Constraint Propagation and Typing

COM2001 ADVANCED PROGRAMMING TOPICS

Type declarations

- Recall that Haskell lets you declare many kinds of type, but they're all treated the same way

```
data RGB = Red | Green | Blue
data Block = Sides Float Float Float
data Encapulate a = Enc a
data Stack a = Empty | Push a (Stack a)
```

- We can also declare **constrained** types

Constrained types and functions

- An ordered tree can only be defined if its elements can be ordered
- We can constrain the **data** declaration

```
data (Ord a) => OTree a
    =  EmptyTree
    |  Node (OTree a) a (OTree a)
push :: a -> OTree a -> OTree a
```

- Or we can constrain the **function** declaration

```
data OTree a
    =  EmptyTree
    |  Node (OTree a) a (OTree a)
push :: (Ord a) => a -> OTree a -> OTree a
```

Beware: type synonyms

- You can't add constraints inside a **type** declaration


```
type (Num a)
  => NumberPair a = (a, a)
```

- Constrain the functions instead

```
type NumberPair a = (a, a)

addNP :: (Num a) => NumberPair a -> a
addNP (x, y) = x + y
```

Computing the type of an expression


```
elem :: Eq a => a -> [a] -> Bool
sort :: Ord b => [b] -> [b]
(.)  :: (d -> e) -> (c -> d)->(c -> e)
```

- What is the type of **elem . sort** ?
- We can find out in GHCi by using the **:t** command.

***Main>** :t elem . sort
elem . sort :: (Ord b) => [b] -> [[b]] -> Bool

How does GCHi work this out?

Haskell has three basic rules for establishing the type of an expression

- Function application
- Type instantiation
- Abstraction

Reminder: Function declarations

- Function declarations take the form

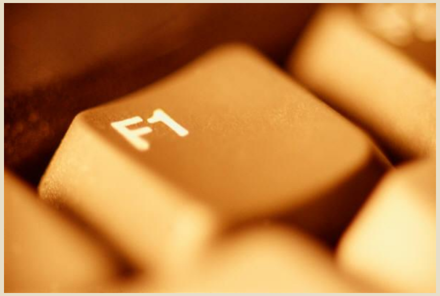
```
f ::   Constraints   =>   Type
```

- The function f has type **Type** provided the constraints are satisfied.
- If the constraints are not satisfied, the function is meaningless


A DECLARATION
UNITED STATES OF AMERICA,
JOHN HANCOCK,

Function application

- From
  - `f :: Cons`$_f$ `=> a -> b`
  - `x :: Cons`$_x$ `=> a`
- Deduce
  - `f x :: (Cons`$_f$`, Cons`$_x$`) => b`



"If f is a function from type a to type b, and you provide it with an argument x of type a, the result (f applied to x) will be of type b"

a and **b** can be any data types
In particular, they might be the **same** type

Example: Function application

- From
  - f :: $\texttt{Cons}_f$ => a -> b
  - x :: $\texttt{Cons}_e$ => a
- Deduce
  - f x :: ($\texttt{Cons}_f$, $\texttt{Cons}_e$) => b

From:

```
double    :: Num a => a -> a
5         :: Num a => a
```

Deduce:

```
double 5 :: (Num a, Num a) => a
```

Type instantiation

- From

  $f :: Cons_f \Rightarrow a$

- Deduce

  $f :: Cons_f\{sigma/b\} \Rightarrow a\{sigma/b\}$

Notation: expr{sigma/b} means "replace all occurrences of b in expr with sigma."
For example

- (Either (List **b**) (Tree c)){**z**/b} = Either (List **z**) (Tree c)
- (Num **a**, Show b){**List a**/a} = (Num (**List a**), Show b)

Example: Type instantiation

- From

  $f :: Cons_f \Rightarrow a$

- Deduce

  $f :: Cons_f\{sigma/b\} \Rightarrow a\{sigma/b\}$

```
   (+)  ::   Num a          =>  a    -> a    -> a
So (+)  ::  (Num a){Int/a}  =>  (a   -> a    -> a){Int/a}
So (+)  ::   Num Int        =>  Int  -> Int  -> Int
```

"Provided Int is in Num [which it is], then (+) has type Int -> Int -> Int". So

```
(+)  :: Int -> Int -> Int
```

Abstraction

This tells us the type of an anonymous function

- From

  `(x :: a)` implies `e ::` $\texttt{Cons}_e$ `=> b`

- Deduce

  `(\x -> e) ::` $\texttt{Cons}_e$ `=> a -> b`

The expression e will typically refer to x. If we assume that x has some arbitrary type a, and this tells us that e then has type b (which will probably depend on a), we deduce that (\x -> e) has type a -> b.

Example: Abstraction

What is the type of the function \x -> (x == x) ?

- **From**

  `(x :: a)` implies `e :: Cons`$_e$ `=> b`

- **Deduce**

  `(\x -> e) :: Cons`$_e$ `=> a -> b`

Assume x :: a

e is (x == x)

What type is e?

```
   (==)     :: Eq a => a -> a -> Bool  -- defn of (==)
        x   ::          a              -- assumed
So (==) x   :: Eq a =>      a -> Bool  -- func. applic.
          x ::              a          -- assumed
So (==) x x :: Eq a =>           Bool  -- func. applic.
```

```
\x -> (x == x)  ::  Eq a => a -> Bool
```

Main example: elem . sort

Recall that **elem . sort** is shorthand for **((.) elem) sort**

Find the type of **(.) elem** first, then apply this to **sort**

```
(.)        ::          (d ->        e         )->(c->d)->(c->e)
     elem  :: Eq a => a -> ([a] -> Bool)
```

We can't apply **(.)** to **elem** immediately, because the types don't match. We need to use type instantiation – we have to change the type of **(.)** by replacing **d** with **a**, and **e** with **([a]->Bool)**.

```
    (.)       :: (d -> e) -> (c -> d) -> (c -> e)
 So (.){a/d} :: (a -> e) -> (c -> a) -> (c -> e)
 So (.)       :: (a -> e) -> (c -> a) -> (c -> e)
```

The expression **(.)** doesn't mention **a** (it's just a dot inside brackets) so **(.){a/d}** is the same as **(.)**

Main example: elem . sort

Recall that **elem . sort** is shorthand for **((.) elem) sort**
Find the type of **(.) elem** first, then apply this to **sort**

```
(.)     ::           (a ->       e        )->(c->a)->(c->e)
   elem :: Eq a => a -> ([a] -> Bool)
```

We still need to replace **e** with **([a]->Bool)**.

```
 (.)                ::    (a -> e)
                       -> (c -> a)
                       -> (c -> e)
So (.){([a]->Bool)/e} ::  (a -> ([a]->Bool))
                       -> (c -> a)
                       -> (c -> ([a]->Bool))
So (.) :: (a->([a]->Bool)) -> (c->a) -> (c->([a]->Bool))
```

Once again, the expression **(.)** doesn't mention **e**

Main example: elem . sort

Recall that **elem . sort** is shorthand for **((.) elem) sort**
Find the type of **(.) elem** first, then apply this to **sort**

```
(.)       ::          (a->([a]->Bool)) -> (c->a) -> (c->([a]->Bool))
    elem :: Eq a => a->([a]->Bool)
(.) elem :: Eq a =>                       (c->a) -> (c->([a]->Bool))
```

`(.) elem :: Eq a => (c->a) -> (c->([a]->Bool))`

We've found the type of **(.) elem**. Now we apply it to **sort**. We'll need to do more type instantiation along the way.

Main example: elem . sort

Recall that **elem . sort** is shorthand for **((.) elem) sort**
Find the type of **(.) elem** first, then apply this to **sort**

```
 (.) elem      :: Eq a          => ( c -> a ) -> (c->([a]->Bool))
          sort ::         Ord b =>  [b]->[b]
```

We need to replace both **a** and **c** with **[b]** to make the types match.

```
(.) elem      ::  Eq [b]         => ([b]->[b]) -> ([b]->([[b]]->Bool))
         sort ::           Ord b  =>  [b]->[b]
```

Therefore, by function application,

```
(.) elem sort :: (Eq [b],Ord b) =>                    [b]->([[b]]->Bool)
```

```
(elem . sort) :: (Eq [b],Ord b) => [b]->([[b]]->Bool)
```

And finally…


```
(elem . sort) :: (Eq [b],Ord b) => [b]->([[b]]->Bool)
```

***Main>** :t elem . sort
elem . sort :: (Ord b) => [b] -> [[b]] -> Bool

Why are these different?

- The definition of **Ord** starts
  class (Eq a) => Ord a where …
- So **Ord b** implies **Eq b**
- But **[b]** is in **Eq** whenever **b** is
- So **Ord b** implies **Eq [b]**

Therefore, the Eq [b] constraint is already included in the Ord b constraint. There is no need to state it explicitly.