

Matching and Modifying with Generics

Neil Brown and Adam Sampson

Computing Laboratory
University of Kent
UK

28 May 2008

Talk Outline

- Using “Scrap Your Boilerplate” generic programming to build a library for pattern matching
 - We can make eliminate duplication in patterns, and make them composable
- Show how to make Haskell code shorter and simpler by using generics

Background

- We write a compiler for concurrent languages using Haskell
- It is a nanopass compiler – executes many isolated compiler transformations on a central abstract syntax tree (AST)
- We use test-driven development (mainly using HUnit)

Compiler transformation

- Example transformation: flatten assignments
- Turn parallel assignments into multiple sequential assignments with temporary variables
- We want to test the transformation

$x, y := y, x$



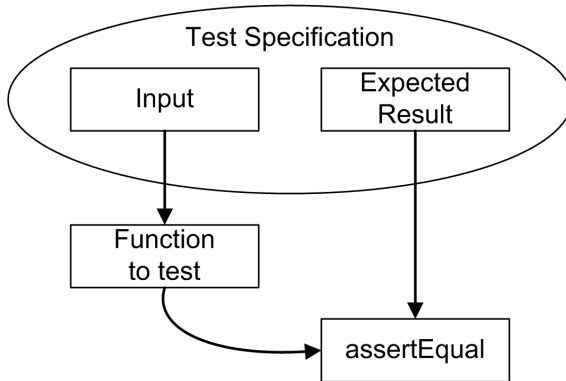
SEQ

$t := x$

$x := y$

$y := t$

Unit testing



Compiler transformation – test input

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

Source Language

`x, y := y, x`

Abstract Syntax Tree (AST)

```
Assign (SourcePos 1 1)
  [Variable (SourcePos 1 1) "x"
  , Variable (SourcePos 1 1) "y"]
  [Variable (SourcePos 1 1) "y"
  , Variable (SourcePos 1 1) "x"]
```

Compiler transformation – test input

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

Source Language

Abstract Syntax Tree (AST)

`x, y := y, x`

`sp = SourcePos 1 1`

Assign sp

```
[Variable sp "x"
 ,Variable sp "y"]
[Variable sp "y"
 ,Variable sp "x"]
```

Compiler transformation – test input

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

Source Language

`x, y := y, x`

Abstract Syntax Tree (AST)

```
sp = SourcePos 1 1  
var x = Variable sp x
```

Assign sp

```
[var "x", var "y"]  
[var "y", var "x"]
```

Compiler transformation – test input

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

Source Language

`x, y := y, x`

Abstract Syntax Tree (AST)

```
sp = SourcePos 1 1
var x = Variable sp x
swap vars = Assign sp vars (reverse vars)

swap [var "x", var "y"]
```

Constructing output – bad

- 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"]
```

- Problems:
 - Temporary name ("t") is automatically generated
 - Want to separate tests from method of name generation

The problem – matching

- Exact name is not important, as long as the two instances both have the same name. Use pattern matching:

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

The problem with patterns

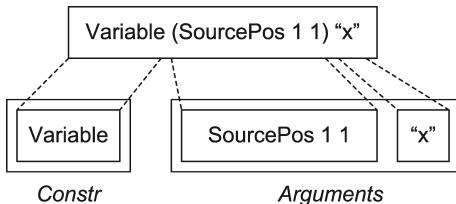
- Patterns cannot be abbreviated, nor easily composed
- We can solve this using generics
- Not a new language extension, just uses generics in normal Haskell

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

Scrap Your Boilerplate (SYB) basics

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



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

Patterns as a data type

- We represent patterns as a value of type Pattern:

```
data Pattern = MatchAnything  
            | String   :@ Pattern  
            | ConstructedItem Constr [Pattern]
```

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

```
toPattern :: Data a => a -> Pattern
```

Example pattern

- We want to match Variable _ "x":

ConstructedItem

```
(toConstr (Variable (SourcePos 1 1) ""))  
[MatchAnything,  
toPattern "x"]
```

Example pattern

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

```
ConstructedItem  
  (toConstr (Variable undefined undefined))  
  [MatchAnything,  
   toPattern "x"]
```

Example pattern

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

```
mVariable x y = Structure
  (toConstr (Variable undefined undefined))
  [toPattern x, toPattern y]
___ = MatchAnything
```

```
mVariable ___ "x"
```

Converting our earlier pattern into a Pattern

```

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:

```

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

matchPattern patt

```

Simplifying the pattern

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

```
matchPattern patt
```

Simplifying the pattern

```
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

```
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`

Tree modification

- Details of how to use SYB to easily modify a particular node in tree structures without a unique identifier for the node are in the paper

Our experience

- Generics allowed the code in our compiler to be:
 - Much shorter
 - More readable
- Writing tests is easier, so our code is better tested

Questions?