Propositions-as-Types & Dependent Types

15-150 M21

Lecture 0811
11 August 2021

0 Propositions-as-Types

Formal logic is the study of *propositions*, which are formal statements that can be either true or false.

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \to \psi \mid \varphi \vee \psi$$

In formal logic, we formalize the reasoning of mathematics by formally *proving* propositions.

We do so by combining *axioms* to form deductions. The axioms are "tautological" (obviously true) statements of logic, e.g.

- $\varphi \to \varphi$
- $\varphi \to \psi \to \varphi$
- $(\varphi \to \psi) \to (\psi \to \theta) \to \varphi \to \theta$
- If φ and $\varphi \to \psi$, then ψ

What does this look like?

Observation:

Tautologies of Formal Logic look like HOF types!

Propositions-as-Types

Idea: we'll associate *propositions* (statements that can be true or false) with *types*: a proposition P is the type of proofs that P is true.

- A proposition P is "true" if it is **inhabited**: there exists some $w : P$ *witnessing the truth of* P.
- An **uninhabited** type is a "false" proposition: there is no witness/proof of its truth

Purpose:

- Use the tools of type theory/functional programming to reason about formal logic
- Utilize logic inside of functional programming

- Can take the conjunction $P_1 \wedge P_2$ of two propositions to get another one: P_1 **and** P_2. A witness of $P_1 \wedge P_2$ consists of a witness to the truth of P_1 and a witness of the truth of P_2.

  $(w1, w2)$ witnesses the truth of $P_1 \wedge P_2$ iff w_1 witnesses the truth of P_1 and w_2 witnesses the truth of P_2

  So conjunction is represented by *product types*

- Can take the disjunction $P_1 \vee P_2$ of two propositions to get another one: P_1 **or** P_2. A witness of $P_1 \vee P_2$ consists of either a witness to the truth of P_1 or a witness of the truth of P_2.

  w witnesses the truth of $P_1 \vee P_2$ iff w witnesses the truth of P_1 or w witnesses the truth of P_2

  So conjunction is represented by *sum types*:

```
datatype ('a,'b) plus = inL of 'a | inR of 'b
```

- The proposition $P_1 \to P_2$ represents implication: P_1 **implies** P_2. A witness of $P_1 \to P_2$ is a way of obtaining a witness $w' : P_2$, given a witness $w : P_1$.

  f witnesses the truth of $P_1 \to P_2$ iff for all witnesses w of P_1, there's a witness $f(w)$ of P_2

  So implication is represented by *function types*

Proving propositions

- $P \to P$

```
fn p => p
```

- $P \to (Q \to P)$

```
fn p => fn q => q
```

- $P \land Q \to P$

```
fn (p,q) => p
```

Matching up formal logic with type theory

- The proposition $\neg P$ represents negation: $\neg P$ means "not P". A witness of $\neg P$ is a proof by contradiction of P: a witness that, if P were true, then absurdity would follow.

$$\neg\mathtt{T}\text{ is defined to be }\mathtt{T\ \ ->\ \ void}$$

where `void` is the type with no elements:

```
datatype void = Void of void  (* No base case
  *)
```

Proving propositions

- $(P \to Q) \to \neg Q \to \neg P$

```
fn g => fn nq => nq o g
```

- $P \to \neg\neg P$
  $P \to (P \to \bot) \to \bot$

```
fn p => fn np => np(p)
```

Idea: Type Families

```
T -> Type
```

In SML,

```
(op >=) : int * int -> bool
```

where `(m >= n)` is `true` if `m` is greater-than-or-equal-to `n`, and `false` otherwise.

What if instead we did

```
(op Geq) : int * int -> Type
```

where `(m Geq n)` is *inhabited* if `m` is greater-than-or-equal-to `n`, and *uninhabited* otherwise?

```
infix Geq
(* (op Geq) : int * int -> Type *)
fun m Geq 0 = unit
  | 0 Geq n = void
  | m Geq n =
      let
        datatype result =
          GeqSucc of (m-1) Geq (n-1)
      in
        result
      end
```

```
GeqSucc(GeqSucc(GeqSucc())) : 8 Geq 3
```

```
fun isNonEmpty [] = void
  | isNonEmpty (x::xs) = unit
```

```
fun hd (L : 'a list) (p : isNonEmpty L) =
    let val x::_ = L in x end
```

In order to call this, you would need to supply not just `L` but `p`. This would be impossible if `L=[]`, since there are no values `p : isNonEmpty []`.

1 Dependent Types

Notice something funny about the type of this function:

```
fun hd (L : 'a list) (p : isNonEmpty L) =
```

The type of the second argument *depends* on the value of the first. How do we make sense of this?

Given a type family `B : A -> Type`, the **dependent product** type, written

$$\texttt{(a : A) -> B(a)} \quad \text{or} \quad \prod_{\texttt{a:A}} \texttt{B(a)}$$

is the type of functions `f`, where, for each `a:A`, `f(a) : B(a)`.

- The proposition $\forall x P(x)$ is universal quantification. A witness of $\forall x P(x)$ is a way to take an arbitrary x and produce a witness of $P(x)$.

```
(x : T) -> P(x)
```

- $\forall x(P(x) \to Q(x)) \to \forall x P(x) \to \forall x Q(x)$

```
fn pPQ => fn pP => fn x => pPQ x (pP(x))
```

Existential Quantification?

Given a type family `B : A -> Type`, the **dependent sum** type, written

$$\texttt{(a : A, B(a))} \qquad \text{or} \qquad \sum_{\texttt{a:A}} \texttt{B(a)}$$

is the type of pairs `(a,b)`, where `a:A` and `b:B(a)`.

- The proposition $\exists x P(x)$ is existential quantification. A witness of $\exists x P(x)$ is some x and a witness of $P(x)$.

Proving propositions

- $\forall x(P(x) \to Q(x)) \to \exists x P(x) \to \exists x Q(x)$

```
fn pPQ => fn (a,pa) => (a,pPQ a pa)
```

```
fun mulNat (m : int, pm : m Geq 0)
           (n : int, pn : n Geq 0)
         : (z : int, pz : z Geq 0)
     = (m*n, ...)
```

```
fun fact (n:int) (pn : n Geq 0)
     : (res : int, pres : res Geq 0) =
  case n of
    0 => (1,())
  | _ => mulNat
         (n,pn)
         (fact (n-1) ((*some p : n-1 Geq 0 *)))
```

Thank you!