

## ***Scrap more boilerplate***

Ralf Lämmel, VU and CWI, Amsterdam

Simon Peyton Jones, Microsoft Research, Cambridge

Subtitles:

- Generic programming *in* Haskell cont'd
- Generic functions for traversal, eq, show, read, generate
- A run-time reflection API for Haskell data
- Generalised nominal type-safe cast

# Scrap your boilerplate — as of TLDI'03

The paradise benchmark for conciseness + adaptiveness

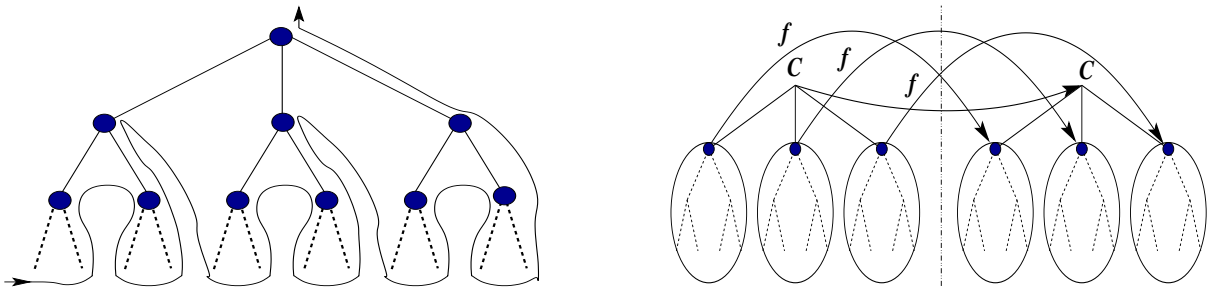
```
increase :: Float → Company → Company  
increase k = everywhere (id `extT` (incS k))
```

```
incS :: Float → Salary → Salary  
incS k (S s) = S (s * (1+k))
```

---

<i>everywhere</i>	...	apply a transformation at all nodes
<i>extT</i>	...	extend a generic transformation in a type

*everywhere* defined in terms of *gmapT*



everywhere :: Data a => (forall b. Data b => b -> b) -> a -> a  
 everywhere f x = f (gmapT (everywhere f) x)

## *gmapT* defined in terms of *gfoldl*

```
gmapT f = gfoldl k id
  where k capp x = capp (f x)
```

## Generic fold

$$gfoldl f z (C x_1 \dots x_n) = (\dots (z C 'f' x_1) \dots) 'f' x_n$$

## List fold (for comparison)

$$foldl f z [x_1, \dots, x_n] = (\dots (z 'f' x_1) \dots) 'f' x_n$$

## Other gmap combinators

- gmapQ — query immediate subterms; return list
- gmapM — gmapT but with monadic threading
- ...

## Primitives so far

### The *Data* class

```
class Typeable t ⇒ Data t where
  -- Elisions avoid brain damage
  gfoldl :: ... → ... → t → r t
```

### The Typeable class

```
class Typeable x -- opaque
```

### The primitive for type-safe cast

```
cast :: (Typeable a, Typeable b) ⇒ a → Maybe b
```

## Now let's try generic oldies

### Bit-stream serialisation and de-serialisation

```
data2bits :: Data a => a -> [Bit]
bits2data :: Data a => [Bit] -> Maybe a
```

### Generic show and read

```
gshow :: Data a => a -> String
gread :: Data a => String -> Maybe a
```

### Generic equality

```
geq :: Data a => a -> a -> Bool
```

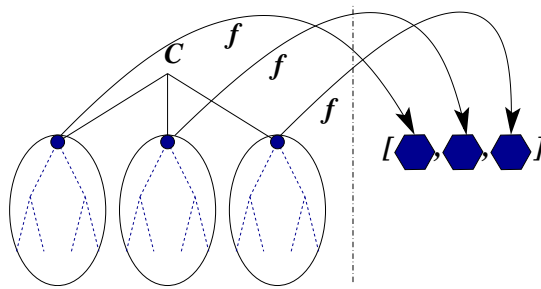
### Type erasure and validation

```
data2tree :: Data a => a -> Tree String
tree2data :: Data a => Tree String -> Maybe a
```

## Scrap show-like boilerplate

```
gshow t
= "("
++ ??? -- How to get a handle on constructor?
++ concat (intersperse " " (gmapQ gshow t))
++ ")"
```

We recurse into subterms with *gmapQ*



## Give me a little reflection

### An abstract datatype for constructors

```
data Constr -- opaque
```

### Retrieve outermost constructor from a value

```
class Typeable a ⇒ Data a where  
  ...  
  toConstr :: a → Constr
```

### Map opaque constructor to string

```
showConstr :: Constr → String
```

## Scrap show-like boilerplate

```
gshow :: Data a => a -> String
gshow t
  = "("
  ++ showConstr (toConstr t)
  ++ concat (intersperse " " (gmapQ gshow t))
  ++ ")"
```

### Possible elaborations

- Minimum parenthesisation
- Treatment of infix operators
- Special case for string
- Special case for polymorphic lists

## Scrap bitty boilerplate

```
data Bit = Zero | One
data2bits :: Data a => a -> [Bit]
data2bits
  = ??? -- How to encode the constructor?
  ++ concat (gmapQ data2bits t)
```

We need to map constructors to bits.

## Give me a little more reflection

### Map constructor to constructor index

```
constrIndex :: Constr → ConIndex
type ConIndex = Int    -- Starts at 1
```

### An abstract datatype for datatypes

```
data DataType -- opaque
```

### Retrieve information about data type

```
class Typeable a ⇒ Data a where
  ...
  dataTypeOf :: a → DataType
```

### Map *DataType* to maximum constructor index

```
maxConstrIndex :: DataType → ConIndex
```

## Scrap boilerplate for binary serialisation

```
data2bits :: Data a => a -> [Bit]
data2bits t
  = encodeCon (dataTypeOf t) (toConstr t)
  ++ concat (gmapQ data2bits t)
```

### The encoder for constructors

```
encodeCon :: DataType -> Constr -> [Bit]
encodeCon ty con = natToBin (max-1) (idx-1)
  where
    max = maxConstrIndex ty
    idx = constrIndex con

-- Return a binary representation
natToBin :: Int -> Int -> [Bit]
```

## From generic show to generic read

### A naive completion of the reflection API

```
class Typeable a ⇒ Data a where
  ...
  toConstr    :: a → Constr  -- as before
  fromConstr  :: Constr → a  -- to be replaced
```

#### Observations:

- Subterms of `(fromConstr con)` could be  $\perp$ .
- One could use `gmapM` to fill in subterms.
- This approach would require laziness.
- This approach won't work for strict datatypes.

## Building terms in a monad

```
fromConstrM :: (Monad m, Data a)
              => (forall b. Data b => m b)
              -> Constr -> m a
```

Perhaps, place *fromConstrM* in the *Data* class?

```
instance Data a => Data [a] where
  ...
  fromConstrM f con = case showConstr con of
    "[]"  -> return []
    "(:" -> do { h <- f; t <- f; return (h:t) }
```

Note: In real code, we don't pattern match on constructor strings.

## Generic read in terms of *fromConstrM*

```
gread :: Data a => String -> Maybe a
gread input = runDec input readM

readM :: Data a => DecM a
readM = read_help
  where
    read_help :: DecM a
      = do { let ty = dataTypeOf ( $\perp$ ::a)
              ; constr ← parseConstr ty
              ; fromConstrM readM constr }
```

### Notes:

- We ignore issues of parenthesisation and spaces.
- We use lexically-scoped type variables for convenience.
- `parseConstr` finds constructor by string comparison.

## The appreciated generic unfold

$gunfold\ k\ z\ con = k^a (z\ confun)$

where

$a$  is the arity of the constructor  $con$ , and

$confun$  is the constructor function corresponding to  $con$ .

## Place *gunfold* in the *Data* class!

```
instance Data a => Data [a] where
  ...
  gunfold k z con = case showConstr con of
    "[]"  -> z []
    "(:)" -> k (k (z (:)))
```

## Define *fromConstrM* in terms of *gunfold*

```
fromConstrM f = gunfold k z
  where
    k mcapp = do { capp ← mcapp; e ← f; return (capp e) }
    z = return
```

# Scrap your generative boilerplate

## Given: syntax of a little language

```
data Prog = Prog Dec Stat
data Dec  = Nodec | Ondec Id Type | Manydecs Dec Dec
data Id   = A | B
data Type = Int | Bool
data Stat = Noop | Assign Id Exp | Seq Stat Stat
data Exp  = Zero | Succ Exp
```

## Wanted: many terms

```
> genUpTo 3 :: [Prog]
[Prog Nodec Noop,Prog Nodec (Assign A Zero),
 Prog Nodec (Assign B Zero),Prog Nodec (Seq Noop
 Noop),Prog (Ondec A Int) Noop,Prog (Ondec A Int)
 (Assign A Zero),Prog (Ondec A Int) (Assign B Zero),
 Prog (Ondec A Int) (Seq Noop Noop),
 ... ]
```

## The generic generator

```
genUpTo :: ∀ a. Data a ⇒ Int → [a]
genUpTo 0 = []
genUpTo d = result
  where
    result = concat (map recurse cons)
```

## Retrieve constructors of the requested type

```
cons :: [Constr]
cons = dataTypeConstrs -- from reflection API
      $ dataTypeOf (head result)
```

## Find all terms headed by a specific Constr

```
recurse :: Data a ⇒ Constr → [a]
recurse = fromConstrM (genUpTo (d-1))
```

## Scrap your boilerplate for equality

```
geq :: Data a => a -> a -> Bool
geq x y
  = toConstr x == toConstr y
  && and (??? geq x y) -- How to descend?
```

We need to traverse two parameters simultaneously.

Again, we generalise list-processing.

Single and two-parameter traversal for lists

```
map      :: (b → c) → [b] → [c]
zipWith :: (a → b → c) → [a] → [b] → [c]
```

Likewise for data

```
gmapQ      :: Data a
            ⇒ (∀ b. Data b ⇒ b → r)
            → a → [r]

gzipWithQ  :: (Data a1, Data a2)
            ⇒ (∀ b1 b2. (Data b1, Data b2)
                ⇒ b1 → b2 → r)
            → a1 → a2 → [r]
```

Likewise for generic operations other than queries.

## Scrap your boilerplate for equality

### Very polymorphic generic equality based on zipping

```
geq' :: (Data a1, Data a2) => a1 -> a2 -> Bool
geq' x y
  = toConstr x == toConstr y
  && and (gzipWithQ geq' x y)
```

### A properly restricted generic equality

```
geq :: Data a => a -> a -> Bool
geq = geq'
```

## How to define *gzipWithQ*?

### Reconsideration: the type of generic queries

```
type GenericQ r =  $\forall$  a. Data a  $\Rightarrow$  a  $\rightarrow$  r
```

### Reconsideration: *gmapQ*'s type

```
gmapQ :: GenericQ r  $\rightarrow$  GenericQ [r]
```

### An insightful type for *gzipWithQ*

```
gzipWithQ :: GenericQ (GenericQ r)  
            $\rightarrow$  GenericQ (GenericQ [r])
```

### Types suggest the definition

1. Compute a list of generic queries from first argument.
2. Perform an accumulating fold over the second operand.

## Intermediate summary

### The *Data* class — reflection-enabled

```
class Typeable a ⇒ Data a where
  gfoldl      :: ... -- scary type omitted
  gunfold     :: ... -- scary type omitted
  dataTypeOf :: a → DataType
  toConstr    :: a → Constr
```

There are observers and constructors for `DataType` and `Constr`.  
GHC supports deriving `(Typeable, Data)`.  
Let's look into generic function extension again: *extT*, *extQ*, ...

# Generic function extension — reconsidered

## Back to paradise

```
increase :: Float → Company → Company
increase k = everywhere (id `extT` (incS k))

incS :: Float → Salary → Salary
incS k (S s) = S (s * (1+k))
```

## Extension of generic transformations

```
extT :: (Typeable a, Typeable b)
      ⇒ (a → a) → (b → b) → a → a

extT fnaa fnbb arg
  = case cast fnbb of
      Just fnaa' → fnaa' arg
      Nothing    → fnaa  arg
```

## Type-safe nominal cast

### The type says it all

```
cast :: (Typeable a, Typeable b) => a -> Maybe b
```

### Cast demo

```
ghci> (cast 'a') :: Maybe Char
Just 'a'
ghci> (cast 'a') :: Maybe Bool
Nothing
ghci> (cast True) :: Maybe Bool
Just True
```

### View cast as a primitive

```
cast x = r
  where
    r :: Maybe b =
      if typeof x == typeof (⊥ :: b)
      then Just (unsafeCoerce x) -- don't mind!!!!
      else Nothing
```

## Another example of generic function extension

### Variations on generic show

```
ghci> gshow "foo"  -- so far
"(: 'f' (: 'o' (: 'o' [])))"
ghci> gshow "foo"  -- wanted
"\foo\""
```

### A generic show with a special case for strings

```
gshow :: Data a => a -> String
gshow = gshowFullyGeneric `extQ` showString
showString :: String -> String
showString s = ... -- espace special characters
```

## Cast at its limits

### Extension of generic queries

```
extQ :: (Typeable a, Typeable b)
      => (a -> r) -> (b -> r) -> (a -> r)
extQ fnar fnbr a = case cast a of ...
```

### Extension of generic transformations

```
extT :: (Typeable a, Typeable b)
      => (a -> a) -> (b -> b) -> a -> a
extT fnaa fnbb a = case cast fnbb of ...
```

### Extension of generic, monadic transformations

```
extM :: ( Typeable (m a), ... ) -- Sigh!
      => (a -> m a) -> (b -> m b) -> a -> m a
extM fnama fnbmb a = case cast fnbmb of ...
```

Constraints long, bogus, and expensive.

# Generalised cast

## Parameterise in type constructor on typeable

```
gcast :: (Typeable a, Typeable b) => c a -> Maybe (c b)
gcast ... = ... -- implementation as before
```

## Extension of generic, monadic transformations

```
extM :: (Typeable a, Typeable b)
      => (a -> m a) -> (b -> m b) -> (a -> m a)
```

```
extM fnama fnbmb
= case gcast (M fnbmb) of
    Just (M fnama') -> fnama'
    Nothing          -> fnama
```

```
newtype M m a = M (a -> m a) -- fake type-level lambda
```

# Generalised cast at its limits

## Variations on generic show

```
ghci> gshow [True,False] -- so far
"(: True (: False []))"
ghci> gshow [True,False] -- wanted
"[True,False]"
```

## A case for polymorphic lists

```
gshow :: Data a => a -> String
gshow = gshowFullyGeneric
      `extQ` showList -- TYPE ERROR!
      `extQ` showString

showList :: Data b => [b] -> String
showList xs
= "["
++ concat (intersperse "," (map gshow xs))
++ "]"
```

## Generalising cast once more

Quantify *ext1Q* over type constructor *t* of kind “\* → \*”

```
ext1Q :: (Typeable a, Typeable1 t)
      => (a → r)
      → (∀ b. Data b => t b → r)
      → (a → r)
```

Cast with dictionary instantiation for *b*

```
ext1Q fnar fntbr
  = case dataCast1 (Q fntbr) of
      Just (Q fnar') → fnar'
      Nothing        → fnar

newtype Q r a = Q (a → r) -- fake type-level lambda
```

*dataCast*'s type is practically enforced!

```
dataCast1 :: (Typeable1 t, Data a)
          => (∀ b. Data b => c (t b))
          → Maybe (c a)
```

## Generalising cast once more (cont'd)

### At a glance

- *dataCast1* needs to go to the *Data* class.
  - *Data* instances of kind “\*” cast to *Nothing*.
  - *Data* instances of kind “\* → \*” cast via *gcast1*:

```
gcast1 :: (Typeable1 s, Typeable1 t) -- New
        => c (s a) → Maybe (c (t a))
```

```
gcast  :: (Typeable a, Typeable b) -- For comparison
        => c a → Maybe (c b)
```

- Ultimately, we want kind polymorphism.

## Conclusion

- Generic functions are first-class citizens in Haskell.
- We can traverse like the mother of traversal.
- We can serialise and de-serialise like a dream.
- We can build terms out of nothing.
- We are not afraid of multi-parameter traversal.
- We can extend generic functions in a given type.
- Static analyses and optimisations are worth an effort.
- Please go ahead and use this stuff with GHC.

Thank you for your attention.