

Tock

Every compilation begins with a single pass

Neil Brown and Adam Sampson

Computing Laboratory
University of Kent

29 November 2007

Outline

- 1 Introduction
- 2 Abstract Syntax Tree (AST) Representation
- 3 Generics
 - Generics for nanopass
 - Difficulties with generics
- 4 Testing
 - Testing frameworks
 - Pattern-matching for tests

Quel est Tock?

- A new compiler for concurrent languages
 - Actually a translator
 - “Translator from **o**ccam to **C** from **K**ent”
- Uses nanopass architecture
- Implemented in Haskell

Nanopass compilation

- Parse source into abstract syntax tree
- Many small passes, each doing one thing to the AST
 - Transformations, annotations, checks...
 - Simple → easy to write and test
 - Easy to add new passes
- Final AST is the output from the compiler

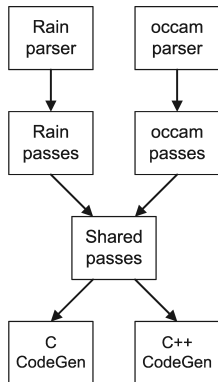


A brief history of Tock

- In A.D. 2007, it was becoming increasingly clear that we needed a new occam compiler. . .
- Adam wrote the initial occam-to-C Tock
 - Based on FCO “spike solution” compiler
 - Ideas from Matt, Christian and Damian’s “42” nanopass compiler
- Neil joined the project to add support for Rain and C++
 - and has done most of the work since then!

A multi-language compiler

- Multiple frontends
 - occam 2.1
 - Rain
- Multiple backends
 - C99 with CCSP
 - C++ with C++CSP
- Relatively straightforward to add new frontends and backends



An extensible compiler

- occam is very much still under development
- We want to experiment with new language and runtime features
- Should be accessible for student projects too
 - Our students know Haskell (in theory)

Why Haskell?

- Haskell has a really good parser library (Parsec)
- Other ideas from “42” looked transferable
- Lots of Haskell expertise at Kent
- Lots of existing compilers in Haskell

Representing the AST

- Existing nanopass frameworks are dynamically-typed. . .
- First attempt had just one data type
- Easy to write code to pattern-match over
- Very easy to make mistakes, though!

```
data Node = Seq [Node]  
  | While Node Node  
  | Name String  
  | ...
```

Better AST representation

- One data type per production
- Much more robust, easier to read
 - and it's what other compilers do

data Process = Seq [Process]
| While Expression Process
| ...

data Expression = Conversion Type Expression
| ...

- But how do we write transformations?

Scrap Your Boilerplate

- We use the Data.Generics module, which comes with GHC
- Provides introspection and dynamic typing in Haskell

```
foo :: Process -> Process
```

```
foo' :: Typeable a => a -> a
```

```
foo' = mkT foo
```

- Now foo' can be applied to any Typeable type

Scrapping Our Boilerplate

- everywhere applies a generic function everywhere that it can within a data structure
- We can use this to write transformation passes
 - e.g. find all the Names in the AST and change dots to underscores

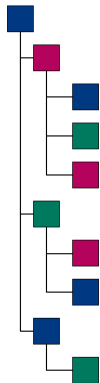
cStyleNames = everywhere (mkT doName)

where

doName :: Name -> Name

doName (Name s)

= Name [if c == '.' then '_' else c | c <- s]



Digging too deep

- everywhere really does look *everywhere*
 - e.g. at all the Chars in all the Strings...
- Very slow!
- No problem
 - Data.Generics provides lower-level functions so you can build your own traversals
 - Wrote a custom traversal function that skips over “uninteresting” types

Don't go any further

- Sometimes everywhere is not the traversal we want
 - e.g. we only want the outermost part of a recursive type
Par (Par a b) (Par c d)
- No problem
 - We can write different traversal functions for different passes

Well, *nearly* any type. . .

- Data.Generics has some limitations
- In particular, parametric types are awkward to work with
 - e.g. **data** Maybe a =Just a | Nothing
- Problem – we haven't found a good workaround!
 - As a result, the AST types aren't quite as neat as they should be. . .

Test-driven development

- Passes are small so the tests can be small.
- Passes can be tested independently.
- We test the parser, the passes and the code generation backends.

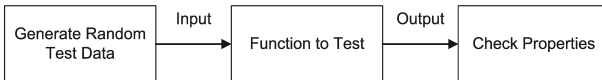
The test frameworks we use

■ HUnit

- Like JUnit; simple tests with assertions
- Best approach for most tests

■ QuickCheck

- Automatically generates testcases!
- Best for tests where properties of the output can be simply expressed



Some example tests

-- *HUnit example:*

```
assertEqual "Test 0"  
  "foo_bar"  
  (cStyleNames "foo.bar")
```

-- *QuickCheck example:*

```
prop_Reverse :: [Int] -> Bool  
prop_Reverse xs = reverse (reverse xs) == xs
```

Ignoring elements

Some items in the AST can be ignored when testing

- e.g. source positions from the parser

```
if (a) { }
```

parses as:

```
If  
  (Pos 0)  
  (Expr (Pos 4) (JustVar "a"))  
  (Block (Pos 7) [])
```

Ignoring elements

Could match the output exactly:

```
assertEqual "If Test 0"
  ( If (Pos 0) (Expr (Pos 4) (JustVar "a")) (Block (Pos 7) [])
    (parse "if (a) { }")
```

But it's more robust to use a pattern match:

```
assertBool "If Test 0" (checkIfTest0 (parse "if (a) { }"))
```

where

```
checkIfTest0
  ( If _
    (Expr _ (JustVar "a"))
    (Block _ [])
  ) = True
checkIfTest0 _ = False
```

Avoiding duplicate code

Testing that a pass generates the correct code to swap two variables:

```
temp42 := y
y := x
x := temp42
```

This has to be matched as:

Block

```
[ Assign (VarList ["temp42"]) (ExprList [JustVar "y"])
, Assign (VarList ["y"]) (ExprList [JustVar "x"])
, Assign (VarList ["x"]) (ExprList [JustVar "temp42"])
]
```

Avoiding duplicate code

We can remove some of the duplication using a helper function:

```
assertEqual "tempAssignTest 0"  
  (Block  
    [ singleAssign "temp42" "y"  
      , singleAssign "y" "x"  
      , singleAssign "x" "temp42"  
    ])  
  testOutput
```

```
singleAssign lhs rhs = Assign (VarList [lhs]) (ExprList [JustVar rhs])
```

But it's still got the name of the temporary hard-coded into it...

Making sure AST elements match

We don't care what the temporary variable is called – just that it's the same one that's used in both cases:

```
assertBool "tempAssignTest 0" (checktempAssignTest0 testOutput)
```

```
checktempAssignTest0 (Block
  [ Assign (VarList [tempVar0]) (ExprList [JustVar "y"])
  , Assign (VarList ["y"]) (ExprList [JustVar "x"])
  , Assign (VarList ["x"]) (ExprList [JustVar tempVar1])
  ])
  = tempVar0 == tempVar1
checktempAssignTest0 _ = False
```

But now we can't use our helper function any more.

Pattern-matching for tests

- Use Haskell's pattern matching?
 - Can't reuse repeated bits of pattern
 - Can't give a *helpful* error when the pattern match fails
- Language extensions?
 - First-class patterns (and similar)
 - But this isn't implemented yet. . .
- Parser combinators?
 - Would work, but the syntax is awful (especially for large patterns)

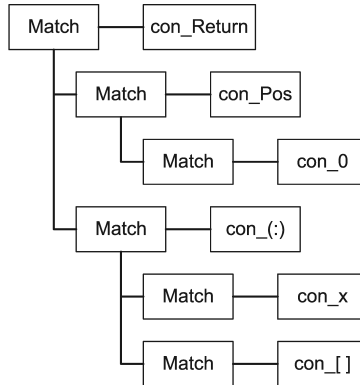
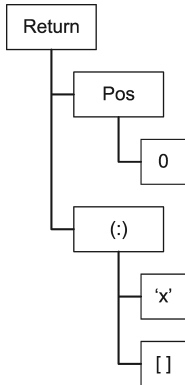
Generic pattern-matching

- Built our own pattern-matching library using Data.Generics

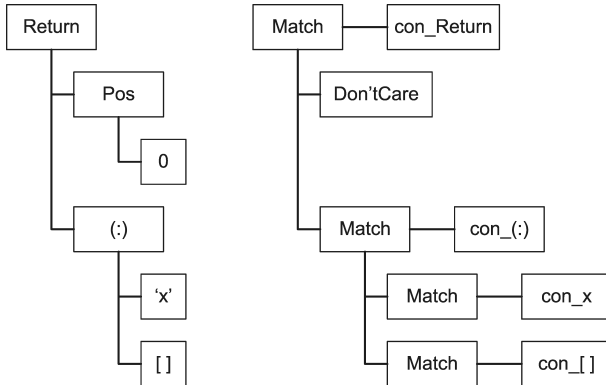
```
data Pattern =  
  Don'tCare  
  | Named String Pattern  
  | Match Constr [Pattern]
```

- Can match any Haskell data (not functions)

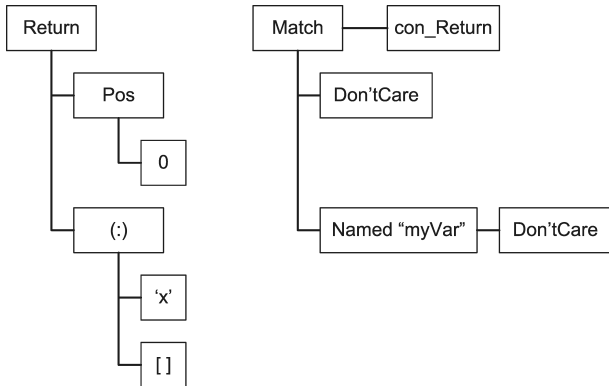
Building a pattern: return "x";



Building a pattern: return "x";



Labeling a pattern: return ??;



label @@ patt =Named label patt

Patterns are coming to town

- A Pattern to match our assignment example:

```
assertPatternMatch "tempAssignTest 0"  
  (mBlock  
    [ singleAssign' ("tempVar" @@ Don'tCare) "y"  
      , singleAssign' "y" "x"  
      , singleAssign' "x" ("tempVar" @@ Don'tCare)  
    ])  
  testOutput  
  
singleAssign' lhv rhv  
  = mAssign (mVarList [lhv]) (mExprList [mJustVar rhv])
```

Pattern's little helpers

- Definitions of the helper functions:

mBlock = tag1 Block

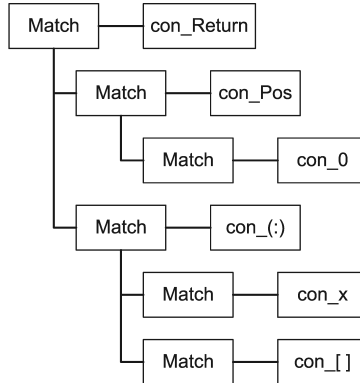
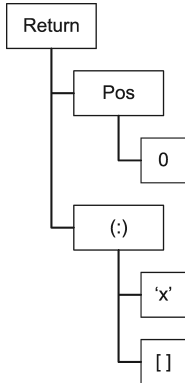
mAssign = tag2 Assign

tag2 con x0 x1 = Match (toConstr con) [checkPatt x0, checkPatt x1]

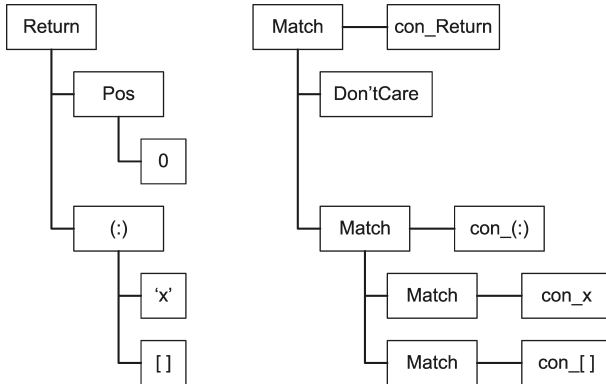
checkPatt :: Typeable a => a -> Pattern

checkPatt x = **if** x *has type Pattern* **then** cast x **else** makePatt x

Masking a pattern



Masking a pattern



Generic pattern matching

- We can mask out a special Pos value:

```
( If  
  (Pos 0)  
  (Expr (Pos 4) (JustVar "a"))  
  (Block (Pos 7) []))
```

```
assertPatternMatch "If Test 0"  
  (stopCaring badPos  
    ( If badPos  
      (Expr badPos (JustVar "a"))  
      (Block badPos [])))  
  (parse "if (a) { }")  
where badPos = Pos 999
```

Generic pattern matching

- Removed duplication, improved readability
- More helpful errors when match fails
- We can match anything that's an instance of Typeable
 - Functions are trickier
- Patterns are checked for consistency as they're applied
- Speed's not great
 - Doesn't matter for tests
 - But if it were faster, we could use it in real passes

† Insert generic swear-word here

- Bad Constr equality
 - `toConstr False == toConstr Nothing †`
- No easy way to discover number of parameters for a Constr
 - Have to map “const undefined” over the sub-terms and count the length of the returned list †
- Can't pass a type as a parameter
 - Must use things like: “undefined :: Int”

Conclusion

- Tock is built on nanopass architecture
- Generics allow us to implement small passes easily
- Small passes → easy to test
 - ~1800 unit tests in the compiler
- Generics also allow us to test passes cleverly

Any questions?

Same time, same place

Next week in the second talk:

- Why occam is interesting to compile
- Parser combinators in Haskell
- Navigating tree structures, generically
- Generating C/C++
- Reflections on using Haskell