Matching and Modifying with Generics

Neil Brown and Adam Sampson

Computing Laboratory
University of Kent
UK

28 May 2008

University of Kent | Computing

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

Background

- 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)

Compiler transformation

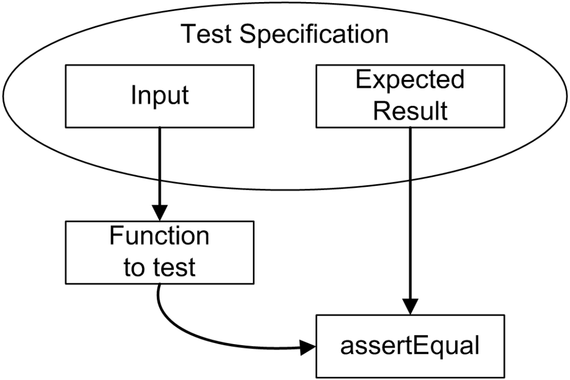
- Example transformation: flatten assignments
- Turn parallel assignments into multiple sequential assignments with temporary variables
- We want to test the transformation

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

→

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

Unit testing



University of Kent | Computing

Compiler transformation – test input

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

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

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

University of Kent | Computing

Compiler transformation – test input

- 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 = SourcePos 1 1

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

University of Kent | Computing

Compiler transformation – test input

- 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 = SourcePos 1 1
var x = Variable sp x

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

University of Kent | Computing

Compiler transformation – test input

- 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 = 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"]]
```

- 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

University of Kent | Computing

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

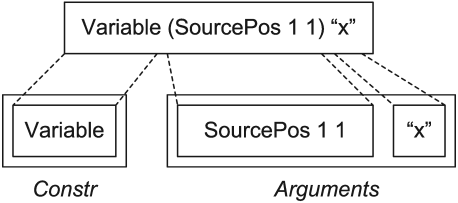
- Patterns cannot be abbreviated, nor easily composed
- We can solve this using generics
- Not a new language extension, just uses generics in normal Haskell

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
```

Patterns as a data type

- We represent patterns as a value of type Pattern:

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

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

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

University of Kent | Computing

Example pattern

- We want to match Variable _ "x":

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

Example pattern

- We want to match Variable _ "x":

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

Example pattern

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

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

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
```

University of Kent Computing

Simplifying the pattern

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

matchPattern patt
```

University of Kent | Computing

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

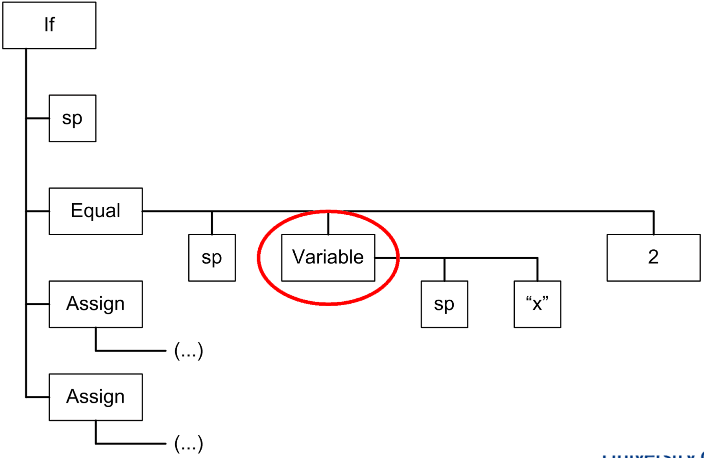
```
var x = mVariable __ x
lhs <:=> rhs = mAssign __ [lhs] [rhs]

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`

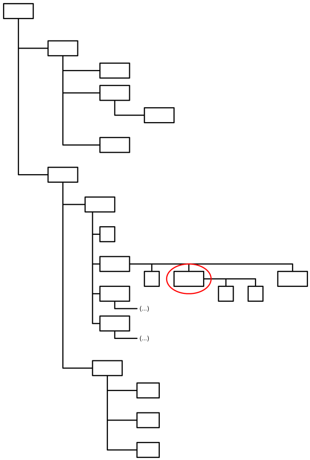
Modifying a tree

Modifying a tree



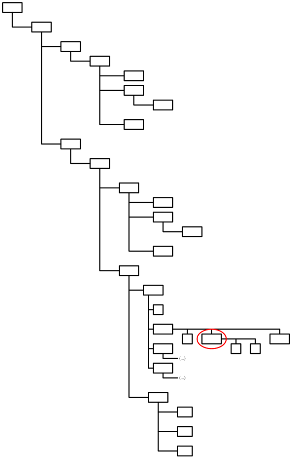
University of Kent | Computing

Modifying a tree

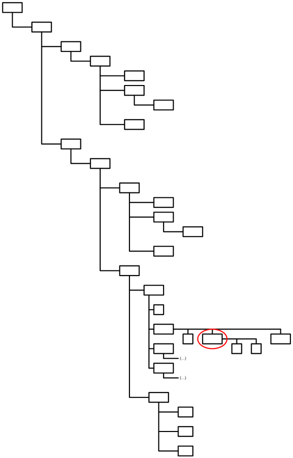


University of Kent | Computing

Modifying a tree



Identifying the right place

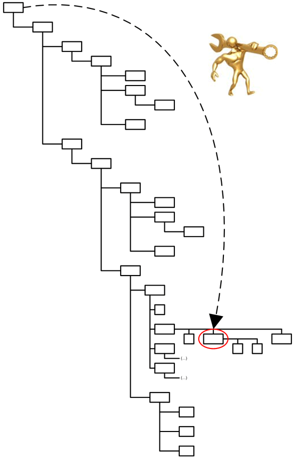


- There are no unique identifiers for nodes
  - Awkward to add them
- Cannot match by equality – we only want to modify a particular use of variable “x”
- Only uniquely identifying thing is the position

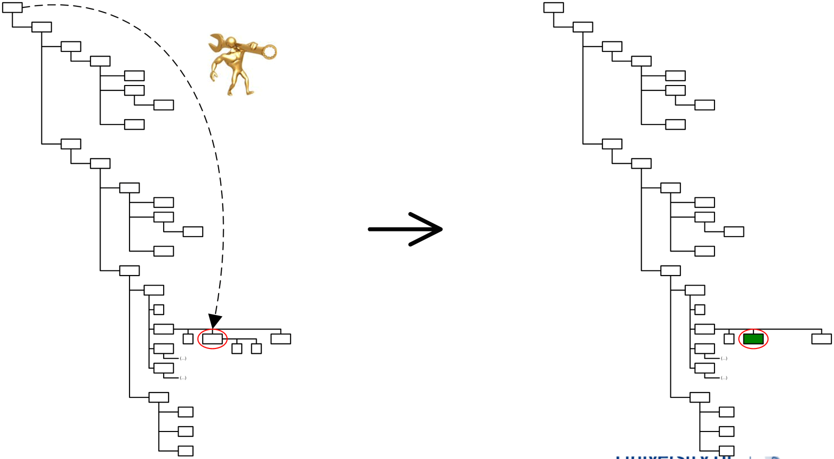
Modifying a single node

Expression –>MyMonad Expression

Modifying a tree

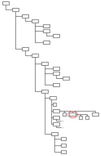
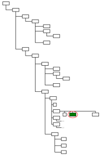


Modifying a tree



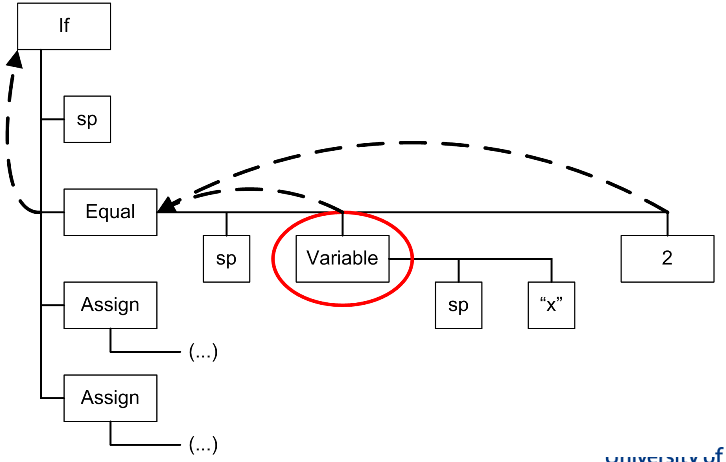
University of Kent Computing

Wrapping the modifier

→( → )

(Expression -> MyMonad Expression) -> (AST -> MyMonad AST)

Modifying a tree



University of Kent | Computing

Pure Haskell Solution

```
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)})
```

Pure Haskell solution

```
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')})
```

Generics solution

- Define decompN functions (see paper), and helper functions:

```
decomp3 :: (Monad m, Data b, Typeable a0, Typeable a1, Typeable a2)
  => (a0 -> a1 -> a2 -> b) ->
     (a0 -> m a0) -> (a1 -> m a1) -> (a2 -> m a2) -> (b -> m b)

mod2of3 con f = decomp3 con return f return
mod3of3 con f = decomp3 con return return f
```

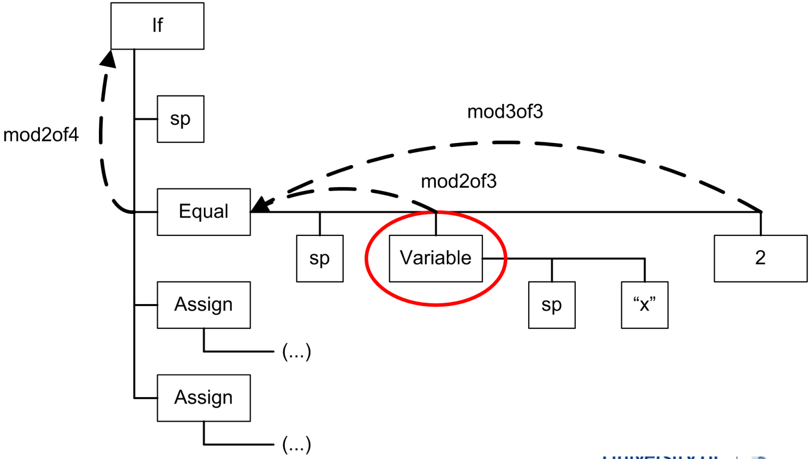
University of Kent | Computing

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)
```

University of Kent | Computing

Composing modifiers



Summary

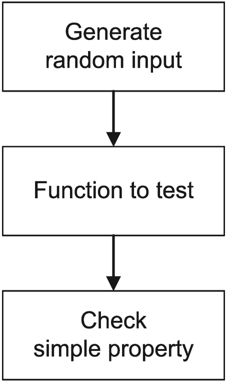
- Used SYB generics for two interesting applications:
  1. Pattern-matching
  2. Tree modification
- Not type-safe, and a little ad-hoc
- But: made our code shorter and more powerful
- Generics are a useful tool for doing even small things that are awkward in Haskell

Questions?

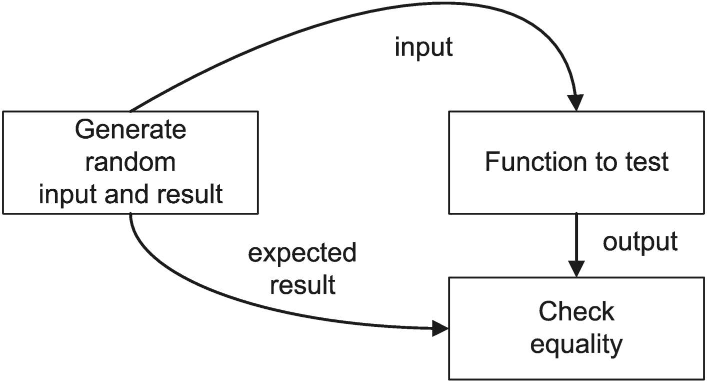
Why can't Pattern be parameterised?

```
data Pattern a = Anything
               | String :@ (Pattern a)
               | Structure Constr [Pattern a]
```

Ideal QuickCheck scenario

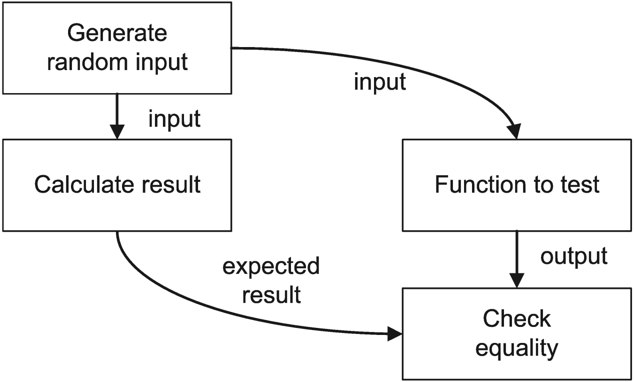


Common QuickCheck scenario



University of Kent | Computing

Redundant QuickCheck scenario



University of Kent | Computing