The `printf` Problem

- Consider the `printf` function in C:

```
printf ("Hello World!\n");
printf ("Name: %s", name);
printf ("ASCII value = %d, Character = %c\n", ch, ch);
```

- The number and type of arguments `printf` expects depends on the format str

```
int printf (const char *format, ...)
```

The `printf` Problem

- The actual ***type*** of `printf` depends on the ***value*** of its first argument

- Can we do something similar in Haskell?

```
printf :: <FormatInfo> -> <Some type depending on FormatInfo>
```

- The **type** of the format information must reflect which and how many arguments are expected

  - can't be a regular string

The `printf` Problem

- Example:

```
“%s is %d years old”
```

- Our representation: what kind of information do we need to represent on
  - value level
  - type level?

```
S (L " is " (I  (L " years old" X))) ::  Format '[String, Int]
```

```
data Format (fmt :: [*]) where
 X :: Format '[]
 L :: …
 S :: …
 I :: …
```

The `printf` Problem

- Mapping the format type to the type of the `printf` function:

```
type family FormatArgsThen (fmt :: [*]) (ty :: *) :: *
type instance FormatArgsThen '[]        ty = ty
type instance FormatArgsThen (t ': fmt) ty = t -> FormatArgsThen fmt ty
```

Problem: Distinguish values of identical representation

- Mars climate orbiter failure:
  - disintegrated, as trajectory was too close to Mars’ atmosphere
  - calculated impulse was in pound-seconds instead of newton-seconds
- How can we use the type systems to avoid such problems?
  - trade-off between safety and overhead

Phantom types

- A type whose type parameter doesn't show up on the right hand side:

```
newtype Length a = Length Double
  deriving (Show, Eq, Ord)
```

- Can be used when side conditions are not reflected in the representations
  - e.g., should only be possible to add lengths if given in the same unit, but both represented as double precision floating point number

Smart Constructors

- Functions which call a constructor, and usually check some side conditions:

```
newtype IPAddr = IPAddr (Int, Int, Int, Int)

mkIPAddr :: Int -> Int -> Int -> Int -> Maybe IPAddr
mkIPAddr n1 n2 n3 n4
  | n1 >= 0 && n1 /= 10 & ... = Just $ IPAddr (n1, n2, n3, n4)
```

Back to GADTs & type families

- We have seen examples of what we can do with type families:

```
type family (+) (n :: Nat) (m :: Nat) :: Nat
type instance 'Z      + m = m
type instance ('S n) + m = 'S (n + m)
```

```
data Vec a (n :: Nat)  where
  Nil   :: Vec a 'Z
  (:::) :: a -> Vec a n -> Vec  a ('S n)


(++) :: Vec a n -> Vec a m -> Vec a (n + m)
Nil         ++ xs = xs
(x ::: xs) ++ ys = x ::: (xs ++ ys)
```

Back to GADTs & type families

- The extra power doesn't come for free:
  - type annotations often required

```
data Vec a (n :: Nat)  where
  Nil   :: Vec a 'Z
  (:::) :: a -> Vec a n -> Vec  a ('S n)

(++) :: Vec a n -> Vec a m -> Vec a (n + m)
Nil        ++ xs = xs
(x ::: xs) ++ ys = x ::: (xs ++ ys)
```

type can't be derived automatically

Back to GADTs & type families

- Define a function which discards all odd elements from a vector

```
removeOdd (x ::: xs)
  | odd x      = removeOdd xs
  | otherwise = x ::: removeOdd xs
```

- What is the type of this function?

```haskell
type family (+) (n :: Nat) (m :: Nat) :: Nat
type instance 'Z + m = m
type instance ('S n) + m = 'S (n + m)
```

```haskell
data Vec a (n :: Nat)  where
  Nil   :: Vec a 'Z
  (:::) :: a -> Vec a n -> Vec  a ('S n)


(++) :: Vec a n -> Vec a m -> Vec a (n + m)
Nil          ++ xs = xs
(x ::: xs) ++ ys = x ::: (xs ++ ys)
```

left hand side (arguments):

```
Nil :: Vec a ‘Z
xs  :: Vec a m
```

n ~ ‘Z

?

```
Vec a m  ~  Vec (‘Z + m)
```

right hand side (result):

```
xs  :: Vec a m
```

```haskell
type family (+) (n :: Nat) (m :: Nat) :: Nat
type instance 'Z + m = m
type instance ('S n) + m = 'S (n + m)
```

```haskell
data Vec a (n :: Nat)  where
  Nil   :: Vec a 'Z
  (:::) :: a -> Vec a n -> Vec  a ('S n)


(++) :: Vec a n -> Vec a m -> Vec a (n + m)
Nil         ++ xs = xs
(x ::: xs) ++ ys = x ::: (xs ++ ys)
```

left hand side (arguments):

```
   :: Vec a k
(x ::: xs) :: Vec a ('S k)          n ~ 'S k                ?
       ys   :: Vec a m                    Vec a 'S( k + m)  ~  Vec a  (('S k) + m))
```

right hand side (result):

```
  x ::: (xs ++ ys) :: Vec a 'S(k + m)
```

UNSW THE UNIVERSITY OF NEW SOUTH WALES SYDNEY • AUSTRALIA