Type families

Haskell and Cryptocurrencies

Dr. Andres Löh, Well-Typed LLP
Dr. Lars Brünjes, IOHK

2018-03-02

INPUT OUTPUT

Goals

- Introduce type-level functions, also known as type families.

Appending vectors

Appending two vectors

```
(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

For vectors?

Appending two vectors

```
(++) :: [a] -> [a] -> [a]
[]        ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

For vectors?

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

How to complete the type?

Natural number addition

```
(+) :: Nat -> Nat -> Nat
Zero  + n = n
Suc m + n = Suc (m + n)
```

Natural number addition

```
(+) :: Nat -> Nat -> Nat
Zero  + n = n
Suc m + n = Suc (m + n)
```

In a dependently-typed language:

```
(++) :: Vec a m -> Vec a n -> Vec a (m + n)
```

Unfortunately, we *cannot promote functions*.

Use a GADT

GADTs express *relations* on the type level.

Every function is a relation …

Use a GADT

GADTs express *relations* on the type level.

Every function is a relation …

```haskell
data Plus :: Nat -> Nat -> Nat -> * where
  PlusZ ::                     Plus Zero    n n
  PlusS :: Plus m n n' -> Plus (Suc m) n (Suc n')
```

```haskell
(++) :: Plus m n p -> Vec m a -> Vec n a -> Vec p a
(++) PlusZ     Nil       ys = ys
(++) (PlusS p) (x :* xs) ys = x :* (++) p xs ys
```

Use a GADT

GADTs express *relations* on the type level.

Every function is a relation …

```
data Plus :: Nat -> Nat -> Nat -> * where
  PlusZ ::                     Plus Zero    n n
  PlusS :: Plus m n n' -> Plus (Suc m) n (Suc n')
```

```
(++) :: Plus m n p -> Vec m a -> Vec n a -> Vec p a
(++) PlusZ     Nil       ys = ys
(++) (PlusS p) (x :* xs) ys = x :* (++) p xs ys
```

While interesting (and perhaps even useful), it's quite inconvenient to have to provide a `Plus` argument by hand.

Type family

```haskell
(+) :: Nat -> Nat -> Nat
Zero  + n = n
Suc m + n = Suc (m + n)
```

```haskell
type family (+) (m :: Nat) (n :: Nat) :: Nat where
  Zero  + n = n
  Suc m + n = Suc (m + n)
```

Type family

```
(+) :: Nat -> Nat -> Nat
Zero  + n = n
Suc m + n = Suc (m + n)
```

```
type family (+) (m :: Nat) (n :: Nat) :: Nat where
  Zero  + n = n
  Suc m + n = Suc (m + n)
```

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

Evaluating a type family in GHCi

```
GHCi> :kind! Suc Zero + Suc Zero
Suc Zero + Suc Zero :: Nat
 = 'Suc ('Suc Zero)
```

Let's look at the types

```
data Vec :: Nat -> * -> * where
  Nil  :: (n ~ Zero  ) => Vec n a
  (:*) :: (n ~ Suc n') => a -> Vec n' a -> Vec n a

type family (+) (m :: Nat) (n :: Nat) :: Nat where
  Zero  + n = n
  Suc m + n = Suc (m + n)

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

Let’s look at the types

```
data Vec :: Nat -> * -> * where
  Nil  :: (n ~ Zero   ) => Vec n a
  (:*) :: (n ~ Suc n') => a -> Vec n' a -> Vec n a

type family (+) (m :: Nat) (n :: Nat) :: Nat where
  Zero  + n = n
  Suc m + n = Suc (m + n)

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

In the first case, `m ~ Zero`:

```
ys :: Vec n a
   ~ Vec (Zero + n) a  -- type family
   ~ Vec (m + n) a     -- m ~ Zero
```

Let's look at the types

```
data Vec :: Nat -> * -> * where
  Nil  :: (n ~ Zero  ) => Vec n a
  (:*) :: (n ~ Suc n') => a -> Vec n' a -> Vec n a

type family (+) (m :: Nat) (n :: Nat) :: Nat where
  Zero  + n = n
  Suc m + n = Suc (m + n)

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

In the second case, `m ~ Suc m'`:

```
x :* (xs ++ ys) :: Vec Suc (m' + n) a
                 ~ Vec (Suc m' + n) a   -- type family
                 ~ Vec (m + n) a        -- m ~ Suc m'
```

Exercise

Define

```
interleave :: Vec n a -> Vec n a -> Vec (n + n) a
```

such that e.g.

```
GHCi> interleave (1 :* 2 :* Nil) (3 :* 4 :* Nil)
1 :* (3 :* (2 :* (4 :* Nil)))
```

Exercise

Define

```
interleave :: Vec n a -> Vec n a -> Vec (n + n) a
```

such that e.g.

```
GHCi> interleave (1 :* 2 :* Nil) (3 :* 4 :* Nil)
1 :* (3 :* (2 :* (4 :* Nil)))
```

Note: This is difficult, and won't easily work. Can you see why?

A failing attempt

```
interleave :: Vec n a -> Vec n a -> Vec (n + n) a
interleave Nil       Nil       = Nil
interleave (x :* xs) (y :* ys) =
  x :* y :* interleave xs ys  -- type error!
```

A failing attempt

```
interleave :: Vec n a -> Vec n a -> Vec (n + n) a
interleave Nil       Nil       = Nil
interleave (x :* xs) (y :* ys) =
  x :* y :* interleave xs ys  -- type error!
```

We learn `n ~ Suc n'`, and the type of the RHS is:

```
  Vec Suc Suc (n' + n') a
~ Vec Suc (n + n') a
~ Vec (Suc n + n') a
```

A failing attempt

```
interleave :: Vec n a -> Vec n a -> Vec (n + n) a
interleave Nil       Nil       = Nil
interleave (x :* xs) (y :* ys) =
  x :* y :* interleave xs ys  -- type error!
```

We learn `n ~ Suc n'`, and the type of the RHS is:

```
  Vec Suc Suc (n' + n') a
~ Vec Suc (n + n') a
~ Vec (Suc n + n') a
```

But we need:

```
  Vec (Suc n' + Suc n') a
~ Vec (n + Suc n') a
```

No magic theorem prover!

The only equations GHC knows for type families are the equations that appear in the definition:

```
type family (+) (m :: Nat) (n :: Nat) :: Nat where
  Zero  + n = n
  Suc m + n = Suc (m + n)
```

No magic theorem prover!

The only equations GHC knows for type families are the equations that appear in the definition:

```haskell
type family (+) (m :: Nat) (n :: Nat) :: Nat where
  Zero  + n = n
  Suc m + n = Suc (m + n)
```

So we know

```haskell
Zero  + n ~ n
Suc m + n ~ Suc (m + n)
```

but not these:

```haskell
n + Zero   ~ n
Suc n + n' ~ n + Suc n'  -- our case
```

A workaround

```
type family (*) (m :: Nat) (n :: Nat) :: Nat where
  Zero  * n = Zero
  Suc m * n = n + (m * n)
```

A workaround

```haskell
type family (*) (m :: Nat) (n :: Nat) :: Nat where
  Zero  * n = Zero
  Suc m * n = n + (m * n)
```

Try again:

```haskell
interleave :: Vec n a -> Vec n a -> Vec ... a
```

A workaround

```haskell
type family (*) (m :: Nat) (n :: Nat) :: Nat where
  Zero  * n = Zero
  Suc m * n = n + (m * n)
```

Try again:

```haskell
interleave :: Vec n a -> Vec n a -> Vec (n * Two) a
interleave Nil       Nil       = Nil
interleave (x :* xs) (y :* ys) =
  x :* y :* interleave xs ys
```

Proving properties

Defining `reverse` on vectors

```
reverse :: Vec n a -> Vec n a
reverse xs = go xs Nil  -- fails
  where
    go :: Vec m a -> Vec n a -> Vec (m + n) a
    go Nil       acc = acc
    go (x :* xs) acc = go xs (x :* acc)  -- fails
```

This does not type check for similar reasons as our `interleave` example.

A simple property

```haskell
thmPlusZero :: SNat n -> (n + Zero) :~: n
thmPlusZero SZero    = Refl
thmPlusZero (SSuc s) =
  gcastWith (thmPlusZero s) Refl
```

A simple property

```
thmPlusZero :: SNat n -> (n + Zero) :~: n
thmPlusZero SZero    = Refl
thmPlusZero (SSuc s) =
  gcastWith (thmPlusZero s) Refl
```

In the first case, `n ~ Zero`.

So we have to prove

```
   (n + Zero)     :~: n
 ~ (Zero + Zero) :~: Zero
 ~ Zero           :~: Zero
```

and this can be done by using `Refl`.

A simple property

```haskell
thmPlusZero :: SNat n -> (n + Zero) :~: n
thmPlusZero SZero     = Refl
thmPlusZero (SSuc s) =
  gcastWith (thmPlusZero s) Refl
```

In the second case, `n ~ Suc n'`.

So we have to prove

```haskell
   (n + Zero)       :~: n
 ~ (Suc n' + Zero) :~: Suc n'
 ~ Suc (n' + Zero) :~: Suc n'
```

So if we know the theorem for `n'`, we are done.

Another simple property

```haskell
thmPlusSuc :: SNat m -> SNat n
           -> (m + Suc n) :~: (Suc (m + n))
thmPlusSuc SZero    _ = Refl
thmPlusSuc (SSuc s) n =
  gcastWith (thmPlusSuc s n) Refl
```

This follows exactly the same principle as `thmPlusZero`.

Using the properties

```
reverse :: Vec n a -> Vec n a
reverse xs =
  gcastWith (thmPlusZero (length xs)) $ go xs Nil
  where
    go :: Vec m a -> Vec n a -> Vec (m + n) a
    go Nil       acc = acc
    go (x :* xs) acc =
      gcastWith
        (thmPlusSuc (length xs) (length acc)) $
      go xs (x :* acc)
```

Exercise

Reimplement `interleave` with the original type using properties:

```
interleave :: Vec n a -> Vec n a -> Vec (n + n) a
```

There are at least two options:

- Just reuse `thmPlusSuc` in the definition of `interleave`.
- Prove that `n * Two` equals `n + n` and use the definition of `interleave` with the other type.

Associated types

Open type families

```
type family Elt as :: *

class Sequence (as :: *) where
  filter :: (Elt as -> Bool) -> as -> as
   ...
```

Open type families

```
type family Elt as :: *

class Sequence (as :: *) where
  filter :: (Elt as -> Bool) -> as -> as
  ...
```

```
type instance Elt [a] = a

instance Sequence [a] where
  filter = L.filter
```

```
type instance Elt Text = Char

instance Sequence Text where
  filter = T.filter
```

Open type families

```
type family Elt as :: *

class Sequence (as :: *) where
  filter :: (Elt as -> Bool) -> as -> as
  ...
```

```
type instance Elt [a] = a

instance Sequence [a] where
  filter = L.filter
```

```
type instance Elt Text = Char

instance Sequence Text where
  filter = T.filter
```

Instances cannot overlap, but can be added at any time.

Associated types

```
class Sequence (as :: *) where
  type Elt as :: *
  filter :: (Elt as -> Bool) -> as -> as
   ...
```

Associated types

```
class Sequence (as :: *) where
  type Elt as :: *
  filter :: (Elt as -> Bool) -> as -> as
  ...
```

```
instance Sequence [a] where
  type Elt [a] = a
  filter = L.filter
```

```
instance Sequence Text where
  type Elt Text = Char
  filter = T.filter
```

This is mainly a syntactic difference.

Recap

We've seen open type families / associated types before in the context of monad transformer interface classes, where they can e.g. be used to map a monad to its state type, as an alternative to functional dependencies.

Example: Treating references uniformly

```
newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

```
newSTRef   :: a -> ST s (STRef s a)
readSTRef  :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
```

```
TVar       :: a -> STM (TVar a)
readTVar   :: TVar a -> STM a
writeTVar  :: TVar a -> a -> STM ()
```

A solution

```
class HasRef (m :: * -> *) where
  type Ref m :: * -> *
  new   :: a -> m (Ref m a)
  read  :: Ref m a -> m a
  write :: Ref m a -> a -> m ()
```

A solution

```haskell
class HasRef (m :: * -> *) where
  type Ref m :: * -> *
  new   :: a -> m (Ref m a)
  read  :: Ref m a -> m a
  write :: Ref m a -> a -> m ()
```

```haskell
instance HasRef IO where
  type Ref IO = IORef
  new   = newIORef
  read  = readIORef
  write = writeIORef
```

The other instances are analogous.

Injectivity

Example: Different representations of data

This is somewhat contrived, but illustrates a very real and very frequent problem when dealing with type families:

```
class Compactable (a :: *) where
  type Compact a :: *
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Compact a -> Int

instance Compactable Int
instance Compactable a => Compactable [a]
```

A strange error

```
Couldn't match type ‘ Compact a ’ with ‘ Compact a0 ’
Expected type: ‘ Compact a -> Int ’
Actual type: ‘ Compact a0 -> Int ’
NB: ‘ Compact ’ is a type function, and may not be injective
The type variable ‘ a0 ’ is amgiguous
In the ambiguity check for ‘ size ’
To defer the ambiguity check to use sites,
    enable AllowAmbiguousTypes
```

We can try enabling `AllowAmbiguousTypes`, but then ...

A strange error

```
test = size (compact [1, 2, 3])
```

A strange error

```
test = size (compact [1, 2, 3])
```

```
Couldn't match expected type ' Compact a0 '
   with actual type ' Compact [t0] '
NB: ' Compact ' is a type function, and may not be injective
The type variables ' a0 ', ' t0 ' are ambiguous
In the first argument of ' size ',
   namely ' (compact [1, 2, 3]) '
In the expression:  size (compact [1, 2, 3])
```

A strange error

```
test = size (compact [1, 2, 3])
```

Couldn't match expected type ‘ `Compact a0` ’
  with actual type ‘ `Compact [t0]` ’
NB: ‘ `Compact` ’ is a type function, and may not be injective
The type variables ‘ `a0` ’, ‘ `t0` ’ are ambiguous
In the first argument of ‘ `size` ’,
  namely ‘ `(compact [1, 2, 3])` ’
In the expression: `size (compact [1, 2, 3])`

```
test = size
  (compact ([1, 2, 3] :: [Int]) :: Compact [Int])
```

is not improving anything.

Explaining the error

```haskell
test = size (compact [1, 2, 3] :: Compact [Int])
```

```haskell
size :: Compactable a => Compact a -> Int
compact [1, 2, 3] :: Compact [Int]
```

Explaining the error

```
test = size (compact [1, 2, 3] :: Compact [Int])
```

```
size :: Compactable a => Compact a -> Int
compact [1, 2, 3] :: Compact [Int]
```

So we have to unify:

```
Compact [Int] ~ Compact a
```

It seems like `a ~ [Int]` is an obvious solution, but is it the only one?

Type families need not be injective

```haskell
type Compact [a] = Array Int (Compact a)
type Compact Int = Int
```

```haskell
newtype Count = Count Int
type Compact Count = Int
```

Type families need not be injective

```haskell
type Compact [a] = Array Int (Compact a)
type Compact Int = Int
```

```haskell
newtype Count = Count Int
type Compact Count = Int
```

Now:

```haskell
Compact [Int] ~ Array Int Int ~ Compact [Count]
```

Injectivity

In general, a function `f` is called *injective* if `f x ~ f y` implies `x ~ y`.

Injectivity

In general, a function `f` is called injective if `f x ~ f y` implies `x ~ y`.

Datatypes (both `data` and `newtype`) are *always* injective, but type families (and type synonyms) are generally not.

Recognizing problematic functions

```haskell
class Compactable (a :: *) where
  type Compact a :: *
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Compact a -> Int
```

Recognizing problematic functions

```haskell
class Compactable (a :: *) where
  type Compact a :: *
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Compact a -> Int
```

In `size`, the type variable `a` appears only as an argument to a type family – it's impossible to use this function in practice.

Solution 1: Making the type family injective

```
class Compactable (a :: *) where
  type Compact a = (r :: *) | r -> a
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Compact a -> Int

instance Compactable Int
instance Compactable a => Compactable [a]

test = size (compact [1, 2, 3] :: Compact [Int])
```

Solution 1: Making the type family injective

```
class Compactable (a :: *) where
  type Compact a = (r :: *) | r -> a
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Compact a -> Int

instance Compactable Int
instance Compactable a => Compactable [a]

test = size (compact [1, 2, 3] :: Compact [Int])
```

This requires `TypeFamilyDependencies`.

Solution 1: Making the type family injective

```
class Compactable (a :: *) where
  type Compact a = (r :: *) | r -> a
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Compact a -> Int

instance Compactable Int
instance Compactable a => Compactable [a]

test = size (compact [1, 2, 3] :: Compact [Int])
```

This requires `TypeFamilyDependencies`.

The function `test` is now accepted. GHC enforces injectivity. Straight-forward, but not all type families are injective, so not always an option.

Solution 2: Redesigning the class hierarchy

```
class
  Size (Compact a) => Compactable (a :: *) where
  type Compact a :: *
  compact   :: a -> Compact a
  uncompact :: Compact a -> a

class Size a where
  size :: a -> Int
```

Probably the best solution in this situation (but again it is not always applicable).

Solution 2: Redesigning the class hierarchy

```
class
  Size (Compact a) => Compactable (a :: *) where
  type Compact a :: *
  compact   :: a -> Compact a
  uncompact :: Compact a -> a

class Size a where
  size :: a -> Int
```

Probably the best solution in this situation (but again it is not always applicable).

Now

```
test = size (compact ([1, 2, 3] :: [Int]))
```

typechecks as long as `Size (Compact [Int])` holds.

Solution 3: Using a proxy

```haskell
data Proxy (a :: k) = Proxy

class Compactable (a ::*) where
  type Compact a ::*
  compact    :: a -> Compact a
  uncompact :: Compact a -> a
  size       :: Proxy a -> Compact a -> Int
```

The additional argument is annoying, but this always works.

```haskell
test = size (Proxy :: Proxy [Int])
  (compact ([1, 2, 3] :: [Int]))
```

Solution 3: Using a proxy

```
data Proxy (a :: k) = Proxy

class Compactable (a :: *) where
  type Compact a :: *
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Proxy a -> Compact a -> Int
```

The additional argument is annoying, but this always works.

```
test = size (Proxy :: Proxy [Int])
  (compact ([1, 2, 3] :: [Int]))
```

```
data Tagged (a :: k) b = Tagged b

size :: Tagged a (Compact a) -> Int  -- another option
```

Solution 4: use explicit type application

The original class:

```
class Compactable (a ::*) where
  type Compact a ::*
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Compact a -> Int
```

```
test = size @[Int]
  (compact ([1, 2, 3] :: [Int]))
```

Allows us to explicitly instantiate `a` – requires both `AllowAmbiguousTypes` and `TypeApplications` (and a recent GHC).

Solution 5: Writing an inverse

If the function actually is injective, we can "prove" it by writing an inverse:

```
type family Uncompact (a :: *) :: *

class (Uncompact (Compact a) ~ a)
        => Compactable (a :: *) where
  type Compact a :: *
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Compact a -> Int
```

Solution 5: Writing an inverse

If the function actually is injective, we can “prove” it by writing an inverse:

```
type family Uncompact (a :: *) :: *

class (Uncompact (Compact a) ~ a)
        => Compactable (a :: *) where
  type Compact a :: *
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Compact a -> Int
```

Extra work, but now, by applying `Uncompact`, we can solve

```
Compact a ~ Compact b
```

Data families

Data families

Next to

```
type family X...
```

there's also

```
data family X...
```

that allows

```
data instance...
newtype instance...
```

(And associated datatypes correspondingly.)

Tempting because they're always injective – not useful if you work with types that already exist.

Generalizing singleton types

We can use a *kind-indexed* data family to make singletons less ad-hoc.

Before:

```
data SNat :: Nat -> * where
  SZero :: SNat Zero
  SSuc  :: SNat n -> SNat (Suc n)
```

Generalizing singleton types

We can use a *kind-indexed* data family to make singletons less ad-hoc.

Before:

```
data SNat :: Nat -> * where
  SZero :: SNat Zero
  SSuc  :: SNat n -> SNat (Suc n)
```

Now:

```
data family Sing (a :: k)

data instance Sing (n :: Nat) where
  SZero :: Sing Zero
  SSuc  :: Sing n -> Sing (Suc n)
```

Constraint kinds

Classes have kinds

```
Eq         :: * -> Constraint
Functor    :: (* -> *) -> Constraint
MonadState :: * -> (* -> *) -> Constraint
```

Classes have kinds

```
Eq         :: * -> Constraint
Functor    :: (* -> *) -> Constraint
MonadState :: * -> (* -> *) -> Constraint
```

By viewing constraints as kind, we can e.g.

- define class synonyms using `type`,
- parameterize types and classes over constraints,
- define constraint families.

Computing a constraint

Using constraint kinds and type families, we can finally provide a proper `Show` instance for environments.

Computing a constraint

Using constraint kinds and type families, we can finally provide a proper `Show` instance for environments.

In order to e.g. show

```
I 'x' :* I False :* I LT :* Nil ::
  Env '[Char, Bool, Ordering] I
```

we need to know

```
(Show (I Char), Show (I Bool), Show (I Ordering))
```

Mapping a constraint over a list

```haskell
type family All
  (c :: k -> Constraint) (xs :: [k]) ::
  Constraint where
  All c '[]        = ()
  All c (x ': xs) = (c x, All c xs)
```

Mapping a constraint over a list

```
type family All
  (c :: k -> Constraint) (xs :: [k]) ::
  Constraint where
  All c '[]         = ()
  All c (x ': xs) = (c x, All c xs)
```

Halfway there:

```
GHCi> :kind! All Show '[Char, Bool, Ordering]
All Show '[Char, Bool, Ordering] :: Constraint
 = (Show Char, (Show Bool,
     (Show Ordering, () :: Constraint)))
```

Mapping over a type-level list

```
type family Map
  (f :: k1 -> k2) (xs :: [k1]) :: [k2] where
  Map f '[]        = '[]
  Map f (x ': xs) = (f x) ': (Map f xs)
```

Mapping over a type-level list

```
type family Map
  (f :: k1 -> k2) (xs :: [k1]) :: [k2] where
  Map f '[]        = '[]
  Map f (x ': xs) = (f x) ': (Map f xs)
```

Now we can compute what we want:

```
GHCi> :kind! All Show
                (Map I '[Char, Bool, Ordering])
All Show
  (Map I '[Char, Bool, Ordering]) :: Constraint
 = (Show (I Char), (Show (I Bool),
     (Show (I Ordering), () :: Constraint)))
```

Deriving the instance

```haskell
data Env :: [*] -> (* -> *) -> * where
  Nil  :: Env '[] f
  (:*) :: f t -> Env ts f -> Env (t ': ts) f

deriving instance
  All Show (Map f xs) => Show (Env xs f)
```

Higher-order functions?

More about `Map`

So we have type-level map on type-level lists:

```
GHCi> :kind! Map I '[Int, Bool]
Map I '[Int, Bool] :: [*]
 = '[I Int, I Bool]
GHCi> :kind! Map Maybe '[Int, Bool]
Map Maybe '[Int, Bool] :: [*]
 = '[Maybe Int, Maybe Bool]
GHCi> Map Suc '[Zero, Suc Zero]
Map Suc '[Zero, Suc Zero] :: [Nat]
 = '[Suc Zero, Suc (Suc Zero)]
```

No partial application

However, the following fails:

```
type family Const (a :: k1) (b :: k2) :: k1 where
  Const a b = a
```

```
problem :: Env '[Int, Char] (Const Bool)
problem = True :* False :* Nil  -- type error
```

The error message says that `Const` should have two arguments, but only one was provided.

No partial application (contd.)

- Just like type synonyms, type families must always be fully applied.
- In particular, an argument of function kind cannot be filled with a partially applied type family.
- Therefore, the use of higher-order functions in type-level programming is severely limited.

No partial application (contd.)

- Just like type synonyms, type families must always be fully applied.
- In particular, an argument of function kind cannot be filled with a partially applied type family.
- Therefore, the use of higher-order functions in type-level programming is severely limited.
- There are some tricks to work around this. If you are interested, have a look at the `singletons` package.

Summary

Reflections on type-level programming

- We can push the limits of the type system quite a bit.
- Non-trivial invariants about our types can be expressed and statically checked.
- Many erroneous values or program states become impossible to construct.

Reflections on type-level programming

- We can push the limits of the type system quite a bit.
- Non-trivial invariants about our types can be expressed and statically checked.
- Many erroneous values or program states become impossible to construct.
- However, all this has a price ...

The cost of type-level programming

- We need many new language extensions that not every Haskeller does feel familiar with.
- Type signatures and error messages become much less clear.
- As soon as we have to provide extra help for the type system (singleton types, proofs), there is substantially more work involved in programming.

When is it useful?

- Some lightweight use of features is generally ok (e.g. the occasional rank-2 type, or an occasional type family).
- More heavy use pays off if it can be properly hidden. I.e., if it can be used to establish important guarantees about the internals of a tricky library without affecting the interface too much.
- Another use case are programs that otherwise *could not be written as conveniently at all*. In particular programs involving type-driven code generation, such as *datatype-generic programming*.