Uniform Boilerplate and List Processing

Jaap van der Plas

30 November, 2012
Uniform Boilerplate and List Processing

Or: Scrap Your Scary Types

Authors:
- Neil Mitchell
- Colin Runciman

Published in the Proceedings of the Haskell Workshop 2007.
Uniplate

Features and goals:

- Reduces boilerplate just like SYB
- Traversal with value-specific behaviour for one type
- Nice syntax using list comprehensions
- Works with Haskell 98, no language extensions required
Uniplate

Compared to SYB and other libraries:

- Simpler types
- Better performance
- Not as flexible
Uniplate

Originates from research in transformation and analysis of functional programs.

Used in:
- HLint
- derive
- 40 other packages
Example datatype

```
data Expr = Add Expr Expr           |
           Val Int                      |
           Sub Expr Expr               |
           Var String                  |
           Let String Expr Expr        |
           Mul Expr Expr               |
           Neg Expr                    |
           Div Expr Expr               |
```
Uniplate instance implementation

```
instance Uniplate Expr where
  uniplate (Add e1 e2) = plate Add |\* e1 |\* e2
  uniplate (Val i) = plate Val |− i
```

Which produces:

```
uniplate (Add e1 e2) = (children, context) where
  children = [e1, e2]
  context [e1, e2] = Add e1 e2
uniplate (Val i) = (children, context) where
  children = []
  context [] = Val i
```
Uniplate instance implementation

Continued...

uniplate (Sub e1 e2) = plate Sub |∗ e1 |∗ e2
uniplate (Mul a b) = plate Mul |∗ a |∗ b
uniplate (Div a b) = plate Div |∗ a |∗ b
uniplate (Neg e) = plate Neg |∗ e
uniplate (Let s e1 e2) = plate Let |− s |∗ e1 |∗ e2
uniplate x = plate x
Queries and transformation

\[\text{children :: Uniplate } a \Rightarrow a \rightarrow [a]\]
\[\text{children } = \text{fst } \circ \text{uniplate}\]

Return fields of the same type

\[\text{universe :: Uniplate } a \Rightarrow a \rightarrow [a]\]
\[\text{universe } x = x : \text{concatMap universe (children } x)\]

Find all elements of the same type (recursively)

**Example:** Find all divisions by zero

\[\text{countDivZero :: Expr } \rightarrow \text{ Int}\]
\[\text{countDivZero } x = \text{length } [(()] | \text{Div } _{_{}} \text{ (Val } 0) \leftarrow \text{universe } x)\]
Queries and transformation

transform :: Uniplate a ⇒ (a → a) → a → a
transform f x = f $ context $ map (transform f) children where
  (children, context) = uniplate x

Bottom-up transformation (recursively)

Example: Replace subtractions with additions:

simplify x = transform f x where
  f (Sub x y) = Add x (Neg y)
  f (Add x y) | x ≡ y = Mul (Val 2) x
  f x = x
Queries and transformation

descend :: Uniplate a ⇒ (a → a) → a → a

descend f x = context (map f children) where
  (children, context) = uniplate x

Top-down transformation. Not recursive!

**Example:** Replace variables (Let String Expr Expr) with expressions

subst :: [(String, Expr)] → Expr → Expr

subst rep x = case x of
  Let n bind x → Let n (subst rep bind) (subst (filter ((/= n) ∘ fst) rep) x)
  Var x → fromMaybe (Var x) (lookup x rep)
  _ → descend (subst rep) x
Queries and transformation

\[ \text{rewrite} :: \text{Uniplate } a \Rightarrow (a \rightarrow \text{Maybe } a) \rightarrow a \rightarrow a \]
\[ \text{rewrite } f = \text{transform } g \quad \text{where} \]
\[ g \ x = \text{maybe } x \ (\text{rewrite } f) \ (f \ x) \]

Exhaustive transformation, bottom-up.

**Example:** Extended `simplify`, now further simplifies code that was previously simplified:

\[ \text{simplify'} \ x = \text{rewrite } r \ x \quad \text{where} \]
\[ r \ (\text{Sub } x \ y) = \text{Just } \ (\text{Add } x \ (\text{Neg } y)) \]
\[ r \ (\text{Add } x \ y) \mid x \equiv y = \text{Just } \ (\text{Mul } (\text{Val } 2) \ x) \]
\[ r \ _ = \text{Nothing} \]
Queries and transformation

\[ \text{para :: Uniplate } a \Rightarrow (a \to [r] \to r) \to a \to r \]

\[ \text{para op x = op x} \mathbin{\&} \text{map (para op)} \mathbin{\&} \text{children x} \]

Paramorphism; combine the value and results of its children into a result.

**Example:** Calculate maximum depth of expression tree

\[ \text{depth :: Expr} \to \text{Int} \]

\[ \text{depth} = \text{para } (\lambda cs \to 1 + \text{maximum } (0 : cs)) \]
Queries and transformation

contexts :: Uniplate a ⇒ a → [(a, a → a)]
contexts x = (x, id) : f (holes x) where
   f xs = [(y, ctx ◦ context) | (child, ctx) ← xs, (y, context) ← contexts child]

Similar to universe, but also provides functions to rebuild the data structures that value was pulled from.

Example: Mutation testing, increment and decrement every literal

mutants :: Expr [Expr]
mutants x = [c (Val j) | (Val i, c) ← contexts x, j ← [i + 1, i + 1]]
type BiplateType b a = b → ([a], [a] → b)

universeOn :: Uniplate a ⇒ BiplateType b a → b → [a]

universeOn biplate x =
  concatMap universe $ fst $ biplate x

class Uniplate a ⇒ Biplate b a where
  biplate :: BiplateType b a

universeBi :: Biplate b a ⇒ b → [a]

universeBi = universeOn biplate

transformBi :: Biplate b a ⇒ (a → a) → b → b

transformBi = transformOn biplate
Biplate example

**data** Stmt = Assign String Expr | Sequence [Stmt] | If Expr Stmt Stmt | While Expr Stmt

**instance** Biplate Stmt Expr **where**

biplate (Assign a b) = plate Assign |− a |∗ b
biplate (Sequence a) = plate Sequence ||+ a
biplate (If a b c) = plate If |∗ a |+ b |+ c
biplate (While a b) = plate While |∗ a |+ b
Conclusions

The limitation to value-specific behaviour for one type has advantages:

- Simple and short function definitions and types
- Good performance
- Traversal through multiple types is possible with Biplate

| Table 1. Table of lexeme counts for solutions to the test problems using each of Uniplate, SYB and Compos. |
|--------------------------------------------------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
|         | simp   | var     | zero    | const   | ren     | sym     | bill    | incr    | incr1   | Query   | Transform | All      |
| Uniplate| 40     | 12      | 18      | 27      | 16      | 17      | 13      | 21      | 30      | 60      | 134      | 194      |
| SYB     | 43     | 29      | 29      | 30      | 19      | 34      | 21      | 24      | 56      | 113     | 172      | 285      |
| Compos  | 71     | 30      | 32      | 54      | 27      | 36      | 25      | 33      | 40      | 123     | 225      | 348      |

| Table 2. Table of timing results, expressed as multiples of the run-time for a hand-optimised version not using any traversal library. |
|--------------------------------------------------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
|         | simp   | var     | zero    | const   | ren     | sym     | bill    | incr    | incr1   | Query   | Transform | All      |
| Compos  | 1.34   | 1.17    | 1.74    | 1.28    | 1.22    | 1.30    | 2.49    | 1.52    | 1.57    | 1.68    | 1.39      | 1.51     |
| Uniplate Manual | 1.16 | 1.44    | 2.64    | 1.27    | 1.36    | 1.48    | 2.28    | 1.27    | 1.08    | 1.96    | 1.23      | 1.55     |
| Uniplate Direct | 1.22  | 1.61    | 3.28    | 1.21    | 1.18    | 1.38    | 2.35    | 1.19    | 1.16    | 2.15    | 1.19      | 1.62     |
| Uniplate Typeable | 1.43 | 2.09    | 4.81    | 1.42    | 1.37    | 2.63    | 5.86    | 1.53    | 1.53    | 3.85    | 1.46      | 2.52     |
| Uniplate Data | 2.30  | 4.64    | 12.70   | 1.84    | 1.89    | 3.60    | 10.70   | 2.07    | 1.69    | 7.91    | 1.96      | 4.60     |
| SYB     | 2.21   | 5.88    | 16.62   | 2.30    | 2.13    | 5.56    | 24.29   | 3.12    | 2.35    | 13.09   | 2.42      | 7.16     |
Future work

- Further speed improvements:
  - Continuation passing
  - List fusion
- More traversal functions