Avoiding quadratic GHC core code size
Introducing the large-records library
Edsko de Vries and Andres Löh
Haskell Implementors’ Workshop 2021

Work done on behalf of Juspay.
module Before where

data R = MkR {
    field1 :: T 1
    , field2 :: T 2
    -- ...
    , fieldN :: T N
}  
deriving (Show, Eq)

deriveGeneric "'R -- SOP generics"

instance ToJSON R where
toJSON = gtoJSON defaultJsonOptions
module After where

-- ..

largeRecord defaultLazyOptions [d|
  data  R = MkR {|
    field1 :: T 1
    , field2 :: T 2
    -- ...
    , fieldN :: T N
  }
  deriving (Show, Eq)
  |

instance ToJSON R where
  toJSON = gtoJSON
AST size (sum of terms/types/coercions) versus number of record fields, after desugaring.
AST size (sum of terms/types/coercions) versus number of record fields, after the simplifier.
Motivation

OS reported maximum resident set size in KB versus number of record fields. Mean and standard error over 100 compilations (no linking), normalized to a baseline of an empty module.
OS reported elapsed real time (wall clock) in seconds versus number of record fields.
Mean and standard error over 100 compilations (no linking), normalized to a baseline of an empty module.
Sources of quadratic core size

- No way to access/update a record without mentioning every field of the record
- No way to introduce and control sharing at the type level
data R = MkR { 
  f00 :: T 00,
  f01 :: T 01,
  f02 :: T 02
  -- .. lots more ..
  , f98 :: T 98
  , f99 :: T 99
  }

Sources of quadratic core size: Records

data R = MkR {
    f00 :: T 00
    , f01 :: T 01
    , f02 :: T 02
    -- .. lots more ..
    , f98 :: T 98
    , f99 :: T 99
}

f00 :: R -> T 0
f00 = \(r :: R) ->
  \textbf{case} \ r \ \textbf{of}
    \textbf{MkR} \ x00 \ x01 \ x02 \ x03 \ x04 \ x05 \ x06 \ x07 \ x08 \ x09
      x10 \ x11 \ x12 \ x13 \ x14 \ x15 \ x16 \ x17 \ x18 \ x19
    -- .. \textit{lots more} ..
      x90 \ x91 \ x92 \ x93 \ x94 \ x95 \ x96 \ x97 \ x98 \ x99 \ ->
    x00
instance HasField "f00" R (T 00) where
    hasField r = (\x -> r { f00 = x }, f00 r)
instance HasField "f00" R (T 00) where
  hasField r = (\x -> r { f00 = x }, f00 r)

hasField_f00 :: R -> (T 0 -> R, T 0)
hasField_f00 r = (\new -> case r of
  MkR x00 x01 x02 x03 x04 x05 x06 x07 x08 x09
  x10 x11 x12 x13 x14 x15 x16 x17 x18 x19
  -- .. lots more ..
  x90 x91 x92 x93 x94 x95 x96 x97 x98 x99
  ->
  MkR new x01 x02 x03 x04 x05 x06 x07 x08 x09
  x10 x11 x12 x13 x14 x15 x16 x17 x18 x19
  -- .. lots more ..
  x90 x91 x92 x93 x94 x95 x96 x97 x98 x99,
  case r of
  MkR x00 x01 x02 x03 x04 x05 x06 x07 x08 x09
  x10 x11 x12 x13 x14 x15 x16 x17 x18 x19
  -- .. lots more ..
  x90 x91 x92 x93 x94 x95 x96 x97 x98 x99
  ->
x00
Sources of quadratic core size: Records

class (  
  c (T 00)  
, c (T 01)  
, c (T 02)  
  -- .. lots more ..  
, c (T 98)  
, c (T 99)  
) => Constraints_R c
Sources of quadratic core size: Records

class (  
    c (T 00)  
    , c (T 01)  
    , c (T 02)  
    -- .. lots more ..  
    , c (T 98)  
    , c (T 99)  
  ) => Constraints_R c

$p1Constraints_R :: Constraints_R c => c (T 0)  
$p1Constraints_R = \{dict ->
  case dict of
    Constraints_R d00 d01 d02 d03 d04 d05 d06 d07 d08 d09  
    d10 d11 d12 d13 d14 d15 d16 d17 d18 d19  
    -- .. lots more ..  
    d90 d91 d92 d93 d94 d95 d96 d97 d98 d99 ->
    d00

Well-Typed
zipMyRecordWith ::
  Applicative f
  => (forall n. T n -> T n -> f (T n))
  -> R -> R -> f R
zipMyRecordWith f r r' =
  pure MkR
  <$> f (f00 r) (f00 r')
  <$> f (f01 r) (f01 r')
  <$> f (f02 r) (f02 r')
  -- .. lots more ..
  <$> f (f98 r) (f98 r')
  <$> f (f99 r) (f99 r')
Sources of quadratic core size: Type arguments

zipMyRecordWith ::
  Applicative f
  => (forall n. T n -> T n -> f (T n))
  -> R -> R -> f R
zipMyRecordWith f r r' =
  pure MkR
  <*> @(T 00) @(T 01 -> T 02 -> T 03 -> .. -> T 99 -> R) f (f00 r) (f00 r')
  <*> @(T 01) @(T 02 -> T 03 -> .. -> T 99 -> R) f (f01 r) (f01 r')
  <*> @(T 02) @(T 03 -> .. -> T 99 -> R) f (f02 r) (f02 r')
  -- .. lots more ..
  <*> @(T 98) @(T 99 -> R) f (f98 r) (f98 r')
  <*> @(T 99) @(R) f (f99 r) (f99 r')
Sources of quadratic core size: Type arguments

```
data NP :: (k -> Type) -> [k] -> Type where
    Nil :: forall f. NP f '[]
    (:*) :: forall f x xs. f x -> NP f xs -> NP f (x ': xs)
```
Sources of quadratic core size: Type arguments

\textbf{data} \ NP :: (k -> Type) \to [k] \to Type \textbf{where}
\begin{align*}
\text{Nil} & : \textbf{forall} \ f. \ NP \ f \ '[] \\
(:\star) & : \textbf{forall} \ f \ x \ xs. \ f \ x \to NP \ f \ xs \to NP \ f \ (x \ ': xs)
\end{align*}

npFromR :: R \to NP \ T \ IndicesR
npFromR MkR{..} = (f00 :* f01 :* f02 :* f03 :* f04 :* f05 :* f06 :* f07 :* f08 :* f09 :* f10 :* f11 :* f12 :* f13 :* f14 :* f15 :* f16 :* f17 :* f18 :* f19 -- .. lots more .. :* f90 :* f91 :* f92 :* f93 :* f94 :* f95 :* f96 :* f97 :* f98 :* f99 :* Nil)

\textbf{Well-Typed}
Sources of quadratic core size: Type arguments

\[
\text{npFromR MkR}\{..\} =
\begin{align*}
& (:\ast) \@00 \@'\langle 1, 2, .., 98, 99 \rangle f00 ( \\
& (:\ast) \@01 \@'\langle 2, .., 98, 99 \rangle f01 ( \\
& (:\ast) \@02 \@'\langle .., 98, 99 \rangle f02 ( \\
& \quad \quad \text{-- .. lots more ..} \\
& (:\ast) \@98 \@'\langle 99 \rangle f98 ( \\
& (:\ast) \@99 \@'\langle \rangle f99 ( \\
& \text{Nil })))))
\end{align*}
\]
Sources of quadratic core size: Type arguments

```haskell
data SList :: [k] -> Type where
    SNil :: SList '[]
    SCons :: SList xs -> SList (x ': xs)

class SListI (xs :: [k]) where
    sList :: SList xs

instance SListI '[] where
    sList = SNil

instance SListI xs => SListI (x ': xs) where
    sList = SCons sList
```

Well-Typed
Sources of quadratic core size: Type arguments

$dSListl_99 :: SList '[99]
$dSListl_98 :: SList '[98, 99]
$dSListl_97 :: SList '[97, 98, 99]
...

$dSListl_99 = SCons @99 '@'[\] ..
$dSListl_98 = SCons @98 '@'[99] ..
$dSListl_97 = SCons @97 '@'[98, 99] ..
...
type family InterpretTo d xs ys :: Constraint where
    InterpretTo _ '[]' '[]' = ()
    InterpretTo d ('(f, x) ': xs) ('(f, y) ': ys) = (Coercible x (Interpreted d y), InterpretTo d xs ys)
Sources of quadratic core size: Type arguments

```haskell
-- type family InterpretTo d xs ys :: Constraint where
  InterpretTo _ '[]' '[]' = ()
  InterpretTo d ('(f, x) ': xs) ('(f, y) ': ys) = (Coercible x (Interpreted d y), InterpretTo d xs ys)

evidence1 = (,) @X @() ...
evidence2 = (,) @X @(X, ()) ...
evidence3 = (,) @X @(X, (X, ())) ...
```
large-records

largeRecord defaultLazyOptions [d]

```haskell
data R = MkR {
    field1 :: T 1,
    field2 :: T 2
    -- ...
    , fieldN :: T N
}
deriving (Show, Eq)
```

|]
large-record defaultLazyOptions [d]

```haskell
data R = MkR {
    field1 :: T 1
, field2 :: T 2
-- ...
, fieldN :: T N
}

deriving (Show, Eq)
```

```haskell
newtype R = RFromVector {vectorFromR :: Vector Any}
```
large-records: Records

unsafeGetIndexR :: Int -> R -> x
unsafeGetIndexR n (RFromVector r) =
    unsafeCoerce $ unsafeIndex r n

unsafeSetIndexR :: Int -> R -> x -> R
unsafeSetIndexR n r x = RFromVector $
    unsafeUpd (vectorFromR r) [(n, unsafeCoerce x)]
large-records: Records

unsafeGetIndexR :: Int -> R -> x
unsafeGetIndexR n (RFromVector r) =
   unsafeCoerce $ unsafeIndex r n

unsafeSetIndexR :: Int -> R -> x -> R
unsafeSetIndexR n r x = RFromVector $ un
   unsafeUpd (vectorFromR r) [(n, unsafeCoerce x)]

f00 :: R -> T 0
f00 = unsafeGetIndexR 0

instance HasField "f00" R (T 0) where
   hasField r = ( unsafeSetIndexR 0 r
                   , unsafeGetIndexR 0 r
                   )
large-records: Records

unsafeGetIndexR :: Int -> R -> x
unsafeGetIndexR n (RFromVector r) =
  unsafeCoerce $ unsafeIndex r n

unsafeSetIndexR :: Int -> R -> x -> R
unsafeSetIndexR n r x = RFromVector $ unsafeUpd (vectorFromR r) [(n, unsafeCoerce x)]

f00 :: R -> T 0
f00 = unsafeGetIndexR 0

instance HasField "f00" R (T 0) where
  hasField r = ( unsafeSetIndexR 0 r
                , unsafeGetIndexR 0 r
                )

(Pattern synonym awaiting NoFieldSelectors, currently using quasi-quoter.)
newtype Rep f a = Rep (Vector (f Any))

class Constraints_R c where
dictConstraints_R :: Proxy c -> Rep (Dict c) R

instance (c (T 0), c (T 1), c (T 2) {- .. -})
) => Constraints_R c where
dictConstraints_R p = Rep $ fromList [
  unsafeCoerce (dictFor p) (Proxy @(T 0))
  , unsafeCoerce (dictFor p) (Proxy @(T 1))
  -- .. lots more ..
  , unsafeCoerce (dictFor p) (Proxy @(T 99))
]
newtype Rep f a = Rep (Vector (f Any))

class Constraints_R c where
dictConstraints_R :: Proxy c -> Rep (Dict c) R

instance (c (T 0), c (T 1), c (T 2) {- .. -})
                     => Constraints_R c where
dictConstraints_R p = Rep $ fromList [  
  unsafeCoerce (dictFor p) (Proxy @(T 0))
  , unsafeCoerce (dictFor p) (Proxy @(T 1))
  -- .. lots more ..
  , unsafeCoerce (dictFor p) (Proxy @(T 99))
  ]

Note: the $a$ in Rep $f$ $a$ is phantom; we are avoiding type-level lists nearly everywhere.
large-records: Transforms

\[
\text{normalize1} :: \forall d \ f \ x. \\
\text{HasNormalForm} (d \ f) (x \ f) (x \text{ Uninterpreted}) \\
\Rightarrow \text{Proxy } d \\
\Rightarrow \text{Rep } I (x \ f) \\
\Rightarrow \text{Rep } (\text{Interpret } (d \ f)) (x \text{ Uninterpreted})
\]

\text{normalize1} _ = \text{unsafeCoerce}
large-records: Transforms

normalize1 :: forall d f x.
    HasNormalForm (d f) (x f) (x Uninterpreted)
    => Proxy d
    -> Rep I (x f)
    -> Rep (Interpret (d f)) (x Uninterpreted)
normalize1 _ = unsafeCoerce

type HasNormalForm d x y =
    InterpretTo d (MetadataOf x) (MetadataOf y)

type family InterpretTo d xs ys :: Constraint where
    InterpretTo _ '[]' '[]' = ()
    InterpretTo d ('(f, x)' ': xs) ('(f, y)' ': ys) = (Coercible x (Interpreted d y), InterpretTo d xs ys)
type family InterpretTo d xs ys :: Constraint where
  InterpretTo _ '[]' '[]' = ()
  InterpretTo d '(f, x) ': xs '(f, y) ': ys =
    IfEqual x (Interpreted d y)
    (InterpretTo d xs ys)

type family IfEqual x y (r :: k) :: k where
  IfEqual actual actual actual k = k
Conclusions

- Avoiding quadratic core code size surprisingly difficult
- Large-records provides support for records with accessors, lenses and generics that is guaranteed to be linear, but at the cost of (internal) type safety.
- No support for unboxed fields.
- Ideally would solve this in ghc itself
  - Need a way to access and update records/dictionaries
  - Need a way to introduce (and control) sharing at the type level.
- Large-records generics is based on *True Sums of Products*, Edsko de Vries and Andres Löh, WGP 2014.
- Related work: *Scrap Your Type Applications*, Barry Jay and Simon Peyton Jones, MPC 2008