

# Type-level Programming

Advanced Haskell

Andres Löh

10–11 October 2013 — Copyright © 2013 Well-Typed LLP



## Excursion: Writing an interpreter

```
data Expr =  
  Int    Int  
| Bool   Bool  
| IsZero Expr  
| Plus   Expr Expr  
| If     Expr Expr Expr
```

Imagined concrete syntax:

```
if isZero (0 + 1) then False else True
```

Abstract syntax:

```
If (IsZero (Plus (Int 0) (Int 1))) (Bool False) (Bool True)
```

# Evaluation

```
data Val =  
    VInt Int  
  | VBool Bool
```

```
eval :: Expr → Val  
eval (Int n)      = VInt n  
eval (Bool b)     = VBool b  
eval (IsZero e)   = case eval e of  
    VInt n → VBool (n == 0)  
    _      → error "type error"  
eval (Plus e1 e2) = case (eval e1, eval e2) of  
    (VInt n1, VInt n2) → VInt (n1 + n2)  
    _                  → error "type error"  
eval (If e1 e2 e3) = case eval e1 of  
    VBool b → if b then eval e2 else eval e3  
    _       → error "type error"
```



## Evaluation (contd.)

- ▶ Evaluation code is mixed with code for handling type errors.
- ▶ The evaluator uses **tags** (i.e., constructors) to distinguish values – these tags are maintained and checked at run time.
- ▶ Run-time type errors can, of course, be prevented by writing a type checker.
- ▶ But even if we know that we only have type-correct terms, the Haskell compiler does not enforce this.



# Phantom types

A common and useful trick is to introduce a phantom type argument to make additional distinctions:

```
data Expr a =  
  Int    Int  
| Bool   Bool  
| IsZero (Expr Int)  
| Plus   (Expr Int) (Expr Int)  
| If     (Expr Bool) (Expr a) (Expr a)
```

We also need a smart constructor:

```
Plus :: Expr Int → Expr Int → Expr a  
plus :: Expr Int → Expr Int → Expr Int  
plus = Plus
```



## Evaluation of phantom types

- ▶ A phantom type argument prevents us from building ill-typed expressions if we additionally use smart constructors.
- ▶ Phantom type arguments can often be used to impose a more rigid typing discipline on something that's internally (nearly) untyped – for example, they are quite often used in bindings to C libraries.
- ▶ However, in our situation, phantom types are only a partial solution: the evaluator doesn't change.



# The evaluator revisited

```
eval :: Expr → Val
eval (Int n)   = VInt n
eval (Bool b) = VBool b
...
```

Let's look at the first lines of the original version.

We now have `Expr a` as an input, so wouldn't it be nice if we could write

```
eval :: Expr a → a
```

– without any tags?



# The evaluator revisited

```
eval :: Expr a → a
eval (Int n)   = n
eval (Bool b) = b
...
```

Unfortunately, this fails ...

```
Int  :: Int → Expr a
Bool :: Bool → Expr a
```

Our smart constructors don't help us if while we're destructing expressions – we'd really like to have

```
Int  :: Int → Expr Int
Bool :: Bool → Expr Bool
```

and be able to exploit this information during pattern matching!



# A “normal” datatype

```
data Tree a = Leaf a  
           | Node (Tree a) (Tree a)
```

introduces constructors with types:

```
Leaf  :: a → Tree a  
Node  :: Tree a → Tree a → Tree a
```

For a parameterized type, all constructors target the unconstrained type – here, `Tree a`.

In our `Expr a` scenario, we’d like constructors that target only a part of the type, with restricted arguments.



## Generalizing the syntax

### Observation

The type signatures of the data constructors contain at least as much information as the **data** declaration itself.

```
data Tree a where  
  Leaf  :: a → Tree a  
  Node  :: Tree a → Tree a → Tree a
```

is as good as

```
data Tree a = Leaf a  
           | Node (Tree a) (Tree a)
```



# Generalizing the syntax

## Observation

The type signatures of the data constructors contain at least as much information as the **data** declaration itself.

```
data Tree :: * → * where  
  Leaf  :: a → Tree a  
  Node  :: Tree a → Tree a → Tree a
```

In this syntax, we can even easily replace the initial argument with a kind signature.

Furthermore, we have the syntactic freedom to write down type-restricted constructors.



## A GADT for well-typed expressions

```
data Expr =  
  Int    Int  
| Bool   Bool  
| IsZero Expr  
| Plus   Expr Expr  
| If     Expr Expr Expr
```

The original version of the **Expr** datatype.



# A GADT for well-typed expressions

```
data Expr :: * where
  Int    :: Int → Expr
  Bool   :: Bool → Expr
  IsZero :: Expr → Expr
  Plus   :: Expr → Expr → Expr
  If     :: Expr → Expr → Expr → Expr
```

The original version in the new syntax.



# A GADT for well-typed expressions

```
data Expr a =
  Int    Int
| Bool   Bool
| IsZero (Expr Int)
| Plus   (Expr Int) (Expr Int)
| If     (Expr Bool) (Expr a) (Expr a)
```

This is our phantom-typed version.



# A GADT for well-typed expressions

```
data Expr :: * → * where  
  Int    :: Int → Expr a  
  Bool   :: Bool → Expr a  
  IsZero :: Expr Int → Expr a  
  Plus   :: Expr Int → Expr Int → Expr a  
  If      :: Expr Bool → Expr a → Expr a → Expr a
```

The phantom-typed version in the new syntax.



# A GADT for well-typed expressions

```
data Expr :: * → * where  
  Int    :: Int → Expr Int  
  Bool   :: Bool → Expr Bool  
  IsZero :: Expr Int → Expr Bool  
  Plus   :: Expr Int → Expr Int → Expr Int  
  If      :: Expr Bool → Expr a → Expr a → Expr a
```

Restricting the result types – this is called a **generalized algebraic datatype**.

Note that **If** can still construct both **Expr Int** and **Expr Bool**, but we know that the components match.

We need the GADTs and KindSignatures language extensions

...





# Example expressions

```
example =  
  If (IsZero (Plus (Int 0) (Int 1))) (Bool False) (Bool True)
```

is inferred to be of type `Expr Bool`.

```
wrong = IsZero (Bool False)
```

triggers a type error.



## Evaluation revisited

```
eval :: Expr a → a  
eval (Int n)      = n  
eval (Bool b)     = b  
eval (IsZero e)   = eval e == 0  
eval (Plus e1 e2) = eval e1 + eval e2  
eval (If e1 e2 e3) = if eval e1 then eval e2 else eval e3
```

- ▶ No possibility for run-time failure (modulo `undefined`).
- ▶ No tags required.



## (Some) errors are caught

```
eval :: Expr a → a
eval (Int n)   = False
eval (Bool b) = False
```

yields

```
Couldn't match type 'Int' with 'Bool'
In the expression: False
In an equation for eval: eval (Int n) = False
```



## Question

```
data X :: * → * where
  C :: Int → X Int
  D :: X a
f (C n) = [n]
f D     = []
```

What is the type of `f`?

```
f :: X a → [Int]
f :: X a → [a]
```

None of the two types is an instance of the other.

Because of this, type inference for GADT pattern matches is generally not possible, and type signatures for such functions are required.



# Extending expressions

Let us extend the expression types with pair construction and projection:

```
data Expr :: * → * where
  Int    :: Int → Expr Int
  Bool   :: Bool → Expr Bool
  IsZero :: Expr Int → Expr Bool
  Plus   :: Expr Int → Expr Int → Expr Int
  If     :: Expr Bool → Expr a → Expr a → Expr a
  Pair   :: Expr a → Expr b → Expr (a, b)
  Fst    :: Expr (a, b) → Expr a
  Snd    :: Expr (a, b) → Expr b
```

What is remarkable here?

The arguments of **Fst** and **Snd** mention a type variable that does not occur in the result.



# Extending evaluation

```
eval :: Expr a → a
eval ...
eval (Pair x y) = (eval x, eval y)
eval (Fst p)    = fst (eval p)
eval (Snd p)    = snd (eval p)
```



# Existential types

```
example :: Expr Int → Int
example (Fst p) = fst (eval p) + snd (eval p)  -- type error
```

What's wrong here?

What type does `p` have, and `eval p` ?

We know there **exists** a type `t` such that `p :: Expr (Int, t)` , but we have no idea what type `t` is ...

This is why types using this form of type hiding are called **existential types**.



## Existential types (contd.)

Existential types predate GADTs in Haskell, and are independently useful.

Here's a not-so-useful example:

```
data Any :: * where
  Any :: a → Any
```

What does it do?

We can store anything in an `Any` , but after that, we cannot do anything useful with it anymore, because we lost all knowledge what it was.

```
heterogeneousList :: [Any]
heterogeneousList = [Any 2, Any id, Any 'x', Any False]
```

Apart from its length, this list carries no useful information. We cannot recover the types.



## A somewhat more useful example

```
data Stepper :: * → * where  
  Stepper :: s → (s → (a, s)) → Stepper a
```

Here, we store an unknown value of type `s` together with a `step` function we can perform to produce a known result of type `a` and a new unknown value.

```
step :: Stepper a → Stepper a  
step (Stepper x f) = Stepper (snd (f x)) f  
look :: Stepper a → a  
look (Stepper x f) = fst (f x)
```



## Stepper examples

```
counter :: Stepper Int  
counter = Stepper 0 (λn → (n, n + 1))  
fibs :: Stepper Int  
fibs = Stepper (0, 1) (λ(x, y) → (x, (y, x + y)))
```

Actually, a `Stepper` is just a different representation of an infinite stream:

```
streamSupply :: [a] → Stepper a  
streamSupply xs = Stepper xs (λ(x : xs) → (x, xs))  
unroll :: Stepper a → [a]  
unroll (Stepper s f) = unfoldr (Just ∘ f) s
```

A somewhat related representation of streams is used in the `vector` package to implement optimizations of stream transformation pipelines via [stream fusion](#).



- ▶ The name **vector** is also used for (homogeneous) lists with a fixed length.
- ▶ It turns out we can represent vectors in Haskell using GADTs.
- ▶ What we need first is a way to express a length as a type. Different lengths must correspond to different types.



## Peano naturals

A natural number is

- ▶ either **zero**,
- ▶ or the **successor** of a natural number.

As a normal Haskell datatype:

```
data Nat = Zero | Suc Nat
```

But we want **Zero** and **Suc Zero** to be different **types**!



# Type-level naturals

```
data Zero
data Suc n
```

Datatypes without constructors – we’re only interested in using these types as an index in a GADT.

Now:

```
type Three = Suc (Suc (Suc Zero))
```

Unfortunately, nothing keeps us from writing:

```
type Bad = Suc (Suc Bool)
```

(This is now fixable with new extensions to Haskell’s type system – via [promoted datatypes](#).)



## Vectors

Vectors are lists with a fixed number of elements:

```
data Vec :: * → * → * where
  Nil    :: Vec a Zero
  Cons :: a → Vec a n → Vec a (Suc n)
```

We use the second parameter of the `Vec` type to store its length.

```
example :: Vec Char Three
example = Cons 'x' (Cons 'y' (Cons 'z' Nil))
```



## Type-safe **head** and **tail**

```
head :: Vec a (Suc n) → a
head (Cons x xs) = x
tail :: Vec a (Suc n) → Vec a n
tail (Cons x xs) = xs
```

- ▶ No case for **Nil** is required.
- ▶ Actually, a case for **Nil** results in a type error.
- ▶ Applying **head** or **tail** to **Nil** also results in a type error.



## More functions on vectors

```
map :: (a → b) → Vec a n → Vec b n
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

The type states explicitly that the source and target vectors have the same length!

```
zipWith :: (a → b → c) → Vec a n → Vec b n → Vec c n
zipWith op Nil Nil = Nil
zipWith op (Cons x xs) (Cons y ys) = Cons (op x y)
                                         (zipWith op xs ys)
```

We require that the two input vectors have equal length!





## Yet more functions on vectors

```
snoc :: Vec a n → a → Vec a (Suc n)
snoc Nil      y = Cons y Nil
snoc (Cons x xs) y = Cons x (snoc xs y)

reverse :: Vec a n → Vec a n
reverse Nil      = Nil
reverse (Cons x xs) = snoc (reverse xs) x
```

But efficient `reverse` is already much more tricky ...

Let's look at `(++)` instead.



## Appending vectors

```
(++) :: Vec a m → Vec a n → Vec a ...
```

What's the length of the resulting vector?

It should be the sum of `m` and `n`, but `m` and `n` are types?  
How do we compute the sum of `m` and `n` on the type level?



# Appending vectors

```
(++) :: Vec a m → Vec a n → Vec a (Add m n)
```

Turns out this is another use case for [type families](#).

```
type family Add (m :: *) (n :: *) :: *
```



## Type-level addition

```
type family    Add m      n :: *  
type instance Add Zero    n = n  
type instance Add (Suc m) n = Suc (Add m n)
```

```
(++) :: Vec a m → Vec a n → Vec a (Add m n)  
Nil      ++ ys = ys  
Cons x xs ++ ys = Cons x (xs ++ ys)
```

Note that it is important [how](#) we define `Add` ...



# Converting between lists and vectors

Unproblematic:

```
toList :: Vec a n → [a]
toList Nil = []
toList (Cons x xs) = x : toList xs
```

Does not work:

```
fromList :: [a] → Vec a n
fromList [] = Nil
fromList (x : xs) = Cons x (fromList xs)
```

Why?

The type says that the result must be polymorphic in `n`, and it is not!



## From lists to vectors

We can

- ▶ specify the length we expect as an input,
- ▶ hide the length using an existential type.

For the former, we have to reflect type-level natural numbers on the value level:

```
data SNat :: * → * where
  SZero :: Nat Zero
  SSuc  :: Nat n → Nat (Suc n)
```

This is called a **singleton type**. Per natural number type `n`, there is one value of type `SNat n`.



```
fromList :: SNat n → [a] → Maybe (Vec a n)
fromList SZero [] = return Nil
fromList (SSuc n) (x : xs) = Cons x <$> fromList n xs
fromList _ _ = empty
```

We have to know the length in advance.

We can also store the vector in an existential ...



## From lists to vectors (contd.)

```
data VecAny :: * → * where
  VecAny :: Vec a n → VecAny a
fromList :: [a] → VecAny a
fromList [] = VecAny Nil
fromList (x : xs) = case fromList xs of
  VecAny ys → VecAny (Cons x ys)
```

Question: Is a `VecAny` any more useful than a list? After all, we still don't know its length ...



# Comparing the length of vectors

```
equalLength :: Vec a m → Vec b n → Bool
```

Not useful, because

```
if equalLength v w then head (zipWith (, ) v w)
    else ...
```

will not type check. We lose the information that `m` and `n` are equal. Why?



## Learning by testing

```
equalLength :: Vec a m → Vec b n → Bool
```

The type `Bool` tells `us` something, but not the compiler. Both `False` and `True` have the same type, so they're completely interchangeable, as far as the type system is concerned.



```
equalLength :: Vec a m → Vec b n → Maybe (Equal m n)
```

We're trying to compute a **proof** that **m** and **n** are actually equal. We should then be able to use that proof to convince the type checker later.

But how do we implement **Equal** ?



## The **Equal** type

Turns out we can use another GADT:

```
data Equal :: * → * → * where  
  Refl :: Equal a a
```

- ▶ The only situation in which we can construct an **Equal a b** is if we already know that **a** and **b** are equal – the constructor **Refl** represents **reflexivity** of equality.
- ▶ We can then temporarily lose knowledge of equality of the types, but **reveal** it again by **pattern matching** on an equality proof.



## Example: deciding equality

```
equalLength :: Vec a m → Vec b n → Maybe (Equal m n)
equalLength Nil Nil = Just Refl
equalLength (Cons x xs) (Cons y ys) =
  case equalLength xs ys of
    Just Refl → Just Refl
    Nothing → Nothing
equalLength _ _ = Nothing
```

- ▶ If both vectors are `Nil`, the type checker knows that `m` and `n` are both `Zero`, so we can use `Refl`.
- ▶ In the `Cons` case, what we're doing looks like the identity function, but in fact, we're changing the types!



## Using equality

We can no longer use `if - then - else`, but we can use pattern matching:

```
case equalLength v w of
  Just Refl → ... zipWith (,) v w ...
  Nothing → ...
```



# A representation type of types

We can use a GADT to give us a representation for a set of types:

```
data ExprType :: * → * where  
  IntT  :: ExprType Int  
  CharT :: ExprType Char  
  PairT :: ExprType a → ExprType b → ExprType (a, b)
```



## Type inference for expressions

- ▶ The function `fromList` converts a list into a vector. We don't know its length (statically), but it always succeeds.
- ▶ For expressions, if we have both untyped and typed expressions, the corresponding function is `inferType`. We don't know the type of the resulting expression, and it does not always succeed.
- ▶ Storing the typed expression without any further information in an existential is much less useful than to additionally store a representation of the inferred type.





# Type inference for expressions (contd.)

```
data ExprAny :: * where  
  ExprAny :: ExprType a → Expr a → ExprAny  
inferType :: UntypedExpr → Maybe ExprAny  
...
```

```
checkType :: ExprType a → UntypedExpr → Maybe (Expr a)  
checkType r1 e = case inferType e of  
  ExprAny r2 te → case equalType r1 r2 of  
    Just Refl → Just te  
    Nothing → Nothing
```

Here, `equalType` is similar to `equalLength`.



## Generics and dynamics

- ▶ The function `checkType` make use of something that can be seen as a type-safe cast.
- ▶ A representation type such as `ExprType` can generally be combined with a typed value to form a dynamically typed value that can safely be cast back to its original type.
- ▶ Similarly, functions written by pattern matching on a representation type essentially encode functionality that is generic over the represented types.



# Lifting lists to the type level

We can lift other data types to the type level – not just natural numbers:

Normal lists:

```
data List a = Nil | Cons a (List a)
```

Type-level lists:

```
data Nil  
data Cons x xs
```

Makes it even more obvious that type-level programming is rather untyped. (But won't be anymore in GHC 7.6 and later.)



## What's in a type-level list?

We can form lists of types now, and each of these lists forms a different type:

```
type Example = Cons Bool (Cons Char (Cons Ordering Nil))
```

Now, like we defined vectors as lists indexed by natural numbers, we instead define lists indexed by type-level lists.



# Heterogeneous vectors

```
data HVec :: * → * where  
  HNil    :: HVec Nil  
  HCons   :: t → HVec ts → HVec Cons t ts
```

- ▶ We have only one type argument for `HVec`, because the type-level list we index by not only determines the length, but also determines the types of the elements.
- ▶ Note that `HCons` is a (data) constructor, whereas `Cons` is a type.



## Functions on heterogeneous vectors

Example:

```
type family Append xs ys :: *  
type instance Append Nil          ys = ys  
type instance Append (Cons x xs) ys = Cons x (Append xs ys)  
happend :: HVec xs → HVec ys → HVec (Append xs ys)  
happend HNil          ys = ys  
happend (HCons x xs) ys = HCons x (happend xs ys)
```



- ▶ Using GADTs and type families, we can express very advanced constraints on the type level.
- ▶ There are a number of common techniques one can successfully apply: lifting datatypes to the type-level, singleton types, representation types, existential packing, type-safe casting, ...
- ▶ Things can be made a lot more kind-safe with DataKinds and KindPolymorphism, two new Haskell extensions.
- ▶ You have to decide how much effort you want to spend – the more you express on the type-level, the more you have to prove about your programs. (But proofs themselves are Haskell programs again.)