Dependently Typed Programming with Singletons

Richard Eisenberg Stephanie Weirich
The University of Pennsylvania

13 September 2012
Haskell Symposium
Copenhagen, Denmark

Outline

- Introduction to singletons
- How singletons are used to simulate dependently typed programming
- An explanation of the singletons library that automates generation of code working with singletons
- Brief survey of issues confronting a programmer using singletons

Length-indexed Vectors

```
data Nat = Zero | Succ Nat

data Vec :: * → Nat → * where
  VNil :: Vec a ‘Zero
  VCons :: a → Vec a n → Vec a (‘Succ n)
```

Length-indexed Vectors

```
data Nat = Zero | Succ Nat

data Vec :: * → Nat → * where
  VNil :: Vec a 'Zero
  VCons :: a → Vec a n → Vec a ('Succ n)
```

- promoted datatype

Length-indexed Vectors

```
data Nat = Zero | Succ Nat

data Vec :: * → Nat → * where
  VNil :: Vec a 'Zero
  VCons :: a → Vec a n → Vec a ('Succ n)
```

- promoted datatype
- kind Nat contains 'Zero and ('Succ n), where n is of kind Nat

Length-indexed Vectors

```
data Nat = Zero | Succ Nat

data Vec :: * → Nat → * where
  VNil :: Vec a 'Zero
  VCons :: a → Vec a n → Vec a ('Succ n)
```

- promoted datatype
- kind Nat contains 'Zero and ('Succ n), where n is of kind Nat
- 'Zero and ('Succ n) contain no terms

makeEven

The function makeEven takes a vector of any length, along with that vector's length, and returns one of even length, perhaps by repeating the first element.

What is makeEven's type?

makeEven

The function makeEven takes a vector of any length, along with that vector's length, and returns one of even length, perhaps by repeating the first element.

What is makeEven's type?

makeEven :: Nat → Vec a n → Vec a ??

makeEven

The function makeEven takes a vector of any length, along with that vector's length, and returns one of even length, perhaps by repeating the first element.

What is makeEven's type?

~~makeEven :: Nat → Vec a n → Vec a ??~~

makeEven :: Nat → Vec a n → Vec a (NextEven n)

makeEven

The function makeEven takes a vector of any length, along with that vector's length, and returns one of even length, perhaps by repeating the first element.

What is makeEven's type?

~~makeEven :: Nat → Vec a n → Vec a ??~~

~~makeEven :: Nat → Vec a n → Vec a (NextEven n)~~

makeEven :: (n : Nat) → Vec a n → Vec a (NextEven n)

Singleton Types

A singleton type is a member of a family of types, each of which has only one value.

The value of a singleton is isomorphic to the type.

```
data SNat :: Nat → * where
  SZero :: SNat ‘Zero
  SSucc :: SNat n → SNat (‘Succ n)

two :: SNat (‘Succ (‘Succ ‘Zero))
two = SSucc (SSucc SZero)
```

Related Work

Xi & Pfenning (PLDI ’98): Use of singletons to simulate dependent types

Monnier & Haguenauer (PLPV ’10): Proof that singletons are as expressive as dependent types

McBride’s SHE (2009): Preprocessor that generates singleton types

The singletons library: Works with promoted datatypes and generates singleton functions

Related Work

Xi & Pfenning (PLDI '98): Use of singletons to simulate dependent types

Monnier & Haguenauer (PLPV '10): Proof that singletons are as expressive as dependent types

McBride's SHE (2009): Preprocessor that generates singleton types

The singletons library: Works with promoted datatypes and generates singleton functions

SHE can't do that

makeEven

```
makeEven :: (n : Nat) → Vec a n → Vec a (NextEven n)
```

makeEven

~~makeEven :: (n : Nat) → Vec a n → Vec a (NextEven n)~~

makeEven :: SNat n → Vec a n → Vec a (NextEven n)

makeEven

~~makeEven :: (n : Nat) → Vec a n → Vec a (NextEven n)~~

makeEven :: SNat n → Vec a n → Vec a (NextEven n)

makeEven

```
makeEven :: SNat n → Vec a n → Vec a (NextEven n)
makeEven n v =
```

makeEven

```
isEven :: Nat → Bool
isEven Zero = True
isEven (Succ Zero) = False
isEven (Succ (Succ n)) = isEven n

makeEven :: SNat n → Vec a n → Vec a (NextEven n)
makeEven n v =
```

makeEven

```
isEven :: Nat → Bool
isEven Zero = True
isEven (Succ Zero) = False
isEven (Succ (Succ n)) = isEven n

makeEven :: SNat n → Vec a n → Vec a (NextEven n)
makeEven n v =
  if isEven n
    then v
    else case v of
      VCons elt _ → VCons elt v
```

makeEven

```
isEven :: Nat → Bool
isEven Zero = True
isEven (Succ Zero) = False
isEven (Succ (Succ n)) = isEven n

makeEven :: SNat n → Vec a n → Vec a (NextEven n)
makeEven n v =
 if isEven n
  then v
  else case v of
   VCons elt _ → VCons elt v
```

```
Couldn't match expected type `Nat'
         with actual type `SNat n'
    In the first argument of `isEven',
    namely `n'
```

makeEven

```
forget :: SNat n → Nat
forget SZero = Zero
forget (SSucc n) = Succ (forget n)


makeEven :: SNat n → Vec a n → Vec a (NextEven n)
makeEven n v =
  if isEven (forget n)
    then v
    else case v of
      VCons elt _ → VCons elt v
```

makeEven

```
forget :: SNat n → Nat
forget SZero = Zero
forget (SSucc n) = Succ (forget n)


makeEven :: SNat n → Vec a n → Vec a (NextEven n)
makeEven n v =
 if isEven (forget n)
   then v
   else case v of
    VCons elt _ → VCons elt v
```

```
Couldn't match type `n' with
                   `NextEven n'
```

sIsEven

```
sIsEven SZero = STrue
sIsEven (SSucc SZero) = SFalse
sIsEven (SSucc (SSucc n)) = sIsEven n
```

sIsEven

```
sIsEven :: SNat n → SBool ??
sIsEven SZero = STrue
sIsEven (SSucc SZero) = SFalse
sIsEven (SSucc (SSucc n)) = sIsEven n
```

sIsEven

```
type family IsEven (n :: Nat) :: Bool
type instance IsEven ‘Zero = ‘True
type instance IsEven (‘Succ ‘Zero) = ‘False
type instance IsEven (‘Succ (‘Succ n)) = IsEven n

sIsEven :: SNat n → SBool (IsEven n)
sIsEven SZero = STrue
sIsEven (SSucc SZero) = SFalse
sIsEven (SSucc (SSucc n)) = sIsEven n
```

makeEven

```
makeEven :: SNat n → Vec a n → Vec a (NextEven n)
makeEven n v =
  case sIsEven n of
    STrue → v
    SFalse → case v of
      VCons elt _ → VCons elt v
```

makeEven

```
makeEven :: SNat n → Vec a n → Vec a (NextEven n)
makeEven n v =
  case sIsEven n of
    STrue → v
    SFalse → case v of
      VCons elt _ → VCons elt v
```

```
Ok, modules loaded: Main.
```

makeEven

```
makeEven :: SNat n → Vec a n → Vec a (NextEven n)
makeEven n v =
  case sIsEven n of
    STrue → v  -- (True ~ IsEven n)
    SFalse → case v of  -- (False ~ IsEven n)
      VCons elt _ → VCons elt v
```

```
Ok, modules loaded: Main.
```

makeEven

```
type family NextEven (n :: Nat) :: Nat
type instance NextEven n = If (IsEven n) n (Succ n)

makeEven :: SNat n → Vec a n → Vec a (NextEven n)
makeEven n v =
  case sIsEven n of
    STrue → v  -- (True ~ IsEven n)
    SFalse → case v of  -- (False ~ IsEven n)
      VCons elt _ → VCons elt v
```

```
Ok, modules loaded: Main.
```

The singletons Library

- Coding with singletons requires duplication:
  - The original, unrefined datatype/function
  - The promoted type (automatic)/type family
  - The singleton type/function on singletons
- The singletons library does the work for you, using Template Haskell

```
data Nat = Zero | Succ Nat

isEven :: Nat → Bool
isEven Zero = True
isEven (Succ Zero) = False
isEven (Succ (Succ n)) = isEven n
```

```
import Data.Singletons

$(singletons [d|
   data Nat = Zero | Succ Nat

   isEven :: Nat → Bool
   isEven Zero = True
   isEven (Succ Zero) = False
   isEven (Succ (Succ n)) = isEven n
 |])
```

```
import Data.Singletons

$(singletons [d|
    data Nat = Zero | Succ Nat

    isEven :: Nat → Bool
    isEven Zero = True
    isEven (Succ Zero) = False
    isEven (Succ (Succ n)) = isEven n
  |])
```

```
data SNat :: Nat → * where
  SZero :: SNat ‘Zero
  SSucc :: SNat n → SNat (‘Succ n)
```

```
type family IsEven (n :: Nat) :: Bool
type instance IsEven ‘Zero = ‘True
type instance IsEven (‘Succ ‘Zero) = ‘False
type instance IsEven (‘Succ (‘Succ n)) = IsEven n
```

```
sIsEven :: SNat n → SBool (IsEven n)
sIsEven SZero = STrue
sIsEven (SSucc SZero) = SFalse
sIsEven (SSucc (SSucc n)) = sIsEven n
```

The Maybe Singleton

```
data Maybe a = Nothing | Just a

data SMaybe :: Maybe k → * where
```

The Maybe Singleton

```
data Maybe a = Nothing | Just a

data SMaybe :: Maybe k → * where
  SNothing :: SMaybe ‘Nothing
```

The Maybe Singleton

```
data Maybe a = Nothing | Just a

data SMaybe :: Maybe k → * where
  SNothing :: SMaybe 'Nothing
  SJust ::            → SMaybe ('Just x)
```

The Maybe Singleton

```
data Maybe a = Nothing | Just a

data SMaybe :: Maybe k → * where
  SNothing :: SMaybe 'Nothing
  SJust :: S      x → SMaybe ('Just x)
```

The Maybe Singleton

```
data Maybe a = Nothing | Just a

data SMaybe :: Maybe k → * where
  SNothing :: SMaybe 'Nothing
  SJust :: S?? x → SMaybe ('Just x)
```

The singletons Encoding

data family Sing (a :: k)

- Sing is a kind-indexed data family
- Sing branches only on its kind k
- In System FC, Sing has two arguments: a kind and a type. The type is ignored.

The singletons Encoding

```
data family Sing (a :: k)

data instance Sing (a :: Nat) where
  SZero :: Sing ‘Zero
  SSucc :: Sing n → Sing (‘Succ n)
```

The singletons Encoding

```
data family Sing (a :: k)

data instance Sing (a :: Nat) where
  SZero :: Sing 'Zero
  SSucc :: Sing n → Sing ('Succ n)

data instance Sing (a :: Maybe k) where
  SNothing :: Sing 'Nothing
  SJust :: Sing x → Sing ('Just x)
```

The singletons Encoding

```
data family Sing (a :: k)

data instance Sing (a :: Nat) where
  SZero :: Sing ‘Zero
  SSucc :: Sing n → Sing (‘Succ n)

data instance Sing (a :: Maybe k) where
  SNothing :: Sing ‘Nothing
  SJust :: Sing x → Sing (‘Just x)

justTwo :: Sing (‘Just (‘Succ (‘Succ ‘Zero)))
justTwo = SJust (SSucc (SSucc SZero))
```

Implicit Parameters

Implicit Parameters

```
makeEven :: {SNat n} → Vec a n → Vec a (NextEven n)
```

Implicit Parameters

```
class SingI (a :: k) where
  sing :: Sing a    -- produce singleton from dictionary
```

~~makeEven :: {SNat n} → Vec a n → Vec a (NextEven n)~~

```
makeEven :: ∀n. SingI n ⇒
                Vec a n → Vec a (NextEven n)
```

Implicit Parameters

```
class SingI (a :: k) where
  sing :: Sing a    -- produce singleton from dictionary

makeEven :: {SNat n} → Vec a n → Vec a (NextEven n)

makeEven :: ∀n. SingI n ⇒
                    Vec a n → Vec a (NextEven n)
makeEven v =
  case sIsEven (sing :: Sing n) of
    STrue → v
    SFalse → case v of
      VCons elt _ → VCons elt v
```

Haskell has Kind Classes!

Haskell has Kind Classes!

```
class SingKind (k :: □) where ...
```

Haskell has Kind Classes!

~~class SingKind (k :: □) where …~~

"type Any :: k"

import GHC.Exts

class (a ~ Any) ⇒ SingKind (a :: k) where …

Observations

- Programming with singletons uses techniques familiar to Haskellers (writing functions!) to simulate dependent types
- GHC's error messages are helpful and (relatively) easy to understand
- It is possible to translate dependently typed code from Agda with relatively few changes
- Still a problem: we cannot promote GADTs

Why Not Use Agda?

Why Not Use Agda?

- Phase separation (type erasure)

Why Not Use Agda?

- Phase separation (type erasure)
- Industrial-strength, optimizing compiler

Additional Topics in Paper

- Full details of encoding, with design decisions
- Extended example translating a richly-typed database access interface from Agda into Haskell using singletons
- A comparison between different ways to write dependently typed code in Haskell
- Suggestions for future extensions of the language to better support dependent types

cabal install singletons