕ's just something of type

$$\mathbb{N} \to \mathrm{Set}$$

Induction on Natural Numbers

We said induction is a mathematical principle for proving that a property holds of *all* natural numbers.

In Agda property of ℕ's just something of type

$$\mathbb{N} \to \mathrm{Set}$$

Given P : ℕ → Set, (P 0) is the statement that the property holds of 0.

Practice Problem

Write a data type which represents the predicate on natural numbers "n is nonzero".