Programming with Dependent Types

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

Coq Tutorial, ITP’15

August 27, 2015

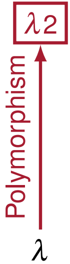
What are Dependent Types?

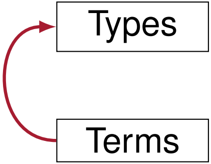
Types

Terms

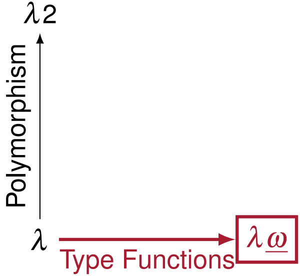
$\boxed{\lambda}$

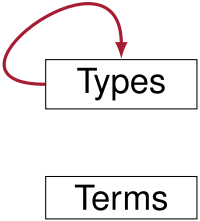
What are Dependent Types?





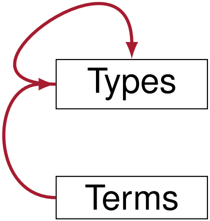
What are Dependent Types?



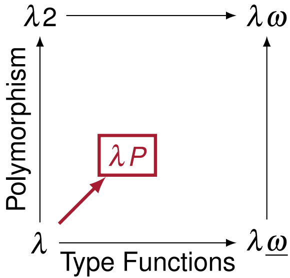


What are Dependent Types?

$$\begin{tikzcd}
\lambda 2 \arrow[r] & \lambda\omega \\
\lambda \arrow[u, "\text{Polymorphism}"] \arrow[r, "\text{Type Functions}"'] & \lambda\underline{\omega} \arrow[u]
\end{tikzcd}$$



What are Dependent Types?


$\lambda 2$

$\lambda\omega$

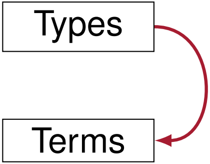
Polymorphism

λP

λ

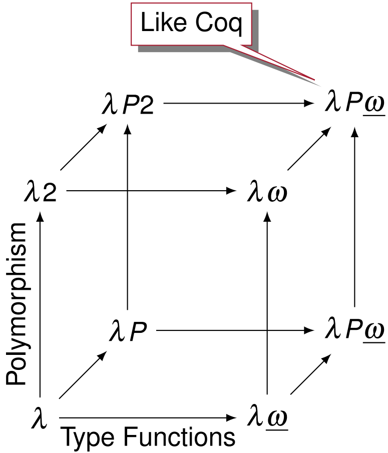
Type Functions

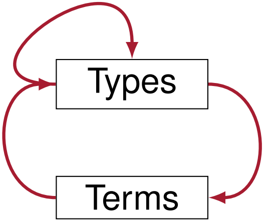
$\lambda\underline{\omega}$


Types

Terms

What are Dependent Types?





A Few Examples

- Equality

  $\mathtt{eq} : \forall\, \mathtt{T} : \mathtt{Type}, \mathtt{T} \rightarrow \mathtt{T} \rightarrow \mathtt{Prop}$

A Few Examples

- Equality

  `eq : ∀ T : Type, T → T → Prop`

- Dependent pairs

  `{ x : T & f x }`

A Few Examples

- Equality
  `eq : ∀ T : Type, T → T → Prop`
- Dependent pairs
  `{ x : T & f x }`
- Vectors (length-indexed lists)
  `vector : ∀ T : Type, nat → Type`

A Few Examples

- Equality

  $\texttt{eq} : \forall\, \texttt{T} : \texttt{Type}, \texttt{T} \to \texttt{T} \to \texttt{Prop}$

- Dependent pairs

  $\{\ \texttt{x} : \texttt{T}\ \&\ \texttt{f x}\ \}$

- Vectors (length-indexed lists)

  $\texttt{vector} : \forall\, \texttt{T} : \texttt{Type}, \texttt{nat} \to \texttt{Type}$

- Equality decision procedures

  $\forall\, \texttt{n m} : \texttt{nat}, \{\texttt{n} = \texttt{m}\} + \{\texttt{n} \neq \texttt{m}\}$

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

```
match v : vector 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
```

Dependent Pattern Matching: “in” clause

Outer Type: f l

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

Dependent Pattern Matching: “in” clause

Outer Type: f l

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

Inner Type: f 0

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)

Dependent Pattern Matching: “in” clause

```
match pf : a = b
   in _ = X
   return f X
with
| eq_refl ⇒ _
end
```

Dependent Pattern Matching: “in” clause

Outer Type: f b

```
match pf : a = b
   in _ = X
   return f X
with
| eq_refl ⇒ _
end
```

Dependent Pattern Matching: “in” clause

Outer Type: f b

```
match pf : a = b
   in _ = X
   return f X
with
| eq_refl ⇒ _
end
```

Inner Type: f a

Dependent Pattern Matching: “as” clause

```
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

```
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

Outer Type: f l v

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

: vector l’

Dependent Pattern Matching: “as” clause

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

Outer Type: f l v

: vector l’

Inner Type: f 0 Vnil

Dependent Pattern Matching: “as” clause

Outer Type: f l v

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

: vector l’

Inner Type: f 0 Vnil

Inner Type: f (S n) (Vcons n x xs)

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. `tuple nat` 3 = `nat` $*$ `nat` $*$ `nat`

✗ Pattern match on the index

✓ Avoid limitations such as positivity