Uniform Boilerplate and List Processing

Neil Mitchell, Colin Runciman

www.cs.york.ac.uk/~ndm/uniplate

A Simple Expression Type

```
data Expr = Add  Expr Expr
          | Sub  Expr Expr
          | Mul  Expr Expr
          | Div  Expr Expr
          | Neg  Expr
          | Val  Int
          | Var  String
```

Task: Variable Occurrences

Type signature is optional

The interesting bit!

```
variables :: Expr → [String]
variables (Var  x   ) = [x]
variables (Val  x   ) = []
variables (Neg x    ) = variables x
variables (Add x y ) = variables x ++ variables y
variables (Sub x y ) = variables x ++ variables y
variables (Mul  x y ) = variables x ++ variables y
variables (Div  x y ) = variables x ++ variables y
```

Repetition

Dependent on constructors

Using Uniplate

Type signature still optional

variables :: Expr → [String]

variables x = [y | Var y ← universe x]

List comprehension

Uniplate function

- Concise, Haskell 98, Robust, Fast

What is Uniplate?

- A library for generic traversals
  - A bit like SYB (Scrap Your Boilerplate)

- Concise – shorter than others
- Quick – focus on performance
- Compatible – Haskell 98
  - Optional multi-parameter type classes

Uniform Types!

- Most traversals have value-specific behaviour for just *one type*
- Elements of one type can be a list
  - Exploit list processing
- This decision makes Uniplate:
  - Simpler
  - Less general

Generic Traversals

- Queries
  - Take a value
  - Extract some information
  - The ‘variables’ example is a query
- Transformations
  - Create a new value, based on the original

Generic Queries

universe :: Uniplate $\alpha \Rightarrow \alpha \rightarrow [\alpha]$

- Returns all values of the *same type* found within the value

universe (Mul (Val 3) (Var “y”)) =
  [Mul (Val 3) (Var “y”), Val 3, Var “y”]

Generic Transformations

transform :: Uniplate $\alpha \Rightarrow (\alpha \to \alpha) \to \alpha \to \alpha$

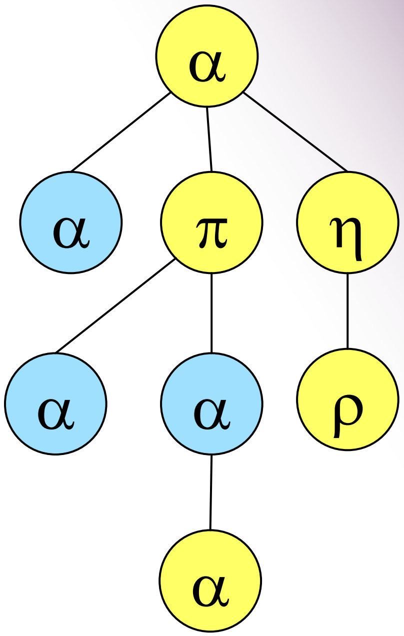
- Apply the function to each value of the *same type*, in a bottom-up manner

```
removeSub = transform f
   where f (Sub x y) = Add x (Neg y)
         f x = x
```

Several other transformation functions

The Uniplate class

$$\text{class Uniplate } \alpha \text{ where}$$
$$\text{uniplate} :: \alpha \rightarrow ([\alpha], [\alpha] \rightarrow \alpha)$$



- Given a value, returns
  1. Maximal substructures of the same type
  2. A function to generate a new value with new children

Traversals upon uniplate

```
universe x = x : concatMap universe children
  where (children, context) = uniplate x

transform f x =
        f $ context $ map (transform f) children
  where (children, context) = uniplate x
```

- Other useful functions in paper

Container types

data Stmt = ... | Assign String Expr | ...

- Stmt contains types of Expr
- How do we manipulate the Expr?
- Biplate is the answer
  - Less common, but more general

Biplate traversals

$universeBi :: Biplate\ \beta\ \alpha \Rightarrow \beta \rightarrow [\alpha]$

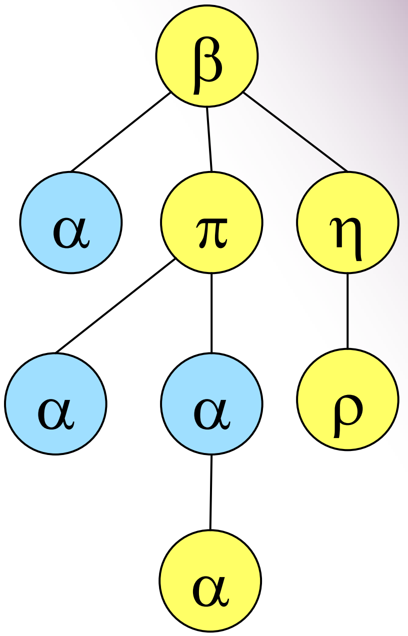
$transformBi ::$

$Biplate\ \beta\ \alpha \Rightarrow (\alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \beta$

- α is target type, β is container type
- Requires multi-parameter type classes
  - But no functional dependencies

The Biplate class

```
class Biplate β α where
        biplate :: β → ([α], [α] → β)
```



- Given a container, returns
  1. Maximal substructures of the target type
  2. A function to generate a new container with new targets

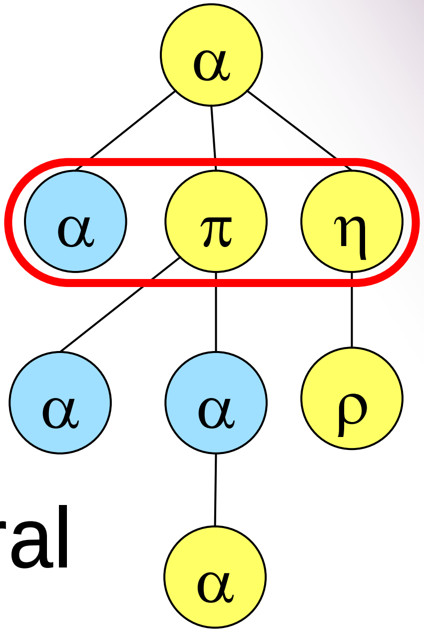
SYB Similarities

- SYB provides similar generic functions
  - universe is a bit like everything
  - transform is a bit like everywhere

```
removeSub = everywhere (mkT f)
  where f (Sub x y) = Add x (Neg y)
        f x = x
```

SYB Differences

- In SYB children are the direct sub-expressions of *any* type
- Uniplate is *same* type



- SYB traversals are more general
- SYB has runtime reflection
- SYB requires rank-2 types

“Paradise Benchmark”

sum [s | S s ← universeBi x]

let billS (S s) = s in
everything (+) (0 `mkQ` billS)

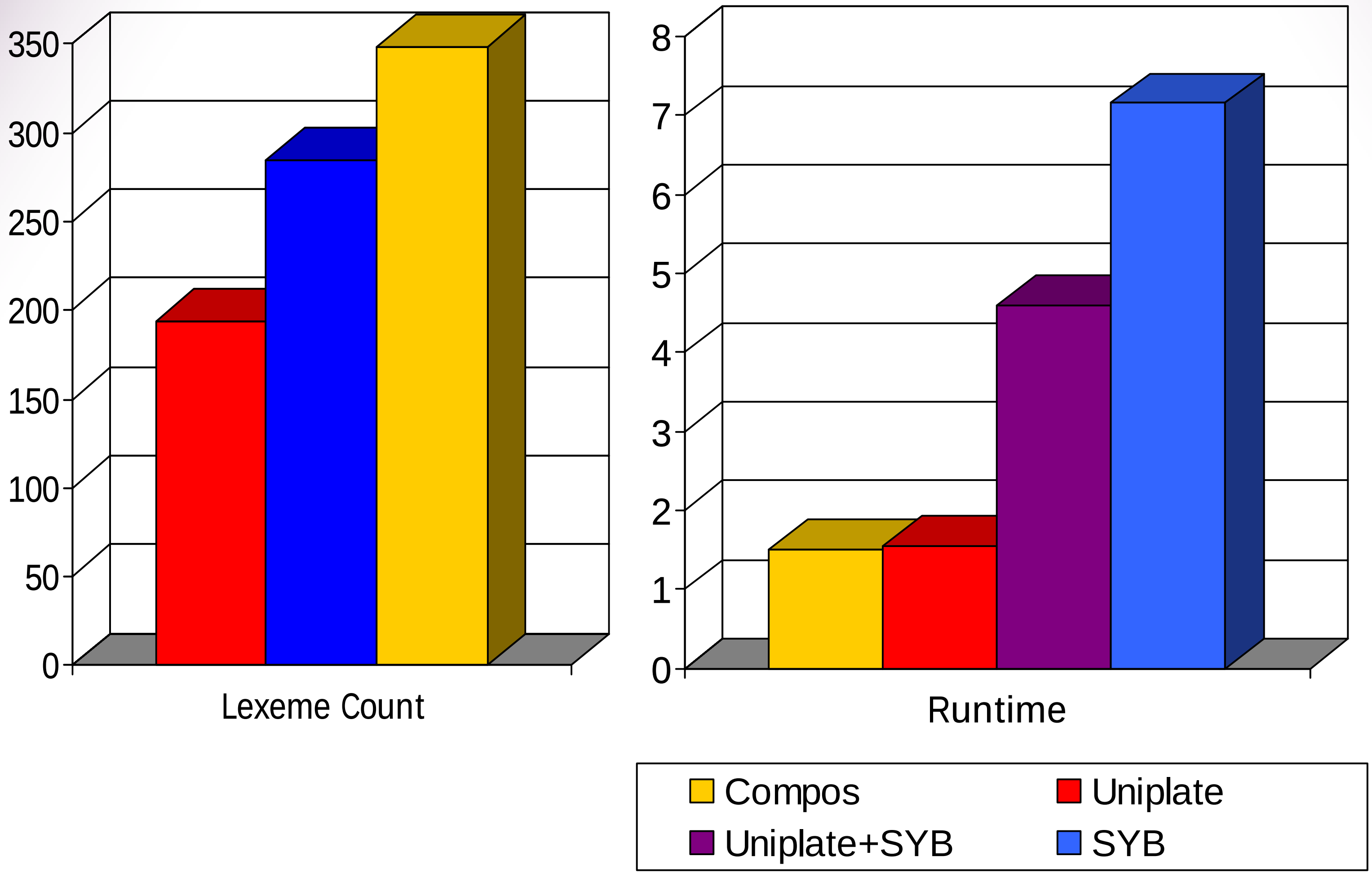
let incS k (S s) = S (s+k) in
transformBi (incS k)

let incS k (S s) = S (s+k) in
everywhere (mkT (incS k))

Uniplate Instances

1. Manual: Implemented directly
   – Can be generated using Data.Derive/TH
2. Direct: Using combinators
3. Typeable: Using Typeable class
4. Data: In terms of Data/Typeable
   – Using GHC this allows automatic deriving
   – Very simple to use

Benchmarks



350
300
250
200
150
100
50
0
Lexeme Count
8
7
6
5
4
3
2
1
0
Runtime
Compos
Uniplate
Uniplate+SYB
SYB

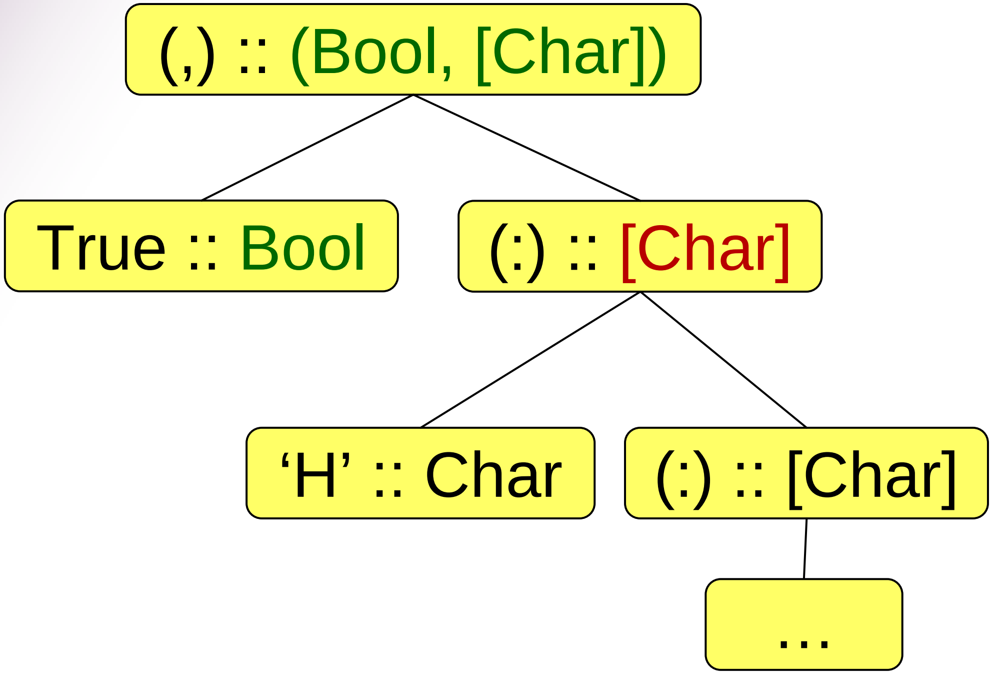
Outperforming SYB, 1

```
universe x = x : concatMap universe children
  where (children, context) = uniplate x
```

- A simple universe/everywhere is $O(n^2)$ in the depth of the value
- Can use continuation passing and foldr/build list fusion

Outperforming SYB, 2

- Operating on Bool in (True, “Haskell”)



- Uniplate knows the *target* type

(Bool, [Char])	Contains
Bool	Target
[Char]	Skip
Char	Skip

Computed with SYB
Stored in a CAF

Uniplate touches 3 components
SYB touches 16

Uniplate Applications

- Supero – program optimiser
- Catch – analysis tool (over 100 uses)
- Reach – another analysis tool
- Yhc/ycr2js – a compiler
- Reduceron – FPGA compiler
  - Lambda lifter in 12 lines
- Available on Hackage (go download it)

Conclusion

- Boilerplate should be scrapped
- We have focused on uniform traversals
- Disadvantage
  - Not always applicable
- Advantages
  - Simpler, more portable, no “scary types”, faster