Uniform Boilerplate and List Processing

Or: Scrap Your Scary Types

Neil Mitchell
University of York, UK
ndm@cs.york.ac.uk

Colin Runciman
University of York, UK
colin@cs.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 = [y | 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.

It can be obtained from the website at `http://www.cs.york.ac.uk/~ndm/uniplate/`. A copy of the library has also been released, and is available on Hackage[2].

1.2 Road map

§2 introduces the traversal combinators that we propose, along with short examples. §3 discusses how these combinators are implemented in terms of a single primitive. §4 extends this approach to multi-type traversals, and §5 covers the extended implementation. §6 investigates some performance optimisations. §7 gives comparisons with other approaches, using examples such as the "paradise" benchmark. §8 presents related work, §9 makes concluding remarks and suggests directions for future work.

2. Queries and Transformations

We define various traversals, using the Expr type defined in the introduction as an example throughout. We divide *traversals* into two categories: queries and transformations. A *query* is a function that takes a value, and extracts some information of a different type. A *transformation* takes a value, and returns a modified version of the original value. All the traversals rely on the class Uniplate, an instance of which is assumed for Expr. The definition of this class and its instances are covered in §3.

2.1 Children

The first function in the Uniplate library serves as both a function, and a definition of terminology:

```
children :: Uniplate α ⇒ α → [α]
```

The function children takes a value and returns *all maximal proper substructures of the same type*. For example:

```
children (Add (Neg (Var "x")) (Val 12)) =
  [Neg (Var "x"), Val 12]
```

The children function is occasionally useful, but is used more commonly as an auxiliary in the definition of other functions.

2.2 Queries

The Uniplate library provides a the universe function to support queries.

```
universe :: Uniplate α ⇒ α → [α]
```

This function takes a data structure, and returns a list of *all* structures of the same type found within it. For example:

```
universe (Add (Neg (Var "x")) (Val 12)) =
  [Add (Neg (Var "x")) (Val 12)
  , Neg (Var "x")
  , Var "x"
  , Val 12]
```

One use of this mechanism for querying was given in the introduction. Using the universe function, queries can be expressed very concisely. Using a list-comprehension to process the results of universe is common.

Example 1

Consider the task of counting divisions by the literal 0.

```
countDivZero :: Expr → Int
countDivZero x = length [() | Div _ (Val 0) ← universe x]
```

Here we make essential use of a feature of list comprehensions: if a pattern does not match, then the item is skipped. In other syntactic constructs, failing to match a pattern results in a pattern-match error. □

[2] `http://hackage.haskell.org/`

2.3 Bottom-up Transformations

Another common operation provided by many boilerplate removal systems (Lämmel and Peyton Jones 2003; Visser 2004; Lämmel and Visser 2003; Ren and Erwig 2006) applies a given function to every subtree of the argument type. We define as standard a bottom-up transformation.

```
transform :: Uniplate α ⇒ (α → α) → α → α
```

The result of `transform f x` is `f x'` where x' is obtained by replacing each α-child x_i in x by `transform f` x_i.

Example 2

Suppose we wish to remove the Sub constructor assuming the equivalence: $x-y \equiv x+(-y)$. To apply this equivalence as a rewriting rule, at all possible places in an expression, we define:

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

This code can be read: apply the subtraction rule where you can, and where you cannot, do nothing. Adding additional rules is easy. Take for example: $x+y = 2*x$ **where** $x \equiv y$. Now we can add this new rule into our existing transformation:

```
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
```

Each equation corresponds to the natural Haskell translation of the rule. The transform function manages all the required boilerplate. □

2.4 Top-Down Transformation

The Scrap Your Boilerplate approach (Lämmel and Peyton Jones 2003) (known as SYB) provides a top-down transformation named everywhere′. We describe this traversal, and our reasons for *not* providing it, even though it could easily be defined. We instead provide descend, based on the composOp operator (Bringert and Ranta 2006).

The `everywhere' f` transformation applies f to a value, then recursively applies the transformation on all the children of the freshly generated value. Typically, the intention in a transformation is to apply f to *every node exactly once*. Unfortunately, `everywhere' f` does not necessarily have this effect.

Example 3

Consider the following transformation:

```
doubleNeg (Neg (Neg x)) = x
doubleNeg x             = x
```

The intention is clear: remove all instances of double negation. When applied in a bottom-up manner, this is the result. But when applied top-down some nodes are missed. Consider the value Neg (Neg (Neg (Neg (Val 1)))); only the outermost double negation will be removed. □

Example 4

Consider the following transformation:

```
reciprocal (Div n m) = Mul n (Div (Val 1) m)
reciprocal x         = x
```

This transformation removes arbitrary division, converting it to divisions where the numerator is always 1. If applied once to each subtree, this computation would terminate successfully. Unfortunately, top-down transformation treats the generated Mul as being transformed, but cannot tell that the generated Div is the result of a transformation, not a fragment of the original input. This leads to a non-termination error. □

As these examples show, when defining top-down transformations using everywhere′ it is easy to slip up. The problem is that the program cannot tell the difference between freshly created constructors, and values that come originally from the input.

So we do support top-down transformations, but require the programmer to make the transformation more explicit. We introduce the descend function, inspired by the Compos paper (Bringert and Ranta 2006).

```
descend :: Uniplate α ⇒ (α → α) → α → α
```

The result of descend f x is obtained by replacing each α-child x_i in x by f x_i. Unlike everywhere′, there is *no recursion* within descend.

Example 5

Consider the addition of a constructor Let String Expr Expr. Now let us define a function subst to replace free variables with given expressions. In order to determine which variables are free, we need to "remember" variables that are bound as we descend[3]. We can define subst using a descend transformation:

```
subst :: [(String, Expr)] → Expr → Expr
subst rep x =
  case x of
    Let name bind x → Let name (subst rep bind)
      (subst (filter ((≠ name) ∘ fst) rep) x)
    Var x → fromMaybe (Var x) (lookup x rep)
    _ → descend (subst rep) x
```

The Var alternative may return an Expr from rep, but no additional transformation is performed on this value, since all transformation is made explicit. In the Let alternative we explicitly continue the subst transformation. □

2.5 Transformations to a Normal Form

In addition to top-down and bottom-up transformations, we also provide transformations to a normal form. The idea is that a rule is applied exhaustively until a normal form is achieved. Consider a rewrite transformation:

```
rewrite :: Uniplate α ⇒ (α → Maybe α) → α → α
```

A rewrite-rule argument r takes an expression e of type α, and returns either Nothing to indicate that the rule is not applicable, or Just e' indicating that e is rewritten by r to e'. The intuition for rewrite r is that it applies r exhaustively; a postcondition for rewrite is that there must be no places where r could be applied. That is, the following property must hold:

```
propRewrite r x = all (isNothing ∘ r) (universe (rewrite r x))
```

One way to define the rewrite function uses transform:

```
rewrite :: Uniplate α ⇒ (α → Maybe α) → α → α
rewrite f = transform g
  where g x = maybe x (rewrite f) (f x)
```

This definition tries to apply the rule everywhere in a bottom-up manner. If at any point it makes a change, then the new value has the rewrite applied to it. The function only terminates when a normal form is reached.

A disadvantage of rewrite is that it may check unchanged subexpressions repeatedly. Performance sensitive programmers might prefer to use an explicit transformation, and manage the rewriting themselves. We show under which circumstances a bottom-up transformation obtains a normal form, and how any transformation can be modified to ensure a normal form.

2.5.1 Bottom-Up Transformations to a Normal Form

We define the function always that takes a rewrite rule r and produces a function appropriate for use with transform.

```
always :: (α → Maybe α) → (α → α)
always r x = fromMaybe x (r x)
```

What restrictions on r ensure that the property rewrite r $x \equiv$ transform (always r) x holds? It is sufficient that the constructors on the right-hand side of r do not overlap with the constructors on the left-hand side.

Example 2 (revisited)

Recall the simplify transformation, as a rewrite:

```
r (Sub x y)           = Just $ Add x (Neg y)
r (Add x y) | x ≡ y = Just $ Mul (Val 2) x
r _                   = Nothing
```

Here Add occurs on the right-hand side of the first line, and on the left-hand side of the second. From this we can construct a value where the two alternatives differ:

```
let x = Sub (Neg (Var "q")) (Var "q")

rewrite    r           x ≡ Mul (Val 2) (Var "q")
transform (always r) x ≡ Add (Var "q") (Neg (Var "q"))
```

To remedy this situation in the original simplify transformation, whenever the right-hand side introduces a new constructor, f may need to be reapplied. Here only one additional f application is necessary, the one attached to the construction of an Add value.

```
f (Sub x y)           = f $ Add x (Neg y)
f (Add x y) | x ≡ y = Mul (Val 2) x
f x                   = x
```
□

2.6 Action Transformations

Rewrite transformations apply a set of rules *repeatedly* until a normal form is found. One alternative is an action transformation, where each node is visited and transformed *once*, and state is maintained and updated as the operation proceeds. The standard technique is to thread a monad through the operation, which we do using transformM.

Example 6

Suppose we wish to rename each variable to be unique:

```
uniqueVars :: Expr → Expr
uniqueVars x = evalState (transformM f x) vars
  where
    vars = ['x' : show i | i ← [1..]]

    f (Var i) = do y : ys ← get
                   put ys
                   return (Var y)
    f x       = return x
```

The function transformM is a monadic variant of transform. Here a *state monad* is used to keep track of the list of names not yet

[3] For simplicity, we ignore issues of hygienic substitution that may arise if substituted expressions themselves contain free variables.

used, with evalState computing the result of the monadic action, given an initial state vars. □

2.7 Paramorphisms

A paramorphism is a fold in which the recursive step may refer to the recursive component of a value, not just the results of folding over them (Meertens 1992). We define a similar recursion scheme in our library.

```
para :: Uniplate α ⇒ (α → [r] → r) → α → r
```

The para function uses the functional argument to combine a value, and the results of para on its children, into a new result.

Example 7

Compiler writers might wish to compute the *depth of expressions*:

```
depth :: Expr → Int
depth = para (λ_ cs → 1+maximum (0 : cs))
```
□

2.8 Contexts

The final operation in the library seems to be a novelty – we have not seen it in any other generics library, even in those which attempt to include all variations (Ren and Erwig 2006). This operation is similar to contextual pattern matching (Mohnen 1996).[4]

```
contexts :: Uniplate α ⇒ α → [(α, α → α)]
```

This function returns lists of pairs (x, f) where x is an element of the data structure which would have been returned by universe, and f replaces the hole which x was removed from.

Example 8

Suppose that mutation testing requires all expressions obtained by incrementing or decrementing *any single* literal in an original expression.

```
mutants :: Expr → [Expr]
mutants x = [c (Val j) | (Val i, c) ← contexts x
                       , j ← [i−1, i+1]]
```
□

In general, contexts has the following properties:

```
propUniverse x = universe x ≡ map fst (contexts x)
propId       x = all (≡ x) [b a | (a, b) ← contexts x]
```

2.9 Summary

We present signatures for all our methods in Figure 1, including several monadic variants. In our experience, the most commonly used operations are universe and transform, followed by transformM and descend.

3. Implementing the Uniplate class

Requiring each instance of the Uniplate class to implement ten separate methods would be an undue imposition. Instead, given a type specific instance for a *single* auxiliary method with a pair as result, we can define *all ten* operations generically, at the class level. The auxiliary is:

```
uniplate :: Uniplate α ⇒ α → ([α], [α] → α)
uniplate x = (children, context)
```

The *children* are all the maximal proper substructures of the same type as x; the *context* is a function to generate a new value, with a different set of children. The caller of *context* must ensure

[4] This function was contributed by Eric Mertens.

```
module Data.Generics.Uniplate where

children    :: Uniplate α ⇒ α → [α]
contexts    :: Uniplate α ⇒ α → [(α, α → α)]
descend     :: Uniplate α ⇒ (α → α) → α → α
descendM    :: (Uniplate α, Monad m) ⇒
               (α → m α) → α → m α
para        :: Uniplate α ⇒ (α → [r] → r) → α → r
rewrite     :: Uniplate α ⇒ (α → Maybe α) → α → α
rewriteM    :: (Uniplate α, Monad m) ⇒
               (α → m (Maybe α)) → α → m α
transform   :: Uniplate α ⇒ (α → α) → α → α
transformM  :: (Uniplate α, Monad m) ⇒
               (α → m α) → α → m α
universe    :: Uniplate α ⇒ α → [α]
```

Figure 1. All Uniplate methods.

```
class Uniplate α where
  uniplate :: α → ([α], [α] → α)

instance Uniplate Expr where
  uniplate (Neg a  ) = ([a]   , λ[a']    → Neg a'   )
  uniplate (Add a b) = ([a, b], λ[a', b'] → Add a' b')
  uniplate (Sub a b) = ([a, b], λ[a', b'] → Sub a' b')
  uniplate (Mul a b) = ([a, b], λ[a', b'] → Mul a' b')
  uniplate (Div a b) = ([a, b], λ[a', b'] → Div a' b')
  uniplate x         = ([]    , λ[]     → x        )
```

Figure 2. The Uniplate class and an instance for Expr.

that the length of the list given to *context* is the same as the length of *children*. The result pair splits the information in the value, but by combining the *context* with the *children* the original value can be recovered:

```
propId x = x ≡ context children
  where (children, context) = uniplate x
```

3.1 Operations in terms of uniplate

All ten operations of §2 can be defined in terms of uniplate very concisely. We define four functions as examples.

```
children :: Uniplate α ⇒ α → [α]
children = fst ∘ uniplate

universe :: Uniplate α ⇒ α → [α]
universe x = x : concatMap universe (children x)

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

descend :: Uniplate α ⇒ (α → α) → α → α
descend f x = context $ map f children
  where (children, context) = uniplate x
```

The common pattern is to call uniplate, then operate on the current children, often calling *context* to create a modified value. Some of these definitions can be made more efficient – see §6.1.

3.2 Writing Uniplate instances

We define a Uniplate instance for the Expr type in Figure 2.

$\mathcal{D}[\![\textbf{data}\ d\ v_1 ... v_n = a_1 ... a_m]\!] =$
  $\mathcal{N}[\![d]\!]\ v_1 ... v_n\ x = \textbf{case}\ x\ \textbf{of}\ \mathcal{C}[\![a_1]\!]\ ...\ \mathcal{C}[\![a_m]\!]$
  where x is fresh

$\mathcal{C}[\![c\ t_1 ... t_n]\!] =$
  $c\ y_1 ... y_n \rightarrow \text{UNIT}\ c <\!\!> \mathcal{T}[\![t_1]\!]\ y_1 <\!\!> ... <\!\!> \mathcal{T}[\![t_n]\!]\ y_n$
  where $y_1 ... y_n$ are fresh

$\mathcal{T}[\![\mathsf{TargetType}\ ]\!] = \text{TARGET}$
$\mathcal{T}[\![\mathsf{PrimitiveType}]\!] = \text{UNIT}$
$\mathcal{T}[\![d\ t_1 ... t_n\ ]\!] = \mathcal{N}[\![d]\!]\ \mathcal{T}[\![t_1]\!]\ ...\ \mathcal{T}[\![t_n]\!]$
$\mathcal{T}[\![v\ ]\!] = v$

$\mathcal{N}$ is an injection to fresh variables

Figure 3. Derivation rules for Uniplate instances.

The distinguishing feature of our library is that the children are defined in terms of their type. While this feature keeps the traversals simple, it does mean that rules for *deriving* instance definitions are not purely syntactic, but depend on the types of the constructors. We now describe the derivation rules, followed by information on the DERIVE tool that performs this task automatically. (If we are willing to make use of Multi-Parameter Type Classes, simpler derivation rules can be used: see §5.)

3.3 Derivation Rules

We can define derivation rules for the *children* and *context* functions, allowing the definition:

```
instance Uniplate Type where
  uniplate x = (children x, context x)
```

Alternatively, it is possible to define one single function which generates both elements of the pair at once, avoiding the need to examine each value twice (see §6.2 for an example).

We model the derivation of an instance by describing a derivation from a data type to a set of declarations. The derivation rules have three functional parameters: $(<\!\!>)$, UNIT and TARGET. By varying these parameters we derive either *children* or *context* functions.

The derivation rules are given in Figure 3. The $\mathcal{D}$ rule takes a data type declaration, and defines a function over that data type. The $\mathcal{C}$ rule defines a case alternative for each constructor. The $\mathcal{T}$ rule defines type specific behaviour: a type is either the target type on which an instance is being defined, or a primitive such as Char, or an algebraic data type, or a free type variable.

Applying $\mathcal{D}$ to Expr, the result is:

$\mathcal{N}[\![\mathsf{Expr}]\!]\ x = \textbf{case}\ x\ \textbf{of}$
  $\mathsf{Val}\ y_1 \rightarrow \text{UNIT}\ \mathsf{Val} <\!\!> \text{UNIT}\ y_1$
  $\mathsf{Var}\ y_1 \rightarrow \text{UNIT}\ \mathsf{Var} <\!\!> \mathcal{N}[\![\mathsf{List}]\!]\ \text{UNIT}\ y_1$
  $\mathsf{Neg}\ y_1 \rightarrow \text{UNIT}\ \mathsf{Neg} <\!\!> \text{TARGET}\ y_1$
  $\mathsf{Add}\ y_1\ y_2 \rightarrow \text{UNIT}\ \mathsf{Add} <\!\!> \text{TARGET}\ y_1 <\!\!> \text{TARGET}\ y_2$
  $\mathsf{Sub}\ y_1\ y_2 \rightarrow \text{UNIT}\ \mathsf{Sub} <\!\!> \text{TARGET}\ y_1 <\!\!> \text{TARGET}\ y_2$
  $\mathsf{Mul}\ y_1\ y_2 \rightarrow \text{UNIT}\ \mathsf{Mul} <\!\!> \text{TARGET}\ y_1 <\!\!> \text{TARGET}\ y_2$
  $\mathsf{Div}\ y_1\ y_2 \rightarrow \text{UNIT}\ \mathsf{Div} <\!\!> \text{TARGET}\ y_1 <\!\!> \text{TARGET}\ y_2$

$\mathcal{N}[\![\mathsf{List}]\!]\ v_1\ x = \textbf{case}\ x\ \textbf{of}$
  $[\,] \rightarrow \text{UNIT}\ [\,]$
  $(:)\ y_1\ y_2 \rightarrow \text{UNIT}\ (:) <\!\!> v_1\ y_1 <\!\!> \mathcal{N}[\![\mathsf{List}]\!]\ v_1\ y_2$

3.3.1 Defining *children*

To derive the *children* function, the derivations are applied with the following parameter values.

$\text{UNIT} = \mathsf{const}\ [\,]$
$\text{TARGET} = (:[\,])$
$(<\!\!>) = (+\!\!+)$

$children\ x = \mathcal{N}[\![\mathsf{Type}]\!]\ x$

The generated function is a traversal which visits every value in the data type. A list is created of all the target types by placing the targets into lists, combining lists using $(+\!\!+)$, and skipping uninteresting values.

From these definitions we can do some reasoning. For example, $\mathsf{list} \equiv \mathsf{concatMap}$, and $\mathsf{concatMap}\ (\mathsf{const}\ [\,]) \equiv \mathsf{const}\ [\,]$. This information can be used to simplify some instances.

3.3.2 Defining *context*

For *context* functions we apply the derivation rules with the following parameter values.

$\textbf{type}\ \mathsf{Cont}\ t\ \alpha = [\alpha] \rightarrow (t, [\alpha])$

$\text{UNIT} :: t \rightarrow \mathsf{Cont}\ t\ \alpha$
$\text{UNIT}\ x\ ns = (x, ns)$

$\text{TARGET} :: \alpha \rightarrow \mathsf{Cont}\ \alpha\ \alpha$
$\text{TARGET}\ x\ (n : ns) = (n, ns)$

$(<\!\!>) :: \mathsf{Cont}\ (a \rightarrow b)\ \alpha \rightarrow \mathsf{Cont}\ a\ \alpha \rightarrow \mathsf{Cont}\ b\ \alpha$
$(<\!\!>)\ a\ b\ ns_1 = \textbf{let}\ (a', ns_2) = a\ ns_1$
$\qquad\qquad\qquad\qquad\ (b', ns_3) = b\ ns_2$
$\qquad\qquad\qquad\ \textbf{in}\ \ (a'\ b', ns_3)$

$context\ x\ ns = \mathsf{fst}\ (\mathcal{N}[\![\mathsf{Type}]\!]\ x\ ns)$

The central Cont type is an extension to the type of *context* which takes a list of children to substitute into a value, and returns both the new value, and the children which were not used. By returning the unused children the $(<\!\!>)$ operation is able to determine both the new value for a (namely a'), and the remaining list of children (namely ns_2), sequencing the use of the children. The TARGET function consumes a child, and the UNIT function returns the children unmodified.

3.4 Automated Derivation of uniplate

Applying these derivation rules is a form of boilerplate coding! The DrIFT tool (Winstanley 1997) derives instances automatically given rules depending only on the information contained in a type definition. However DrIFT is unable to operate with certain Haskell extensions (TEX style literate Haskell, C pre processor), and requires a separate pre-processing stage.

In collaboration with Stefan O'Rear we have developed the DERIVE tool (Mitchell and O'Rear 2007). DERIVE is based on Template Haskell (Sheard and Jones 2002) and has predefined rules for derivation of Uniplate instances. It has special rules to remove redundant patterns to produce simpler and more efficient instances.

Example 9

```
data Term = Name String
          | Apply Term [Term]
            deriving ( {-! Uniplate ! -} )
```

Running the DERIVE tool over this file, the generated code is:

```
instance Uniplate Term where
  uniplate (Name x1   ) = ([]     , λ_          → Name x1   )
  uniplate (Apply x1 x2) = (x1 : x2, λ(n : ns) → Apply n ns)
```

□

4. Multi-type Traversals

We have introduced the Uniplate class and an instance of it for type Expr. Now let us imagine that Expr is merely the expression type in a language with statements:

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

We could define a Uniplate instance for Stmt, and so perform traversals upon statements too. However, we may run into limitations. Consider the task of finding all literals in a Stmt – this requires boilerplate to find not just inner statements of type Stmt, but inner expressions of type Expr.

The Uniplate class takes a value of type α, and operates on its substructures of type α. What we now require is something that takes a value of type β, but operates on the children of type α within it – we call this class Biplate. Typically the type β will be a container of α. We can extend our operations by specifying how to find the α's within the β's, and then perform the standard Uniplate operations upon the α type. In the above example, α = Expr, and β = Stmt.

We first introduce UniplateOn, which requires an explicit function to find the occurrences of type α within type β. We then make use of Multi-parameter type classes (MPTC's) to generalise this function into a type class, named Biplate.

4.1 The UniplateOn Operations

We define operations, including universeOn and transformOn, which take an extra argument relative to the standard Uniplate operators. We call this extra argument biplate: it is a function from the containing type (β) to the contained type (α).

```
type BiplateType β α = β → ([α], [α] → β)
biplate :: BiplateType β α
```

The intuition for biplate is that given a structure of type β, the function should return the largest substructures in it of type α. If $\alpha \equiv \beta$ the original value should be returned:

```
biplateSelf :: BiplateType α α
biplateSelf x = ([x], λ[x'] → x')
```

We can now define universeOn and transformOn. Each takes a biplate function as an argument:

```
universeOn :: Uniplate α ⇒ BiplateType β α → β → [α]
universeOn biplate x =
  concatMap universe $ fst $ biplate x

transformOn :: Uniplate α
            ⇒ BiplateType β α → (α → α) → β → β
transformOn biplate f x =
  context $ map (transform f) children
  where (children, context) = biplate x
```

These operations are similar to the original universe and transform. They unwrap β values to find the α values within them, operate using the standard Uniplate operations for type α, then rewrap if necessary. If α is constant, there is another way to abstract away the biplate argument, as the following example shows.

Example 10

The Yhc.Core library (Golubovsky et al. 2007), part of the York Haskell Compiler (Yhc), makes extensive use of Uniplate. In this library, the central types include:

```
data Core     = Core String [String] [CoreData] [CoreFunc]

data CoreFunc = CoreFunc String String CoreExpr

data CoreExpr = CoreVar  String
              | CoreApp  CoreExpr [CoreExpr]
              | CoreCase CoreExpr [(CoreExpr, CoreExpr)]
              | CoreLet  [(String, CoreExpr)] CoreExpr
                  -- other constructors
```

Most traversals are performed on the CoreExpr type. However, it is often convenient to start from one of the other types. For example, coreSimplify::CoreExpr → CoreExpr may be applied not just to an individual expression, but to all expressions in a function definition, or a complete program. If we are willing to freeze the type of the second argument to biplate as CoreExpr we can write a class:

```
class UniplateExpr β where
    uniplateExpr :: BiplateType β CoreExpr

universeExpr   x = universeOn   uniplateExpr x
transformExpr x = transformOn uniplateExpr x

instance Uniplate CoreExpr
instance UniplateExpr Core
instance UniplateExpr CoreFunc
instance UniplateExpr CoreExpr
instance UniplateExpr β ⇒ UniplateExpr [β]
```

□

This technique has been used in the Yhc compiler. The Yhc compiler is written in Haskell 98 to allow for bootstrapping, so only the standard single-parameter type classes are available.

4.2 The Biplate class

If we are willing to make use of *multi-parameter type classes* (Jones 2000) we can define a class Biplate with biplate as its sole method. We do not require functional dependencies.

```
class Uniplate α ⇒ Biplate β α where
    biplate :: BiplateType β α
```

We can now implement universeBi and transformBi in terms of their On counterparts:

```
universeBi :: Biplate β α ⇒ β → [α]
universeBi = universeOn biplate

transformBi :: Biplate β α ⇒ (α → α) → β → β
transformBi = transformOn biplate
```

In general the move to Biplate requires few code changes, merely the use of the new set of Bi functions. To illustrate we give generalisations of two examples from previous sections, implemented using Biplate. We extend the variables and simplify functions to work on Expr, Stmt or many other types.

Example from §1 (revisited)

```
variables :: Biplate β Expr ⇒ β → [String]
variables x = [y | Var y ← universeBi x]
```

The equation requires only one change: the addition of the Bi suffix to universe. In the type signature we replace Expr with

Biplate β Expr $\Rightarrow \beta$. Instead of requiring the input to be an Expr, we merely require that from the input we know how to reach an Expr. □

Example 2 (revisited)

```
simplify :: Biplate β Expr ⇒ β → β
simplify x = transformBi f x
  where f (Sub x y) = Add x (Neg y)
        f x         = x
```

In this redefinition we have again made a single change to the equation: the addition of Bi at the end of transform. □

5. Implementing Biplate

The complicating feature of biplate is that when defining Biplate where $\alpha \equiv \beta$ the function does not descend to the children, but simply returns its argument. This "same type" restriction can be captured either using the type system, or using the Typeable class (Lämmel and Peyton Jones 2003). We present three methods for defining a Biplate instance – offering a trade-off between performance, compatibility and volume of code.

1. Direct definition requires $O(n^2)$ instances, but offers the highest performance with the fewest extensions.
2. The Typeable class can be used, requiring $O(n)$ instances and no further Haskell extensions, but giving worse performance.
3. The Data class can be used, providing fully automatic instances with GHC, but requiring the use of rank-2 types, and giving the worst performance.

All three methods can be fully automated using DERIVE, and all provide a simplified method for writing Uniplate instances. The first two methods require the user to define instances of auxiliary classes, PlateAll and PlateOne, on top of which the library defines the Uniplate and Biplate classes. The Biplate class definition itself is independent of the method used to implement its instances. This abstraction allows the user to start with the simplest instance scheme available to them, then move to alternative schemes to gain increased performance or compatibility.

5.1 Direct instances

Writing direct instances requires the Data.Generics.PlateDirect module to be imported. This style requires a maximum of n^2 instance definitions, where n is the number of types which contain each other, but gives the highest performance and most type-safety. The instances still depend on the type of each field, but are easier to define than the Uniplate instance discussed in §3.2. Here is a possible instance for the Expr type:

```
instance PlateOne Expr where
  plateOne (Neg a  ) = plate Neg |* a
  plateOne (Add a b) = plate Add |* a |* b
  plateOne (Sub a b) = plate Sub |* a |* b
  plateOne (Mul a b) = plate Mul |* a |* b
  plateOne (Div a b) = plate Div |* a |* b
  plateOne x         = plate x
```

Five infix combinators ($|*$, $|+$, $|-$, $||*$ and $||+$) indicate the structure of the field to the right. The $|*$ combinator says that the value on the right is of the target type, $|+$ says that a value of the target type *may occur* in the right operand, $|-$ says that values of the target type *cannot occur* in the right operand. $||*$ and $||+$ are versions of $|*$ and $|+$ used when the value to the right is a *list* either of the target type, or of a type that may contain target values. The law plate f $|-$ $x \equiv$ plate (f x) justifies the definition presented above.

This style of definition naturally expands to the multi-type traversal. For example:

```
instance PlateAll Stmt Expr where
  plateAll (Assign   a b  ) = plate Assign   |- a |* b
  plateAll (Sequence a    ) = plate Sequence ||+ a
  plateAll (If       a b c) = plate If       |* a |+ b |+ c
  plateAll (While    a b  ) = plate While    |* a |+ b
```

From the definitions of PlateOne and PlateAll the library can define Uniplate and Biplate instances. The information provided by uses of $|-$ and $|+$ avoids redundant exploration down branches that do not have the target type. The use of $||*$ is an optimisation which allows a list of the target type to be directly manipulated with biplate instead of producing and consuming this list twice. The use of $||+$ avoids the definition of additional instances.

In the worst case, this approach requires an instance for each container/contained pair. In reality few traversal pairs are actually needed. The restricted pairing of types in Biplate instances also gives increased type safety; instances such as Biplate Expr Stmt do not exist.

In our experience definitions using these combinators offer similar performance to hand-tuned instances; see §7.2 for measurements.

5.2 Typeable based instances

Instead of writing $O(n^2)$ class instances to locate values of the target type, we can use the Typeable class to *test at runtime* whether we have reached the target type. We present derivations much as before, based this time only on combinators $|+$ and $|-$:

```
instance (Typeable α, Uniplate α) ⇒ PlateAll Expr α where
  plateAll (Neg a  ) = plate Neg |+ a
  plateAll (Add a b) = plate Add |+ a |+ b
  plateAll (Sub a b) = plate Sub |+ a |+ b
  plateAll (Mul a b) = plate Mul |+ a |+ b
  plateAll (Div a b) = plate Div |+ a |+ b
  plateAll _         = plate x

instance (Typeable α, Uniplate α) ⇒ PlateAll Stmt α where
  plateAll (Assign   a b  ) = plate Assign   |- a |+ b
  plateAll (Sequence a    ) = plate Sequence |+ a
  plateAll (If       a b c) = plate If       |+ a |+ b |+ c
  plateAll (While    a b  ) = plate While    |+ a |+ b
```

The $|+$ combinator is the most common, denoting that the value on the right may be of the target type, or may contain values of the target type. However, if we were to use $|+$ when the right-hand value was an Int, or other primitive type we did not wish to examine, we would require a PlateAll definition for Int. To omit these unnecessary instances, we can use $|-$ to indicate that the type is not of interest.

The Data.Generics.PlateTypeable module is able to automatically infer Biplate instances given a PlateAll instance. Alas this is not the case for Uniplate. Instead we must explicitly declare:

```
instance Uniplate Expr where
  uniplate = uniplateAll

instance Uniplate Stmt where
  uniplate = uniplateAll
```

The reader may wonder why we cannot define:

```
instance PlateAll α α ⇒ Uniplate α where
  uniplate = uniplateAll
```

```
repChildren :: (Data α, Uniplate β, Typeable α, Typeable β)
            ⇒ α → ([β], [β] → α)
repChildren x = (children, context)
  where
    children = concat $ gmapQ (fst ∘ biplate) x

    context xs = evalState (gmapM f x) xs
    f y = do let (cs, con) = biplate y
             (as, bs) ← liftM (splitAt $ length cs) get
             put bs
             return $ con as
```

Figure 4. Code for Uniplate in terms of Data.

Consider the Expr type. To infer Uniplate Expr we require an instance for PlateAll Expr Expr. But to infer this instance we require Uniplate Expr – which we are in the process of inferring! [5]

5.3 Using the Data class

The existing Data and Typeable instances provided by the SYB approach can also be used to define Uniplate instances:

```
import Data.Generics
import Data.Generics.PlateData

data Expr = ... deriving (Typeable, Data)
data Stmt = ... deriving (Typeable, Data)
```

The disadvantages of this approach are (1) *lack of type safety* – there are now Biplate instances for many pairs of types where one is not a container of the other; (2) *compiler dependence* – it will only work where Data.Generics is supported, namely GHC at the time of writing.[6] The clear advantage is that there is almost no work required to create instances.

How do we implement the Uniplate class instances? The fundamental operation is given in Figure 4. The repChildren function descends to each of the child nodes, and is guarded by a Typeable cast to ensure that $\alpha \not\equiv \beta$. The operation to get the children can be done using gmapQ. The operation to replace the children is more complex, requiring a state monad to keep track of the items to insert.

The code in Figure 4 is not optimised for speed. Uses of splitAt and length require the list of children to be traversed multiple times. We discuss improvements in §6.2.

6. Performance Improvements

This section describes some of the performance improvements we have been able to make. First we focus on our optimisation of universe, using continuation passing and some foldr/build fusion properties (Peyton-Jones et al. 2001). Next we turn to our Data class based instances, improving them enough to outperform SYB itself.

6.1 Optimising the universe function

Our initial universe implementation was presented in §3.1 as:

```
universe :: Uniplate on ⇒ on → [on]
universe x = x : concatMap universe (children x)
```

A disadvantage is that concatMap produces and consumes a list at every level in the data structure. We can fix this by using continuations:

```
universe x = f x []
  where f :: Uniplate on ⇒ on → [on] → [on]
        f x rest = x : concatCont (map f $ children x) rest

concatCont [] rest       = rest
concatCont (x : xs) rest = x (concatCont xs rest)
```

Now we only perform one reconstruction. We can do even better using GHC's list fusion (Peyton-Jones et al. 2001). The user of universe is often a list comprehension, which is a *good consumer*. We can make concatCont a good consumer, and f a *good producer*:

```
universe :: Uniplate on ⇒ on → [on]
universe x = build (f x)
  where
  f :: Uniplate on ⇒ on → (on → res → res) → res → res
  f x cons nil = x `cons`
    concatCont (map (flip f cons) $ children x) nil

concatCont xs rest = foldr ($) rest xs
```

6.2 Optimising PlateData

Surprisingly, it is possible to layer Uniplate over the Data instances of SYB, with better performance than SYB itself. The first optimisation is to generate the two members of the uniplate pair with only one pass over the data value. We cannot use SYB's gmapM or gmapQ – we must instead use gfoldl directly. We also make use of continuation passing style in some places.

With this first improvement in place we perform much the same operations as SYB. But the overhead of list creation in uniplate makes traversals about 15% slower than SYB.

The next optimisation relies on the extra information present in the Uniplate operations – namely the target type. A boilerplate operation walks over a data structure, looking for target values to process. In SYB, the target values may be of *any* type. For Uniplate the target is a *single uniform* type. If a value is reached which is not a container for the target type, no further exploration is required of the values children. Computing which types are containers for the target type can be done relatively easily in the SYB framework (Lämmel and Peyton Jones 2004):

```
data DataBox = ∀ α • (Typeable α, Data α) ⇒ DataBox α

contains :: (Data α, Typeable α) ⇒ α → [DataBox]
contains x = if isAlgType dtyp then concatMap f ctrs else []
  where
    f c = gmapQ DataBox (asTypeOf (fromConstr c) x)
    ctrs = dataTypeConstrs dtyp
    dtyp = dataTypeOf x
```

The contains function takes a *phantom* argument x which is never evaluated. It returns all the fields of all possible constructors of x's type, along with a type representation from typeOf. Hence all types can be divided into three sets:

1. The singleton set containing the type of the target.
2. The set of other types which *may* contain the target type.
3. The set of other types which *do not* contain the target type.

We compute these sets for each type only once, by using a CAF inside the class to store it. The cost of computing them is small. When examining a value, if its type is a member of set 3 we can prune the search. This trick is surprisingly effective. Take for

[5] GHC has co-inductive or recursive dictionaries, but Hugs does not. To allow continuing compatibility with Hugs, and the use of fewer extensions, we require the user to write these explicitly for each type.

[6] Hugs supports the required rank-2 types for Data.Generics, but the work to port the library has not been done yet.

example an operation over Bool on the value (True, "Haskell"). The SYB approach finds 16 subcomponents, Uniplate touches only 3 subcomponents.

With all these optimisations we can usually perform both queries and transformations faster than SYB. In the benchmarks we range from 4% worse to 127% better, with an average of 56% faster. Full details are presented in §7.2.

7. Results and Evaluation

We evaluate our boilerplate reduction scheme in two ways: firstly by the *conciseness of traversals* using it (i.e. the amount of boilerplate it removes), and secondly by its *runtime performance*. We measure conciseness by counting lexemes – although we concede that some aspects of concise expression may still be down to personal preference. We give a set of nine example programs, written using Uniplate, SYB and Compos operations. We then compare both the conciseness and the performance of these programs. Other aspects, such as the clarity of expression, are not so easily measured. Readers can make their own assessment based on the full sources we give.

7.1 Boilerplate Reduction

As test operations we have taken the first three examples from this paper, three from the Compos paper (Bringert and Ranta 2006), and the three given in the SYB paper (Lämmel and Peyton Jones 2003) termed the "Paradise Benchmark". In all cases the Compos, SYB and Uniplate functions are given an appropriately prefixed name. In some cases, a helper function can be defined in the same way in both SYB and Uniplate; where this is possible we have done so. Type signatures are omitted where the compiler is capable of inferring them. For SYB and Compos we have used definitions from the original authors where available, otherwise we have followed the guidelines and style presented in the corresponding paper.

7.1.1 Examples from this Paper

Example from §1 (revisited)

```
uni_variables x = [y | Var y ← universe x]

syb_variables = everything (++) ([] `mkQ` f)
  where f (Var y) = [y]
        f _       = []

com_variables :: Expr a → [String]
com_variables x = case x of
  Var y → [y]
  _ → composOpFold [] (++) com_variables x
```

Only Compos needs a type signature, due to the use of GADTs. List comprehensions allow for succinct queries in Uniplate. □

Example 1 (revisited)

```
uni_zeroCount x = length [() | Div _ (Val 0) ← universe x]

syb_zeroCount = everything (+) (0 `mkQ` f)
  where f (Div _ (Val 0)) = 1
        f _               = 0

com_zeroCount :: Expr a → Int
com_zeroCount x = case x of
  Div y (Val 0) → 1+com_zeroCount y
  _ → composOpFold 0 (+) com_zeroCount x
```

In the Uniplate solution the list of () is perhaps inelegant. However, Uniplate is the only scheme that is able to use the standard length function: the other two express the operation as a fold. Compos requires additional boilerplate to continue the operation on Div y. □

```
data Stm = SDecl  Typ Var | SAss    Var Exp
         | SBlock [Stm]   | SReturn Exp
data Exp = EStm Stm       | EAdd Exp Exp
         | EVar  Var      | EInt  Int
data Var = V String
data Typ = T_int          | T_float
```

Figure 5. Data type from Compos.

Example 2 (revisited)

```
simp (Sub x y)           = simp $ Add x (Neg y)
simp (Add x y) | x ≡ y = Mul (Val 2) x
simp x                   = x

uni_simplify = transform simp

syb_simplify = everywhere (mkT simp)

com_simplify :: Expr a → Expr a
com_simplify x = case x of
  Sub a b → com_simplify $
    Add (com_simplify a) (Neg (com_simplify b))
  Add a b → case (com_simplify a, com_simplify b) of
              (a', b') | a' ≡ b'   → Mul (Val 2) a'
                       | otherwise → Add a' b'
  _ → composOp com_simplify x
```

This is a modified version of simplify discussed in §2.5.1. The two rules are applied everywhere possible. Compos does not provide a bottom-up transformation, so needs extra boilerplate. □

7.1.2 Multi-type examples from the Compos paper

The statement type manipulated by the Compos paper is given in Figure 5. The Compos paper translates this type into a GADT, while Uniplate and SYB both accept the definition as supplied.

As the warnAssign function from the Compos paper could be implemented much more neatly as a query, rather than a monadic fold, we choose to ignore it. We cover the remaining three functions.

Example 11 (rename)

```
ren (V x) = V ("_" ++ x)

uni_rename = transformBi ren

syb_rename = everywhere (mkT ren)

com_rename :: Tree c → Tree c
com_rename t = case t of
  V x → V ("_" ++ x)
  _ → composOp com_rename t
```

The Uniplate definition is the shortest, as there is only one constructor in type Var. As Compos redefines all constructors in one GADT, it cannot benefit from this knowledge. □

Example 12 (symbols)

```
uni_symbols x = [(v, t) | SDecl t v ← universeBi x]
```

Table 1. Table of lexeme counts for solutions to the test problems using each of Uniplate, SYB and Compos.

	simp	var	zero	const	ren	syms	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	syms	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

```
type Manager  = Employee
type Name     = String
type Address  = String
data Company  = C [Dept]
data Dept     = D Name Manager [Unit]
data Unit     = PU Employee | DU Dept
data Employee = E Person Salary
data Person   = P Name Address
data Salary   = S Integer
```

Figure 6. Paradise Benchmark data structure.

```
syb_symbols = everything (++) ([] `mkQ` f)
  where f (SDecl t v) = [(v, t)]
        f _           = []

com_symbols :: Tree c → [(Tree Var, Tree Typ)]
com_symbols x = case x of
  SDecl t v → [(v, t)]
  _ → composOpMonoid com_symbols x
```

Whereas the Compos solution explicitly manages the traversal, the Uniplate solution is able to use the built-in universeBi function. The use of lists again benefits Uniplate over SYB. □

Example 13 (constFold)

```
optimise (EAdd (EInt n) (EInt m)) = EInt (n+m)
optimise x = x

uni_constFold = transformBi optimise

syb_constFold = everywhere (mkT optimise)

com_constFold :: Tree c → Tree c
com_constFold e = case e of
  EAdd x y → case (com_constFold x, com_constFold y) of
               (EInt n, EInt m) → EInt (n+m)
               (x', y') → EAdd x' y'
  _ → composOp com_constFold e
```

The constant-folding operation is a bottom-up transformation, requiring all subexpressions to have been transformed before an enclosing expression is examined. Compos only supports top-down transformations, requiring a small explicit traversal in the middle. Uniplate and SYB both support bottom-up transformations. □

7.1.3 The Paradise Benchmark from SYB

The Paradise benchmark was introduced in the SYB paper (Lämmel and Peyton Jones 2003). The data type is shown in Figure 6. The idea is that this data type represents an XML file, and a Haskell program is being written to perform various operations over it. The Compos paper includes an encoding into a GADT, with tag types for each of the different types.

We have made one alteration to the data type: Salary is no longer of type Float but of type Integer. In various experiments we found that the rounding errors for floating point numbers made different definitions return different results.[7] This change is of no consequence to the boilerplate code.

Example 14 (increase)

The first function discussed in the SYB paper is increase. This function increases every item of type Salary by a given percentage. In order to fit with our modified Salary data type, we have chosen to increase all salaries by k.

```
incS k (S s) = S (s+k)

uni_increase k = transformBi (incS k)

syb_increase k = everywhere (mkT (incS k))

com_increase :: Integer → Tree c → Tree c
com_increase k c = case c of
  S s → S (s+k)
  _ → composOp (com_increase k) c
```

In the Compos solution all constructors belong to the same GADT, so instead of just matching on S, all constructors must be examined. □

Example 15 (incrOne)

The incrOne function performs the same operation as increase, but only within a named department. The one subtlety is that if the named department has a sub-department with the same name, then the salaries of the sub-department should only be increased once. We are able to reuse the increase function from the previous section in all cases.

```
uni_incrOne d k = descendBi f
  where f x@(D n _ _) | n ≡ d     = uni_increase k x
                      | otherwise = descend f x
```

[7] Storing your salary in a non-exact manner is probably not a great idea!

```
syb_incrOne :: Data a ⇒ Name → Integer → a → a
syb_incrOne d k x | isDept d x = syb_increase k x
                  | otherwise  = gmapT (syb_incrOne d k) x
  where isDept   d = False `mkQ` isDeptD d
        isDeptD d (D n _ _) = n ≡ d

com_incrOne :: Name → Integer → Tree c → Tree c
com_incrOne d k x = case x of
  D n _ _ | n ≡ d → com_increase k x
  _ → composOp (com_incrOne d k) x
```

The SYB solution has grown substantially more complex, requiring two different utility functions. In addition syb_incrOne now *requires* a type signature. Compos retains the same structure as before, requiring a case to distinguish between the types of constructor. For Uniplate we use descend rather than transform, to ensure no salaries are incremented twice. □

Example 16 (salaryBill)

The final function is one which sums all the salaries.

```
uni_salaryBill x = sum [s | S s ← universeBi x]

syb_salaryBill = everything (+) (0 `mkQ` billS)
  where billS (S s) = s

com_salaryBill :: Tree c → Integer
com_salaryBill x = case x of
  S s → s
  _ → composOpFold 0 (+) com_salaryBill x
```

Here the Uniplate solution wins by being able to use a list comprehension to select the salary value out of a Salary object. The Uniplate class is the only one that is able to use the standard Haskell sum function, not requiring an explicit fold. □

7.1.4 Uniplate compared to SYB and Compos

In order to measure conciseness of expression, we have taken the code for all solutions and counted the number of lexemes – using the lex function provided by Haskell. A table of results is given in Table 1. The definitions of functions shared between SYB and Uniplate are included in both measurements. For the incrOne function we have not included the code for increase as well.

The Compos approach requires much more residual boilerplate than Uniplate, particularly for queries, bottom-up transformations and in type signatures. The Compos approach also requires a GADT representation.

Compared with SYB, Uniplate seems much more similar. For queries, Uniplate is able to make use of list comprehensions, which produces shorter code and does not require encoding a manual fold over the items of interest. For transformations, typically both are able to use the same underlying operation, and the difference often boils down to the mkT wrappers in SYB.

7.2 Runtime Overhead

This section compares the speed of solutions for the nine examples given in the previous section, along with hand-optimised versions, using no boilerplate removal library. We use four Uniplate instances, provided by:

Manual: These are Uniplate and Biplate instances written by hand. We have chosen not to use continuation-passing to implement these instances, as it quickly becomes complex!

Direct: These instances use the direct combinators from §5.1.

Typeable: These instances use the Typeable combinators from §5.2.

Data: These instances use the SYB Data instances directly, as described in §5.3.

For all data types we generate 100 values at random using QuickCheck (Claessen and Hughes 2000). In order to ensure a fair comparison, we define one data type which is the same as the original, and one which is a GADT encoding. All operations take these original data types, transform them into the appropriate structure, apply the operation and then unwrap them. We measure all results as multiples of the time taken for a hand-optimised version. We compiled all programs with GHC 6.6 and -O2 on Windows XP.

The results are presented in Table 2. Using Manual or Direct instances, Uniplate is roughly the same speed as Compos – but about 50% slower than hand-optimised versions. Using the Data instances provided by SYB, we are able to outperform SYB itself! See §6 for details of some of the optimisations used.

8. Related Work

The Uniplate library is intended to be a way to remove the boilerplate of traversals from Haskell programs. It is far from the first library to attempt boilerplate removal.

The SYB library (Lämmel and Peyton Jones 2003) is perhaps the most popular boilerplate removal system in Haskell. One of the reasons for its success is tight integration with the GHC compiler, lowering the barrier to use. We have compared directly against traversals written in SYB in §7.1, and have also covered how to implement Uniplate in terms of SYB in §5.3. In our experience most operations are shorter and simpler than the equivalents in SYB, and we are able to operate without the extension of rank-2 types. Most of these benefits stem directly from our definition of children as being the children of the same uniform type, contrasting with the SYB approach of all direct children.

The SYB library is, however, more powerful than Uniplate. If you wish to visit values of different type in a single traversal, Uniplate is unsuitable. The Data and Typeable methods have also been pushed further in successive papers (Lämmel and Peyton Jones 2004, 2005) – in directions Uniplate may be unable to go.

The Compos library (Bringert and Ranta 2006) is another approach to the removal of boilerplate, requiring GADTs (Peyton Jones et al. 2006) along with rank-2 types. The Compos library requires an existing data type to be rewritten as a GADT. The conversion from standard Haskell data structures to GADTs currently presents several problems: they are GHC specific, deriving is not supported on GADTs, and GADTs require explicit type signatures. The Compos approach is also harder to write instances for, having no simple instance generation framework, and no automatic derivation tool (although one could be written). The inner composOp operator is very powerful, and indeed we have chosen to replicate it in our library as descend. But the Compos library is unable to replicate either universe or transform from our library.

The Stratego tool (Visser 2004) provides support for generic operations, focusing on both the operations and the strategies for applying them. This approach is performed in an *untyped* language, although a typed representation can be modelled (Lämmel 2003). Rather than being a Haskell library, Stratego implements a domain specific language that can be integrated with Haskell.

The Strafunski library (Lämmel and Visser 2003; Lämmel 2002) has two aspects: generic transformations and queries for trees of any type; and features to integrate components into a larger programming system. Generic operations are performed using strategy combinators which can define special case behaviour for particular types, along with a default to perform in other situations. The

Strafunski library is integrated with Haskell, primarily providing support for generic programming in application areas that involve traversals over large abstract syntax trees.

The Applicative library (McBride and Paterson 2007) works by threading an Applicative operation through a data structure, in a similar way to threading a Monad through the structure. There is additionally a notion of Traversable functor, which can be used to provide generic programming. While the Applicative library can be used for generic programming, this task was not its original purpose, and the authors note they have "barely begun to explore" its power as a generic toolkit.

Generic Programming There are a number of other libraries which deal with generic programming, aimed more at writing *type generic* (or *polytypic*) functions, but which can be used for boilerplate removal. The *Haskell generics suite*[8] showcases several approaches (Weirich 2006; Hinze 2004; Hinze and Jeuring 2003).

9. Conclusions and Future Work

We have presented the Uniplate library. It defines the classes Uniplate and Biplate, along with a small set of operations to perform queries and transformations. We have illustrated by example that the boilerplate required in our system is less than in others (§7.1), and that we can achieve these results without sacrificing speed (§7.2). Our library is both practical and portable, finding use in a number of applications, and using fewer extensions to the Haskell language than alternatives.

The restriction to a uniformly typed value set in a traversal allows the power of well-developed techniques for list processing such as list-comprehensions to be exploited. We feel this decision plays to Haskell's strengths, without being limiting in practice.

There is scope for further speed improvements: for example, use of continuation passing style may eliminate tuple construction and consumption, and list fusion may be able to eliminate some of the intermediate lists in uniplate. We have made extensive practical use of the Uniplate library, but there may be other traversals which deserve to be added.

The use of boilerplate reduction strategies in Haskell is not yet ubiquitous, as we feel it should be. We have focused on simplicity throughout our design, working within the natural typed design of Haskell, rather than trying to extend it. Hopefully the removal of complicated language features (particularly 'scary' types) will allow a wider base of users to enjoy the benefits of boilerplate-free programming.

Acknowledgments

The first author is a supported by an EPSRC PhD studentship. Thanks to Björn Bringert, Jules Bean and the anonymous reviewers for feedback on an earlier drafts of this paper; Eric Mertens for helpful ideas; and Stefan O'Rear for work on DERIVE.

References

Björn Bringert and Aarne Ranta. A pattern for almost compositional functions. In *Proc. ICFP '06*, pages 216–226. ACM Press, 2006.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. ICFP '00*, pages 268–279. ACM Press, 2000.

Dimitry Golubovsky, Neil Mitchell, and Matthew Naylor. Yhc.Core - from Haskell to Core. *The Monad.Reader*, (7):45–61, April 2007.

Ralf Hinze. Generics for the masses. In *Proc. ICFP '04*, pages 236–243. ACM Press, 2004. ISBN 1-58113-905-5.

Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and theory. In *Summer School on Generic Programming*, volume 2793 of *LNCS*, pages 1–56. Springer-Verlang, 2003.

Mark P. Jones. Type classes with functional dependencies. In *Proc ESOP '00*, volume 1782 of *LNCS*, pages 230–244. Springer-Verlang, 2000.

R. Lämmel and J. Visser. A Strafunski Application Letter. In *Proc. PADL'03*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.

Ralf Lämmel. The sketch of a polymorphic symphony. In *Proc. of International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002)*, volume 70 of *ENTCS*. Elsevier Science, 2002.

Ralf Lämmel. Typed generic traversal with term rewriting strategies. *Journal of Logic and Algebraic Programming*, 54:1–64, 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proc. TLDI '03*, volume 38, pages 26–37. ACM Press, March 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proc. ICFP '04*, pages 244–255. ACM Press, 2004.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proc. ICFP '05*, pages 204–215. ACM Press, September 2005.

Conor McBride and Ross Paterson. Applicative programming with effects. *JFP*, 17(5):1–13, 2007.

Lambert G. L. T. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

Neil Mitchell and Stefan O'Rear. Derive - project home page. `http://www.cs.york.ac.uk/~ndm/derive/`, March 2007.

Neil Mitchell and Colin Runciman. A static checker for safe pattern matching in Haskell. In *Trends in Functional Programming (2005 Symposium)*, volume 6, pages 15–30. Intellect, 2007.

Markus Mohnen. Context patterns in Haskell. In *Implementation of Functional Languages*, pages 41–57. Springer-Verlag, 1996.

Matthew Naylor and Colin Runciman. Finding inputs that reach a target expression. In *Proc. SCAM '07*. IEEE Computer Society, September 2007. To appear.

Simon Peyton-Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Proc. Haskell '01*, pages 203–233. ACM Press, 2001.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proc. ICFP '06*, pages 50–61. ACM Press, 2006.

Deling Ren and Martin Erwig. A generic recursion toolbox for Haskell or: scrap your boilerplate systematically. In *Proc. Haskell '06*, pages 13–24. ACM Press, 2006.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proc. Haskell Workshop '02*, pages 1–16. ACM Press, 2002.

Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 216–238. Spinger-Verlag, June 2004.

Philip Wadler. List comprehensions. In Simon Peyton Jones, editor, *Implementation of Functional Programming Languages*. Prentice Hall, 1987.

Stephanie Weirich. RepLib: a library for derivable type classes. In *Proc. Haskell '06*, pages 1–12. ACM Press, 2006.

Noel Winstanley. Reflections on instance derivation. In *1997 Glasgow Workshop on Functional Programming*. BCS Workshops in Computer Science, September 1997.

[8] `http://darcs.haskell.org/generics/`