Type-Level Literals

• Like ordinary (non-overloaded) literals, but at the type-level.
• Type level natural numbers:
  0, 1, 2, 3, ... :: Nat
• Type level symbols:
  "hello", "some label", ... :: Symbol
Type-level literals are enabled by the DataKinds extension.

```
-- part 1
{-# LANGUAGE DataKinds, KindSignatures #-}

-- part 2
{-# LANGUAGE TypeOperators, GADTs #-}

-- part 3
{-# LANGUAGE PolyKinds, MultiParamTypeClasses #-
{-# LANGUAGE FlexibleInstances, FlexibleContexts #-
{-# LANGUAGE UndecidableInstances #-}
```
module Array where

The module GHC.TypeLits provides useful functions for working with singleton types:

import GHC.TypeLits
import Foreign
import Control.Monad
An array is a pointer with the invariant that it points to the required number of elements:

```haskell
newtype Array (size :: Nat) a = A (Ptr a)
```

An index into an array is an integer with the invariant that it is in the range \([ 0 .. size - 1 ]\):

```haskell
newtype Ix (size :: Nat) = I Int
deriving Show
```
Given values of the correct types, we can work with arrays without bounds checking:

```
arrayElem :: Storable a => Array n a -> Ix n -> Ptr a
arrayElem (A arr) (I i) = advancePtr arr i

arrayPeek a i = peek (arrayElem a i)
arrayPoke a i v = poke (arrayElem a i) v
```
Singleton types connect type-level literals with run-time values.

\[ \text{Sing} :: \text{Nat} \rightarrow * \]

Each type has only one interesting value, called \text{sing}:

\[ \text{sing} :: \text{Sing} 0 \]
\[ \text{sing} :: \text{Sing} 4096 \]

The name \text{sing} is overloaded for each literal (class \text{SingI}).
Example: Array Size

arraySize :: SingI n => Array n a -> Sing n
arraySize _ = sing
From Singletons to Numbers

To access the run-time value for a singleton, use:

\[
\text{fromSing} :: \text{Sing } n \rightarrow \text{ Integer}
\]

Examples:

\[
\begin{align*}
\text{fromSing} (\text{sing} :: \text{Sing 0}) &= 0 \\
\text{fromSing} (\text{sing} :: \text{Sing 4096}) &= 4096
\end{align*}
\]

We can convert to other numeric types too:

\[
\text{singToNum} :: \text{Num } a \Rightarrow \text{ Sing } (n :: \text{ Nat}) \rightarrow a
\]

\[
\text{singToNum} = \text{fromInteger} \cdot \text{fromSing}
\]
Example: Creating New Arrays

-- With explicit size parameter.
arrayNew' :: Storable a => Sing n -> IO (Array n a)
arrayNew' size = A 'fmap' mallocArray (singToNum size)
Example: Creating New Arrays

-- With explicit size parameter.

arrayNew' :: Storable a => Sing n -> IO (Array n a)
arrayNew' size = A `fmap` mallocArray (singToNum size)

-- Automatically inferred size.

arrayNew :: (Storable a, SingI n) => IO (Array n a)
arrayNew = withSing arrayNew'

{-
withSing :: SingI n => (Sing n -> b) -> b
withSing f = f sing
-}
-- Dynamic check

index' :: Sing n -> Int -> Maybe (Ix n)
index' size n = do guard (0 <= n && n < singToNum size)
  return (I n)


index :: SingI n => Int -> Maybe (Ix n)
index = withSing index’
Example: Creating Index Values

-- Dynamic check
index' :: Sing n -> Int -> Maybe (Ix n)
index' size n = do guard (0 <= n && n < singToNum size)
                   return (I n)

index :: SingI n => Int -> Maybe (Ix n)
index = withSing index'

-- Access all elements
indexes :: SingI n => Array n a -> [Ix n]
indexes arr = [ I i | i <- [ 0 .. size - 1 ] ]
              where size = singToNum (arraySize arr)
Example: Putting It All Together

arrayDump :: (SingI n, Storable a, Show a) => Array n a -> IO ()
arrayDump arr = mapM_ (print <=< arrayPeek arr)
(indexes arr)

dump arr = do
  print arr
  arrayDump arr
  return arr
Example: Putting It All Together

arrayDump :: (SingI n, Storable a, Show a) => Array n a -> IO ()
arrayDump arr = mapM_ (print <=< arrayPeek arr)
  (indexes arr)

example :: IO (Array 12 Char)
example = do arr <- arrayNew
  mapM_ (\i -> arrayPoke arr i 'a')
  (indexes arr)
  arrayDump arr
  return arr
newtype Sing (n :: Nat) = SNat Integer

fromSing (SNat n) = n

class SingI n where sing :: Sing n

-- Built-into GHC
instance SingI 0 where sing = SNat 0
instance SingI 1 where sing = SNat 1
instance SingI 2 where sing = SNat 2
...
The Real Types

The types for working with singletons are more general: they also support type-level symbols and other custom singletons.

data family Sing n

newtype instance Sing (n :: Nat) = SNat Integer
newtype instance Sing (n :: Symbol) = SNat String
...

The operations are also overloaded for other kinds too:

fromSing (sing :: Sing "Hello") == "Hello"
Computation With Type Naturals

- Introduced via special type families/predicates:
  
  \[
  (+), (*), (^) :: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \\
  (\leq) :: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Constraint}
  \]

- No user-defined instances, custom solver in GHC.

- Work in progress:
  
  - Works well when working with known constants
  - Currently improving support for abstract reasoning.
Dynamic Checks with Improved Types

\[
singAdd :: Sing a \rightarrow Sing b \rightarrow Sing (a + b)
\]
\[
singAdd x y = \text{case } isZero x \text{ of}
\]
\[
\text{IsZero} \rightarrow y
\]
\[
\text{IsSucc } n \rightarrow \text{singSucc } (\text{singAdd } n y)
\]

\{\text{- data IsZero :: Nat } \rightarrow* \}

\text{where}
\[
\text{IsZero} :: \text{IsZero } 0
\]
\[
\text{IsSucc} :: \!(\text{Sing } n) \rightarrow \text{IsZero } (n + 1) \}
\]
Dynamic Checks with Improved Types

\[
singAdd :: Sing a \rightarrow Sing b \rightarrow Sing (a + b)
\]
\[
singAdd x y = \text{case isZero } x \text{ of}
\]
\[
\quad \text{IsZero} \rightarrow y
\]
\[
\quad \text{IsSucc } n \rightarrow \text{singSucc } (\text{singAdd } n \ y)
\]

\[
\{- \text{ data IsZero :: Nat } \rightarrow \ast
\]
\[
\quad \text{where}
\]
\[
\quad \text{IsZero} :: \text{IsZero } 0
\]
\[
\quad \text{IsSucc} :: !(\text{Sing } n) \rightarrow \text{IsZero } (n + 1) -\}
\]

\[
singSucc :: Sing a \rightarrow Sing (a + 1)
\]
\[
singSucc x = \text{unsafeSingNat } (\text{fromSing } x + 1)\]
Some Remaining Issues

- Lazy vs. Strict type-function evaluation
  - GHC preserves type synonyms
    
    \[
    \text{add} \ (S :: S \, 1) \ (S :: S \, 2) :: S \ \text{Nat} \ (1 + 2)
    \]

- Literals and class instances
  
  \[
  \text{instance} \ C \ a \quad \text{-- ok}
  \]
  \[
  \text{instance} \ C \ 1 \quad \text{-- ok}
  \]
  \[
  \text{instance} \ C \ (n + 1) \quad \text{-- not ok}
  \]

- Nicer notations for writing singletons?
  
  - Avoid \text{sing} :: \text{Sing} 3
Alternative Design for Value Literals

class FromLiteral n a where
  fromLiteral :: Sing n -> a

-- Overloaded numbers
instance FromLiteral (n :: Nat) Integer where
  fromLiteral x = fromSing x

-- Overloaded strings
instance FromLiteral (s :: Symbol) String where
  fromLiteral x = fromSing x

-- Restricted literals
instance (n <= 255) => FromLiteral (n :: Nat) Word8 where
  fromLiteral x = fromInteger (fromSing x)
class FromLiteral n a where
  fromLiteral :: Sing n -> a

  -- Overloaded numbers
instance FromLiteral (n :: Nat) Integer where
  fromLiteral x = fromSing x

  -- Overloaded strings
instance FromLiteral (s :: Symbol) String where
  fromLiteral x = fromSing x
class FromLiteral n a where
    fromLiteral :: Sing n -> a

-- Overloaded numbers
instance FromLiteral (n :: Nat) Integer where
    fromLiteral x = fromSing x

-- Overloaded strings
instance FromLiteral (s :: Symbol) String where
    fromLiteral x = fromSing x

-- Restricted literals
instance (n <= 255) => FromLiteral (n :: Nat) Word8 where
    fromLiteral x = fromInteger (fromSing x)
• Type level naturals and symbols are in GHC 7.6

• Computation with type-level available on branch type-nats

• Please try it out and send me feedback!