GADTs

Constructors of ADTs must target type T

↓

```
data Either a b where
  Left  :: a -> Either a b
  Right :: b -> Either a b
```

Does it make sense to lift these restrictions on a b

```
data Expr
  = LitI    Int
    LitB    Bool
    IsZero  Expr
    Plus    Expr Expr
    If      Expr Expr Expr
```

We can get an AST and form concrete syntax which gets parsed

- Haskell doesn't allow it's returning diff. types because they are expressions which require a consensus on types. (unlike the AST above)

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

→ Expr a
→ Expr a
→ ...
→ ...

the variable a here is a phantom type because it is never used

Now we have to make sure to construct well-typed terms. smart constructors hide information and provide a well-typed API. Now, how does an interp for this tree work?

Interpreter

We need the type information for evaluation

```
eval :: Expr a -> Val
eval (ListI n) = VInt
```

} annoying Val to type check

```
eval (IsZero e) =
  case eval e of
    VInt n -> VBool (n == 0)
    _      -> error "type error"
```

} Using the nice API should always work

This is a lot of boiler plate to take out of the box, inspect, and put back. We also have type checkers!

We need to write a relaxed ADT to allow for polymorphic type parameters. It's still all Expr a, but a can be anything! There are GADTs

```
eval :: Expr a -> a
eval (ListI n)  = n
eval (ListB b)  = b
eval (IsZero e) = eval e == 0

eval (If e1 e2 e3)
  | eval e1 = eval e2
  | otherwise = eval e3
```

} much cleaner and we can pattern match on it!

- Better correctness guarantees, but globals are hard!

- Pattern matching on GADTs requires type signatures.

```
data X a where
  C :: Int -> X Int
  D ::        X a
  E :: Bool -> X Bool
```

```
f (C n) = [n]
f D     = []
f (E n) = [n]
```

Without the annotations the compiler cannot infer the types for the constructed lists.

Because all types are implicitly $\forall a, \forall b$: foo : Expr a -> Expr b will break and instead quantify $\forall a, \exists b$ on:

foo (Fst e) = snd (eval e) } existentially quantified

* It is now in the box, but we cannot touch it.

- The more expressive the type system, the less runtime errors.

We can define (for example) peano arithmetic at the type-level:

data Zero
data Succ n } "phantom type"

data Vec a n where
Nil :: Vec a Zero
Cons :: a -> Vec a n
-> Vec a (Succ n)

Type-safe head and tail

head :: Vec a (Succ n) -> a
head (Cons x xs) = x

tail :: Vec a (Succ n) -> Vec a n
tail :: (Cons x xs) = xs

$\forall a, \forall b$. map :: (a -> b) -> [a] -> [b]

↳ we can't accidentally write map f (Cons x xs) = map f xs

- Not fully-dependent types (Haskell tries to fake it tho)

Making this it would make the type system non decidable (unless you're C++ and you timeout instead)

- What about the append function?

  vappend :: Vec a m -> Vec a n -> Vec a ??

  L> construct explicit evidence

  L> type families

Given $n, m \in \mathbb{N}$ what is their sum?

GADTs can describe the graph of addition:

```
data Sum m n s where
  SumZero :: Sum Zero n n
  SumSucc :: Sum m n s -> Sum (Succ m) n (Succ s)
```

Now we can say:

```
append :: Sum m ns -> Vec a m -> Vec a n -> Vec a s
append SumZero  Nil       ys = ys
append (SumSucc p) (Cons x xs) ys = Cons x (Append p xs ys)
```

- We are doing relational programming on the type-level.

We can use existential types for reflecting type-level natural numbers as singletons:

```
data SNat n where
  SZero :: SNat Zero
  SSucc :: SNat n -> SNat (Succ n)
```

- A fromList :: SNat n -> [a] -> Vec a n might still fail dynamically!

```
if equalLength xs ys
  then zipVec xs ys
  else zipVec xs xs
```

```
zipVec :: Vec a n -> Vec b n
                  -> Vec (a,b) n
```

- Does not type-check because the compiler is not convinced this is correct. There is no type-level information that these are the same.

A GADT can witness that two types are equal:

```
data Equal a b where
  Refl :: Equal a b
```

} only axiom

```
refl :: Equal a a
sym :: Equal a b -> Equal b a
trans :: Equal a b -> Equal b c -> Equal a c

refl       = Refl
sym Refl   = Refl
trans Refl Refl = Refl

eqLength :: Vec a m -> Vec b n -> Maybe (Equal m n)
eqLength     Nil     = Just Refl
eqLength (Cons x xs) (Cons y ys) =
  | Just Refl <- eqLength xs ys = Just Refl
eqLength _ _                    = Nothing
```

- Now if we get nothing it is not the same. If we get something we have a proof term.
- We can now talk about type equality and phantom types to express GADTs.