Playing with Haskell Data

Neil Mitchell
Overview

- The “boilerplate” problem
- Haskell’s weakness (really!)
- Traversals and queries
- Generic traversals and queries
- Competitors (SYB and Compos)
- Benchmarks
Data structures

- A tree of *typed* nodes
- Parent/child relationship is important
A concrete data structure

data Expr = Val Int
  | Neg Expr
  | Add Expr Expr
  | Sub Expr Expr

- Simple arithmetic expressions
Task: Add one to every Val

\[
\begin{align*}
\text{inc} & : \text{Expr} \rightarrow \text{Expr} \\
\text{inc} \ (\text{Val} \ i) & = \text{Val} \ (i+1) \\
\text{inc} \ (\text{Neg} \ x) & = \text{Neg} \ (\text{inc} \ x) \\
\text{inc} \ (\text{Add} \ x \ y) & = \text{Add} \ (\text{inc} \ x) \ (\text{inc} \ y) \\
\text{inc} \ (\text{Sub} \ x \ y) & = \text{Sub} \ (\text{inc} \ x) \ (\text{inc} \ y)
\end{align*}
\]

- What is the worst thing about this code?
Many things!

1. If we add Mul, we need to change
2. The action is one line, obscured
3. Tedious, repetitive, dull
4. May contain subtle bugs, easy to overlook
5. Way too long
The boilerplate problem

- A lot of tasks:
  1. Navigate a data structure (boilerplate)
  2. Do something (action)

- Typically boilerplate is:
  - Repetitive
  - Tied to the data structure
  - Much bigger than the action
Compared to Pseudo-OO

```java
class Expr
class Val : Expr {int i}
class Neg : Expr {Expr a}
class Add : Expr {Expr a, b}
class Sub : Expr {Expr a, b}
```

1) Java/C++ are way too verbose to fit on slides!
Inc, in Pseudo-OO

```c
void inc(x) {
    if (x is Val) x.i += 1;
    if (x is Neg) inc(x.a)
    if (x is Add) inc(x.a); inc(x.b)
    if (x is Mul) inc(x.a); inc(x.b)
}
```

Casts, type evaluation etc omitted
Haskell’s weakness

- OO actually has a lower complexity
  - Hidden very effectively by horrible syntax
- In OO objects are deconstructed
- In Haskell data is deconstructed \textit{and} reconstructed
- OO destroys original, Haskell keeps original
Comparing inc for Add

- Haskell
  \[ \text{inc} \ (\text{Add} \ x \ y) = \text{Add} \ (\text{inc} \ x) \ (\text{inc} \ y) \]
- OO
  \[ \text{if} \ (x \ \text{is} \ \text{Add}) \ \text{inc}(x.a); \ \text{inc}(x.b) \]

- Both deconstruct Add (follow its fields)
- Only Haskell rebuilds a new Add
Traversals and Queries

- What are the common forms of “boilerplate”?
  - Traversals
  - Queries

- Other forms do exist, but are far less common
Traversals

- Move over the entire data structure
- Do “action” to each node
- Return a new data structure

- The previous example (inc) was a traversal
Queries

- Extract some information out of the data
- Example, what values are in an expression?
A query

vals :: Expr -> [Int]
vals (Val i) = [i]
vals (Neg x) = vals x
vals (Add x y) = vals x ++ vals y
vals (Mul x y) = vals x ++ vals y

- Same issues as traversals
Generic operations

- Identify primitives
  - Support lots of operations
  - Neatly
  - Minimal number of primitives

- These goals are in opposition!

- Here follow my basic operations…
Generic Queries

allOver :: a -> [a]
The vals query

vals x = [i | Val i <- allOver x]

- Uses Haskell list comprehensions – very handy for queries
- Can anyone see a way to improve on the above?
- Short, sweet, beautiful 😊
More complex query

- Find all negative literals that the user negates:

\[ \{ i \mid \text{Neg (Val } i) \leftarrow \text{allOver } x \}, \ i < 0 \] 

- Rarely gets more complex than that
Generic Traversals

- Have some “mutator”
- Apply to each item

\[
\text{traversal} :: (a \to a) \to a \to a
\]

5. Bottom up
6. Top down – automatic
7. Top down – manual
Bottom-up traversal

\[
\text{mapUnder} :: (a \to a) \to a \to a
\]
The inc traversal

\[
\text{inc } x = \text{mapUnder } f \ x \\
\text{where} \\
f (\text{Val } x) = \text{Val } (x+1) \\
f \ x = x
\]

- Say the action (first line)
- Boilerplate is all do nothing
Top-down queries

- Bottom up is almost always best
- Sometimes information is pushed down
- Example: Remove negation of add

\[ f(\text{Neg}(\text{Add } x \ y)) = \text{Add}(\text{Neg } x) (\text{Neg } y) \]

- Does not work, \( x \) may be \( \text{Add} \)

\[ f(\text{Neg}(\text{Add } x \ y)) = \]
\[ \text{Add}(f(\text{Neg } x)) (f(\text{Neg } y)) \]
Top-down traversal

mapOver :: (a -> a) -> a -> a

Produces one element per call
One element per call?

- Sometimes a traversal does not produce one element
- If zero made, need to explicitly continue
- In two made, wasted work
- Can write an explicit traversal
Top-down manual

compos :: (a -> a) -> a -> a
Compos

\[
\text{nong} \ (\text{Neg} \ (\text{Add} \ x \ y)) = \text{Add} \ (\text{nong} \ (\text{Neg} \ x)) \ (\text{nong} \ (\text{Neg} \ y))
\]
\[
\text{nong} \ x = \text{compos} \ \text{nong} \ x
\]

- Compos does no recursion, leaves this to the user
- The user explicitly controls the flow
Other types of traversal

- Monadic variants of the above

  - `allOverContext :: a -> [(a, a -> a)]`
  - Useful for doing something once

  - `fold :: ([r] -> a) -> (x -> a -> r) -> x -> r`
  - `mapUnder` with a different return
The Challenge

Pick an operation
Will code it up “live”
Traversals for your data

- Haskell has *type classes*
- \( \text{allOver :: Play a => a -> [a]} \)

- Each data structure has its own methods
- \( \text{allOver Expr /= allOver Program} \)
Minimal interface

- Writing 8+ traversals is annoying
- Can define all traversals in terms of one:

```
replaceChildren :: x -> ([x], [x] -> x)
```

- Get all children
- Change all children
replaceChildren :: x -> ([x], [x] -> x)
(children, generate) = replaceChildren x

- generate children == x
- @pre generate y
  length y == length children
Some examples

mapOver f x = gen (map (mapOver f) child)
where (child, gen) = replaceChildren (f x)

mapUnder f x = f (gen child2)
where (child, gen) = replaceChildren x
    child2 = map (mapUnder f) child

allOver x = x : concatMap allOverOver child
Where (child, gen) = replaceChildren x
Writing `replaceChildren`

- A little bit of thought
- Reasonably easy
- Using GHC, these instances can be derived automatically
Competitors: SYB + Compos

- Not Haskell 98, GHC only
- Use scary types...

- Compos
  - Provides compos operator and fold
- Scrap Your Boilerplate (SYB)
  - Very generic traversals
Compos

- Based on GADT’s
- No support for bottom-up traversals

compos ::
    (forall a. a -> m a) ->
    (forall a b. m (a -> b) -> m a -> m b) ->
    (forall a. t a -> m (t a)) ->
    t c -> m (t c)
Scrap Your Boilerplate (SYB)

- Full generic traversals
- Based on similar idea of children
  - But is actual children, of different types!

\[
gfoldl ::
(fordall a b. Term a \Rightarrow w (a \to b)
  \Rightarrow a \to w b)
\Rightarrow (fordall g. g \Rightarrow w g)
\Rightarrow a \to w a
\]
SYB vs Play, children

Diagram showing a comparison between SYB and Play for children.
Traversals are based on types:

0 `mkQ` f

f :: Expr -> Int

mkQ converts a function on Expr, to a function on all types

Then apply mkQ everywhere
Paradise benchmark

salaryBill :: Company -> Float
salaryBill = everything (+) (0 `mkQ` billS)

billS :: Salary -> Float
billS (S f) = f

salaryBill c = case c of
  S s -> s
  _ -> composOpFold 0 (+) salaryBill c

salaryBill x = sum [x | S x <- allOverEx x]
Runtime cost - queries
Runtime cost - traversals

- Play SYB Under
- Play SYB Over
- Play SYB Compos
- SYB
- Play Under
- Play Over
- Play Compos
- Compos
- Raw
In the real world?

- Used in Catch about 100 times
- Used in Yhc.Core library
- Used by other people
  - Yhc Javascript converter
  - Settings file converter
Conclusions

- Generic operations with simple types
- Only 1 simple primitive

If you only remember two operations:
  - allOver – queries
  - mapUnder – traversals