Dependent Types

Hype for Types

November 9, 2021

Safe Printing

Detypify

Consider these well typed expressions:

```
sprintf "nice"
sprintf "%d" 5
sprintf "%s,%d" "wow" 32
```

What is the type of `sprintf`?

Detypify

Consider these well typed expressions:

```
sprintf "nice"
sprintf "%d" 5
sprintf "%s,%d" "wow" 32
```

What is the type of `sprintf`? Well... it depends.

Types have types too

The type of `sprintf` *depends* on the value of the argument.
In order to compute the type of `sprintf`, we'll need to write a function that takes a string (List char), and returns a *type*!

Types have types too

The type of `sprintf` *depends* on the value of the argument.
In order to compute the type of `sprintf`, we'll need to write a function that takes a string (List char), and returns a *type*!

```
-- sprintf s : formatType s
```

Types have types too

The type of `sprintf` *depends* on the value of the argument.
In order to compute the type of `sprintf`, we'll need to write a function that takes a string (List char), and returns a *type*!

```
-- sprintf s : formatType s

formatType : List char → Type
formatType [] = List char
formatType ("%" :: "d" :: cs) = int → formatType cs
formatType ("%" :: "s" :: cs) = string → formatType cs
formatType (_ :: cs) = formatType cs
```

Quantification

Ok, we can express the type of `sprintf s` for some argument `s`, but what's the type of `sprintf`?

Quantification

Ok, we can express the type of `sprintf s` for some argument `s`, but what's the type of `sprintf`?
Recall that when we wanted to express a type like "`A -> A` for all `A`", we introduced universal quantification over *types*: $\forall$ `A.A -> A`.

Quantification

Ok, we can express the type of `sprintf s` for some argument `s`, but what's the type of `sprintf`?
Recall that when we wanted to express a type like "`A -> A` for all `A`", we introduced universal quantification over *types*: $\forall$ `A.A -> A`.
What if we had universal quantification over *values*?

Quantification

Ok, we can express the type of `sprintf s` for some argument `s`, but what's the type of `sprintf`?
Recall that when we wanted to express a type like "`A -> A` for all `A`", we introduced universal quantification over *types*: $\forall$ `A.A -> A`.
What if we had universal quantification over *values*?

```
sprintf : (s : List char) → formatType s
```

Curry-Howard Again

What kind of proposition does quantification over values correspond to?

Curry-Howard Again

What kind of proposition does quantification over values correspond to?

$$(x : \tau) \to A \;\equiv\; \forall x : \tau . A$$

This type can also be written like so:

1. $\forall (x : \tau) \to A$
2. $\forall x : t . A$
3. $\Pi_{x:\tau} A$

Question:

Do we need two kinds of arrow now?
One for dependent quantification and one normal?

Curry-Howard Again

What kind of proposition does quantification over values correspond to?

$$(x : \tau) \to A \;\equiv\; \forall x : \tau . A$$

This type can also be written like so:

1. $\forall (x : \tau) \to A$
2. $\forall x : t . A$
3. $\Pi_{x:\tau} A$

Question:

Do we need two kinds of arrow now?
One for dependent quantification and one normal?
Nope!
$A \to B \equiv (\_ : A) \to B$

Some Rules

$$\frac{\Gamma, x : \tau \vdash e : A \quad \Gamma, x : \tau \vdash A : \mathit{Type}}{\Gamma \vdash \lambda(x : \tau)e : (x : \tau) \to A} \qquad \frac{\Gamma \vdash e_1 : (x : \tau) \to A \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : [e_2/x]A}$$

Vectors Again

If we can write functions from values to types, can we define new types which depend on values?

Vectors Again

If we can write functions from values to types, can we define new types which depend on values?

```
data Vec : Type → Nat → Type where
  Nil :  (a : Type) → Vec a 0
  Cons : (a : Type) → (n : Nat) →
             a → Vec a n → Vec a (n+1)


n = 1 + 2


xs : Vec string n
xs =
    Cons string 2 "hype" (
        Cons string 1 (Int.toString (n+1)) (
            Cons string 0 "types" (Nil string)))
```

Vectors are actually usable now!

```
val append : (a : Type) → (n m : Nat) →
             Vec a n →
             Vec a m →
             Vec a (n + m)

val repeat : (a : Type) → (n : Nat) →
             a →
             Vec a n


val filter : (a : Type) → (n : Nat) →
             (a → bool) →
             Vec a n →
             Nat × Vec a ??
```

Duality

If we can quantify over the argument to a function, can we quantify over the left element of a tuple?

Duality

If we can quantify over the argument to a function, can we quantify over the left element of a tuple?
Yes!

$$(x : \tau) \times A \equiv \exists x : \tau . A$$

This type can also be written:

1. $\{x : \tau \mid A\}$
2. $\Sigma_{x:\tau} A$

As before, $A \times B \equiv (\_ : A) \times B$

```
val filter : (a : Type) → (n : Nat) →
             (a → bool) →
             Vec a n →
             (m : Nat) × Vec a m
```

More Rules

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : [e_1/x]A \quad \Gamma, x : \tau \vdash A : \mathit{Type}}{\Gamma \vdash (e_1, e_2) : (x : \tau) \times A}$$

$$\frac{\Gamma \vdash e : (x : \tau) \times A}{\Gamma \vdash \pi_1\ e : \tau} \qquad \frac{\Gamma \vdash e : (x : \tau) \times A}{\Gamma \vdash \pi_2\ e : [\pi_1\ e/x]A}$$

Ok, so what?

Contracts are actually pretty nice

A familiar frustration for 150 students and TAs:

```
(* REQUIRES : input sequence is sorted *)
val search : int → int seq → int option

> search 3 [5,4,3] ==> NONE
(* "search is broken!" *)
(* piazza post ensues *)
```

Contracts are actually pretty nice

A familiar frustration for 150 students and TAs:

```
(* REQUIRES : input sequence is sorted *)
val search : int → int seq → int option

> search 3 [5,4,3] ==> NONE
(* "search is broken!" *)
(* piazza post ensues *)
```

The 122 solution:

```
//@requires is_sorted(xs)
```

Nice, but only works at runtime.

Contracts are actually pretty nice

A familiar frustration for 150 students and TAs:

```
(* REQUIRES : input sequence is sorted *)
val search : int → int seq → int option

> search 3 [5,4,3] ==> NONE
(* "search is broken!" *)
(* piazza post ensues *)
```

The 122 solution:

```
//@requires is_sorted(xs)
```

Nice, but only works at runtime. What if passing `search` a non-sorted list was *type error*?

A simpler example

```
(* REQUIRES : second argument is greater than zero *)
val div : Nat → Nat → Nat
```

Comment contracts are not great, solutions?

A simpler example

```
(* REQUIRES : second argument is greater than zero *)
val div : Nat → Nat → Nat
```

Comment contracts are not great, solutions?

```
val div : Nat → Nat → Nat option
```

Incurs runtime cost to check for zero, and you still have to fail if it happens.

A simpler example

```
(* REQUIRES : second argument is greater than zero *)
val div : Nat → Nat → Nat
```

Comment contracts are not great, solutions?

```
val div : Nat → Nat → Nat option
```

Incurs runtime cost to check for zero, and you still have to fail if it happens.

```
val div : Nat → (n : Nat) × (1 ≤ n) →Nat
```

Dividing by zero is impossible! And we incur no runtime cost to prevent it.

A simpler example

```
(* REQUIRES : second argument is greater than zero *)
val div : Nat → Nat → Nat
```

Comment contracts are not great, solutions?

```
val div : Nat → Nat → Nat option
```

Incurs runtime cost to check for zero, and you still have to fail if it happens.

```
val div : Nat → (n : Nat) × (1 ≤ n) →Nat
```

Dividing by zero is impossible! And we incur no runtime cost to prevent it.
What does a value of type $(n : Nat) \times (1 \leq n)$ look like?

$$(3, \text{conceptsHW1.pdf}) : (n : Nat) \times (1 \leq n)$$

Question:

What goes in the PDF?

15-151 Refresher

What constitutes a proof of $n \leq m$?

15-151 Refresher

What constitutes a proof of $n \leq m$?
We just have to define what $(\leq)$ means!

1. $\forall n.\ 0 \leq n$
2. $\forall m\ n.\ n \leq m \Rightarrow n + 1 \leq m + 1$

This looks familiar!

15-151 Refresher

What constitutes a proof of $n \leq m$?
We just have to define what $(\leq)$ means!

1. $\forall n.\ 0 \leq n$
2. $\forall m\ n.\ n \leq m \Rightarrow n + 1 \leq m + 1$

This looks familiar!

```
data _≤_ : Nat →Nat → Type where
  LeqZ : (n : Nat) → 0 ≤ n
  LeqS : (n : Nat) → (m : Nat) →
             n ≤ m →(n + 1) ≤ (m + 1)
```

15-151 Refresher

What constitutes a proof of $n \leq m$?
We just have to define what $(\leq)$ means!

1. $\forall n.\ 0 \leq n$
2. $\forall m\ n.\ n \leq m \Rightarrow n + 1 \leq m + 1$

This looks familiar!

```
data _≤_ : Nat →Nat → Type where
  LeqZ : (n : Nat) → 0 ≤ n
  LeqS : (n : Nat) → (m : Nat) →
               n ≤ m →(n + 1) ≤ (m + 1)
```

$$LeqZ\ 3 : 0 \leq 3$$

15-151 Refresher

What constitutes a proof of $n \leq m$?
We just have to define what $(\leq)$ means!

1. $\forall n.\ 0 \leq n$
2. $\forall m\ n.\ n \leq m \Rightarrow n + 1 \leq m + 1$

This looks familiar!

```
data _≤_ : Nat →Nat → Type where
  LeqZ : (n : Nat) → 0 ≤ n
  LeqS : (n : Nat) → (m : Nat) →
           n ≤ m →(n + 1) ≤ (m + 1)
```

$$LeqZ\ 3 : 0 \leq 3$$
$$LeqZ\ 43 : 0 \leq 43$$

15-151 Refresher

What constitutes a proof of $n \leq m$?
We just have to define what $(\leq)$ means!

1. $\forall n.\ 0 \leq n$
2. $\forall m\ n.\ n \leq m \Rightarrow n + 1 \leq m + 1$

This looks familiar!

```
data _≤_ : Nat →Nat → Type where
  LeqZ : (n : Nat) → 0 ≤ n
  LeqS : (n : Nat) → (m : Nat) →
             n ≤ m →(n + 1) ≤ (m + 1)
```

$$LeqZ\ 3 : 0 \leq 3$$
$$LeqZ\ 43 : 0 \leq 43$$
$$LeqS\ 0\ 2\ (LeqZ\ 2) : 1 \leq 3$$

15-151 Refresher

What constitutes a proof of $n \leq m$?
We just have to define what $(\leq)$ means!

1. $\forall n.\ 0 \leq n$
2. $\forall m\ n.\ n \leq m \Rightarrow n + 1 \leq m + 1$

This looks familiar!

```
data _≤_ : Nat →Nat → Type where
  LeqZ : (n : Nat) → 0 ≤ n
  LeqS : (n : Nat) → (m : Nat) →
           n ≤ m →(n + 1) ≤ (m + 1)
```

$$\begin{gathered} LeqZ\ 3 : 0 \leq 3 \\ LeqZ\ 43 : 0 \leq 43 \\ LeqS\ 0\ 2\ (LeqZ\ 2) : 1 \leq 3 \\ (3, LeqS\ 0\ 2\ (LeqZ\ 2)) : (n : Nat) \times (1 \leq n) \end{gathered}$$

Some Sort of Contract

```
data NatList : Type where
  Nil : NatList
  Cons : Nat → NatList → NatList

data Sorted : NatList → Type where
  NilSorted : Sorted Nil
  SingSorted : (n : Nat) → Sorted (Cons n Nil)
  ConsSorted : (n m : Nat) → (xs : NatList) →
               n ≤ m →
               Sorted (Cons m xs) →
               Sorted (Cons n (Cons m xs))

val search : Nat →
             (xs : NatList) →
             Sorted xs →
             Nat option
```

A Type for Term Equality

If we can express a relation like less than or equal, how about equality?

A Type for Term Equality

If we can express a relation like less than or equal, how about equality?

```
data Eq : (a : Type) → a → a → Type where
  Refl : (a : Type) → (x : a) → Eq a x x

symm : (a : Type) (x y : a) → Eq a x y → Eq a y x
symm a x y (Refl A q) = Refl A q

trans : (a : Type) (x y z : a) → Eq a x y → Eq a y z →
Eq a x z
trans a x y z (Refl A q) (Refl _ _) = Refl A q


plus_comm : (n m : Nat) → Eq Nat (n + m) (m + n)
inf_primes : (n : nat) →
             (m : Nat) × ((m > n) × (Prime m))
```