Chapter 22

Matching and Modifying with Generics

Neil C. C. Brown[1], Adam T. Sampson[2]
Category: Research Paper

Abstract: Haskell has powerful features such as pattern matching and recursive data types for building tree structures easily. These features are not without problems, however: patterns are not composable and cannot be abbreviated; it can be hard to modify a specific part of a large tree structure without unique identifiers. This paper explains how a particular Haskell generics approach, Scrap Your Boilerplate, can be used to solve both of these problems in Haskell, without the need for meta-programming or new language features.

22.1 INTRODUCTION

Haskell allows programmers to use pattern matching in several contexts, most notably function equations, to match data structures and bind variables. One problem with Haskell's pattern matching is that the entirety of a pattern must be written together; patterns are not composable or reusable. In addition, pattern matching cannot enforce the equality of two subcomponents; for example, they cannot match a list with the first two elements being equal.

Haskell's data structures can be recursive and mutually recursive, which allows complex heterogenous tree structures to be expressed. Data in functional languages is immutable, so tree structures are modified by walking the entirety of a tree and effectively building a new tree. It is difficult to 'remember' the location of a particular sub-tree during one traversal and later traverse to it again later, in the absence of unique identifiers for nodes.

[1]Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK
neil@twistedsquare.com
[2]Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK
ats@offog.org

In this paper we show how to use a generics library to solve both of these shortcomings. We have made composable patterns with more advanced capabilities than those of standard Haskell pattern matching (described in section 22.2). We also solve the problem of recording the position of an arbitrary sub-tree in a complex heterogenous tree in order to traverse to it again later (see section 22.3).

The generics approach used here is Scrap Your Boilerplate (SYB); a Haskell library that supports generic programming in Haskell using two commonly available language extensions (rank-2 polymorphism and a type-safe cast operator) [5, 6, 7]. It has since formed the basis for other further work [2, 4, 8]. SYB is available in the GHC Haskell compiler in the module *Data.Generics*.

22.2 PATTERN MATCHING

Haskell supports pattern matching over data-types, allowing different constructors of an algebraic data type to be handled separately. Pattern-matching can:

- match complex structures of any depth according to their structure, and
- bind of elements of the match to variables, to be used later on.

However, pattern-matching:

- lacks support for composition or abbreviation, and
- is unable to relate sub-elements of a structure to each other while pattern matching.

We will explain a technique that retains the first two capabilities, while solving the latter two shortcomings. We treat binding of variables orthogonally from structural constraints in patterns (such as *Just* `[_,_]`). For our purposes here, we will therefore consider a standard variable binding in a pattern match *x* to be *x*`@_`; that is, a binding without structural constraint.

22.2.1 Compiler Examples

We use Haskell to write Tock, a source-to-source nanopass compiler for parallel languages, compiling from languages such as occam-π [13] into C or C++. The idea behind the nanopass methodology is that the compiler is composed of a large number of small independent passes that each alter a central Abstract Syntax Tree (AST) in one regard (for example, resolving names) [9]. An added advantage of the nanopass methodology is that the passes are easy to unit-test.

We use pattern-matching to check that the actual output value of a pass matches our expectation. For example, occam-π allows parallel assignments. One pass in the compiler takes parallel assignments of the form on the left below (that assigns `x` to `y` in parallel with `y` to `x`), and flattens them into a sequential form involving temporary variables on the right, where `t` is a new (uniquely-named) temporary variable:

```
x, y := y, x
```

```
SEQ
  t := x
  x := y
  y := t
```

For simplicity, we will omit discussing the issue of declaring `t`.

Our Haskell function for checking the output of the transformation pass is[3]:

```
check (SeqBlock [Assign _ [Variable _ temp0] [Variable _ "x"],
                 Assign _ [Variable _ "x"] [Variable _ "y"],
                 Assign _ [Variable _ "y"] [Variable _ temp1]])
      = temp0 == temp1
check _ = False
```

The problems with pattern-matching are apparent in this example: there is a lot of duplication, and we cannot directly check that the two temporary variables are the same – instead we must check that after the pattern match. Imagine if we also want to check that two consecutive swaps get transformed properly:

```
check2 (SeqBlock [Assign _ [Variable _ temp0] [Variable _ "x"],
                  Assign _ [Variable _ "x"] [Variable _ "y"],
                  Assign _ [Variable _ "y"] [Variable _ temp1]],
                  Assign _ [Variable _ temp2] [Variable _ "a"],
                  Assign _ [Variable _ "a"] [Variable _ "z"],
                  Assign _ [Variable _ "z"] [Variable _ temp3]])
       = (temp0 == temp1) && (temp2 == temp3) && (temp0 /= temp2)
check2 _ = False
```

Despite the common aspects with our first *check* function, there is no simple way to share the common code, because the patterns are not composable or abbreviatable.

We cannot form an expected value to check against, because the temporary variables have dynamically generated names that are guaranteed to be unique. We do not want to test the exact name of the temporary variables (which would tie all our tests to our method of generating unique names), but instead test the relationship between the various temporary variable names.

The problems with pattern-matching are in contrast to the ease of generating the input data for the passes, where we can remove all duplication:

```
assign lhs rhs = Assign sp lhs rhs
var n = Variable sp n
swap vars = assign vars (reverse vars)

input1 = swap [var "x", var "y"]
input2 = SeqBlock sp [input1, swap [var "a", var "z"]]
```

We will demonstrate how our pattern-matching techniques can remove duplication from our patterns, and allow different parts of the pattern to be given the same label and automatically be matched with each other.

[3]For a complete definition of our AST types, see appendix A

22.2.2 Approach

As others have noted [2], every data item in Haskell is a constructor with a number of parameters/arguments (integers and similar types being nullary constructors). This is apparent in the syntax for a pattern in Haskell; each part of a pattern is a constructor, accompanied by a number of sub-patterns to match the constructor's arity. This matches with SYB's spine view of data, where each data item is decomposed into a constructor (of type *Constr*) and a number of sub-items. We therefore marry SYB with pattern matching.

Our approach is centred around one new data-type (accompanied by a helper function: double-underscore):

```
data Pattern = Anything | (:@) String Pattern | Structure Constr [Pattern]
               deriving (Show, Typeable)
__ = Anything
```

This *Pattern* is powerful enough to match anything in Haskell that is an instance of *Data*[4], a type-class in SYB that can be automatically derived by GHC. *Anything* can match anything. The `:@` operator is like the `@` in standard Haskell pattern matching, applying a label to a pattern. As indicated earlier, the label *x* in a standard pattern match is here written as *x*`:@__`. Any two parts of the pattern with the same label must have the same value to match successfully. The *Structure* pattern matches a specific data constructor, and then matches the first pattern in the list with the first argument to the constructor and so on.

We can build patterns such as the following to match a *Variable*, ignore its *SourcePos* (the first argument, and thus the first item in the list) and capture its name (the second argument, hence the second item in the list):

```
Structure (toConstr (Variable undefined undefined)) [__, "varName":@__]
```

The *toConstr* function is from the SYB library, and has type:

```
toConstr :: Data a => a -> Constr
```

In order to call *toConstr*, we must pass it a fully-formed instance of the **type** *Variable*. The **constructor** *Variable* is actually of type *SourcePos* `->` *String* `->` *Variable*. Therefore we need to give the constructor two accompanying arguments to produce a term of type *Variable*. Because *toConstr* never evaluates its argument, the simple uniform approach of using *undefined* for all arguments is valid. Helper functions are provided to easily use this technique:

```
con2 :: Data b => (a0 -> a1 -> b) -> Constr
con2 c = toConstr (c undefined undefined)

Structure (con2 Variable) [__, "varName":@__]
```

The items in the list for the second argument to *Structure* must all themselves be patterns. This can be particularly long-winded when matching patterns for

[4] *Pattern* is also an instance of *Data*, but we must derive this manually as *Constr* is not itself an instance of *Data*.

lists (in this case, *String*s); we must form patterns of a list in its cons-form. For example, to match a variable named *x*:

```
Structure (con2 Variable) [__,
  Structure (con2 ((:) :: Char -> String -> String))
    [ Structure (con0 'x') [],
      Structure (([]) :: String) []]]
```

An obvious step would be to create a helper function for the *String* type (or, more generally, for lists). But we can simplify matters by taking advantage of the dynamic typing of the *Typeable* class. *Typeable* is a super-class of *Data* that provides a *cast* function that performs a safe run-time cast.

Functions of the form *patt*N are provided, that take a constructor, followed by the appropriate number of arguments (*N*), and form a pattern using the *Structure* constructor and the *toPattern* function:

```
patt2 :: (Data c0, Data c1, Data b) => (a0 -> a1 -> b) -> c0 -> c1 -> Pattern
```

In the type of the *patt2* function, the types expected by the constructor (*a0* and *a1*) are separated from the types given to *patt2* (*c0* and *c1*); indeed, the *a0* and *a1* types are effectively part of a declaration of kind (∗ -> ∗ -> ∗) rather than type. This is what allows the second (and third) arguments to be either a *Pattern*, or an item of the correct type for the constructor (or an item of an unrelated type – forming an invalid pattern, as discussed later on in section 22.2.3).

We can now write this very simple pattern: `patt2 Variable __ "x"`. Note how the second argument to the *patt2* function is a *Pattern*, whereas the third argument is a *String*. No type annotations or extra functions are needed to differentiate the two. To simplify our life further, we can even define helper functions such as: `var = patt2 Variable __`.

We can now write our original full patterns (from section 22.2.1) as follows:

```
var = patt2 Variable __
label x = x :@ __
lhs <:=> rhs = patt3 Assign __ [lhs] [rhs]
seq = patt2 SeqBlock

swap var1 var2 temp
  = [temp <:=> var1,
     var1 <:=> var2,
     var1 <:=> temp]

check t
 = patternMatch (seq (swap (var "x") (var "y") (var (label "temp")))) t

check2 t
 = do items <- patternMatch (seq
        (swap (var "x") (var "y") (var (label "temp"))
        (swap (var "a") (var "z") (var (label "temp2"))
       ) t
      assertNotEqual (Map.lookup items "temp") (Map.lookup items "temp2")
```

Note that the `"x"` and `"y"` items are *String*s (variable names), whereas `"temp"` is a label for a *Pattern*; it is just coincidence that they look similar here (and that `"temp"` labels a *String* item). The *var* function accepts either *String* or *Pattern*, due to the previously discussed mechanisms. Finally, there is no check for quality after the pattern match; the match itself takes care of checking the two items labelled `"temp"` are the same, and that the two items labelled `"temp2"` are the same. We must check that the two items are distinct separately, as we do not always need this restriction, so it is not built into the pattern-matching mechanisms.

22.2.3 Implementation

Creating Patterns

The implementations of the *patt*N functions are shown below. They all use the *toPattern* function, which takes a value of a type belonging to the *Data* class and turns it into a *Pattern* automatically. If the value is already of type *Pattern*, it is used as-is. Otherwise the function uses *toConstr*, *gmapQ* and recursion to turn the entire value into a *Pattern*. The *gmapQ* function maps a generic query over all sub-terms of a constructor and returns a list of the results.

```
patt2 :: (Data c0, Data c1, Data b) => (a0 -> a1 -> b) -> c0 -> c1 -> Pattern
patt2 c x0 x1 = Structure (con2 c) [toPattern x0, toPattern x1]

toPattern :: Data a => a -> Pattern
toPattern x = case cast x of
  Just x' -> x'
  Nothing -> Structure (toConstr x) (gmapQ toPattern x)
```

Matching Labelled Items

We chose to implement the mechanism for ensuring items with the same label have the same value by using an associative map in a state monad. The first time a label is encountered, the appropriate value is inserted into the map. Every further time the label is found in the pattern, the new value is compared to the stored value. If they are not equal, the pattern match fails. We store the values (which can be of any type) inside a simple wrapper type that maintains the *Data* constraint on the item:

```
data AnyDataItem = forall a. Data a => ADI a
```

The equality operation for this type uses the *geq* function (a generic equality comparison provided by SYB). For a dynamic type such as this, we are not aware of any way to automatically use a type's native *Eq* instance (when available). This would be preferable to using *geq*, as it would allow programmers to provide their own custom definition of equality for two items with the same label.

Pattern Matching

The implementation for matching a *Pattern* against a data structure is straightforward, descending the pattern and zipping together the two structures. When the match fails, we give an error that highlights exactly which part of the match failed, noting the expected value/pattern, and the actual value.

```
patternMatch :: (Data a, Data b) =>
                a -> b -> Either String (Map String AnyDataItem)
patternMatch x y = evalStateT (patternMatch' x y) Map.empty

type PatternM = StateT (Map String AnyDataItem) (Either String)

patternMatch' :: (Data a, Data b) => a -> b -> PatternM ()
patternMatch' Anything _ = return ()
patternMatch' (label :@ item) x
    = patternMatch' item x >> recordItem label (ADI x)
patternMatch' (Structure con items) x
  | not (any (constrEq con) (dataTypeConstrs (dataTypeOf x)))
    = throwError ... -- Inconsistent pattern
  | not (constrEq con (toConstr x)) = throwError ... -- Constructors not equal
  | otherwise = foldl Map.union (sequence (gmapFuncsQ funcs x))
    where funcs = map patternMatch' items

recordItem :: String -> AnyDataItem -> PatternM ()
recordItem label x
  = do m <- get
       case Map.lookup label m of
         Just y -> if x == y then return ()
                             else throwError ... -- Labelled items did not match
         Nothing -> put (Map.insert label x m)
```

We describe the implementation of *gmapFuncsQ* function in section 22.4 – it maps a list of query functions over the sub-terms of a given value.

One unexpected problem arose with equality over the *Constr* type. The way *Eq* for *Constr* has been defined (only comparing the indices of the constructor), only *Constr* items of the same type can be compared safely. This means that if we automatically derive *Eq* for the *Pattern* type then we encounter the slightly unexpected equality: *Structure* (*toConstr False*) [] == *Structure* (*toConstr LT*) []?[5] Thus we use our own *constrEq* function.

Invalid Patterns

With standard Haskell patterns, the compiler checks the types in a pattern. In our library, *Pattern*s can be nested together regardless of what type they match, and thus it is possible to write a pattern which can never match. For example, the pattern *patt2 Variable* 2 3 will compile correctly, even though it is nonsensical.

[5] *LT* being from the type *Ordering*

We cannot easily remove this problem by, for example, parameterising *Pattern* according to the type it matches. The *Structure* item holds a list of patterns that need to match differently typed items, so we would have to wrap up the *Pattern* items to hide their type; thus losing any advantage of giving them a type in the first instance.

Instead we must detect this error at run-time. When comparing two constructors (e.g. 2 from the above pattern, and the actual *SourcePos* constructor) we first check that their type is the same. If not, then instead of failing with a "Did not match" message, we fail with an "Inconsistent pattern – can never match" message.

22.2.4 Modifiable Patterns

The parser in our compiler is itself a pass and so, like all passes, it has unit tests. The *SourcePos* tags cause some problems in this regard. When parsing our short piece of test code, the parser records the source position into the *SourcePos* items. In order to specify the expected output exactly, we would have to figure out the exact source positions of each item in our input string – a laborious process, not to mention that inserting an extra space would ruin all the expected data!

Our first solution was to define a special *SourcePos* tag (*anySP*) and implement custom equality so that *anySP* was deemed equal to any other *SourcePos* tag. Another solution would be to define a custom equality for other items in our AST that ignores the inner *SourcePos* tags. However, our *Pattern*s allow for a different solution. When parsing the statement "`x :=y`" we write our expected test data as normal data (i.e. no *Pattern*s) using an arbitrary value, *replaceSP*, for any *SourcePos* items:

```
var = Variable replaceSP
expectedVal = Assign replaceSP [var "x"] [var "y"]
```

This expected output is matched against the actual output by turning it into a *Pattern*, and replacing all the *replaceSP* tags with the *Anything* pattern, using a simple *replacePattern* function[6]:

```
replacePattern :: Pattern -> Pattern -> Pattern -> Pattern
replacePattern _ _ Anything = Anything
replacePattern find replacement original @(n :@ p)
  | original == find = replacement
  | otherwise = n :@ (replacePattern find replacement p)
replacePattern find replacement original @(Structure c ps)
  | original == find = replacement
  | otherwise = Structure c (map (replacePattern find replacement) ps)

expectedPatt = replacePattern (toPattern replaceSP) Anything
                              (toPattern expectedVal)
```

[6]It is assumed, at least for demonstration here, that you would not want to replace all your *Anything* patterns.

Thus the expected value can be written as normal (using any helper functions that may have been defined for constructing normal values), then later it can be turned into a *Pattern*, manipulated by masking out particular values as *Anything* patterns and finally matched against the actual test output. Note that the function is not restricted to transforming values to *Anything* patterns – it can also replace specified values with labelled patterns, or any arbitrary pattern.

22.2.5 Related Work

The lack of composition and abbreviation of Haskell's patterns has been noted many times in the past and several different solutions have been proposed [12, 11].

One such solution is first-class patterns [11]. First-class patterns provide pattern-combinators for easy and powerful composition of patterns, but introduce an elaborate mechanism for binding parts of a pattern. They do not solve the problem of enforcing the equality of repeated labels.

A simple proposal is pattern synonyms[7] – macro-substitution for patterns. These would help to abbreviate patterns, and allow some straight-forward composition, but they do not provide anything beyond that.

The idea behind another solution, views, is to transform the data into a view (by performing some computation), and then pattern match (or some analogous operation) on this view of the data [12]. The transformations are functions and can therefore be composed. We could define a transformation that – in our example – extracted the *temp0* and *temp1* items from the data structure. We could even make the view succeed (most view functions return a *Maybe* type) only when *temp0* and *temp1* match.

We need a different pattern for each test. Therefore to use views for our purposes, we would likely need to customise the view for each test, which would not save much effort over writing out a whole pattern.

Unification extends pattern-matching by allowing related sub-terms (that must match each other) and also allowing symmetric matching (the right-hand side can be a pattern too) [3, 10]. Thus, unification subsumes the capabilities of our approach, which by contrast is smaller and simpler.

Perhaps the closest work to our solution is the idea of a pattern logic that uses monadic combinators [1]. The combinators allow patterns to be easily composed, and include powerful quantifiers ($\forall$ and $\exists$). They can also be used to compare different parts of the pattern against each other when matching. While the capabilities of the pattern logic are comparable to the approach detailed here, the practical implementation is different. In specific, programmers must define the necessary combinators for their custom data types, whereas we take advantage of GHC's automatic derivation of the *Data* type-class. The pattern logic also lacks the ability to easily turn normal data into its *Pattern* form; combinators must be constructed independently of the data-type being matched.

[7] http://hackage.haskell.org/trac/haskell-prime/wiki/PatternSynonyms

22.3 TREE NAVIGATION

In our compiler, we build a control-flow graph (CFG) while walking through the AST. We then perform various analyses on the CFG, and often want to modify the AST based on our analysis (for example, to remove an unused variable). This means modifying a specific part of the AST, without any explicit unique identifier to tell different nodes apart[8]. More generally, we wish to modify a particular sub-tree that is uniquely identified solely by its location in the whole tree.

We will frame our discussion around the following example. Consider a code fragment that selects between two assignments using an **if** statement:

```
If sp (EqualConst sp (Variable sp "x") 2)
  (Assign sp [Variable sp "a"] ]Variable sp "x"])
  (Assign sp [Variable sp "b"] [Variable sp "y"])
```

The analysis would reveal (based on our control flow graph) that in the first assignment, x is always equal to the constant 2, and thus the constant value could be substituted in accordingly. We cannot simply replace all instances of x in the whole AST; this would wrongly substitute for x in the condition of the **if** statement. A constraint on our implementation is that the modifications we make to the tree in our compiler are always inside a monad.

22.3.1 Non-Generic Approach

When we walk the tree the first time, building the control-flow graph, we should record our position in the tree by building up a function. One can imagine this function as being a wrapper that carries a modifier function to the correct part of the tree. In terms of figure 22.1, we want to provide a modifier for a *Variable*, and be able to apply this modifier to the root of the tree. Conceptually we want the wrapper to traverse the arrow marked *A* in the diagram.

We build up the wrapper function by composing a wrapper function at each step descending the tree, composing arrows *B* and *C* to get *A*. Each wrapper will have the following type (*m* being the monad):

```
type ModifierFunc m inner outer = (inner -> m inner) -> (outer -> m outer)
```

The wrappers are composed using function composition (the `.` operator). We build up the function as we walk the tree, storing the *ModifierFunc* in a state monad. We omit the construction of the control-flow graph here in order to focus on the modifier functions.

The tree walking functions are named *recordVarUse*$*$. They are monadic and use the *addVarUseToState* function that takes a variable and a modifier wrapper for that variable, and stores the two in the state. Should we wish to modify a variable later on, we can lookup its associated modifier wrapper, wrap our modifier and apply it to the tree.

[8] Due to the manipulations performed by our passes, source positions are not guaranteed to be unique.

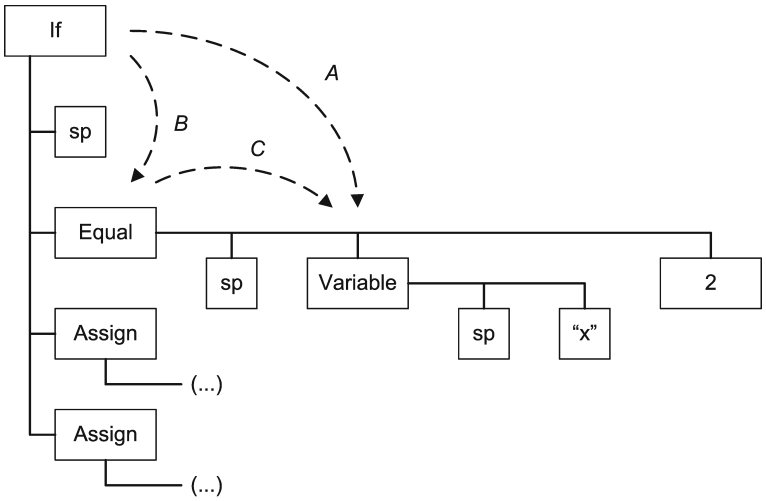

FIGURE 22.1. Our example data shown in tree (spine) form

The *recordVarUse* functions are as follows:

```
recordVarUse (If _ cond thenClause elseClause) mod
  = do recordVarUseCond cond (mod .
         \f (If x0 e x2 x3) -> do {e' <- f e ; return (If x0 e' x2 x3)})
       recordVarUse thenClause (mod .
         \f (If x0 x1 th x3) -> do {th' <- f th ; return (If x0 x1 th' x3)})
       recordVarUse elseClause (mod .
         \f (If x0 x1 x2 el) -> do {el' <- f el ; return (If x0 x1 x2 el')})

recordVarUseCond (EqualConst _ lhs _) mod
  = addVarUseToState lhs (mod .
      \f (EqualConst x0 v x2) -> do {v' <- f v ; return (EqualConst x0 v' x2)})
```

It is important that the trees returned by the functions use the *x1*, *x2* and *x3* parameters to the modifier function rather than using *cond*, *thenClause* and *elseClause*; it is possible that we may modify both clauses later on, so the latter values will be stale (unmodified) when the second modification function is called.

22.3.2 Generic Approach

The approach detailed in the previous section works, but it is clear that it is verbose and error-prone, for example the then/else clauses might be accidentally swapped.

The approach is very formulaic; all elements are left unmodified except the particular item of interest, which is modified by a function. The three lambda expressions in the *recordVarUse* function only differ in which item (2nd/3rd/4th) is modified, which suggests that the approach could be automated. What we would like to write for the top *recordVarUse* clause is:

```
recordVarUse mod (If _ cond thenClause elseClause)
  = do recordVarUseCond cond (mod . mod2of4 If)
       recordVarUse thenClause (mod . mod3of4 If)
       recordVarUse elseClause (mod . mod4of4 If)
```

Each *modNofM* function modifies the appropriate argument of the given constructor, leaving the rest unchanged. The type of the *mod2of4* function is as follows:

```
mod2of4 :: (Monad m, Data b, Typeable a0, Typeable a1, Typeable a2, Typeable a3)
  => (a0 -> a1 -> a2 -> a3 -> b) -> ModifierFunc m a1 b
```

We have to declare a number of constraints on the type of the function, mainly to satisfy the requirements of the SYB library. Note that the main reason that we pass the constructor (e.g. *If*) to the *mod2of4* function is to help the type system; we specify the four types (*a0*, *a1*, *a2*, *a3*) and which type the *ModifierFunc* acts on (the second; *a1*).

Here is how we define *mod2of4* and, for comparison, *mod3of4* in terms of another function, *decomp4*, which applies the given four monadic functions to the four arguments of the constructor:

```
mod2of4 con f = decomp4 con return f return return
mod3of4 con f = decomp4 con return return f return

decomp4 :: (Monad m, Data b, Typeable a0, Typeable a1, Typeable a2, Typeable a3)
  => (a0 -> a1 -> a2 -> a3 -> b) ->
     (a0 -> m a0) -> (a1 -> m a1) -> (a2 -> m a2) -> (a3 -> m a3) -> (b -> m b)
```

The *gmapM* transformation function – which is the obvious way of implementing *decomp4* – maps one generic function over the sub-terms. This generic function adjusts its behaviour solely according to the type of the sub-terms; if the 3rd and 4th items have identical types, they will be processed identically. The generic function cannot know which of the same-typed items it is operating on. The way we solve this problem is detailed in section 22.4 – using the functions described in that section, we can easily define *decomp4*:

```
decomp4 con f0 f1 f2 f3 x
  = do when (not (constrEq con x)) (fail "Invalid tree-walk")
       gmapFuncsM [mkM' f0, mkM' f1, mkM' f2, mkM' f3] x
```

See section 22.2.3 for an explanation of why we use a custom constrEq function rather than the equality operator. We compare the constructors before applying the functions as a sanity check; the error that we give is analogous to a non-exhaustive pattern match error in our original non-generic approach. An error will not occur as long as the modifier function is applied to the original tree and the sub-branches being modified are disjoint.

Our approach divorces the structure of the tree from the types involved. We can view the tree through the spine view of SYB; a series of nodes (constructors) with a fixed number of child nodes (the arguments to the constructors).

Although not very elegant, our approach is a working practical solution to our problem. The code needed to build up wrapper types for traversing a tree is now very small, and we did not use any pre-processing techniques or extra language systems; our solution is entirely build on Haskell and the SYB library.

We are not aware of any technique similar to the one presented here. The idea is similar to other work regarding customised traversals and generics [4, 8], but that work has always focused on applying modifiers according to type, not by position in the tree.

22.4 GENERIC PROGRAMMING BY INDEX IN SYB

Both of the solutions presented in this paper need to apply generic functions to sub-terms where the functions differ by *index*, rather than by *type*. The SYB library is designed to support the latter.

The first problem is that we require a list of generic queries/monadic transformations. These have the following types:

```
type GenericQ r = forall a . Data a => a -> r
type GenericM m = forall a . Data a => a -> m a
```

It is not possible in Haskell to construct a list with type [*GenericQ Bool*] due to the rank-2 types involved. Therefore, to work around this problem the SYB library supplies wrappers:

```
newtype GenericQ' r = GQ { unGQ :: (GenericQ r) }
newtype GenericM' m = GM { unGM :: (Data a => a -> m a) }
```

It is possible to create a list of [*GenericQ' Bool*]. Lists of query/monadic transformation functions given to the helper functions we now describe must be supplied using these wrappers.

The first function we require is to apply a list of generic query functions to the sub-terms of a constructor and return the list of the results. For this we can use the *gmapQi* function from the SYB library that queries one specific indexed child of a constructor.

```
gmapQi :: Data a => Int -> GenericQ u -> a -> u
```

```
gmapFuncsQ :: Data a => [GenericQ' u] -> a -> [u]
gmapFuncsQ funcs x = [gmapQi n (unGQ f) x | (n, f) <- zip [0..] funcs]
```

We also need a similar function for monadic transformation. Since there is no *gmapMi* equivalent (only *gmapM*), we must use a different technique. We use a monad state transformer to record a list of modification functions. Each function application removes the next function from the head of the list (in effect, a stack) and applies that:

```
gmapM :: (Monad m, Data a) => GenericM m -> a -> m a
```

```
popAndApply :: (Monad m, Data a) => a -> StateT [GenericM' m] m a
```

```
popAndApply x = do (f:fs) <- get
                   put fs
                   lift ((unGM f) x)

gmapFuncsM :: (Monad m, Data a) => [GenericM' m] -> a -> m a
gmapFuncsM funcs x = evalStateT (gmapM popAndApply x) funcs
```

The *lift* function is needed to lift the inner monadic value up into the *StateT* monad. The *evalStateT* function executes the state monad transformer action with the given state (*funcs*), and returns the result, discarding the state – which should always be the empty list at the end.

Finally, we define a useful helper version of the *mkM* function, that also wraps its return value. The *mkM* function turns a type-specific monadic modification into a generic modification that applies *return* to all other types.

```
mkM' :: (Monad m, Typeable a) => (a -> m a) -> GenericM' m
mkM' f = GM {unGM = mkM f}
```

22.5 CONCLUSIONS

This paper has detailed two interesting uses of SYB to solve practical problems we have encountered in programming a compiler in Haskell. It is apparent that SYB is a powerful tool that allows us to build powerful systems with all the details neatly hidden away behind the *Data* type-class.

One problem with SYB is that it does not integrate well with the notion of type-classes in Haskell. There are several points where we would like to construct a generic function along the lines of *show* `extQ` *gshow*; that is, to use an instance of *Show* wherever it is available, but to resort to the *Data*-based SYB function *gshow* otherwise. SYB cannot express such concepts. Although the *Typeable* type supports dynamic typing, it cannot dynamically reveal type-class membership – if this is at all feasible in a Haskell implementation.

The examples in this paper have been simplified from the open-source Tock code. We have not yet separated our generic utilities into a reusable library; if you would be interested in using them in other projects, please contact the authors.

22.6 ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their extensive comments. Thanks are also due to Chris Booth – without his enthusiasm for test-driven development, this work would not have been started.

A ABSTRACT SYNTAX TREE TYPE

Throughout this paper we use the following simplified AST structure (for brevity we have omitted the *deriving* clauses):

```
data SourcePos = SourcePos String Int Int −− filename, line number, column
data Statement = Assign SourcePos [Variable] [Variable] −− dest, source
               | If SourcePos Condition Statement Statement −− then, else
               | SeqBlock SourcePos [Statement]
data Variable = Variable SourcePos String
data Condition = EqualConst SourcePos Variable Int
```

REFERENCES

[1] O. Chitil and F. Huch. A pattern logic for prompt lazy assertions. In *Implementation and Application of Functional Languages, 18th International Workshop, IFL 2006*, LNCS 4449, pages 126–144, April 2007.

[2] R. Hinze, A. Löh, and B. C. d. S. Oliveira. "Scrap Your Boilerplate" Reloaded. In *Proceedings of the Eighth International Symposium on Functional and Logic Programming (FLOPS 2006)*, 2006.

[3] P. Jansson and J. Jeuring. Functional pearl: Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, 1998.

[4] R. Lämmel. Scrap your boilerplate with XPath-like combinators. In *POPL'07, Proceedings*. ACM Press, Jan. 2007.

[5] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI*, pages 26–37, 2003.

[6] R. Lämmel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ICFP 2004*, pages 244–255. ACM Press, 2004.

[7] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP 2005*, pages 204–215. ACM Press, Sept. 2005.

[8] D. Ren and M. Erwig. A generic recursion toolbox for haskell or: scrap your boilerplate systematically. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 13–24, New York, NY, USA, 2006. ACM Press.

[9] D. Sarkar, O. Waddell, and R. K. Dybvig. A nanopass infrastructure for compiler education. In *ICFP 2004*, pages 201–212. ACM Press, 2004.

[10] T. Sheard and E. Pasalic. Two-level types and parameterized modules. *Journal of Functional Programming*, 14(5):547–587, 2004.

[11] M. Tullsen. First class patterns. In *PADL*, pages 1–15, 2000.

[12] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings, 14th Symposium on Principles of Programming Languages*, pages 307–312. ACM, 1987.

[13] P. H. Welch and F. R. M. Barnes. Communicating mobile processes: introducing occam-pi. In *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.