FUN WITH TYPE FUNCTIONS

Simon Peyton Jones (Microsoft Research)
Chung-Chieh Shan (Rutgers University)
Oleg Kiselyov (Fleet Numerical Meteorology and Oceanography Center)
class Num a where
  (+), (*) :: a -> a -> a
  negate   :: a -> a

square :: Num a => a -> a
square x = x*x

instance Num Int where
  (+)    = plusInt
  (*)    = mulInt
  negate = negInt

plusInt :: Int -> Int -> Int
mulInt  :: Int -> Int -> Int
negInt :: Int -> Int

test = square 4 + 5 :: Int
class GNum a b where
  (+) :: a -> b -> ???

instance GNum Int Int where
  (+) x y = plusInt x y

instance GNum Int Float where
  (+) x y = plusFloat (intToFloat x) y

test1 = (4::Int) + (5::Int)
test2 = (4::Int) + (5::Float)

plusInt :: Int -> Int -> Int
plusFloat :: Float -> Float -> Float
intToFloat :: Int -> Float

Allowing more good programs
class GNum a b where
  (+) :: a -> b -> ???

- Result type of (+) is a **function of the argument types**

```haskell
class GNum a b where
  type SumTy a b :: *
  (+) :: a -> b -> SumTy a b
```

- Each method gets a type signature
- Each associated type gets a kind signature
Each instance declaration gives a “witness” for \texttt{SumTy}, matching the kind signature

\begin{verbatim}
instance GNum Int Int where
    type SumTy Int Int = Int
    (+) x y = plusInt x y

instance GNum Int Float where
    type SumTy Int Float = Float
    (+) x y = plusFloat (intToFloat x) y
\end{verbatim}
Type functions

- SumTy is a type-level function
- The type checker simply rewrites
  - SumTy Int Int --> Int
  - SumTy Int Float --> Float
  whenever it can
- But (SumTy t1 t2) is still a perfectly good type, even if it can’t be rewritten. For example:

```haskell
data T a b = MkT a b (SumTy a b)
```
Eliminate bad programs

- Simply omit instances for incompatible types

```haskell
newtype Dollars = MkD Int

instance GNum Dollars Dollars where
  type SumTy Dollars Dollars = Dollars
  (+) (MkD d1) (MkD d2) = MkD (d1+d2)

-- No instance GNum Dollars Int

test = (MkD 3) + (4::Int) -- REJECTED!
```
Consider a finite map, mapping keys to values.

Goal: the data representation of the map depends on the type of the key:
- **Boolean key**: store two values (for F,T resp)
- **Int key**: use a balanced tree
- **Pair key** (x,y): map x to a finite map from y to value; ie use a trie!

Cannot do this in Haskell...a good program that the type checker rejects.
class Key k where
  data Map k :: * -> *
  empty :: Map k v
  lookup :: k -> Map k v -> Maybe v
...insert, union, etc....

data Maybe a = Nothing | Just a

Map is indexed by k, but parametric in its second argument.
class Key k where
  data Map k :: * -> *
  empty :: Map k v
  lookup :: k -> Map k v -> Maybe v
  ...insert, union, etc....

instance Key Bool where
  data Map Bool v = MB (Maybe v) (Maybe v)
  empty = MB Nothing Nothing
  lookup True (MB _ mt) = mt
  lookup False (MB mf _) = mf
class Key k where
  data Map k :: * -> *
  empty :: Map k v
  lookup :: k -> Map k v -> Maybe v

instance (Key a, Key b) => Key (a,b) where
  data Map (a,b) v = MP (Map a (Map b v))
  empty = MP empty
  lookup (ka,_kb) (MP m) = case lookup ka m of
    Nothing -> Nothing
    Just m2 -> lookup kb m2
Optimising data structures

- **Goal:** the data representation of the map depends on the **type** of the key
  - **Boolean key:** SUM
    
    ```haskell
data Map Bool v = MB (Maybe v) (Maybe v)
```
  - **Pair key** (x,y): PRODUCT
    
    ```haskell
data Map (a,b) v = MP (Map a (Map b v))
```

- **What about List key** [x]: SUM of PRODUCT + RECURSION?
Lists

instance (Key a) => Key [a] where
  data Map [a] v = ML (Maybe v) (Map (a,[a]) v)
  empty = ML Nothing empty
  lookup [] (ML m0 _) = m0
  lookup (h:t) (ML _ m1) = lookup (h,t) m1

- Note the cool recursion: these Maps are potentially infinite!
- Can use this to build a trie for (say) Int toBits :: Int -> [Bit]
Types with special maps

- Easy to accommodate types with non-generic maps: just make a type-specific instance

```haskell
instance Key Int where
  data Map Int elt = IM (Data.IntMap.Map elt)
  empty = IM Data.IntMap.empty
  lookup k (IM m) = Data.IntMap.lookup m k

module Data.IntMap where
  data Map elt = ...
  empty :: Map elt
  lookup :: Map elt -> Int -> Maybe elt
  ...etc...
```
Memo functions

- One way: when you evaluate \((f \ x)\) to give \(\text{val}\), add \(x \rightarrow \text{val}\) to \(f\)'s memo table, by side effect.

- A nicer way: build a (lazy) table for all possible values of \(x\)

```haskell
class Memo k where
  data Table k :: * -> *
  toTable :: (k->r) -> Table k r
  fromTable :: Table k r -> (k->r)

  memo :: Memo k => (k->r) -> k -> r
  memo f = fromTable (toTable f)
```
class Memo k where
  data Table k :: * -> *
  toTable :: (k->r) -> Table k r
  fromTable :: Table k r -> (k->r)

instance Memo Bool where
  data Table Bool w = TBbool w w
  toTable f = TBbool (f True) (f False)
  fromTable (TBbool x y) b = if b then x else y

- Table contains (lazily) pre-calculated results for both True and False
instance (Memo a) => Memo [a] where
  data Table [a] w
  = TList w (Table a (Table [a] w))
As with Map, the memo table is infinite (second use of laziness)

class Memo k where
    data Table k :: * -> *
    toTable :: (k->r) -> Table k r
    fromTable :: Table k r -> (k->r)
instance Memo Int where
  data Table Int w = TInt (Table [Bool] w)

  toTable f = TInt (toTable (∖bs ->
                        f (bitsToInt bs)))

  fromTable (TInt t) n = fromTable t (intToBits n)

class Memo k where
  data Table k :: * -> *
  toTable :: (k->r) -> Table k r
  fromTable :: Table k r -> (k->r)
Dynamic programming

fib :: Int -> Int
fib = fromTable (toTable fib')
    where
        fib' :: Int -> Int
        fib' 0 = 1
        fib' 1 = 1
        fib' n = fib (n-1) + fib (n-2)

- Recursive calls are to the memo'd function
DATA PARALLEL HASKELL
Arrays of pointers to boxed numbers are Much Too Slow

Arrays of pointers to pairs are Much Too Slow

Idea!

Representation of an array depends on the element type
class Elem a where
    data [:a:]
    index :: [:a:] -> Int -> a

instance Elem Double where
    data [:Double:] = AD ByteArray
    index (AD ba) i = ...

instance (Elem a, Elem b) => Elem (a,b) where
    data [:((a,b)):] = AP [:a:] [:b:]
    index (AP a b) i = (index a i, index b i)
Nested arrays

We do not want this for [: [:Float:] :]
The flattening transformation

- Concatenate sub-arrays into one big, flat array
- Operate in parallel on the big array
- Segment vector keeps track of where the sub-arrays are

- Lots of tricksy book-keeping!
- Possible to do by hand (and done in practice), but very hard to get right
- Bleloch showed it could be done systematically
concatP, segmentP are constant time
And are important in practice
CONSTRAINT KINDS
A long-standing problem

```
class Collection c where
    insert :: a -> c a -> c a

instance Collection [] where
    insert x [] = [x]
    insert x (y:ys)
        | x==y      = y : ys
        | otherwise = y : insert x ys
```

Does not work! We need `Eq`!
A long-standing problem

class Collection c where
    insert :: Eq a => a -> c a -> c a

instance Collection [] where
    insert x [] = [x]
    insert x (y:ys)
        | x==y     = y : ys
        | otherwise = y : insert x ys

instance Collection BalancedTree where
    insert = ...needs (>)...
We want the constraint to vary with the collection \( c \)!

```haskell
class Collection c where
  type X c a :: Constraint
  insert :: X c a => a -> c a -> c a

instance Collection [] where
  type X [] a = Eq a
  insert x [] = [x]
  insert x (y:ys)
    | x==y      = y : ys
    | otherwise = y : insert x ys
```

An associated type of the class

For lists, use `Eq`
We want the constraint to vary with the collection \(c\)!

```haskell
class Collection c where
  type X c a :: Constraint
  insert :: X c a => a -> c a -> c a

instance Collection BalancedTree where
  type X BalancedTree a = (Ord a, Hashable a)
  insert = ...(>)...hash...
```

For balanced trees use (Ord,Hash)
**Associated constraints!**

- Lovely because, it is simply a combination of
  - Associated types (existing feature)
  - Having Constraint as a kind

- No changes at all to the intermediate language!

\[ \kappa ::= \ast \mid \kappa \to \kappa \]

\[ \forall k. \kappa \mid k \]

\[ \text{Constraint} \]
BABY SESSION TYPES
Baby session types (BST)

- addServer :: In Int (In Int (Out Int End))
- addClient :: Out Int (Out Int (In Int End))

- Type of the process expresses its protocol

- Client and server should have dual protocols:
  - run addServer addClient -- OK!
  - run addServer addServer -- BAD!
Baby session types

- addServer :: In Int (In Int (Out Int End))
- addClient :: Out Int (Out Int (In Int End))

```
data In v p = In (v -> p)
data Out v p = Out v p
data End = End
```

NB punning
Nothing fancy here

- addClient is similar
But what about run???

run :: ??? -> ??? -> End

A process

A co-process

class Process p where
    type Co p
    run :: p -> Co p -> End

- Same deal as before: Co is a type-level function that transforms a process type into its dual
Just the obvious thing really
PRINTF
C: `sprintf("Hello%s.", name)`

Format descriptor is a string; absolutely no guarantee the number or types of the other parameters match the string.

Haskell: `(sprintf "Hello%s." name)`??
- No way to make the `type` of `(sprintf f)` depend on the `value` of `f`
- But we can make the `type` of `(sprintf f)` depend on the `type` of `f`!
sprintf :: F f -> SPrintf f

sprintf (Lit "Day") :: String
-- Like printf("Day")

sprintf (Lit "Day " `Cmp` int) :: Int -> String
-- Like printf("Day %n")

sprintf (Lit "Day " `Cmp` int `Cmp` Lit "Month" `Cmp` string)
:: Int -> String -> String
-- Like printf("Day %n Month %s")
data F f where
  Lit :: String -> F L
  Val :: Parser val -> Printer val -> F (V val)
  Cmp :: F f1 -> F f2 -> F (f1 `C` f2)

data L
data V a
data C a b

type Parser a = String -> [(a,String)]
type Printer a = a -> String

int :: F (Val Int)
int = Val (..parser for Int..) (..printer for Int..)
data F f where
  Lit :: String -> F L
  Val :: Parser val -> Printer val -> F (V val)
  Cmp :: F f1 -> F f2 -> F (f1 `C` f2)

int :: F (Val Int)
int = Val (...parser for Int...) (..printer for Int)

f_ld = Lit "day" :: F L
f_lds = Lit "day" `Cmp` Lit "s" :: F (L `C` L)
f_dn = Lit "day " `Cmp` int :: F (L `C` V Int)
f_nds = int `Cmp` Lit " day" `Cmp` Lit "s" :: F (V Int `C` L `C` L)
What we’d like to say

data F :: Fmt -> * where
  Lit :: String -> F L  
  Val :: Parser val -> Printer val -> F (V val)  
  Cmp :: F f1 -> F f2 -> F (C f1 f2)

data Fmt = L | V * | C Fmt Fmt

type Parser a = String -> [(a,String)]
type Printer a = a -> String

<table>
<thead>
<tr>
<th>Format</th>
<th>Kindedness</th>
</tr>
</thead>
<tbody>
<tr>
<td>F L</td>
<td>Well kinded</td>
</tr>
<tr>
<td>F (L <code>C</code> L)</td>
<td>Well kinded</td>
</tr>
<tr>
<td>F Int</td>
<td>Ill kinded</td>
</tr>
<tr>
<td>F (Int <code>C</code> L)</td>
<td>Ill kinded</td>
</tr>
</tbody>
</table>

But can’t (quite) write this yet
Now we can write the type of `sprintf`:

```
sprintf :: F f -> SPrintf f
```

The type-level counterpart to `sprintf`

<table>
<thead>
<tr>
<th>Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SPrintf L</code></td>
<td><code>String</code></td>
</tr>
<tr>
<td><code>SPrintf (L </code>C<code> L)</code></td>
<td><code>String</code></td>
</tr>
<tr>
<td><code>SPrintf (L </code>C<code> V Int)</code></td>
<td><code>Int -&gt; String</code></td>
</tr>
<tr>
<td><code>SPrintf (V Int </code>C<code>L</code>C<code> L)</code></td>
<td><code>Int -&gt; String</code></td>
</tr>
<tr>
<td><code>SPrintf (V Int </code>C<code>L</code>C<code> V Int)</code></td>
<td><code>Int -&gt; Int -&gt; String</code></td>
</tr>
</tbody>
</table>

No type classes here: we are just doing type-level computation
type SPrintf f = TPrinter f String

type family TPrinter f x

type instance TPrinter L x = x

type instance TPrinter (V val) x = val -> x

type instance TPrinter (C f1 f2) x = TPrinter f1 (TPrinter f2 x)

- The `C` constructor suggests a (type-level) accumulating parameter

"Type family" declares a type function without involving a type class
Back to sprintf

\[
\text{sprintf} :: F \ f \rightarrow \text{SPrintf} \ f
\]

\[
\text{sprintf} \ (f1 \ `\text{Cmp}` \ f2) = ???
\]

-- \text{sprintf f1} :: \text{Int} \rightarrow \text{Bool} \rightarrow \text{String (say)}
-- \text{sprintf f2} :: \text{Int} \rightarrow \text{String}
-- These don’t compose!
Back to `sprintf`

- Use an accumulating parameter (a continuation), just as we did at the type level

```haskell
-- Define a function to print strings

-- `sprintf` defines a function that takes a partial `sprintf` function and a continuation

```haskell
-- `print` is defined to use the `sprintf` function

```haskell
```
Same format descriptors for scan

\texttt{sscanf :: F f -> SS\texttt{scanf}} \texttt{f}

* Same format descriptor
* Result type computed by a different type function (of course)
EQUALITY CONSTRAINTS
What is the type of union?

union :: Coll c => c -> c -> c

But we could sensibly union any two collections whose elements were the same type

eg  c1 :: BitSet, c2 :: [Char]
Equality predicates

- But we could sensibly union any two collections whose elements were the same type
  
  \[ c_1 :: \text{BitSet}, c_2 :: [\text{Char}] \]

- Elem is not \textit{injective}
Equality predicates

union :: (Coll c1, Coll c2, Elem c1 ~ Elem c2) => c1 -> c2 -> c2
union c1 c2 = foldl1 insert c2 (elems c1)

insert :: Coll c => c -> Elem c -> c
elems :: Coll c => c -> [Elem c]
The paper: more examples “Fun with type functions”

- Machine address computation
  \[\text{add} :: \text{Pointer } n \rightarrow \text{Offset } m \rightarrow \text{Pointer } (GCD \ n \ m)\]

- Tracking state using Hoare triples
  \[
  \text{acquire} :: (\text{Get } n \ p \sim \text{Unlocked}) \\
  \rightarrow \text{Lock } n \rightarrow \text{M } p \ (\text{Set } n \ p \ \text{Locked}) \ ()
  \]
  [Diagram: Lock-state before and Lock-state after]

- Type level computation tracks some abstraction of value-level computation; type checker assures that they “line up”.

- Need strings, lists, sets, bags at type level
Type families let you do type-level computation

Data families allow the data representation to vary, depending on the type index

They fit together very naturally with type classes. How else could you write

```haskell
f :: F a -> Int
f x = ???  -- Don’t know what F a is!
```

Wildly popular in practice
“Program correctness is a basic scientific ideal for Computer Science”

Types have made a huge contribution to this ideal

More sophisticated type systems threaten both Happy Properties:
1. Automation is harder
2. The types are more complicated (MSc required)

Some complications (2) are exactly due to ad-hoc restrictions to ensure full automation

At some point it may be best to say “enough fooling around: just use Coq”. But we aren’t there yet

Haskell is a great place to play this game
data F f where
    Lit :: String -> F L
    Val :: Parser val -> Printer val -> F (V val)
    Cmp :: F f1 -> F f2 -> F (C f1 f2)

sprintf f = print f (\s -> s)

print :: F f -> (String -> a) -> TPrinter f a
print (Lit s) k = k s
...

In this RHS we know that f~L
Equality predicates are nothing new

data F f where
  Lit :: String -> F L
  Val :: Parser val -> Printer val -> F (V val)
  Cmp :: F f1 -> F f2 -> F (C f1 f2)

sprintf f = print f (\s -> s)

print :: F f -> (String -> a) -> TPrinter f a
print (Lit s) k = k s
...

In this RHS we know that f~L

data F f where
  Lit :: (f ~ L) => String -> F f
  Val :: (f ~ V val) => ... -> F f
  Cmp :: (f ~ C f1 f2) => F f1 -> F f2 -> F f
Completely subsumes functional dependencies

\[ \text{class } C \text{ a b } | \text{ a->b, b->a where...} \]

If I have evidence for \( (C \text{ a b}) \), then I have evidence that \( F1 \text{ a } \sim \text{ b}, \) and \( F2 \text{ b } \sim \text{ a} \)

\[ \text{class } (F1 \text{ a } \sim \text{ b}, F2 \text{ b } \sim \text{ a}) \Rightarrow C \text{ a b where} \]

\[ \text{ type } F1 \text{ a} \]
\[ \text{ type } F2 \text{ b} \]
\[ ... \]