Scrap Your Boilerplate

Kalmer Apinis

Department of Computer Science
University of Tartu

Ettevõte “genCom”

andmestruktuur

```
data Company  = C [Dept]
data Dept     = D Name Manager [SubUnit]
data SubUnit  = PU Employee | DU Dept
data Employee = E Person Salary
data Person   = P Name Address
data Salary   = S Float
type Manager  = Employee
type Name     = String
type Address  = String
```

Ettevõte “genCom”

andmed

```
genCom :: Company
genCom = C [D "Research" ralf [PU joost, PU marlow],
            D "Strategy" blair []]

ralf, joost, marlow, blair :: Employee
ralf   = E (P "Ralf" "Amsterdam")   (S 8000)
joost  = E (P "Joost" "Amsterdam")  (S 1000)
marlow = E (P "Marlow" "Cambridge") (S 2000)
blair  = E (P "Blair" "London")     (S 100000)
```

Ettevõte “genCom”

ülesannded

- Soovime teha funktsiooni, mis tõstaks kogu ettevõttes palka.

```
increase :: Float -> Company -> Company
```

- Soovime teha funktsiooni, mis muudaks konkreetse inimese aadressi andmestruktuuris.

```
moveP :: Name -> Address -> Company -> Company
```

Ettevõte “genCom”

increase1

```
increase1 k (C ds) = C (map (incD k) ds)

incD :: Float -> Dept -> Dept
incD k (D nm mgr us) =
  D nm (incE k mgr) (map (incU k) us)

incU :: Float -> SubUnit -> SubUnit
incU k (PU e) = PU (incE k e)
incU k (DU d) = DU (incD k d)

incE :: Float -> Employee -> Employee
incE k (E p s) = E p (incS k s)

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

Ettevõte “genCom”

moveP1

```
moveP1 :: Name -> Address -> Company -> Company
moveP1 p a (C ds) = C (map (movD p a) ds)

movD :: Name -> Address -> Dept -> Dept
movD n a (D nm mgr us) =
  D nm (movE n a mgr) (map (movU n a) us)

movU :: Name -> Address -> SubUnit -> SubUnit
movU n a (PU e) = PU (movE n a e)
movU n a (DU d) = DU (movD n a d)
```

Ettevõte “genCom”

moveP1

```
movE :: Name -> Address -> Employee -> Employee
movE n a (E p s) = E (movP n a p) s

movP :: Name -> Address -> Person -> Person
movP n a p@(P name address)
  | n == name  = P name a
  | otherwise  = p
```

Ettevõte “genCom”

tähelepanekud

- suurem osa koodist on *boilerplate*
  - `incD, incU, incE, movD, movU, movE`

Ettevõte “genCom”

tähelepanekud

- suurem osa koodist on *boilerplate*
  - `incD, incU, incE, movD, movU, movE`
- pole taaskasutatav
  - ```
    movD :: Name -> Address -> Dept -> Dept
    ```
    vs.
    ```
    incD :: Float -> Dept -> Dept
    ```
    ```

Ettevõte “genCom”

tähelepanekud

- suurem osa koodist on *boilerplate*
  - `incD, incU, incE, movD, movU, movE`
- pole taaskasutatav
  - ```
    movD :: Name -> Address -> Dept -> Dept
    ```
    vs.
    ```
    incD :: Float -> Dept -> Dept
    ```
- **proovime** teha taaskasutatva *boilerplate*
    ```

Halb mõte

walkC

```
walkC :: (Dept -> Dept) ->
         (SubUnit -> SubUnit) ->
         (Employee -> Employee) ->
         (Person -> Person) ->
         (Salary -> Salary) ->
         Company -> Company

increase2 k = walkC id id id id          (incS k)
moveP2 p a  = walkC id id id (movP p a) id
```

Halb mõte

tähelepanekud

- *boilerplate* on üks

Halb mõte

tähelepanekud

- *boilerplate* on üks
- väga palju parameetreid
  - üks iga alamstruktuuri kohta

Halb mõte

tähelepanekud

- *boilerplate* on üks
- väga palju parameetreid
  - üks iga alamstruktuuri kohta
- andmestruktuuri muutumisel tuleb muuta *boilerplate* definitsioone

Scrap Your Boilerplate

increase3 ja moveP3

```
everywhere :: Data a
           => (forall b. Data b => b -> b)
           -> a -> a

mkT :: (Typeable a, Typeable b)
    => (b -> b) -> a -> a
```

Scrap Your Boilerplate

increase3 ja moveP3

```
everywhere :: Data a
           => (forall b. Data b => b -> b)
           -> a -> a

mkT :: (Typeable a, Typeable b)
    => (b -> b) -> a -> a

increase3 :: Float -> Company -> Company
increase3 k = everywhere (mkT (incS k))

moveP3 :: Name -> Address -> Company -> Company
moveP3 p a = everywhere (mkT (movP p a))
```

Type extension

cast

```
class Typeable

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

proovime

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

Type extension

mkT

```
mkT :: (Typeable a, Typeable b)
    => (b -> b) -> a -> a

mkT f = case cast f of
  Just g -> g
  Nothing -> id
```

proovime

```
Prelude> (mkT not) True
False
Prelude> (mkT not) 'a'
'a'
```

Type extension

extT

```
extT :: (Typeable a, Typeable b)
     => (a -> a) -> (b -> b) -> a -> a

(q `extT` f) a = case cast a of
  Just b -> f b
  Nothing -> q a
```

mitmest funktsioonist koosnev

```
incAllMoveP k p a =
  everywhere (mkT (incS k) `extT` (movP p a))
```

everywhere

ühe taseme läbimine

```
-- rakendame funktsiooni alamosadele
class Typeable a => Data a where
  gmapT :: (forall b. Data b => b -> b) -> a -> a

instance Data Employee where
  gmapT f (E per sal) = E (f per) (f sal)

instance Data Bool where
  gmapT f x = x

instance Data a => Data [a] where
  gmapT f [] = []
  gmapT f (x:xs) = f x : f xs
```

everywhere

rekursiivne läbimine

```
-- rakendame muutmist alt-üles
everywhere :: Data a
           => (forall b. Data b => b -> b)
           -> a -> a

everywhere f x = f (gmapT (everywhere f) x)
```

Päringud

queries

Seni vaatasime transformereid kujul:

```
forall a. Data a => a -> a
```

Nüüd vaatame päringuid:

```
forall a. Data a => a -> R
```

Päringud

näide

```
salaryBill :: Company -> Float
salaryBill = everything (+) (0 `mkQ` billS)

billS :: Salary -> Float
billS (S f) = f
```

proovime

```
*Main> salaryBill genCom
111000.0
```

Päringud

mkQ

```
mkQ :: (Typeable a, Typeable b)
    => r -> (b -> r) -> a -> r
(r `mkQ` q) a = case cast a of
  Just b  -> q b
  Nothing -> r
```

proovime

```
Prelude> (22 `mkQ` ord) 'a'
97
Prelude> (22 `mkQ` ord) 'b'
98
Prelude> (22 `mkQ` ord) True
22
```

Päringud

mitme funktsiooni tõstmiseks extQ

```
extQ :: (Typeable a, Typeable b)
     => (a -> r) -> (b -> r) -> (a -> r)

(q `extQ` f) a = case cast a of
  Just b -> f b
  Nothing -> q a
```

Päringud

Data

```
class Typeable a => Data a where
  gmapT :: (forall b. Data b => b -> b) -> a -> a
  gmapQ :: (forall b. Data b => b -> r) -> a -> [r]

instance Data Employee where
  gmapT = ... nagu enne ...
  gmapQ f (E p s) = [f p, f s]

instance Data a => Data [a] where
  gmapT = ... nagu enne ...
  gmapQ f [] = []
  gmapQ f (x:xs) = [f x, f xs]
```

Päringud

Data

```
instance Data Bool where
  gmapT x = ... nagu enne ...
  gmapQ x = []

everything :: Data a =>
  (r -> r -> r) -> (forall a. Data a => a -> r)
  -> a -> r

everything k f x =
  foldl k (f x) (gmapQ (everything k f) x)
```

Päringud

Näide

```
find :: Name -> Company -> Maybe Dept
find n = everything orElse (Nothing `mkQ` findD n)

findD :: String -> Dept -> Maybe Dept
findD n d@(D n' _ _) | n == n' = Just d
                     | otherwise = Nothing

orElse :: Maybe a -> Maybe a -> Maybe a
x `orElse` y = case x of
  Just _ -> x
  Nothing -> y
```

Päringud

Näide

```
find :: Name -> Company -> Maybe Dept
find n = everything orElse (Nothing `mkQ` findD n)

findD :: String -> Dept -> Maybe Dept
findD n d@(D n' _ _) | n == n' = Just d
                     | otherwise = Nothing

orElse :: Maybe a -> Maybe a -> Maybe a
x `orElse` y = case x of
  Just _ -> x
  Nothing -> y
```

NB!

- tänu Haskell'i laiskusele ei otsita peale leidmist enam edasi

Tüübid

Sama mis enne?

```
everywhere :: (forall b. Data b => b -> b)
           -> (forall a. Data a => a -> a)

type GenericT = forall a. Data a => a -> a

everywhere :: GenericT -> GenericT

type GenericQ r = forall a. Data a => a -> r

everything :: (r -> r -> r)
           -> GenericQ r -> GenericQ r
```

Varakult lõpetamine

everywhereBut

```
everywhereBut :: GenericQ Bool
              -> GenericT -> GenericT
everywhereBut q f x
  | q x = x
  | otherwise = f (gmapT (everywhereBut q f) x)

increase k = everywhereBut names (mkT (incS k))

names :: GenericQ Bool
names = False `mkQ` isName

isName :: String -> Bool
isName n = True
```

Monaadilised transformerid

mkM

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

mkM f = case cast f of
  Just g -> g
  Nothing -> return
```

Monaadilised transformerid

gmapM ja everywhereM

```
class Typeable a => Data a where
  gmapM :: Monad m
        => (forall b. Data b => b -> m b)
        -> a -> m a

everywhereM :: (Monad m, Data a)
            => (forall b. Data b => b -> m b)
            -> a -> m a

everywhereM f x =
  do x' <- gmapM (everywhereM f) x
     f x'
```

Monaadilised transformerid

abstraktne näide

```
lookupSalaries :: Company -> IO Company
lookupSalaries = everywhereM (mkM lookupE)

lookupE :: Employee -> IO Employee
lookupE (E p@(P n _) _) =
  do { s <- dbLookup n; return (E p s) }

dbLookup :: Name -> IO Salary
-- andmebaasist andmete otsimine
```

edasi?

- Kuidas on võrreldavad `gmapT` ja `gmapM`?
- Kas vajame `gmapMQ`?

Typeable

üks võimalik typeable klassi lahendus

```
class Typeable a where
  typeOf :: a -> TypeRep

data TypeRep = TR String [TypeRep]

instance Typeable Int where
  typeOf x = TR "Prelude.Int" []
instance Typeable Bool where
  typeOf x = TR "Prelude.Bool" []
instance Typeable a => Typeable [a] where
  typeOf x = TR "Prelude.List" [typeOf (get x)]
    where
      get :: [a] -> a
      get = undefined
```

cast

cast implementatsioon

```
cast :: (Typeable a, Typeable b) => a -> Maybe b
cast x = r
  where
    r = if typeOf x == typeOf (get r)
      then Just (unsafeCoerce x)
      else Nothing

get :: Maybe a -> a
get x = undefined

unsafeCoerce :: a -> b
```

Kokkuvõte

programmi osad

- Programmeerija-kirjutatud osa
  - lühike teisendusfunktsioon (nt. `incS`)
  - teisendusfunktsioonide tõstmine suuremale andmestruktuurile (kasutades `mkT, extQ`)
- Mehaaniliselt-genereeritud osa
  - omadefineeritud andmestruktuuride `Typeable` ja `Data` instantsid
- Teegi osa
  - väike hulk kombinaatoreid (nt. `mkT, everywhere, everything jne`) mida saab ise laiendada

- GHC >= 6, "-fglasgow-exts"
- osa ghc nimeruumist (`Data.Generics`)
- ghc kompilaator toetab `deriving (Typeable, Data)`