Programming with Dependent Types

Gregory Malecha
gmalecha@gmail.com, eng.ucsd.edu

Coq Tutorial, ITP’15

August 27, 2015
What are Dependent Types?
What are Dependent Types?
What are Dependent Types?

Polymorphism

λ

Type Functions

λω

Types

Terms

Motivation | Dependent Types 2 / 8
What are Dependent Types?

Motivation | Dependent Types
What are Dependent Types?

Motivation | Dependent Types
What are Dependent Types?

Like Coq

Polymorphism

Type Functions

Motivation | Dependent Types
A Few Examples

- **Equality**
  
  \[ \text{eq} : \forall T : \text{Type}, T \to T \to \text{Prop} \]
A Few Examples

- Equality
  \[ \text{eq} : \forall T : \text{Type}, T \rightarrow T \rightarrow \text{Prop} \]

- Dependent pairs
  \[ \{ x : T \& f \ x \} \]
A Few Examples

- Equality
  \[ eq : \forall T : \text{Type}, T \to T \to \text{Prop} \]

- Dependent pairs
  \[ \{ x : T \& f x \} \]

- Vectors (length-indexed lists)
  \[ \text{vector} : \forall T : \text{Type}, \text{nat} \to \text{Type} \]
A Few Examples

- **Equality**
  \[ \text{eq} : \forall T : \text{Type}, T \to T \to \text{Prop} \]

- **Dependent pairs**
  \[ \{ x : T \& f x \} \]

- **Vectors (length-indexed lists)**
  \[ \text{vector} : \forall T : \text{Type}, \text{nat} \to \text{Type} \]

- **Equality decision procedures**
  \[ \forall n m : \text{nat}, \{ n = m \} + \{ n \neq m \} \]
Defining Dependent Types

**Inductively**
- Use inductive families

**Functionally**
- Compute the type from data
  e.g. `tuple nat 3 = nat * nat * nat`
Dependent Pattern Matching: “in” clause

```haskell
match v : vector l

with
| Vnil ⇒ _
| Vcons n x xs ⇒ _
end
```
Dependent Pattern Matching: “in” clause

```plaintext
match v : vector l
  in vector l'
  return f l'
with
  | Vnil ⇒ _
  | Vcons n x xs ⇒ _
end
```
Dependent Pattern Matching: “in” clause

```
match v : vector l
  in vector l'
  return f l'

with
| Vnil ⇒ _
| Vcons n x xs ⇒ _
end
```

Outer Type: f l

Inner Type: f 0

Inner Type: f (S n)

Motivation
Dependent Pattern Matching: “in” clause

```
match v : vector l
    in vector l'
    return f l'
  with
    | Vnil ⇒ _
    | Vcons n x xs ⇒ _
end
```

Outer Type: f l

Inner Type: f 0
Dependent Pattern Matching: “in” clause

```plaintext
match v : vector l
  in vector l'
  return f l'
with
  | Vnil ⇒ _
  | Vcons n x xs ⇒ _
end
```

Outer Type: f l

Inner Type: f 0

Inner Type: f (S n)
Dependent Pattern Matching: “in” clause

\[
\text{match } \text{pf}: a = b \\
\quad \text{in } _= X \\
\quad \text{return } f X \\
\text{with} \\
\quad \text{eq_refl } \Rightarrow _= \_ \\
\text{end}
\]
Dependent Pattern Matching: “in” clause

\[
\text{match } pf: a = b \\
\quad \text{in } _\_ = X \\
\quad \text{return } f X \\
\text{with} \\
\quad | \text{eq_refl } \Rightarrow _\_ \\
\text{end}
\]

Outer Type: \( f b \)

Inner Type: \( f a \)
Dependent Pattern Matching: “in” clause

\[
\text{match } pf : a = b \\
\quad \text{in } _\_ = X \\
\quad \text{return } f X \\
\text{with} \\
\quad | \text{eq_refl } \Rightarrow _\_ \\
\text{end}
\]

Outer Type: $f b$

Inner Type: $f a$

Motivation | Dependent Types
Dependent Pattern Matching: “as” clause

```ocaml
match v : vector l
  as v'
  in vector l'
  return f l' v'
with
  | Vnil  ⇒ _
  | Vcons n x xs ⇒ _
end
```
Dependent Pattern Matching: “as” clause

```plaintext
match v : vector l as v' : vector l'
in vector l'
return f l' v'
with
| Vnil ⇒ _
| Vcons n x xs ⇒ _
end
```
Dependent Pattern Matching: “as” clause

```
match v : vector l 
  as v' : vector l' 
  in vector l' 
  return f l' v' 
with 
| Vnil ⇒ _ 
| Vcons n x xs ⇒ _ 
end
```

Outer Type: f l v

Inner Type: f 0 Vnil

Inner Type: f (S n) (Vcons n x xs)
Dependent Pattern Matching: “as” clause

```latex
match v : vector l
  as v' : vector l'
  in vector l'
  return f l' v'
with
  | Vnil ⇒ _
  | Vcons n x xs ⇒ _
end
```

Outer Type: f l v

Inner Type: f 0 Vnil
Dependent Pattern Matching: “as” clause

match v: vector l
as v': vector l'
in vector l'
return f l' v'
with
| Vnil ⇒ _
| Vcons n x xs ⇒ _
end

Motivation | Dependent Types
Defining Dependent Types

**Inductively**
- Use inductive families

**Functionally**
- Compute the type from data
e.g. `tuple nat 3 = nat * nat * nat`
Defining Dependent Types

Inductively

- Use inductive families
- ❌ Least fixed-points

Functionally

- Compute the type from data
  e.g. `tuple nat 3 = nat * nat * nat`
- ❌ Pattern match on the index
Defining Dependent Types

Inductively
- Use inductive families
- Least fixed-points
- Often irrelevant indices

Functionally
- Compute the type from data e.g. \( \text{tuple} \ \text{nat} \ 3 = \text{nat} * \text{nat} * \text{nat} \)
- Pattern match on the index
- Avoid limitations such as positivity