Dependent Types

Hype for Types

November 18, 2024

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 *)

fun formatType (s : char list) : type =
  case s of
    []                    => char list
  | ('%' :: 'd' :: cs) => (int -> formatType cs)
  | ('%' :: 's' :: cs) => (string -> formatType cs)
  | (_ :: cs)           => formatType cs
```

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 *)

fun formatType (s : char list) : type =
  case s of
    []                 => char list
  | ('%' :: 'd' :: cs) => (int -> formatType cs)
  | ('%' :: 's' :: cs) => (string -> formatType cs)
  | (_ :: cs)          => formatType cs

(* formatType "%d and %s"  =  int -> string -> char list *)
(* sprintf "%d and %s" : int -> string -> char list *)
```

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*: ∀ `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*: ∀ `A.A -> A`.

What if we had universal quantification over *values*?

```
sprintf : (s : char list) -> 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?

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

Curry-Howard Again

What kind of proposition does quantification over values correspond to?

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

This type is sometimes also written as:

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

Curry-Howard Again

What kind of proposition does quantification over values correspond to?

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

This type is sometimes also written as:

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

Question

Seems like we now have two arrow types:

1. Normal: $A \to B$
2. Dependent: $(x : A) \to B$

Do we need both?

Question

Seems like we now have two arrow types:

1. Normal: $A \to B$
2. Dependent: $(x : A) \to B$

Do we need both?

Question

Seems like we now have two arrow types:

1. Normal: $A \to B$
2. Dependent: $(x : A) \to B$

Do we need both? Nope!

$$A \to B \equiv (\_ : A) \to B$$

Some Rules

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

Note on Notation

In SML we write type contructors on the *right*:

```
val cool : int list = [1,2,3,4]
```

[1]Readers may note the parallels to another CS course mantra.

Note on Notation

In SML we write type contructors on the *right*:

```
val cool : int list = [1,2,3,4]
```

But now we have functions in our types, and we apply functions on the left! So let's just write everything on the left. While we are at it, lets make values of type `Type` capital, and their values lowercase:

```
val cool : List Int = [1,2,3,4]
val a : A = (* omitted *)
```

[1]Readers may note the parallels to another CS course mantra.

Note on Notation

In SML we write type contructors on the *right*:

```
val cool : int list = [1,2,3,4]
```

But now we have functions in our types, and we apply functions on the left! So let's just write everything on the left. While we are at it, lets make values of type `Type` capital, and their values lowercase:

```
val cool : List Int = [1,2,3,4]
val a : A = (* omitted *)
```

Question

What is the type of `List`?

[1] Readers may note the parallels to another CS course mantra.

Note on Notation

In SML we write type contructors on the *right*:

```
val cool : int list = [1,2,3,4]
```

But now we have functions in our types, and we apply functions on the left! So let's just write everything on the left. While we are at it, lets make values of type `Type` capital, and their values lowercase:

```
val cool : List Int = [1,2,3,4]
val a : A = (* omitted *)
```

Question

What is the type of `List`?

```
List : Type -> Type
```

List is a function over types!

Types are values[1]

[1] Readers may note the parallels to another CS course mantra.

Vectors Again

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

Vectors Again

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

```
inductive Vec : Type → Nat → Type
| nil  : (A : Type) → Vec A 0
| cons : (A : Type) → (n : Nat) →
           A → Vec A n → Vec A (n+1)

def xs : Vec String 3 :=
  cons String 2 "hype" (
    cons String 1 (toString 4) (
      cons String 0 "types" (nil String)
    )
  )
```

Vectors Again

```
inductive Vec : Type → Nat → Type
| nil  : (A : Type) → Vec A 0
| cons : (A : Type) → (n : Nat) →
            A → Vec A n → Vec A (n+1)

def two := 1 + 0 + 1

def xs : Vec String (6 / two) :=
  cons String two "hype" (
    cons String 1 (toString 4) (
      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 ->
             Vec a ?? (* What should go here? *)
```

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 ->
             Vec a ?? (* What should go here? *)
```

Ponder

How do we describe the return value of `filter`?

Existential Crisis

For filter, we need to return the vector's length, *in addition* to the vector itself:

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

Existential Crisis

For filter, we need to return the vector's length, *in addition* to the vector itself:

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

We want to refer to the left value of a tuple, in the TYPE on the right.

Intuition: existential quantification!

There exists some `n : Nat`, such that we return `Vec a n`.

(We're constructivists, so exists means I actually give you the value)

Duality

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

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?

Specifications are actually pretty nice

Discussion

Do you actually read function contracts/specifications in 122/150?

Specifications are actually pretty nice

Discussion

Do you actually read function contracts/specifications in 122/150?

```
(* REQUIRES : input list is sorted *)
val search : int -> int list -> int option

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

Compile-time Contracts

The 122 solution:

```
int search (int target, int[] arr)
//@requires is_sorted(arr)
{
    ...
}
```

Nice, but only works at runtime.

Compile-time Contracts

The 122 solution:

```
int search (int target, int[] arr)
//@requires is_sorted(arr)
{
    ...
}
```

Nice, but only works at runtime.

What if passing `search` a non-sorted list was a *type error*?

A simpler example

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

Comment contracts aren't good enough. I don't read comments!

A simpler example

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

Comment contracts aren't good enough. I don't read comments!

```
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 aren't good enough. I don't read comments!

```
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 aren't good enough. I don't read comments!

```
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!

```
inductive Le : Nat → Nat → Prop
| zero {n : Nat} : Le 0 n
| step {n m : Nat} : Le n m → Le (Nat.succ n) (Nat.succ m)
```

conceptsHW1.pdf

```
inductive Le : Nat → Nat → Prop
| zero {n : Nat} : Le 0 n
| step {n m : Nat} : Le n m → Le (Nat.succ n) (Nat.succ m)

def ex1  : Le 0 0 := @Le.zero 0
def ex1' : Le 0 0 := Le.zero

def ex2 : Le 0 3 := Le.zero

def ex3 : Le 2 3 := Le.step (Le.step Le.zero)

def ex4 : (n : Nat) ×' (Le 1 n) :=
  3, (Le.step Le.zero)
```

Some Sort of Contract

```
inductive Sorted : List Nat → Prop
| nil_sorted    : Sorted []
| single_sorted : (n : Nat) → Sorted [n]
| cons_sorted   : (n m : Nat) →
                    (xs : List Nat) →
                    Le n m →
                    Sorted (m :: xs) →
                    Sorted (n :: m :: xs)

def search : Nat
           → (xs : List Nat)
           → Sorted xs
           → Option Nat := sorry
```