Implementing Type-Level Literals

Iavor S. Diatchki

September 14, 2012

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 and Various Extensions

Type-level literals are enabled by the `DataKinds` extension.

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

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

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

Useful Modules

```
module Array where
```

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

```
import GHC.TypeLits
import Foreign
import Control.Monad
```

Example: Using Type-Level Naturals for Arrays

An array is a pointer with the invariant that it points to the required number of elements:

```
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 ]`:

```
newtype Ix (size :: Nat)      = I Int
                                deriving Show
```

Example: Accessing Array Elements

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 (specialized to type naturals)

Singleton types connect type-level literals with run-time values.

```
Sing :: Nat -> *
```

Each type has only one interesting value, called `sing`:

```
sing :: Sing 0
sing :: Sing 4096
```

The name `sing` is overloaded for each literal (class `SingI`).

Example: Array Size

```haskell
arraySize :: SingI n => Array n a -> Sing n
arraySize _ = sing
```

From Singletons to Numbers

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

```
fromSing :: Sing n -> Integer
```

Examples:

```
fromSing (sing :: Sing 0)    == 0
fromSing (sing :: Sing 4096) == 4096
```

We can convert to other numeric types too:

```
singToNum :: Num a => Sing (n :: Nat) -> a
singToNum = fromInteger . fromSing
```

Example: Creating New Arrays

```haskell
-- 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
-}
```

Example: Creating Index Values

```haskell
-- 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

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

Example: Putting It All Together

```haskell
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
```

Behind The Scences

```
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:

```
(+), (*), (^) :: Nat -> Nat -> Nat
(<=)          :: Nat -> Nat -> Constraint
```

- No user-defined instaces, 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 -> Sing b -> Sing (a + b)
singAdd x y = case isZero x of
                IsZero   -> y
                IsSucc n -> singSucc (singAdd n y)

{- data IsZero :: Nat -> *
     where
     IsZero :: IsZero 0
     IsSucc :: !(Sing n) -> IsZero (n + 1) -}
```

Dynamic Checks with Improved Types

```
singAdd :: Sing a -> Sing b -> Sing (a + b)
singAdd x y = case isZero x of
                IsZero   -> y
                IsSucc n -> singSucc (singAdd n y)

{- data IsZero :: Nat -> *
     where
     IsZero :: IsZero 0
     IsSucc :: !(Sing n) -> IsZero (n + 1) -}

singSucc :: Sing a -> Sing (a + 1)
singSucc x = unsafeSingNat (fromSing x + 1)
```

Some Remaining Issues

- Lazy vs. Strict type-function evaluation
  - GHC preserves type syninyms

    `add (S :: S 1) (S :: S 2) :: S Nat (1 + 2)`
- Literals and class instances

```
instance C a        -- ok
instance C 1        -- ok
instacne C (n + 1)  -- not ok
```

- Nicer notations for writing singletons?
  - Avoid `sing :: Sing 3`

Alternative Design for Value Literals

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

Alternative Design for Value Literals

```haskell
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
```

Alternative Design for Value Literals

```haskell
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)
```

Status

- Type level naturals and symols are in GHC 7.6
- Coputation with type-level available on branch `type-nats`
- Please try it out and send me feedback!