Matching and Modifying with Generics

Neil Brown and Adam Sampson

Computing Laboratory
University of Kent
UK

28 May 2008
Talk Outline

- Two separate applications of “Scrap Your Boilerplate”
  - generic programming
  - 1. Pattern-matching
  - 2. Modifying large trees
- Show how to make Haskell code shorter and simpler by using generics
We write a compiler for concurrent languages using Haskell

We use test-driven development (mainly using HUnit)

It is a nanopass compiler – executes many isolated compiler transformations on a central abstract syntax tree (AST)
Example transformation: flatten assignments
Turn parallel assignments into multiple sequential assignments with temporary variables
We want to test the transformation

\[
\begin{align*}
\text{SEQ} & \quad \begin{aligned}
\text{x, y := y, x} \\
\text{t := x} \\
x := y \\
y := t
\end{aligned}
\end{align*}
\]
Unit testing

Test Specification

Input

Expected Result

Function to test

assertEqual
We need to construct a fragment of AST (right) to feed into our test, corresponding to the source code (left):

\[
x, y := y, x
\]
We need to construct a fragment of AST (right) to feed into our test, corresponding to the source code (left):

\[
x, y := y, x
\]

\[
\text{sp} = \text{SourcePos} 1 1
\]

Assign sp

\[
\text{[Variable sp "x"}, \text{Variable sp "y"}]\n\]

\[
\text{[Variable sp "y"}, \text{Variable sp "x"}]\n\]
We need to construct a fragment of AST (right) to feed into our test, corresponding to the source code (left):

\[
x, \ y \ := \ y, \ x
\]

\[
sp = \text{SourcePos} \ 1 \ 1
\]
\[
\text{var } x = \text{Variable } sp \ x
\]

Assign sp

\[
[\text{var } "x", \ \text{var } "y"]
\]
\[
[\text{var } "y", \ \text{var } "x"]
\]
We need to construct a fragment of AST (right) to feed into our test, corresponding to the source code (left):

\[
x, y := y, x
\]

\[
\text{sp} = \text{SourcePos} \ 1 \ 1 \\
\text{var } x = \text{Variable } \text{sp} \ x \\
\text{swap vars} = \text{Assign } \text{sp vars (reverse vars)} \\
\text{swap [var "x", var "y"]}
\]
Could try constructing output value to match against:

```
SeqBlock [Assign sp [var "t"] [var "x"],
    Assign sp [var "x"] [var "y"],
    Assign sp [var "y"] [var "t"]
```

But temporary won’t really be called "t" – name will be generated

Don’t want to tie tests to name generation – if we change the name generation we’d have to change all our tests!

Exact name is not important, as long as the two instances both have the same name
The problem – matching

- Can’t check against an expected value. Must use pattern matching:

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

- Can’t easily shorten the pattern!
The problem with patterns

- Patterns cannot be abbreviated, nor easily composed
- We can solve this using generics
- Not a new language extension, just uses generics in normal Haskell
Generic programming

- A generic function is one that does different things to each type, depending on its structure.
- Not to be confused with polymorphism: a polymorphic function is one that does the same thing to whichever type it is applied to.
- We were already using a generic programming technique known as Scrap Your Boilerplate (SYB):
  - It is built around a type-class called Data.
  - GHC, the Haskell compiler, can automatically derive instances of Data.
SYB basics

SYB decomposes data into its constructor and a list of arguments:

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

![Diagram showing the decomposition of data into a constructor and arguments.](image)
Patterns as a data type

- We represent patterns as a value of type Pattern:

```haskell
data Pattern = Anything
               | String :@ Pattern
               | Structure Constr [Pattern]
```

- Can easily convert any item into its equivalent exact pattern (see paper)

```haskell
toPattern :: Data a => a → Pattern
```
Example pattern

- We want to match `Variable _ "x"`:

Structure
  (toConstr (Variable (SourcePos 1 1) ""))
[Anything,
  toPattern "x"]
We want to match Variable _ "x":

Structure
  (toConstr (Variable undefined undefined))
  [Anything,
   toPattern "x"]
Example pattern

- We want to match Variable _ "x":

```plaintext
mVariable x y = Structure
  (toConstr (Variable undefined undefined))
  [toPattern x, toPattern y]
_    = Anything

mVariable _ "x"
```

Converting our earlier pattern into a Pattern:

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

Pattern-match above becomes Pattern below:

```plaintext
patt = mSeqBlock
  [mAssign __ [mVariable __ ("temp": @__)]] [mVariable __ "x"],
  mAssign __ [mVariable __ "x"] [mVariable __ "y"],
  mAssign __ [mVariable __ "y"] [mVariable __ ("temp": @__)]]
```

matchPattern patt
Simplifying the pattern

\[
patt = \text{mSeqBlock} \\
  \quad [\text{mAssign} \ [\text{mVariable} \ ("temp": \@\@)] \ [\text{mVariable} \ "x"], \\
  \quad \text{mAssign} \ [\text{mVariable} \ "x"] \ [\text{mVariable} \ "y"], \\
  \quad \text{mAssign} \ [\text{mVariable} \ "y"] \ [\text{mVariable} \ ("temp": \@\@)]] \\
\]

matchPattern patt
Simplifying the pattern

```plaintext
var x = mVariable __ x

patt = mSeqBlock
   [mAssign __ [var ("temp": @__)] [var "x"],
    mAssign __ [var "x"] [var "y"],
    mAssign __ [var "y"] [var ("temp": @__)]]

matchPattern patt
```
Simplifying the pattern

```
var x = mVariable __ x
lhs <=> rhs = mAssign __ [lhs] [rhs]

patt = mSeqBlock
    [var ("temp": @__) <=> var "x",
     var "x" <=> var "y",
     var "y" <=> var ("temp": @__)]

matchPattern patt
```
Simplifying the pattern

```plaintext
var x = mVariable __ x
lhs <+=> rhs = mAssign __ [lhs] [rhs]

patt = mSeqBlock [t <+=> x, x <+=> y, y <+=> t]
   where
      x = var "x"
      y = var "y"
      t = var "temp": @__

matchPattern patt
```
Pattern matching summary

- We represent patterns as normal Haskell data (with the help of SYB)
- We can manipulate these patterns
  - Pull out common sub-patterns to reduce duplication
  - Replace parts of the pattern
- Code for matching a pattern against data is in the paper
- Patterns are not type-safe – it is possible to create inconsistent patterns (see paper): mVariable __ 7
Modifying a tree
Modifying a tree

If
  sp
  Equal
    sp
    sp
    Variable
      sp
      “x”
      2
  Assign
    ...
Modifying a tree
Modifying a tree
Identifying the right place

- There are no unique identifiers for nodes
  - Awkward to add them
- Cannot match by equality – we only want to modify a particular use of variable “x”
- Only uniquely identifying thing is the position
Modifying a single node

Expression $\rightarrow$ MyMonad Expression
Modifying a tree
Modifying a tree
Wrapping the modifier

(Expression $\rightarrow$ MyMonad Expression) $\rightarrow$ (AST $\rightarrow$ MyMonad AST)
Modifying a tree

- If
  - sp
- Equal
  - sp
  - Variable
  - sp
  - "x"
- Assign
- Assign

(...)

2
analyse (If _ cond thenClause elseClause) mod = do
  analyseExpr cond (mod .
    \f (If sp e x2 x3) -> do {e' <- f e ; return (If sp e' x2 x3)})
analyse (If _ cond thenClause elseClause) mod = do
  analyseExpr cond (mod .
    \f (If sp e x2 x3) -> do {e' <- f e ; return (If sp e' x2 x3)})
  analyse thenClause (mod .
    \f (If sp x1 th x3) -> do {th' <- f th ; return (If sp x1 th' x3)})
  analyse elseClause (mod .
    \f (If sp x1 x2 el) -> do {el' <- f el ; return (If sp x1 x2 el')})

analyseExpr (Equal _ lhs rhs) mod = do
  analyseExpr lhs (mod .
    \f (Equal sp e x2) -> do {e' <- f e ; return (EqualConst sp e' x2)})
  analyseExpr rhs (mod .
    \f (Equal sp x1 e) -> do {e' <- f e ; return (EqualConst sp x1 e')})
Define decompN functions (see paper), and helper functions:

\[
\text{decomp3 :: (Monad m, Data b, Typeable a0, Typeable a1, Typeable a2) => (a0 \rightarrow a1 \rightarrow a2 \rightarrow b) \rightarrow (a0 \rightarrow m a0) \rightarrow (a1 \rightarrow m a1) \rightarrow (a2 \rightarrow m a2) \rightarrow (b \rightarrow m b)}
\]

\[
\text{mod2of3 con f = decomp3 con return f return}
\]

\[
\text{mod3of3 con f = decomp3 con return return f}
\]
Generics solution

analyse (If _ cond thenClause elseClause) mod = do
  analyseExpr cond (mod . mod2of4 If)
  analyse thenClause (mod . mod3of4 If)
  analyse elseClause (mod . mod4of4 If)

analyseExpr (Equal _ lhs rhs) mod = do
  analyseExpr lhs (mod . mod2of3 Equal)
  analyseExpr rhs (mod . mod3of3 Equal)
Composing modifiers
Summary

- Used SYB generics for two interesting applications:
  1. Pattern-matching
  2. Tree modification
- Not type-safe, and a little ad-hoc
- But: made our code shorter and more powerful
- Generics are a useful tool for doing even small things that are awkward in Haskell
Questions?
Why can’t Pattern be parameterised?

data Pattern a = Anything
       | String  : @ (Pattern a)
       | Structure Constr [Pattern a]
Ideal QuickCheck scenario

1. Generate random input
2. Function to test
3. Check simple property
Common QuickCheck scenario

input

Generate random input and result

expected result

Function to test

output

Check equality
Redundant QuickCheck scenario

- Generate random input
- Calculate result
- Function to test
- Check equality