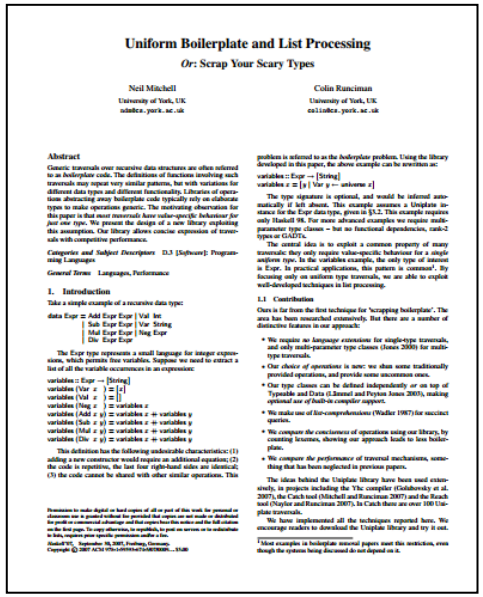
Uniform Boilerplate and List Processing
Or: Scrap Your Scary Types



Uniform Boilerplate and List Processing

Or: Scrap Your Scary Types

Neil Mitchell
University of York, UK
ndmitch.york.ac.uk

Colin Runciman
University of York, UK
colinc.york.ac.uk

Abstract

Generic traversals over recursive data structures are often referred to as *boilerplate* code. The definitions of functions involving such traversals may repeat very similar patterns, but with variations for different data types and different functionality. Libraries of operations abstracting away boilerplate code typically rely on elaborate types to make operations generic. The motivating observation for this paper is that *most traversals have value-specific behaviour for just one type*. We present the design of a new library exploiting this assumption. Our library allows concise expression of traversals with competitive performance.

Categories and Subject Descriptors D.3 [*Software*]: Programming Languages

General Terms Languages, Performance

1. Introduction

Take a simple example of a recursive data type:

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

The Expr type represents a small language for integer expressions, which permits free variables. Suppose we need to extract a list of all the variable occurrences in an expression:

```
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
```

This definition has the following undesirable characteristics: (1) adding a new constructor would require an additional equation; (2) the code is repetitive, the last four right-hand sides are identical; (3) the code cannot be shared with other similar operations. This problem is referred to as the *boilerplate* problem. Using the library developed in this paper, the above example can be rewritten as:

```
variables :: Expr → [String]
variables x = [x | Var y ← universe x]
```

The type signature is optional, and would be inferred automatically if left absent. This example assumes a Uniplate instance for the Expr data type, given in §3.2. This example requires only Haskell 98. For more advanced examples we require multi-parameter type classes – but no functional dependencies, rank-2 types or GADTs.

The central idea is to exploit a common property of many traversals: they only require value-specific behaviour for a *single* uniform *type*. In the variables example, the only type of interest is Expr. In practical applications, this pattern is common[1]. By focusing only on uniform type traversals, we are able to exploit well-developed techniques in list processing.

1.1 Contribution

Ours is far from the first technique for 'scrapping boilerplate'. The area has been researched extensively. But there are a number of distinctive features in our approach:

- We require *no language extensions* for single-type traversals, and only multi-parameter type classes (Jones 2000) for multi-type traversals.
- Our *choice of operations* is new: we shun some traditionally provided operations, and provide some uncommon ones.
- Our type classes can be defined independently *or* on top of Typeable and Data (Lämmel and Peyton Jones 2003), making *optional use of built-in compiler support*.
- We make use of *list-comprehensions* (Wadler 1987) for succinct queries.
- We *compare the conciseness* of operations using our library, by counting lexemes, showing our approach leads to less boilerplate.
- We *compare the performance* of traversal mechanisms, something that has been neglected in previous papers.

The ideas behind the Uniplate library have been used extensively, in projects including the Yhc compiler (Golubovsky et al. 2007), the Catch tool (Mitchell and Runciman 2007) and the Reach tool (Naylor and Runciman 2007). In Catch there are over 100 Uniplate traversals.

We have implemented all the techniques reported here. We encourage readers to download the Uniplate library and try it out.

[1] Most examples in boilerplate removal papers meet this restriction, even though the systems being discussed do not depend on it.

Haskell'07, September 30, 2007, Freiburg, Germany.
Copyright © 2007 ACM 978-1-59593-674-5/07/0009 ...$5.00

Neil Mitchell and

Colin Runciman,

Haskell Workshop, 2007

Simple generics (Haskell ’98)

Hutton’s Razor++

```
data Exp = Lit Int
         | Neg Exp
         | Add Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
         | Div Exp Exp
```

- What literals are in an expression?
- Change all Sub to Add/Neg?

Literals in an expression

```
literals (Lit i  ) = [i]
literals (Neg x  ) = literals x
literals (Add x y) = literals x ++ literals y
literals (Sub x y) = literals x ++ literals y
literals (Mul x y) = literals x ++ literals y
literals (Div x y) = literals x ++ literals y
```

Uniplate in action

- What literals are in an expression?

```
literals x = [i | Lit i <- universe x]
```

- Change all Sub to Add/Neg?

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

* Was called “Play” before Colin renamed it

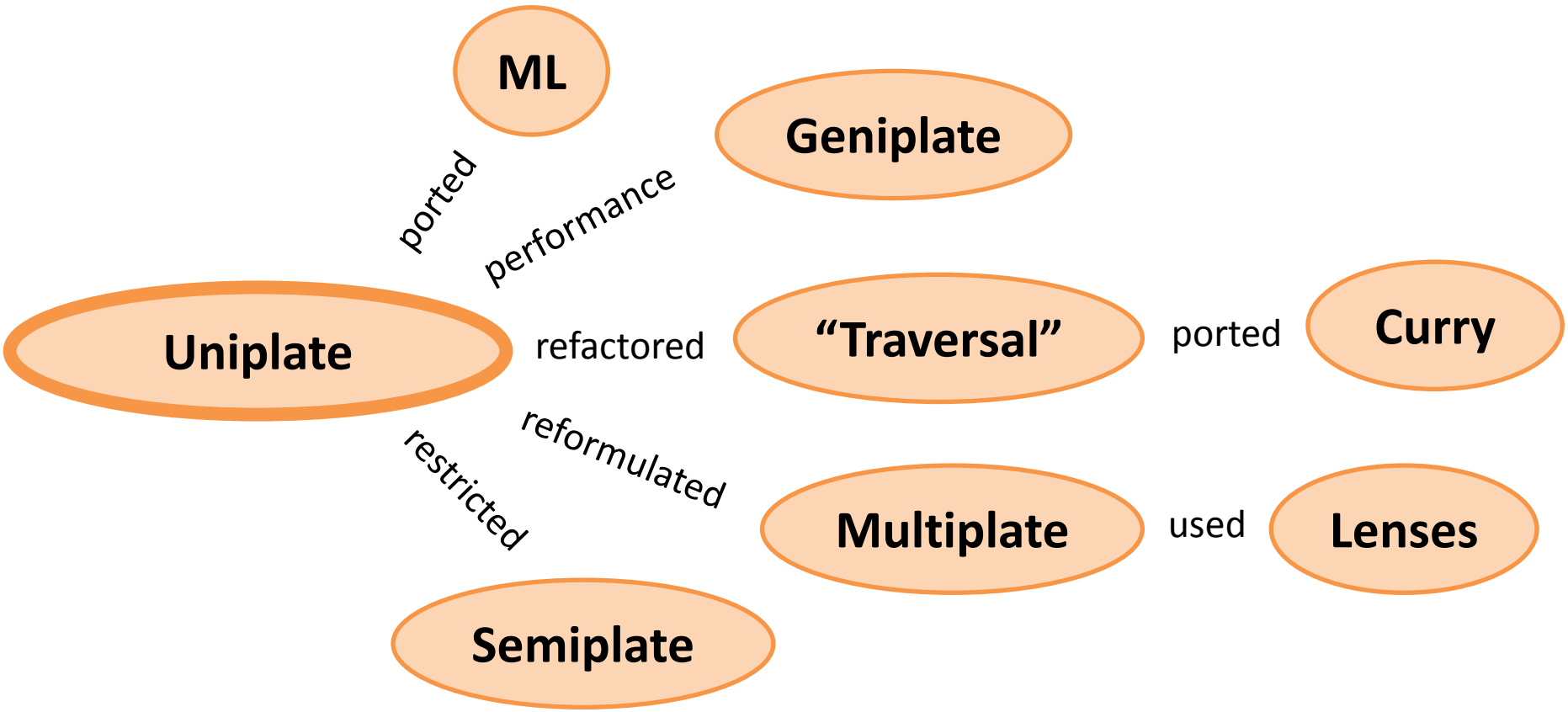
Simplicity of Haskell ‘98

```
class Uniplate a where
    uniplate :: a -> ([a], [a] -> a)

universe :: Uniplate a => a -> [a]
transform :: Uniplate a => (a -> a) -> a -> a
```

Compared to Scrap Your Boilerplate (SYB):

```
class Data a where
    gfoldl :: (forall d b. Data d => c (d -> b) -> c b)
           -> (forall g. g -> c g)
           -> a
           -> c a
```


ML
ported
Geniplate
performance
Uniplate
refactored
“Traversal”
ported
Curry
reformulated
restricted
Multiplate
used
Lenses
Semiplate

Applications (48 on Hackage)

- HLint – Haskell linting tool
- Reduceron – FPGA compiler
- Supero – Haskell optimiser
- Hoogle – Haskell search engine
- NSIS – Windows installer generator
- Scion – IDE backend
- Tamarin prover – Security theorem prover
- Codo notation – Comonad notation
- Yi – text editor
- …

Retrieving re-usable software components by polymorphic type

Colin Runciman and

Ian Toyn, JFP, 1991

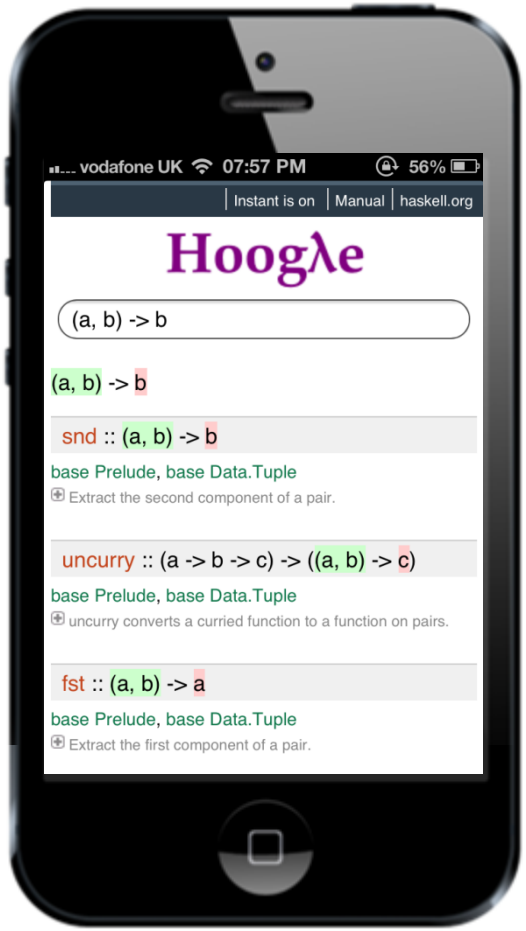
Let's define a type-based search engine!

Mikael Rittri, Using Types as Search Keys in Function Libraries. FPCA 1989

... recent developments in
so-called *hypertext systems* ...

Hoogle (2003-), Yahoo! (2007-)

- Web based, Haskell servers
- Name and type-based search
- Search 8,457 functions
  - vs 203 in 1991
- Many company-local copies
  - Instant reports if it goes down!
- Integrated in FP Complete IDE
  - People were paid to work on it



Hoogλe

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

- What should match?
- In what order?
- Not too slow…

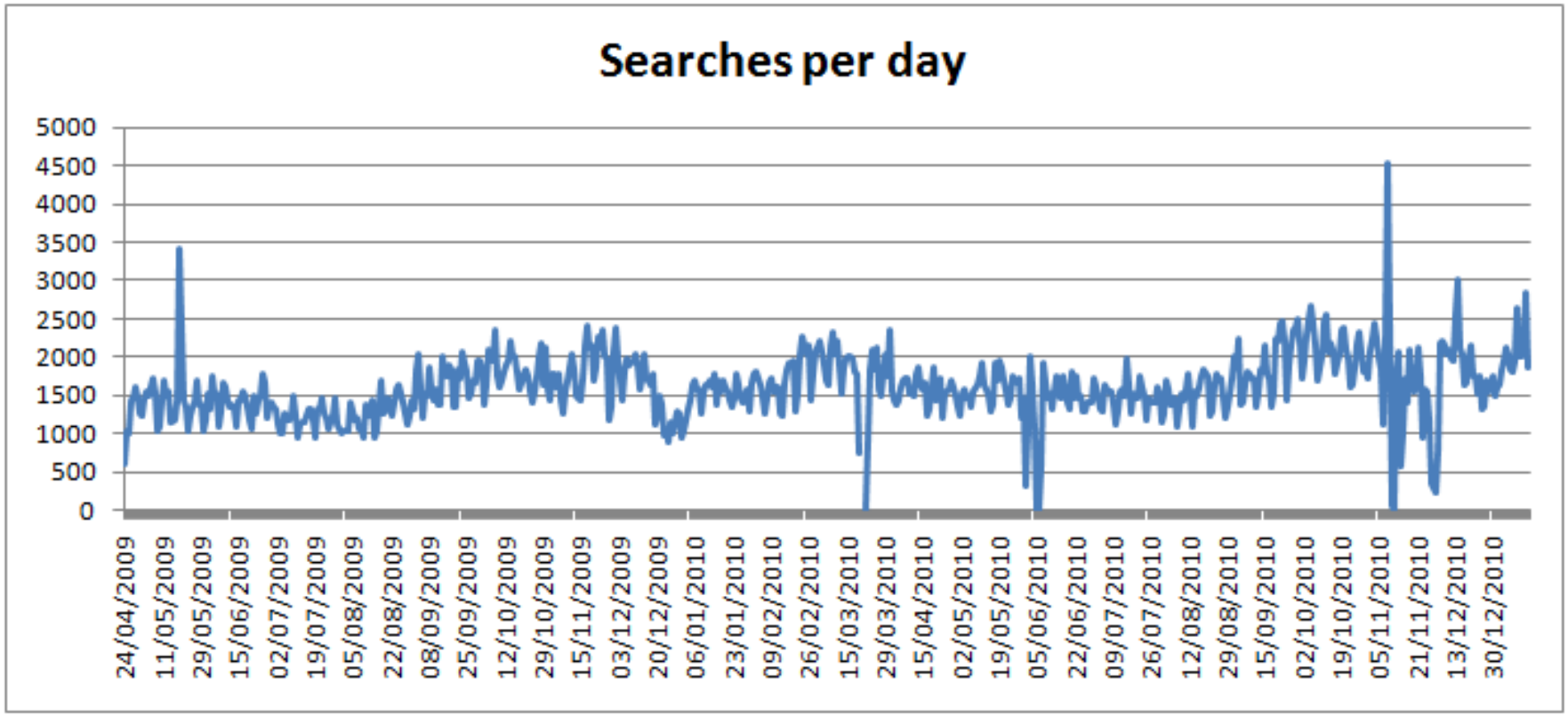
http://haskell.org/hoogle

Hoogλe

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

(a -> a -> a) -> [a] -> a	foldl1, foldr1
(a -> a -> a) -> [a] -> [a]	scanl1, scanr1
Foldable t => (a -> a -> a) -> t a -> a	foldl1, foldr1
(a -> a -> Ordering) -> [a] -> a	minimumBy
(a -> b -> a) -> a -> [b] -> a	foldl
(a -> b -> b) -> b -> [a] -> b	foldr
(a -> a -> Bool) -> [a] -> [a]	nubBy

Hoogle Usage



I would love to update this, but the log file is now 8.4Gb

~30 million searches since 2009

Funny Searches

- Colin Runciman
- :: Colin Runciman
- eastenders
- california public schools portable classes
- diem chuan truong dai hoc su pham ha noi 2008
- ebay consistency version
- videos pornos gratis
- Gia savores de BARILOCHE
- name of Peanuts cartoon bird