Detour: Inductive Types

- **Algebraic Data Types (ADT)**
  - Defined by a set of constructors.
  - Carry a guarantee that any value can only be generated by one of the constructors.

- **Recursion in ADT constructors**
  - Inductive definition: any value is guaranteed to have been created by a finite number of applications of the supplied constructors.

- **Natural numbers**
  - \( S(z) = 1 + z \)
  - \( 1 = S(0) \)
  - \( 2 = S(1) \)
  - \( 3 = S(2) \)
Detour: Inductive Types

- "match" statement

```
E ::= ... | match E with C* end
```

A constructor and some variables

```
match E with C* end
length E := list : List E.
match list with Nil => 0 | Cons x xs => 1 + length xs end
```

Inductive List E
Nil : List E
Cons : E -> List E -> List E

```
A Gentle Introduction

- We already had types depending on type parameters.
- "member polymorphism?"

```
\lambda A B C. \lambda (f:B\to C) \lambda (g:A\to B) \lambda (x:A). f (g x)
```

The type of the variable depends on assignments to the type parameters

```
\Lambda A B C. \lambda (f:B\to C) \lambda (g:A\to B) \lambda (x:A). f (g x)
```

A Gentle Introduction

- Next level: a type depending on the value of an "ordinary" parameter.

Canonical example:

```
Vec n
```

(a polymorphic type that represents an n-ary vector.)

- What would be the types of vector operators?

```
vplus : (n : nat) \to Vec n \to Vec n \to Vec n
vmult : (n : nat) \to real \to Vec n \to Vec n
dot : (n : nat) \to Vec n \to Vec n \to real
```

```
vplus : \forall n. n \to Vec n \to Vec n \to Vec n
vmult : \forall n. n \to real \to Vec n \to Vec n
dot : \forall n. n \to Vec n \to Vec n \to real
```

A Gentle Introduction

- Next level: a type depending on the value of an "ordinary" parameter.

Canonical example:

```
Vec n
```

(a polymorphic type that represents an n-ary vector.)

- What would be the types of vector operators?

```
vplus : (n : nat) \to Vec n \to Vec n \to Vec n
vmult : (n : nat) \to real \to Vec n \to Vec n
dot : (n : nat) \to Vec n \to Vec n \to real
```

```
vplus : \forall n. n \to Vec n \to Vec n \to Vec n
vmult : \forall n. n \to real \to Vec n \to Vec n
dot : \forall n. n \to Vec n \to Vec n \to real
```

```
Vec n
```

Inductive Vec (n : nat) *
VNil : Vec 0
VCons : real \to Vec n \to Vec (n+1)

- E.g. \langle 5,9,6 \rangle =
```
VC cons 5 (VC cons 9 (VC cons 6 (VNil)))
```

- Value of n can be inferred from type of arguments.

Dependent Type Definition

```
Vec n
```

Inductive Vec (n : nat) *
VNil : Vec 0
VCons : real \to Vec n \to Vec (n+1)

- E.g. \langle 5,9,6 \rangle =
```
VC cons 5 (VC cons 9 (VC cons 6 (VNil)))
```

- Value of n can be inferred from type of arguments.

Implicit Arguments

```
Inductive Pair A B *
pair : A \to B \to Pair A B
```

```
pair 2.0 True
```

- A, B can be inferred from types of 2.0, True through unification
Implicit Arguments

Inductive Vec (n : nat)  *
VNil : Vec 0
VCons : real → Vec n → Vec (n+1)
VCons 5 (VCons 9 (VCons 6 VNil))

11

Terms as Types

• A type can be a term — a term can be a type
  • Usually, terms that are used as types have a designated type, Type.
  • There is no practical difference anymore between \( \lambda / \lambda \), or \( e_1 / e_2 \).
  • Our polymorphic types become high-order functions:
    List : Type → Type
    Pair : Type → Type → Type
    Vec : nat → Type → Type

Terms as Types

• Value parameters that have dependencies need a name.

\[
\begin{align*}
\text{vplus} & : (n : \text{nat}) \rightarrow \text{Vec } n \rightarrow \text{Vec } n \\
\text{vmult} & : (n : \text{nat}) \rightarrow \text{real} \rightarrow \text{Vec } n \rightarrow \text{Vec } n \\
\text{dot} & : (n : \text{nat}) \rightarrow \text{Vec } n \rightarrow \text{Vec } n \rightarrow \text{real}
\end{align*}
\]

Terms as Types

• Value parameters that have dependencies need a name.
  • Adopt the \( \forall \) binder.

\[
\begin{align*}
\text{vplus} & : \forall n : \text{nat}. \text{Vec } n \rightarrow \text{Vec } n \\
\text{vmult} & : \forall n : \text{nat}. \text{real} \rightarrow \text{Vec } n \rightarrow \text{Vec } n \\
\text{dot} & : \forall n : \text{nat}. \text{Vec } n \rightarrow \text{Vec } n \rightarrow \text{real}
\end{align*}
\]

Terms as Types

• When are two types deemed “compatible”?
  • Simple answer: when they are \( \beta \)-equivalent.*
  • Type checking becomes undecidable.

* Some additional reductions may be needed (for fix, match).
Dependent match

- When types are dependent, it’s common to have branches of match with (apparently) different types.

```
repeat \( \lambda h \) (e : nat) (n : nat).
match n with
  0 => VMil
  S k => VCons e (h e k)
end

: Vec 0
: Vec (3 k)
```

Dependent match

- When types are dependent, it’s common to have branches of match with (apparently) different types.

```
def getp \( \lambda A B \) (p : Pair A B) (i : nat).
match p with
  pair a b => match n with
    0 => a
    _ => b
  end
  return ifz n A B
end

def pred \( \lambda n \) : nat.
match n with
  0 => ?
  S k => k
end

: ??
: nat

pred 0 = 7 : ??
pred 3 = 2 : nat
```

Dependent match

- When types are dependent, it’s common to have branches of match with (apparently) different types.

```
def getp \( \lambda A B \) (p : Pair A B) (i : nat).
match p with
  pair a b => match n with
    0 => a
    _ => b
  end
  return ifz n A B
end

def pred \( \lambda n \) : nat.
match n with
  0 => ?
  S k => k
end

: ??
: nat

pred 0 = ? : ??
pred 3 = 2 : nat
```
Something Truly Amazing

Recursion Principle

For next time