



Generic programming

Advanced functional programming

Wouter Swierstra

Wrapping up

- Official opening this afternoon at the Centraal Museum
- Pizza tonight at Channable from 18:00 onwards
- Hand in your keys tomorrow morning (09:45-10:00)
- If you want, we can arrange for you to store your luggage in an office upstairs.

Motivation

Similar functionality for different types

- equality, comparison
- mapping over the elements, traversing data structures
- serialization and deserialization
- generating (random) data
- ...

Often, there seems to be an algorithm independent of the details of the datatype at hand. Coding this pattern over and over again is boring and error-prone.

Deriving

We can use Haskell's *deriving* mechanism to get some functionality for free:

```
data Tree = Leaf
  | Node Tree Int Tree
  deriving (Show, Eq)
```

This works for a handful of built-in classes, such as `Show`, `Ord`, `Read`, etc.

But what if we want to derive instances for classes that are not supported?

Example: encoding values

```
data Tree = Leaf | Node Tree Int Tree
```

```
data Bit = 0 | 1
```

```
encodeTree :: Tree -> [Bit]
```

```
encodeTree Leaf = [0]
```

```
encodeTree (Node l x r) = [1] ++ encodeTree l  
                          ++ encodeInt x  
                          ++ encodeTree r
```

We assume a suitable encoding exists for integers:

```
encodeInt :: Int -> [Bit]
```

Example: encoding values

```
data Lam = Var Int
         | App Lam Lam
         | Abs Lam
```

```
encodeLam :: Lam -> [Bit]
```

```
encodeLam (Var n)   = [0] ++ encodeInt n
```

```
encodeLam (App f a) = [1,0] ++ encodeLam f
                    ++ encodeLam a
```

```
encodeLam (Abs e)  = [1,1] ++ encodeLam e
```

Encode: Underlying ideas

In both cases we have seen, we:

- encode the choice between different constructors using sufficiently many bits,
- and append the encoded arguments of the constructor being used in sequence.
- use the encode function being defined at the recursive positions

Goal

Express the underlying algorithm for `encode` in such a way that we do not have to write a new version of `encode` for each datatype anymore.

(Datatype-)Generic Programming

Techniques to exploit the structure of datatypes to define functions by *induction over the type structure*.

Approach taken in this lecture

- define a uniform representation of data types;
- define a functions `to` and `from` to convert values between user-defined datatypes and their representations.
- define your generic function by induction on the structure the representation.

Regular datatypes

Most Haskell datatypes have a common structure:

```
data Pair a b = Pair a b
```

```
data Maybe a = Nothing | Just a
```

```
data Tree a = Tip | Bin (Tree a) a (Tree a)
```

```
data Ordering = LT | EQ | GT
```

Informally:

- A datatype can be parameterized by a number of variables.
- A datatype has a number of constructors.
- Every constructor has a number of arguments.
- Every argument is a variable, a different type, or a recursive call.

Constructing regular datatypes

Idea

If we can describe regular datatypes in a different way, using a limited number of combinators, we can use this structure to define algorithms for all regular datatypes.

We proceed in two steps:

- abstract over recursion
- describe the “remaining” structure systematically.

We can define `fix` in Haskell using the defining property of fixed point combinators:

```
fix f = f (fix f)
```

This lets us capture recursion explicitly – enabling us to memoize computations, for example.

Question

What is the type of `fix`?

Fixpoints

We would like to define a similar fixpoint operation to describe recursion in *datatypes*.

For functions, we *abstract over* the recursive calls:

```
fac :: (Int -> Int) -> Int -> Int
fac = \fac x -> if x == 0 then 1 else x * fac (x-1)
```

For data types, let's do the same:

```
data Tree t = Leaf
  | Node t Int t
```

We introduce a separate *type* parameter corresponding to recursive occurrences of trees.

Type-level fixpoints?

```
data TreeF t = Leaf  
  | Node t Int t
```

Now Tree is not recursive – how can we take compute its fixpoint?

Type-level fixpoints

We can compute the fixpoint of a *type constructor* analogously to the `fix` function:

```
fix f = f (fix f)
```

```
data Fix f = In (f (Fix f))
```

Question

What is the *kind* of `Fix`?

Type-level fixpoints

We can now define trees using our `Fix` datatype:

```
data TreeF t = LeafF
  | NodeF t Int t
```

```
data Fix f = In (f (Fix f))
```

```
type Tree = Fix TreeF
```

The type `TreeF` is called the *pattern functor* of trees.

Question

What is the pattern functor for our data type of lambda terms?

Type-level fixpoints

This construction works equally well for lists:

```
data ListF a xs = NilF
  | ConsF a xs
```

```
data Fix f = In (f (Fix f))
```

```
type List a = Fix (ListF a)
```

Question

Is our type `List a` the same as `[a]`?

Type-level fixpoints

This construction works equally well for lists:

```
data ListF a xs = NilF
  | ConsF a xs
```

```
data Fix f = In (f (Fix f))
```

```
type List a = Fix (ListF a)
```

Question

Is our type `List a` the same as `[a]`?

What does 'the same' mean?

Type isomorphisms

Two types A and B are *isomorphic* if we can define functions

`f :: A -> B`

`g :: B -> A`

such that

`forall (x :: A) . g (f x) = x`

`forall (x :: B) . f (g x) = x`

Types `Fix (ListF a)` and `[a]` are isomorphic

```
from :: (Fix (ListF a)) -> [a]
from (In NilF)           = []
from (In (ConsF x xs))  = x : from xs
```

```
to :: [a] -> Fix (ListF a)
to []           = In NilF
to (x : xs)    = In (ConsF x (to xs))
```

It is relatively easy to see that these are inverses ...

A single step of recursion

Instead of taking the fixpoint, we can also use the pattern functor to observe a single layer of recursion.

To do so, we consider the type `ListF a [a]` – the outermost layer is a `NilF` or `ConsF`; any recursive children are ‘real’ lists.

```
from :: ListF a [a] -> [a]
```

```
from NilF      = []
```

```
from (ConsF x xs) = x : xs
```

```
to :: [a] -> ListF a [a]
```

```
to []          = NilF
```

```
to (x : xs)   = ConsF x xs
```

Once again, these are inverses.

Pattern functors are functors

```
data ListF a r = NilF | ConsF a r
```

```
instance Functor (ListF a) where
```

```
  fmap f NilF      = NilF
```

```
  fmap f (ConsF x r) = ConsF x (f r)
```

Mapping over the pattern functor means applying the function to all recursive positions.

This is different from what `fmap` does on lists, normally!

Pattern functors are functors – contd.

```
data TreeF t = LeafF  
  | NodeF t Int t
```

```
instance Functor TreeF where
```

```
  fmap f (LeafF)      = LeafF
```

```
  fmap f (NodeF l x r) = NodeF (f l) x (f r)
```

Where these pattern functors give us a good way to describe recursive datatypes – how should we write them?

Idea

Haskell data types can typically be described as a combination of a small number of primitive operations.

Building pattern functors systematically

Choice between two constructors can be represented using

```
data (f :+: g) r = L (f r) | R (g r)
```

Choice between constructors can be represented using multiple applications of (`:+:`).

Two constructor arguments can be combined using

```
data (f **: g) r = f r **: g r
```

More than two constructor arguments can be described using multiple applications of (`**:`).

Building pattern functors systematically – contd.

A recursive call can be represented using

```
data I r = I r
```

Constants (such as independent datatypes or type variables) can be represented using

```
data K a r = K a
```

Constructors without argument are represented using

```
data U r = U
```

Example

Our kit of combinators.

```
data (f :+: g) r = L (f r) | R (g r)
```

```
data (f :+: g) r = f r :+: g r
```

```
data I r = I r
```

```
data K a r = K a
```

```
data U r = U
```

```
data ListF a r = NilF | ConsF a r
```

```
type ListS a = U :+: (K a :+: I)
```

The types `ListS a r` and `[a]` are isomorphic.

All simple data types in Haskell can be described using these five combinators.

Excursion: algebraic data types

Haskell's data types are sometimes referred to as **algebraic** datatypes.

What does *algebraic* mean?

Excursion: algebraic data types

Haskell's data types are sometimes referred to as **algebraic** datatypes.

What does *algebraic* mean?

Abstract algebra is a branch of mathematics that studies mathematical objects such as monoids, groups, or rings.

These structures are typically generalizations of familiar sets/operations (such as addition or multiplication on natural numbers).

If you prove a property of these structures from the axioms, this property for every structure satisfying the axioms.

Algebraic datatypes

The $:\ast:$ and $:\text{+}:$ behave similarly to \ast and $+$ on numbers; the u type is similar to 1 .

For example, for any type t we can show $1 \ast t$ is isomorphic to t .

Or for any types t and u , we can show $t \ast u$ is isomorphic to $u \ast t$.

Similarly, $t \text{+} u$ is isomorphic to $u \text{+} t$.

So far we have seen how to represent data types using pattern functors, built from a small number of combinators.

- How can we define *generic functions* – such as the binary encoding example we saw previously?
- How can we convert between user-defined data types and their pattern functor representation?

Defining generic functions

We would like to define a function

```
encode :: f a -> [Bit]
```

that works on all pattern functors f .

Instead, we'll define a slight variation:

```
encode :: (a -> [Bit]) -> f a -> [Bit]
```

which abstracts over the handling of recursive subtrees.

Generic encoding

```
class Encode f where
```

```
  fencode :: (a -> [Bit]) -> f a -> [Bit]
```

```
instance Encode U where
```

```
  fencode _ U = []
```

```
instance Encode (K Int) where
```

```
  -- suitable implementation for integers
```

```
instance Encode I where
```

```
  fencode f (I r) = f r
```

Generic encoding - contd.

```
class Encode f where
```

```
  fencode :: (a -> [Bit]) -> f a -> [Bit]
```

```
instance (Encode f, Encode g) =>
```

```
  Encode (f :+: g) where
```

```
    fencode f (L x) = 0 : fencode f x
```

```
    fencode f (R x) = 1 : fencode f x
```

```
instance (Encode f, Encode g) =>
```

```
  Encode (f **: g) where
```

```
    fencode f (x **: y) =
```

```
      fencode f x ++ fencode f y
```

Where are we now?

Using these instances, we can derive `fencode` for every pattern functor built up from the functor combinators.

How does that give us `encode` for a concrete datatype?

If we have a conversion function

```
from :: [a] -> ListS a [a]
```

we can define

```
encodeList :: [Int] -> [Bit]
```

```
encodeList = fencode encodeList . from
```

The Regular class

We can systematically store the isomorphism using a class:

```
class Regular a where
```

```
  from :: a      -> (PF a) a
```

```
  to   :: PF a a -> a
```

What is PF?

The Regular class

We can systematically store the isomorphism using a class:

```
class Regular a where  
  from  :: a      -> (PF a) a  
  to    :: PF a a -> a
```

What is PF?

```
type family PF a :: * -> *
```

```
instance Regular [a] where
```

```
  from = ...
```

```
  to   = ...
```

```
type instance PF [a] = ListS a
```

Generic encode, again

We can write a generic encoding function:

```
encode :: (Regular a, Encode (PF a)) => a -> [Bit]
encode = fencode encode . from
```

This works for *any* regular data type that can be represented as a pattern functor.

Who does what?

Generic library

Provides the functor combinators and some other helper functions.

Library

Provides generic functions by defining instances for all the functor combinators.

User

Per datatype, provides an isomorphism with the pattern functor. Can then use all the generic functions.

- Available from Hackage.
- Provides generic programming functionality in the style just described.
- Several generic functions are defined, more in `regular-extras`.
- Can automatically derive the pattern functor and isomorphism for a datatype (using Template Haskell).

Limitations of the approach

- Not all types are regular – nested types, mutually recursive types, GADTs are all not supported.
- Encoding type parameters via constants is not optimal. We cannot, for example, generically define the map function over a type parameter using regular.

Recap

- Pattern functors give us the mathematical machinery to describe and recursive datatypes.
- As a result, we can define generic functions (such as `encode`) and patterns of recursion (`cata`);
- Understanding pattern functors lets us express the relation between data types and their folds (Church encodings)
- We can use Haskell's type classes to assemble modular datatypes and functions!
- We can use this technology to define new data structures generically – such as zippers or tries.

Other approaches

There are many generic programming frameworks.

They take different views on the structure of Haskell datatypes and have slightly different strengths and weaknesses.

Some other approaches:

- Scrap your boilerplate (`syb`)
- Uniplate
- EMGM
- `instant-generics`
- `multirec`
- Template Haskell

One point we glossed over in the discussion about the `regular` library is how to convert between our representation type and user-written datatypes.

We can automate this using Template Haskell:

- inspect the datatype definition;
- generate the corresponding `to` and `from` functions.

This works – but requires some programming work – especially if you’re writing your own generic programming library.

GHC now ships with a built-in library for writing generic functions `GHC.Generics`.

This handles all the conversions between representations for you.

It also exposes a great deal of meta-information such as:

- data type names;
- constructor names;
- field projections;
- ...

It has become the *de facto* standard for a lot of generic programming work.

But using it is not always easy – you need to understand *all* of Haskell to write a generic function yourself.

Bonus slides on combining data types

Combining datatypes

In Haskell, whenever we define a data type:

```
data Expr = Val Int | Add Expr Expr
```

We can add new functions freely:

```
eval :: Expr -> Int
```

```
render :: Expr -> String
```

But we cannot add new constructors without modifying the datatype and any functions defined over it.

In object oriented languages, the situation is dual: we can add new subclasses to a class, but adding new methods requires updating every subclass.

The Expression Problem

Phil Wadler dubbed this the Expression Problem:

The expression problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts).

The Expression Problem

Phil Wadler dubbed this the Expression Problem:

The expression problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts).

How can we address the Expression Problem in Haskell?

A naive approach

```
data IntExpr = Val Int | Add Expr Expr
```

```
data MulExpr = Mul IntExpr IntExpr
```

```
type Expr = Either IntExpr MulExpr
```

Question

What is wrong with this approach?

A naive approach

```
data IntExpr = Val Int | Add Expr Expr
```

```
data MulExpr = Mul IntExpr IntExpr
```

```
type Expr = Either IntExpr MulExpr
```

Question

What is wrong with this approach?

We cannot freely mix addition and multiplication.

Solution: work with pattern functors

```
data AddF a = Val Int | Add a a
```

```
data MulF a = Mul a a
```

```
data Expr f = In (f (Expr f))
```

```
type MyExpr = Expr (AddF :+: MulF)
```

Problems

- How can we write functions over expressions?
- Constructing expressions is a pain:

```
addExample :: Expr (MulF :+: AddF)
```

```
addExample = In (Inl (Mul (In (Inr (Val 1)))  
                        (In (Inr (Val 2)))))
```

Functions over expressions

Usually, we write functions through pattern matching on a fixed set of branches.

But pattern matching on our constructors is painful (we have lots of injections in the way).

And this *fixes* the possible patterns that we accept.

Functions over expressions

Usually, we write functions through pattern matching on a fixed set of branches.

But pattern matching on our constructors is painful (we have lots of injections in the way).

And this *fixes* the possible patterns that we accept.

Idea

Use Haskell's class system to *assemble* algebras for us!

Functions over expressions

To define a function over an expression – without knowing the constructors – we introduce a new type class:

```
class Eval f where
  evalAlg :: f Int -> Int

eval :: Eval f => Expr f -> Int
eval = cata evalAlg
```

Functions over expressions

We can now add instance for all the constructors that we wish to support:

```
instance Eval AddF where
```

```
  evalAlg (Add l r) = l + r
```

```
  evalAlg (Val i)   = i
```

```
instance Eval MulF where
```

```
  evalAlg (Mul l r) = l * r
```

```
...
```


Functions over expressions

To assemble the desired algebra, however, we need one more instance:

```
instance (Eval f, Eval g) => Eval (f :+: g) where
  evalAlg x = ...
```

Question

What should this instance be?

Functions over expressions

To assemble the desired algebra, however, we need one more instance:

```
instance (Eval f, Eval g) => Eval (f :+: g) where  
  evalAlg (Inl x) = evalAlg x  
  evalAlg (Inr y) = evalAlg y
```

The Expression Problem

- How can we write functions over expressions?
 - Use type classes
- Constructing expressions is a pain:

```
addExample :: Expr (MulF :+: AddF)
```

```
addExample = In (Inl (Mul (In (Inr (Val 1)))  
                        (In (Inr (Val 2)))))
```

The Expression Problem

- How can we write functions over expressions?
 - Use type classes
- Constructing expressions is a pain:

```
addExample :: Expr (MulF :+: AddF)
```

```
addExample = In (Inl (Mul (In (Inr (Val 1)))  
                        (In (Inr (Val 2)))))
```

Idea

Define smart constructors!

Not so smart constructors

For any fixed pattern functor, we can define auxiliary functions to assemble datatypes:

```
data AddF a = Val Int | Add a a
```

```
type AddExpr = Expr AddF
```

```
add :: AddExpr -> AddExpr -> AddExpr
```

```
add l r = In (Add l r)
```

But how can we handle coproducts of pattern functors?

Automating injections

To deal with coproducts, we introduce a type class describing *how* to inject some 'small' pattern functor `sub` into a larger one `sup`:

```
class (:<:) sub sup where  
  inj :: sub a -> sup a
```

What instances are there?

Instances

```
class (:<:) sub sup where
```

```
  inj :: sub a -> sup a
```

```
instance (:<:) f f where
```

```
  inj = ...
```

```
instance (:<:) f (f :+: g) where
```

```
  inj = ...
```

```
instance ((:<:) f g) => (:<:) f (h :+: g) where
```

```
  inj = ...
```

Question

How should we complete the above definitions?

Instances

```
class (:<:) sub sup where
```

```
  inj :: sub a -> sup a
```

```
instance (:<:) f f where
```

```
  inj = id
```

```
instance (:<:) f (f :+: g) where
```

```
  inj = Inl
```

```
instance ((:<:) f g) => (:<:) f (h :+: g) where
```

```
  inj = inj . Inr
```


Smart constructors

```
inject :: ((<:) g f) => g (Expr f) -> Expr f
```

```
inject = In . inj
```

```
val :: (AddF <: f) => Int -> Expr f
```

```
val x    = inject (Val x)
```

```
add :: (AddF <: f) => Expr f -> Expr f -> Expr f
```

```
add x y  = inject (Add x y)
```

```
mul :: (MulF <: f) => Expr f -> Expr f -> Expr f
```

```
mul x y  = inject (Mul x y)
```

Results!

```
e1 :: Expr AddF
```

```
e1 = val 1 `add` val 2
```

```
v1 :: Int
```

```
v1 = eval e1
```

```
e2 :: Expr (MulF :+: AddF)
```

```
e2 = val 1 `mul` (val 2 `add` val 3)
```

```
v2 :: Int
```

```
v2 = eval e2
```

We can easily add new constructors:

```
data SubF a = SubF a a
```

```
type NewExpr = SubF :+: MulF :+: AddF
```

Or define new functions:

```
class Render f where  
  render :: f String -> String
```

What if we would like to define recursive functions without using folds?

A first attempt might be:

```
class Render f where  
  render :: f (Expr f) -> String
```

What if we would like to define recursive functions without using folds?

A first attempt might be:

```
class Render f where
  render :: f (Expr f) -> String
```

But this is too restrictive! We require `f` and the recursive pattern functors `(Expr f)` to be the same.

Generalizing

A more general type seems better:

```
class Render f where  
  render :: f (Expr g) -> String
```

We can try to define an instance:

```
instance Render Mul where  
  render :: Mul (Expr g) -> String  
  render (Mul l r) = ...
```

Question

How can we complete this instance?

Generalizing

A more general type seems better:

```
class Render f where
  render :: f (Expr g) -> String
```

We can try to define an instance:

```
instance Render Mul where
  render :: Mul (Expr g) -> String
  render (Mul l r) = ...
```

Question

How can we complete this instance?

We cannot make a recursive call! We don't know that the pattern functor `g` can be rendered.

General recursion

```
class Render f where
```

```
  render :: Render g => f (Expr g) -> String
```

```
instance Render Mul where
```

```
  render :: Mul (Expr g) -> String
```

```
  render (Mul l r) = renderExpr l  
                    ++ " * "  
                    ++ renderExpr r
```

```
renderExpr :: Render f => Expr f -> String
```

```
renderExpr (In t) = render t
```


Recap

Datatype generic programming lets us exploit the *structure* of our *datatypes* to generate new *functions* and *types*.

Bonus slides on folds

Beyond simple generic functions

This concept of *pattern functor* gives us the language to study the structure of data structures in greater detail.

The `Foldable` class in Haskell is defined as follows:

```
class Foldable t where
  fold :: Monoid m => t m -> m
```

But not all folds compute monoidal results...

Can we give a more precise account of folds?

Folding lists

We have seen the `fold` on lists many times:

```
foldr :: (a -> r -> r) -> r -> [a] -> r
```

```
foldr op e []      = e
```

```
foldr op e (x:xs) = op x (foldr op e xs)
```

In the other lectures, we saw examples of other folds over natural numbers, trees, etc.

Can we describe this pattern more precisely?

- Replace constructors by user-supplied arguments.
- Recursive substructures are replaced by recursive calls.

Folding lists – contd.

```
foldr :: (a -> r -> r) -> r -> [a] -> r
```

Compare the types of the constructors with the types of the arguments:

```
(:) :: a -> [a] -> [a]
```

```
[] :: a -> [a]
```

```
cons :: a -> r -> r
```

```
nil :: a -> r
```

Folding other structures

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

```
foldNat :: (r -> r) -> r -> Nat -> r
```

```
foldNat s z Zero    = z
```

```
foldNat s z (Suc n) = s (foldNat s z n)
```

Folding other structures

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

```
foldNat :: (r -> r) -> r -> Nat -> r
```

```
foldNat s z Zero      = z
```

```
foldNat s z (Suc n)  = s (foldNat s z n)
```

```
data Lam = Var Int | App Lam Lam | Abs Lam
```

```
foldLam :: (Int -> r) -> (r -> r -> r) -> (r -> r)  
        -> Lam -> r
```

```
foldLam v ap ab (Var n)  = v n
```

```
foldLam v ap ab (App f a) = ap (foldLam v ap ab f)  
                             (foldLam v ap ab a)
```

```
foldLam v ap ab (Abs e)  = ab (foldLam v ap ab e)
```


Catamorphism generically

If we can map over the generic positions, we can express the fold or *catamorphism* generically:

```
cata :: (Regular a, Functor (PF a)) =>
      (PF a r -> r) -> a -> r
cata phi = phi . fmap (cata phi) . from
```

The argument describing how to handle each constructor, $\text{PF } a \ r \rightarrow r$, is sometimes called an *algebra*.

Question

What about the `cata` defined over fixpoints?

Alternatively

Or using our fixpoint operation on types we can write:

```
newtype Fix f = In (f (Fix f))
```

```
cata :: Functor f => (f a -> a) -> Fix f -> a
```

```
cata f (In t) = f (fmap (cata f) t)
```

- The Haskell wiki
 - www.haskell.org/haskellwiki/Generics
 - www.haskell.org/haskellwiki/GHC.Generics
- *A generic Deriving Mechanism for Haskell* by Magalhães et al.
- *Data types a la carte* by Wouter Swierstra;
- *Type-indexed data types* by Jeuring, Loeh, and Hinze.