Dependent Type Theory

Cameron Wong

98-317 Hype For Types

2020-04-16

- Types depend on values
  - `('a, n) vec` as the type of length-n lists
  - `n fin` as the type of naturals less than n

- Lift all values up to the type level
- Instead of complicated encodings like using succ and fin as type-level functions, just refer to values in types
- `nth:  ('a,n) vec -> {x:nat | x < n} -> 'a`

Can also bind arguments, eg

- `repeat: {n:int} -> {x:'a} -> ('a,n) vec`

Advantages

- Very easy to understand
- Requires no fancy tricks like `'a fin`

Driving question:
What does it mean for a type to depend on a value?

Driving question:
What <u>is</u> the type { `x:t | p(x)` }?

What is a refined type?

Types = Sets?

- $\texttt{nat} = \mathbb{N}$
- $\texttt{int} = \mathbb{Z}$
- $\tau_1 \to \tau_2 =$ (set-theoretic function)
- $\tau\ \texttt{list} = \mathbb{N} \to \overline{\tau}$

- Refinements become very simple – just use set comprehension!
  - $\{\texttt{x:t | p(x)}\} = \{x \in T \mid p(x)\}$

Advantages

- Very intuitive
- Can apply existing set theory research to type theory

Disadvantages

- Well...

Types as sets

```
datatype t = T of t -> bool
```

- Let S be the set representing the type `t`
- Certainly, $|S| = |S \to \texttt{bool}|$

Cantor's Theorem
For any set A, $|A| < |\mathcal{P}(A)|$.

Types as sets

- $S \to \texttt{bool}$ is equivalent to $\mathcal{P}(S)$
- Uh-oh...

Disadvantages

- It's unsound!

Types as sets

Recall: Curry-Howard Isomorphism

Curry-Howard Isomorphism

Types are propositions, programs are proofs

Review

Algebraic types (+ functions) correspond to propositional logic (or zeroth-order logic):

- $P \wedge Q$ corresponds to $A \times B$
- $P \vee Q$ corresponds to $A + B$
- $P \Rightarrow Q$ corresponds to $A \rightarrow B$

What about first-order logic?

- $\exists(x : \tau).p(x)$
- $\forall(x : \tau).p(x)$

For any $x : \tau$, $p(x)$ is a proposition.

For any $x : \tau$, $p(x)$ is a ~~proposition~~ type.

p is a function $\tau \rightarrow \texttt{type}$

How to prove $\exists(x : \tau).p(x)$?

Need:

- Some value $v : \tau$
- A proof of the proposition $p(v)$

Need:

- A value $v : \tau$
- A ~~proof~~ program of ~~the proposition~~ type $p(v)$

Need:

- A ~~value~~ expression $v : \tau$
- A ~~proof~~ ~~program~~ expression of ~~the proposition~~ type $p(v)$

A pair of expressions is a tuple!

Dependent tuple: $\Sigma(x : \tau).p(x)$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : p(e_1)}{\Gamma \vdash \langle e_1, e_2 \rangle : \Sigma(x : \tau).p(x)}$$

$$\frac{\Gamma \vdash e : \Sigma(x : \tau).p(x)}{\Gamma \vdash \pi_1 e : \tau}$$

$$\frac{\Gamma \vdash e : \Sigma(x : \tau).p(x)}{\Gamma \vdash \pi_2 e : p(\pi_1 e)}$$

Observation:
If $p(x) = \tau_2$ is a constant function, then $\Sigma(x : \tau_1).p(x)$ is the same as $\tau_1 \times \tau_2$

Observation:

$\tau_1 \times \tau_2$ is "τ_2 added τ_1 times"

- $\Sigma(x : \tau).p(x)$ corresponds to $\exists(x : \tau).p(x)$
- corresponds to $\forall(x : \tau).p(x)$

What is a proof of $\forall(x : \tau).p(x)$?

Given a value $v : \tau$, produce a proof of the proposition $p(v)$

Given a value $v : \tau$, produce a ~~proof~~ expression of ~~the proposition~~ type $p(v)$

This is a <u>function</u> of type $\tau \rightarrow p(v)$

Dependent function: $\Pi(x : \tau).p(x)$

$$\frac{\Gamma, x : \tau \vdash e : p(x)}{\Gamma \vdash \lambda(x : \tau).e : \Pi(x : \tau).p(x)}$$

$$\frac{\Gamma \vdash e_1 : \Pi(x : \tau).p(x) \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : p(e_1)}$$

Observation:
If $p(x) = \tau_2$, then $\Pi(x : \tau_1).p(x)$ is equivalent to $\tau_1 \rightarrow \tau_2$

- $\Sigma(x : \tau).p(x)$ corresponds to $\exists(x : \tau).p(x)$
- $\Pi(x : \tau).p(x)$ corresponds to $\forall(x : \tau).p(x)$

Back to refinements

What is $\{\texttt{x:t | p(x)}\}$?

- `{x:t | p(x)}` is $\Sigma(x : t).p(x)$
- `{x:t} -> p(x)` is $\Pi(x : t).p(x)$

Note that regular functions can be subsumed by Π-types!
$\texttt{int -> int} \rightsquigarrow \Pi(\_ : \texttt{int}).(\lambda\_ : \texttt{int})$

Next Question:
How to prove the proposition $p(x)$?

Next Question:
How to ~~prove the proposition~~ write a program of type $p(x)$?

Next Question:

How to ~~prove the proposition~~ write a program of type $3 < 5$?

What is the definition of $<_{\texttt{nat}}$?

```
fun 0 < s(_) = true
  | _ < 0 = false
  | s(n) < s(m) = n < m
```

What's in a proof?

$$3 < 5 \rightsquigarrow 2 < 4 \rightsquigarrow 1 < 3 \rightsquigarrow 0 < 2 \rightsquigarrow \texttt{true}$$

$$\underbrace{3 < 5}_{\texttt{bool}} \rightsquigarrow \underbrace{2 < 4}_{\texttt{bool}} \rightsquigarrow \underbrace{1 < 3}_{\texttt{bool}} \rightsquigarrow \underbrace{0 < 2}_{\texttt{bool}} \rightsquigarrow \underbrace{\texttt{true}}_{\texttt{bool}}$$

$$\underbrace{3 < 5}_{\texttt{type}} \rightsquigarrow \underbrace{2 < 4}_{\texttt{type}} \rightsquigarrow \underbrace{1 < 3}_{\texttt{type}} \rightsquigarrow \underbrace{0 < 2}_{\texttt{type}} \rightsquigarrow \underbrace{\top}_{\texttt{type}}$$

Curry-Howard

- The type `unit` (or $\mathbf{1}$) corresponds to the proposition $\top$ (true)
- The type `void` (or $\mathbf{0}$) corresponds to the proposition $\perp$ (false)

The type $3 < 5$ is equivalent to `unit`!

$$\texttt{Refl} : (3 < 5)$$

`Refl` = "true by definition"

```
(3, Refl) :  {x:int | x < 5}
```

For usability:

```
3 :  {x:int | x < 5}
```

```
type _ vec = [] : 'a vec
           | (::) : 'a * 'a vec -> 'a vec

(* repeat : int -> 'a -> 'a vec *)
fun repeat = _
```

```
type _ vec = [] : ('a, 0) vec
           | (::) : 'a * ('a, n) vec -> ('a, n+1) vec

(* repeat : {n:nat} -> {x:'a} -> {l:('a,n) vec} *)
fun repeat = _
```

Example

```
type _ vec = [] : ('a, 0) vec
           | (::) : 'a * ('a, n) vec -> ('a, n+1) vec

(* repeat : {n:nat} -> {x:'a} ->
   {l:('a,n) vec | forall (m < n) . nth m l = x}
 *)
fun repeat = _
```

$$repeat : \Pi(n : \texttt{nat}).\Pi(x : \alpha).\Sigma(l : (\alpha, n)\ \texttt{vec}).$$
$$\Pi(m : \Sigma(m' : \texttt{nat}).(m' < n)).(\texttt{nth}\ \ l\ (\pi_1 m) = x)$$

```
fun repeat 0 x = ([], fn (m,p) => _)
  | repeat n x = _
```

Example

p has type $m < 0 \rightsquigarrow \perp$, so $p : \mathbf{0}$

```
fun repeat 0 x = ([], fn (m,p) => abort p)
  | repeat n x = _
```

```
fun repeat 0 x = ([], fn (m,p) => abort p)
  | repeat n x =
      (* xs : ('a, n-1) vec
       * p : {m:nat | m < n} -> nth xs m = x
       *)
      let val (xs, p) = repeat (n-1) x
       in _
      end
```

```
fun repeat 0 x = ([], fn (m,p) => abort p)
  | repeat n x =
      (* xs : ('a, n-1) vec
       * p : {m:nat | m < n} -> nth xs m = x
       *)
      let val (xs, p) = repeat (n-1) x
       (* _: {m:nat | m < n} -> (nth (x::xs) m = x) *)
       in (x::xs, _)
      end
```

Example

```
fun repeat 0 x = ([], fn (m,p) => abort p)
  | repeat n x =
      (* xs : ('a, n-1) vec
       * p : {m:nat | m < n-1} -> nth xs m = x
       *)
      let val (xs, p) = repeat (n-1) x
      (* p' : m < n *)
       in (x::xs, fn (0,p') => Refl (* defn of nth *)
                   | (m,p') => _)
      end
```

Example

```
fun repeat 0 x = ([], fn (m,p) => abort p)
  | repeat n x =
      (* xs : ('a, n-1) vec
       * p : {m:nat | m < n-1} -> nth xs m = x
       *)
      let val (xs, p) = repeat (n-1) x
      (* By definition of <, [p : m < n] is also a
       * proof of m-1 < n-1
       *)
       in (x::xs, fn (0,p') => Refl
                               (* (m-1,p') is
                                * Sigma(x:nat).(x<n-1)
                                *)
                   | (m,p') => p(m-1, p'))
      end
```