Lambda Calculus

Untyped Lambda Calculus

- Application (`$`) and Abstraction ($\lambda x \rightarrow ...$) only
- No type signatures - *every* term is a function
- Eta / Beta reductions as computation

```
foo = baz bar -- application
baz x = x -- abstraction
```

One could see an endless number of Eta reductions available.

Simply-Typed Lambda Calculus

- Now we have the classic `->` function arity type
  - `a -> b -> c` ~ `a -> (b -> c)`
- Unification and type checking via substitution is now a thing
  - We only have dummy terms for types - it only represents arity

```
-- Type checking now just does literal substitution
foo :: x -> a -> x
foo x a = x

bar :: x
bar = ()

baz :: ?
baz = ()

foo bar -- typechecks
foo baz -- ): `x` ~/~ `?`
```

Hindley-Milner Polymorphism

- Now we have type variables
  - We can enforce symmetry `a -> a` and head type checking `a -> b`
  - Identical type variables in a scope must *completely* unify
- Outer-most `forall`

```haskell
foo :: forall x. x -> x
foo x = x

bar :: forall b c. b -> c -> b
bar b c = b

foo foo :: forall x. x -> x -- checks
foo bar :: forall beans pie. beans -> pie -> beans -- checks
```

System-F (Polymorphic Lambda Calculus)

- Rank-N Types
  - nested variable quantification - can't unify nested terms globally

```haskell
foo :: forall x y (u :: [*]).
           (forall a. a -> a) -- ^ `f`
        -> x
        -> y
        -> HList [x:y:u]
foo f x y = HCons (f x) $ HCons (f y) HNil
```

System-FC (Unification and Coercion Constraints)

- Unification is now first class…?
  - Type level terms, called "Coercions", are proofs of the coercion…?
- Reflexive coercion relation `:=:`
  - `Int :: Int :=: Int`

```haskell
foo :: (a :=: Bool) =>
       a -- ^ The `x` term
    -> Int
foo = Λc :: (a :=: Bool). -- proof that `a` can be a `Bool`
      \(x :: a) ->
        if (x `cast` c) -- the successful result, used as a `Bool`
          then 0
          else 1
```