Generalized Algebraic Data Types
Algebraic data types (review)

```haskell
data Expr =
    Lit Int
  | Neg Expr
  | Add Expr Expr
  | Mul Expr Expr
```

Declaring this new data type gives you:

1. new type Expr

2. several “functions” for creating values of type Expr:

```
Lit :: Int -> Expr
Neg :: Expr -> Expr
Add :: Expr -> Expr -> Expr
Mul :: Expr -> Expr -> Expr
```
Limitations of mono-typed expressions

data Expr =
    -- literals
    LitI Int
    | LitB Bool
  -- integers
    | Neg Expr
    | Add Expr Expr
    | Mul Expr Expr
  -- booleans
    | Not Expr
    | Or  Expr Expr
    | And Expr Expr
  -- mixed
    | Equ Expr Expr
    | If  Expr Expr Expr

Problem: can build ill-typed expressions:

Add (LitB True) (LitI 4)

Solutions:

• dynamic typing during evaluation
• separate type-checking phase
• better: use Haskell type system to statically prevent ill-typed expressions
Getting more out of data types

Tool #1: *phantom types*
- type parameter that isn’t an argument to a data constructor
- use to *embed* and *enforce* properties in Haskell types

Tool #2: *generalized algebraic data types*
- write type of each data constructor *explicitly*
  - return types of each data constructor can be different
  - can include class constraints

*Very useful for deeply embedded DSLs!*
- *embed DSL type system in Haskell type system*
Phantom types (interlude)

no data constructors!

data Unsafe
data Safe

newtype Input\texttt{a} = In String \hspace{1cm} \textit{phantom type}

\textit{Create new unvalidated input.}
input :: String \rightarrow Input Unsafe
input = In

\textit{Validate the input. If unsafe, return Nothing.}
validate :: Input Unsafe \rightarrow \textit{Maybe (Input Safe)}
validate (In s) = if isSafe s then Just (In s)
\hspace{1cm} \text{else Nothing}

\textit{Store validated input in the database.}
store :: Input Safe \rightarrow \textit{IO ()}
store (In s) = addToDB s

\textit{type system ensures that we validate before storing!}
Back to typed expressions

Idea: add type information to the data type

data Expr a =
  -- literals
  Lit a
  -- integers
  | Neg (Expr Int)
  | Add (Expr Int) (Expr Int)
  | Mul (Expr Int) (Expr Int)
  -- booleans
  | Not (Expr Bool)
  | Or  (Expr Bool) (Expr Bool)
  | And (Expr Bool) (Expr Bool)
  -- mixed
  | Equ (Expr a) (Expr a)
  | If  (Expr Bool) (Expr a) (Expr a)
Typed expressions

Limitation: return type of all constructors is Expr a!

-- literals
Lit :: a -> Expr a

-- integers
Neg :: Expr Int -> Expr a
Add :: Expr Int -> Expr Int -> Expr a
Mul :: Expr Int -> Expr Int -> Expr a

-- booleans
Not :: Expr Bool -> Expr a
Or :: Expr Bool -> Expr Bool -> Expr a
And :: Expr Bool -> Expr Bool -> Expr a

-- mixed
Equ :: Expr a -> Expr a -> Expr a
If :: Expr Bool -> Expr a -> Expr a -> Expr a

statically ill-typed 😊
Add (Lit True) (Lit 4)

statically “well typed” 😞
Add (And (Lit True) (Lit False)) (Lit 4)
Generalized algebraic data types

Allow you to specify the types of data constructors more precisely

data Expr a where
  -- literals
  Lit :: a -> Expr a
  -- integers
  Neg :: Expr Int -> Expr Int
  Add :: Expr Int -> Expr Int -> Expr Int
  Mul :: Expr Int -> Expr Int -> Expr Int
  -- booleans
  Not :: Expr Bool -> Expr Bool
  Or  :: Expr Bool -> Expr Bool -> Expr Bool
  And :: Expr Bool -> Expr Bool -> Expr Bool
  -- mixed
  Equ :: Eq a => Expr a -> Expr a -> Expr Bool
  If  :: Expr Bool -> Expr a -> Expr a -> Expr a

can even define type class constraints and use this info in functions!

statically ill-typed 😊

Add (Lit True) (Lit 4)

Add (And (Lit True) (Lit False)) (Lit 4)