Fusion

Advanced functional programming - Lecture 11

Wouter Swierstra

Today's lecture

- Auto in Agda demo
- Fusion
- Dependently typed programming in Haskell

Problem?

Suppose we want to compute the sum of squares in Haskell:

```
sumSq :: Int -> Int
sumSq y = sum (map square [1 .. y])
  where
  square x = x * x
```

Evaluation trace

```
sumSq 5
```

Evaluation trace

```
sumSq 5

sum (map square [1,2,3,4,5])
```

Evaluation trace

```
sumSq 5

sum (map square [1,2,3,4,5])

sum [1,4,9,16,25]
```

Evaluation trace

```
sumSq 5

sum (map square [1,2,3,4,5])

sum [1,4,9,16,25]

55
```

Intermediate data structures

Allocating the list of squares requires memory – even though we immediately traverse it.

Such *intermediate data structures* are best avoided when writing efficient code.

We traverse the list *twice* – while we could compute the desired result in a single pass.

Can we do better?

Sum of squares

```haskell
sumSq :: Int -> Int
sumSq y = go 1
  where
  go i
    | i > y     = 0
    | otherwise = square i + go (i + 1)
```

This version no longer computes an intermediate list of squares.

Sum of squares

```haskell
sumSq :: Int -> Int
sumSq y = go 1
  where
  go i
    | i > y     = 0
    | otherwise = square i + go (i + 1)
```

This version no longer computes an intermediate list of squares.

But it doesn't have the nice functional feel to it!

- not modular;
- harder to read;
- harder to maintain.

Challenge

How can we write the functional version…

Challenge

How can we write the functional version…

but optimize it avoid allocating intermediate data structures

Good news and bad

- GHC is really, really good at inlining and partially evaluating functions.
- But only if these functions are not recursive.

And the functions creating intermediate data structures (such as `map` are typically recursive).

Naive approach

What if we teach GHC how to avoid allocating intermediate data structures that arise from certain combinations of functions, such as maps and filters?

```
map f (map g xs) == map (f . g) xs

filter p (filter q xs)
  == filter (\x -> p x && q x) xs
```

GHC lets you specialize **rewrite rules** as compiler pragmas.

Whenever it encounters the left-hand expression, it will replace it with the right-hand expression.

Writing rewrite rules

```
{-# RULES
"mapMap" forall f g xs.
  map f (map g xs) = map (f . g) xs
#-}
```

We can add similar pragmas to inline functions.

Or prioritise to specify the order in which rules are applied.

Note: these equalities are *not* checked by GHC. You can change the meaning of your program all too easily!

Combining `map` and `filter`

##Question

What happens when we encounter `map f (filter p xs)`?

Combining `map` and `filter`

##Question

What happens when we encounter `map f (filter p xs)`?

Answer

Nothing – neither of our rules is triggered.

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Solution: add another rule

```haskell
mapFilter :: (a -> b) -> (a -> Bool) -> [a] -> [b]
mapFilter f p [] = []
mapFilter f p (x : xs) =
  if p x then f x : mapFilter f p xs
         else mapFilter f p xs
```

We can add the custom rewrite rule:

```haskell
map f (filter p xs) == mapFilter f p xs
```

Scaling this up?

- What about `filter p (map f xs)`?
- What happens when we want to handle other functions?
- Or what happens when we have more than two nested calls?

This approach clearly doesn't scale very well.

Take two

Instead of defining *many* rules for each function, we'll try to define functions using a common pattern of recursion – such as a fold.

If we can then explain how to fuse functions defined in this fashion, we can hopefully get performance gains without sacrificing the compositionality.

Using `foldr`

We can define functions like `map`, `filter`, `sum`, `++` as folds:

```
map f = foldr (\a xs -> f a : xs) []
sum = foldr (+) 0
filter p =
  foldr (\x xs -> if p x then x : xs else xs) []
```

Fusing `foldr`

If we now return to our `sumSq` example.

We can define both `sum` and `map` using `foldr`.

How can we fuse these into a single fold?

```
sum (map squares xs)
```

Unfolding definitions we get:

```
foldr (+) 0 (foldr (\x xs -> square x : xs) [] xs)
```

It's still not clear how to proceed…

Foldr vs construction

The `foldr` function deconstructs a list.

But a function like `map` can still build new intermediate lists:

```
map f = foldr (\a xs -> f a : xs) []
```

These are the structures we want to avoid creating!

Fold vs construction

Our functions should avoid calling `(:)` and `[]` directly.

Instead of writing:

```
map f = foldr (\a xs -> f a : xs) []
```

We can write:

```
map f xs =
  let m cons nil =
    foldr (\a xs -> cons (f a) xs) nil xs
  in m (:) []
```

So far, we haven't gained much.

Foldr and build

Lets capture the pattern of instantiating `nil` and `cons` with the actual constructors:

```
build :: ((a -> [a] -> [a]) -> [a] -> t) -> t
build g = g (:) []
```

And redefine our map function as:

```
map f xs =
  let m cons nil =
    foldr (\a xs -> cons (f a) xs) nil xs
  in build m
```

What have we gained?

We can define many other functions in this style:

- using `foldr` to traverse over lists;
- using `build` to construct lists.

What have we gained?

We can define many other functions in this style:

- using `foldr` to traverse over lists;
- using `build` to construct lists.

We can recognize *when* an intermediate data structure is created:

```
foldr c n (build g)
```

That is, we *build* a data structure, only to fold over it later.

This should be avoided!

Fusion

```
foldr c n (build g)
```

- `foldr` will replace `(:)` and `[]` with `c` and `n`;
- `build` will pass `(:)` and `[]` to `g`.

Why not pass `c` and `n` to `g` directly?

```
foldr c n (build g) = g c n
```

Example: `sum (map square [1,2,3,4,5])`

After inlining `map` and `sum` we're left with

```
sumSq =
  foldr (+) 0 (build (\c1 n1 ->
                 (foldr (\x xs -> c (square x) xs)
                        n1 [1,2,3,4,5])))
```

Example: `sum (map square [1,2,3,4,5])`

After inlining `map` and `sum` we're left with

```
sumSq =
  foldr (+) 0 (build (\c1 n1 ->
                 (foldr (\x xs -> c (square x) xs)
                        n1 [1,2,3,4,5])))
```

After applying our rule, we're left with:

```
sumSq y =
  foldr (\x y -> (+) (square x) y) 0 [1,2,3,4,5]
```

To get the fast version we saw previously, we should also define `enumFromTo` in this style...

Foldr fusion

Instead of writing recursive functions directly, we write *algebras* that are passed to a fold.

Instead of creating intermediate structures using constructors directly, we use `build` to create new lists.

A list that is created using build, and then deconstructed using a fold, can be fused away automatically.

Good news and bad

This generalizes nicely to *any* recursive data structure – not just lists.

You *already* know how to do this:

- the `cata` function folds over any data structure;
- we can build new data structures by passing in the corresponding constructors (cf. Church encodings).

But some functions, such as `foldl` or `zip`, are not easily defined as folds.

What about unfolds?

What about working with infinite data types?

```
unfoldr :: (s -> Maybe (a,s)) -> s -> [a]
```

Can we dualize this construction?

Steps

```
data Step a s = Done
  | Yield a s
```

A lazily generated list can be described by:

- either you're done;
- you should produce a new value of type `a` and continue;

This is isomorphic to the `Maybe (a,s)` part of `unfoldr`.

In pseudo-Haskell we can define the arguments to unfold as:

```
data CoList a = exists s . CoList (s -> Step a s) s
```

Colist

We can read out all the elements of a colist as follows:

```
unfold :: CoList a -> [a]
```

Colist

We can read out all the elements of a colist as follows:

```
unfold :: CoList a -> [a]

unfold (CoList step s) =
  go s
    where
    go s = case step s of
             Done -> []
             Yield x s' -> x : go s'
```

Creating colists

The opposite transformation is simple enough:

```
destroy :: [a] -> CoList a
```

Creating colists

The opposite transformation is simple enough:

```
destroy :: [a] -> CoList a

destroy xs = ...
  CoList step xs
    where
    step [] = Done
    step (x:xs) = Yield x xs
```

Map for CoLists

```
mapCL :: (a -> b) -> CoList a -> CoList b
mapCL f (CoList step s) = CoList step' s
  where
  step' s = case step of
              Done -> Done
              Yield x s' -> Yield (f x) s'
```

Note: this function is *not* recursive.

Destroy/unfold

We can define many functions, such as `map` and `filter`, to work on colists rather than lists.

```
map f = unfold . mapCL f . destroy
```

Once we compose two maps, however, we have something of the form:

```
unfold . mapCL f . destroy
  . unfold . mapCL g . destroy
```

If we can get rid of the intermediate `destroy . unfold` – GHC will fuse the two `mapCL` calls for us.

Rewrite rules to the rescue

If we add the following rule:

```
destroy (unfold xs) = xs
```

We can get rid of intermediate data structures!

In practice and theory

This idea is used by libraries such as `Data.Bytestring` and `Data.Text` to let you write efficient Haskell code, without sacrificing the functional look-and-feel.

By programming with algebras (the arguments to folds) and coalgebras (the arguments to unfolds) directly, we can minimize the usage of recursive functions.

This makes optimizing our code much easier!

Church encodings all over again!

Church encodings identify a data type with its fold:

```
type Church f = forall r . (f r -> r) -> r

from :: Church f -> Fix f
from c = f In

to :: Fix f -> Church f
to t = \f -> cata f t
```

What happens when we dualize this?

And identify codata with its unfold?

CoChurch encodings

CoChurch encodings identify a data type with its unfold.

We can define the generic unfold (or *anamorphism*) just as we did for folds:

```
data Fix f = In {unId :: f (Fix f)}

unfold :: Functor f => (a -> f a) -> a -> Fix f
unfold coalg s =
```

CoChurch encodings

CoChurch encodings identify a data type with its unfold.

We can define the generic unfold (or *anamorphism*) just as we did for folds:

```
data Fix f = In {unId :: f (Fix f)}

unfold :: Functor f => (a -> f a) -> a -> Fix f
unfold coalg s =

  In (fmap (unfold coalg) (coalg s))
```

CoChurch encodings

```haskell
data Fix f = In {unId :: f (Fix f)}

unfold :: Functor f => (a -> f a) -> a -> Fix f

data CoChurch f where
  CoChurch :: (r -> f r) -> r -> CoChurch f

to :: Fix f -> CoChurch f
to f = ...

from :: Functor f => CoChurch f -> Fix f
from (CoChurch coalg s) = ...
```

CoChurch encodings

```
data Fix f = In {unId :: f (Fix f)}

unfold :: Functor f => (a -> f a) -> a -> Fix f

data CoChurch f where
  CoChurch :: (r -> f r) -> r -> CoChurch f

to :: Fix f -> CoChurch f
to f = CoChurch unId f

from :: Functor f => CoChurch f -> Fix f
from (CoChurch coalg s) = unfold coalg s
```

Dependent types in Haskell

Agda vs Haskell

Agda is a *dependently typed language*; Haskell is not.

How close can we get in Haskell? How can we transcribe Agda programs to Haskell?

GADTs vs indexed families

GADTs take *types* as arguments; Agda's indexed families may be indexed by *values*.

But…

GADTs vs indexed families

GADTs take *types* as arguments; Agda's indexed families may be indexed by *values*.

But...

Data kind promotion lifts Haskell data types to the *type level*:

```
data Nat = Z | S Nat
```

Allowing us to write

```
data Vec :: Nat -> * -> * where
  Nil :: Vec Z a
  Cons :: a -> Vec n a -> Vec (S n) a
```

Vec in Haskell vs Agda

```
data Vec :: Nat -> * -> * where
  Nil :: Vec Z a
  Cons :: a -> Vec n a -> Vec (S n) a
```

Note that the number characterizing a vector's length *only* occurs in the type-level – it is erased at run-time.

In most dependently typed languages, the Cons constructor also carries the *length* of the tail (albeit implicitly).

There are optimizations (in Idris for example) that *erase* certain values that are not needed at runtime.

Computations

We can perform computations at the type-level using type families:

```
type family Sum (n :: Nat) (m :: Nat) :: Nat
type instance Sum Z m = m
type instance Sum (S k) m = S (Sum k m)
```

And use the computations in our *types*:

```
vappend :: Vec n a -> Vec m a -> Vec (Sum n m) a
```

Once again, all the 'numbers' only exist on the type level and are erased.

Challenges

In Agda we can write a function that takes the first n elements of a vector:

```
vchop :: (n : Nat) -> Vec (Sum n m) a
      -> (Vec n a, Vec m a)
```

Question

Why can we not write this function is Haskell directly?

Challenges

In Agda we can write a function that takes the first `n` elements of a vector:

```
vchop :: (n : Nat) -> Vec (Sum n m) a
      -> (Vec n a, Vec m a)
```

Question

Why can we not write this function is Haskell directly?

We cannot create the dependent function space

```
(n : Nat) -> ...
```

Singletons

The problem

Our natural numbers exist on the *type level*, but we have no way to write dependent functions using the corresponding values.

Singletons

The problem

Our natural numbers exist on the *type level*, but we have no way to write dependent functions using the corresponding values.

The solution

Introduce a separate data type marrying the natural number types and the values.

```
data Natty :: Nat -> * where
  Zy :: Natty Z
  Sy :: Natty n -> Natty (S n)
```

For any `n` there is only one possible value of type `Natty n` – we call `Natty` a *singleton type*.

vtake 2

Using `Natty` we can define `vtake` as follows:

```
vchop :: Natty n -> Vec (Sum n m) a
      -> (Vec n a, Vec m a)
vchop Zy ys = (Nil , ys)
vchop (Sy k) (Cons x xs) =
  let (as,bs) = vchop k xs in
  (Cons x as, bs)
```

General principle

For any *dependent function* in Agda of the form:

```
f :: (x : A) -> T x
```

We can translate this into Haskell as:

```
data SingleA :: A -> Set where
  ...

f :: forall a . Single a -> T a
```

Limitations

- We can only index GADTs by ‘simple’ algebraic data types. There is a lot of work on GADT promotion going on at the moment.
- This does not always work smoothly:

```
vtake :: Natty n -> Vec (Sum n m) a -> Vec n a
```

Does not work. In the recursive call, GHC cannot figure out how to instantiate `m`...

```
Couldn't match type ‘Sum n m0’ with ‘Sum n m’ …
    NB: ‘Sum’ is a type function,
    and may not be injective
    The type variable ‘m0’ is ambiguous
    Expected type:
        Natty n -> Vec (Sum n m) a -> Vec n a
      Actual type:
        Natty n -> Vec (Sum n m0) a -> Vec n a
    In the ambiguity check for the type
    signature for ‘vtake’:
      vtake ::
        forall (n :: Nat) (m :: Nat) a.
        Natty n -> Vec (Sum n m) a -> Vec n a
    To defer the ambiguity check to use sites,
      enable AllowAmbiguousTypes
    In the type signature for ‘vtake’:
      vtake :: Natty n -> Vec (Sum n m) a -> Vec n a
```

Resolving ambiguity

To address this, we need to make the missing information explicit.

One way to do so is by using a *proxy* type, carrying the missing informaton about `m`:

```
data Proxy :: k -> * where
  Proxy :: Proxy i
```

Note: this type does not store any interesting *value* information.

```
vtake :: Natty n -> Proxy m -> Vec (Sum n m) a
      -> Vec n a
vtake Z n xs                = Nil
vtake (S k) n (Cons x xs) = Cons x (vtake k n xs)
```

Calling vtake

To call the `vtake` function we now need to pass in the explicit proxy:

```
xs :: Vec (S (S (S (Z)))) Int
xs = Cons 1 (Cons 2 (Cons 3 Nil))

firstTwo =
  vtake (Sy (Sy Zy)) (Proxy :: Proxy (S Z)) xs
```

Can we get rid of this?

Type classes!

We can use type classes to generate singletons for us:

```
class NATTY (n :: Nat) where
  natty :: Natty n

instance NATTY Z where
  natty = Zy

instance NATTY n => NATTY (S n) where
  natty = Sy natty
```

Using NATTY

Using this type classe, we can avoid specifying some arguments:

```
vtake2 :: NATTY n => Proxy m -> Vec (Sum n m) a
       -> Vec n a
vtake2 p xs = vtake natty p xs
```

Now the singleton `Natty n` is inferred via the `NATTY` type class.

Duplication!

It is clear that these constructions lead to all kinds of duplication. We have seen several flavours of natural numbers:

- value level Nats;
- promoted Nats;
- singleton Nats;
- class constructing singleton Nats.
- ...

Similarly for addition we have:

- addition between values;
- the `Sum` type family.

Can we do better?

The singletons package

The `singletons` package uses Template Haskell to generate type-level data and functions automatically from their value-level counterparts:

```
data Nat = Zero | Succ Nat

$(genSingletons [''Nat ])
```

will generate:

```
data instance Sing (a :: Nat) where
  SZero :: Sing 'Zero
  SSucc :: SingRep n => Sing n -> Sing ( 'Succ n)
```

(which rougly corresponds to our `Natty` type).

Promoting functions

This even works for some functions!

```
$(promote [d|
  plus :: Nat -> Nat -> Nat
  plus Zero m = m
  plus (Succ n) m = Succ (plus n m) |])
```

Generates a type family:

```
type family Plus (n :: Nat) (m :: Nat) :: Nat
type instance Plus 'Zero m = m
type instance Plus ( 'Succ n) m = 'Succ (Plus n m)
```

In summary

- Using singleton types, we can 'fake' dependent types to some degree.
- We sometimes need to pass around more information than we would like, through singletons and proxies.
- Some of this can be automated, using type classes and Template Haskell.

Exam

Will consist of two parts:

1. Open book – feel free to bring four sheets of A4 paper with notes – but you cannot consult the internet or use your laptop. This is made during the exam slot. Goal: test your knowledge and understanding.
2. Take-home – handed out during exam slot. To be handed in before midnight on Friday April 13th through submit. You can use your laptop, internet, etc. Goal: test creativity and insight.

The scoring of the individual questions will be on the exam itself.

Exam

We've covered a lot of different topics. I always try to make the exam illustrative of the material that we covered in class:

- define a monad/foldable/applicative instance for `T`?
- how will lazy evaluation compute `foo(x,y,z)`?
- evaluate lambda term `t`?
- give a Church encoding/pattern functor for `T`.
- give an Agda function computing `bar`

Take-home exam

There will be a few more open problems in the take-home exam – typically those involving complex types in Agda/Dependent Haskell.

Feel free to discuss your ideas with fellow students, but do not share your work.

Questions?