CSE 114A lecture 26

Agenda:

- Polymorphism!

Damas-Hindley-Milner type inference

is known for its support for polymorphism

$\backslash x \rightarrow x :: a \rightarrow a$

$$\frac{(x, T_1) : \Gamma \vdash e :: T_2}{\Gamma \vdash \backslash x \rightarrow e :: T_1 \rightarrow T_2}$$

$\backslash x \rightarrow \backslash y \rightarrow x :: a \rightarrow b \rightarrow a$

$P_1 \rightarrow P_2 \rightarrow P_1$

Think about this program:

```
let id = \x -> x in
  let y = id 5 in
    id (\z -> (z + y))
```

:: Int → Int

(`\z -> (z + y)` : Int → Int)

Quiz question 1: is this program well-typed? If so, what's its type?

```
id   5
↑    ↑
a→a  Int
Int→Int

id (\z -> z + y)
↑   ↑
a→a (Int→Int)
(Int→Int)→(Int→Int)
```

How do we deal with the fact that a polymorphic function might be applied to arguments of different types at different places in a program?

1) When we put a type into the type environment for a let-bound variable, we generalize that type.
(This happens in the ELet case of 'infer'.)

Forall a . a → a

implicit quantifier!

f should have type forall a. a → a

```
let f = \x -> x in
  let y = f 5 in
    f (\z -> z + y)
```

f gets instantiated with a fresh type variable which gets unified with Int → Int

f gets its own instantiation

2) When we use a let-bound variable from the type environment, we instantiate its type with a fresh type variable specific to that one use!
(This happens in the EVar case of 'infer'.)

(This is called "let-polymorphism")

`(\x -> x) 5`