

Tock

A nanopass presentation

Adam Sampson and Neil Brown

Computing Laboratory
University of Kent

6 December 2007

Outline

- 1 Introduction
- 2 occam language features
- 3 Parsing
- 4 Code analysis
 - Tree navigation
 - Usage checking
- 5 Backends
- 6 Reflections
 - How we use Haskell
 - Haskell frustrations

What we're assuming you know

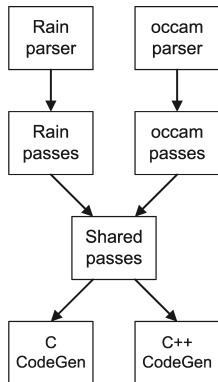
- Haskell syntax
- Monads

Brief recap: what is Tock?

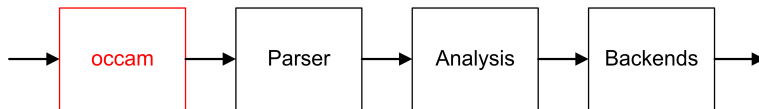
- A new compiler for concurrent languages
 - Actually a translator
 - “Translator from **o**ccam to **C** from **K**ent”
- Uses nanopass architecture
 - Many small independent passes
- Implemented in Haskell

Brief recap: 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



occam



What makes occam interesting to compile

- Parallelism!
- Indentation-based syntax
- Parser needs lots of lookahead
- Compile-time usage checks
 - Parallel safety
 - Definedness
- Abbreviations

Runtime checks

- Various things are checked at runtime in occam
 - Over/underflow on arithmetic operations
 - Data type conversions (e.g. INT32 \rightarrow INT16)
 - Array bounds
- Can compile out these checks when statically proven safe

Abbreviations

- occam doesn't have pointers
- Instead, it has a very powerful reference (“abbreviation”) system

```
INT x:           -- declares variable
INT y IS x:     -- declares reference to variable
VAL INT y2 IS y:
    -- declares read-only reference to variable

PROC foo (INT z) -- procedure that takes a reference
    z := 42
:
```

Array abbreviations

- occam's arrays have some interesting features
 - slicing
 - compile-time and run-time bounds checks

```
[10] INT xs:           -- declares an array
SEQ
xs[15] := 42          -- would fail at compile time
xs[n] := 42           -- would fail at runtime if n >= 10

INT y IS xs[4]:      -- abbreviate a member of the array
y := 42

[4] INT ys IS [xs FROM 5 FOR 4]:
  -- abbreviate slice of array
ys[2] := 42
```

Implementing multi-dimensional arrays

- In C...
 - Pointer for data
 - Array of sizes (one for each dimension)
 - Multidimensional arrays flattened to one dimension
 - Insert bounds checks into C code that accesses arrays
- In C++...
 - Can't just use the STL vector class – no slicing!
 - Wrote our own class to provide the occam array semantics

Retrying

An unusually low-level feature...

```
INT32 i:
```

```
[4]BYTE bs REYPES i:
```

```
[4][3]INT xs:
```

```
[2][6]INT ys RESHAPES xs:
```

- Reinterprets data as a different type
- Like a C pointer cast, but with safety checks
- Straightforward for C or C++...
- Arrays can be reshaped

More retyping

```
CHAN INT ci:
```

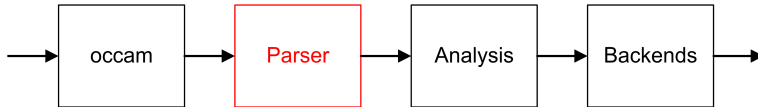
```
CHAN [4]BYTE cb RETYPES ci:
```

- Channels can also be retyped
 - Very awkward for the C++CSP backend!
 - Can't use templated channels

Mobiles

- A *mobile* is a safe mutable reference
- Regular mobiles
 - Only one process can use any particular mobile
 - ... but ownership can be transferred – e.g. by sending the mobile down a channel
 - Can be smart about allocation – recycling, relocating
- Shared mobiles
 - Protected by a lock
 - Requires either reference counting or garbage collection

Parser



Parsec

- Combinator-based parsing library
- Productions are monadic operations that return the thing they parsed

reserved :: String -> Parser String

reserved "CHAN" -- *matches and returns "CHAN"*

- Operators combine productions

dataType = reserved "INT" <|> reserved "BYTE"

channelType = reserved "CHAN" >> dataType

What makes Parsec cool

- Does Prolog-style backtracking
 - No lookahead needed
 - try lets you decide when to commit to a particular branch (like Prolog cut)
- Can pass state around for awkward languages
 - occam parser uses this to track types of symbols
 - e.g. "expected integer constant"
 - But we probably want to avoid this in the future

What makes Parsec *really* cool

- Productions are more-or-less BNF with Haskell syntax

statement

= while <|> communication <|> assignment

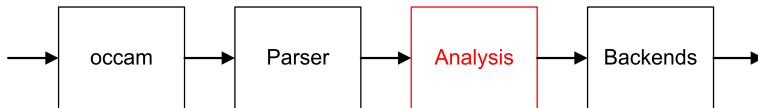
while

= **do** reserved "while"
reserved "("
e <- expression
reserved ")"
s <- statement
return (While e s)

Nanopass parsing

- Our frontends are nanopass-based too
- For occam:
 - Preprocess
 - Tokenise (based on Alex)
 - Join continuation lines
 - Convert indentation to markers
 - Parse
 - Resolve grammar ambiguities
- Can use QuickCheck to generate test code

Analysis



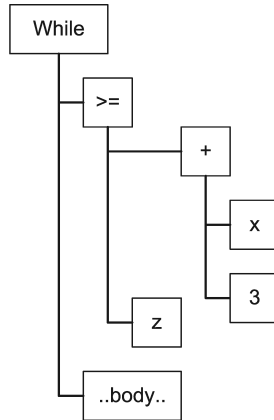
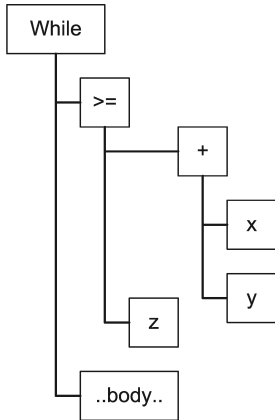
Control-flow graphs

- We need to analyse the program for various purposes
 - Parallel usage checking
 - Definedness checking
 - Optimisation
 - Automobalisation
- Many algorithms are easier to express on a CFG than on the AST directly
- Generating the CFG from the AST is easy

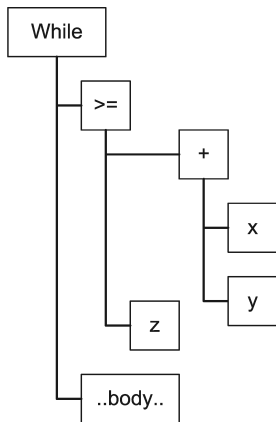
Modifying the AST

- Once we've made a decision based on the CFG, we need to go and change the AST
- But how?
 - No unique identifiers in AST
 - Only unique thing is the position in the tree
 - No easy way to find *and modify* a particular spot in a tree

Navigating the AST



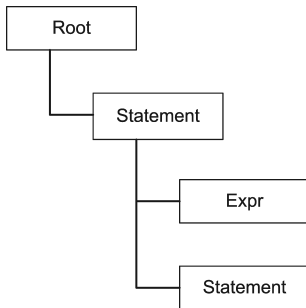
Navigating the AST



- Remember the path through the tree
 - 1st child, 1st child, 2nd child
- Not quite as easy as that in Haskell (the types vary!)

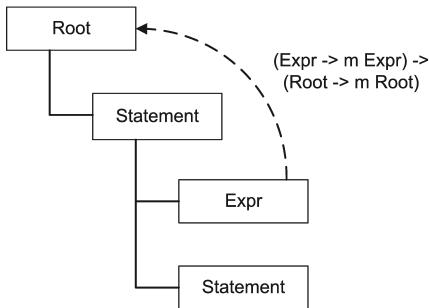
A hop, a skip and a jump

- Wrap modifier functions as we descend the tree



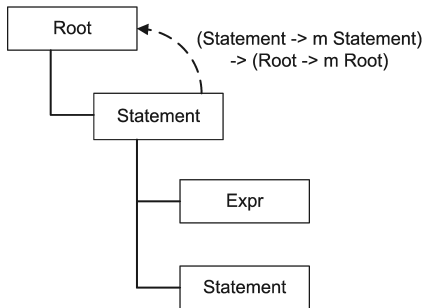
A hop, a skip and a jump

- Wrap modifier functions as we descend the tree



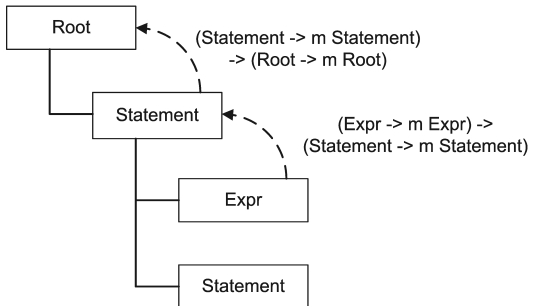
A hop, a skip and a jump

- Wrap modifier functions as we descend the tree



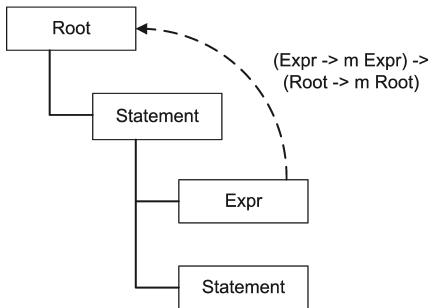
A hop, a skip and a jump

- Wrap modifier functions as we descend the tree



A hop, a skip and a jump

- Wrap modifier functions as we descend the tree



$\sum \lambda_n = \text{Tedium}$

- Writing it out by hand is tiresome:

buildNode (While e s) modifier

= **do** addNode (label e) (append modifier

— *(Expr → m Expr) → (Statement → m Statement)*

(\ f (While e s) → **do** {e' ← f e ; **return** (While e' s)}))

buildNode p (append modifier

— *(Statement → m Statement) → (Statement → m Statement)*

(\ f (While e s) → f s >>= **return** . While e))

...

Generic Route-Building

- First-class patterns would help here, again
- Another use for generics

buildNode (While e s) modifier

= **do** addNode (label e) (change1of2 modifier While)
 buildNode s (change2of2 modifier While)

...

change1of2 modifier con f = append modifier (modify2 con f **return**)

change2of2 modifier con f = append modifier (modify2 con **return f**)

modify2 :: Monad m =>

(a0 -> a1 -> a) -> (a0 -> m a0) -> (a1 -> m a1) -> a -> m a

Definedness checking

- Compiler must detect use of undefined objects
 - e.g. using a mobile you've communicated away
- Reasonably straightforward to do with the CFG
- Must be conservative – if you can't tell it's safe, it isn't!

Making it go faster

- Same type of analysis can be used for optimisations
- In occam, mobile types are explicit

```
SOME.TYPE fixed:  
MOBILE SOME.TYPE mobile:
```

- In Rain, everything appears non-mobile, and the compiler works out whether objects should be moved or copied

```
out ! bigList;                               // Send copy  
bigList = bigList + smallList;  
out ! bigList;                               // Send reference  
bigList = smallList;
```

Parallel usage checking

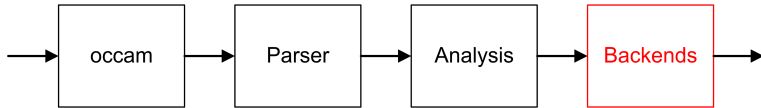
- Compiler must detect and disallow unsafe parallel usage of objects
 - e.g. enforce CREW: Concurrent Read, Exclusive Write
- Annotate AST/CFG nodes with objects used
 - e.g. "reads x , writes y, z "
- Check for consistency at **PAR** blocks

But it's not that simple...

```
[11] CHAN INT cs:  
PAR i = 0 FOR 10  
  pipeline.cell (cs[i]?, cs[i + 1]!)
```

- Array elements can be indexed by an arbitrary expression
- Can't just reason about the whole array
- Need to prove that i and $i + 1$ don't overlap, or that $a[i]$ is defined
 - Existing compiler brute-forces it!
 - Could use a proper solver (e.g. the Omega test)
 - Could do simple pattern matching – $(i + C) \setminus N$ covers 99% of cases
- Again, must be conservative by default

Backends

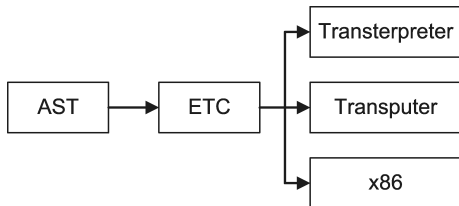


Why compile to C/C++?

- Want to support multiple platforms
 - IA32, x86-64, PowerPC, Cell, ARM. . .
 - Writing code generators is boring!
- For C, use CCSP via the CIF interface
- For C++, use C++CSP
- Existing compilers are very mature and generate good code

Generating code

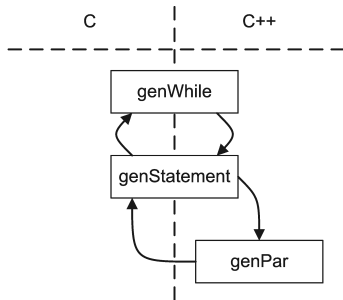
- Going via bytecode throws away a lot of useful information for optimisation



- Generate idiomatic code – as a human would
 - That's what compilers are designed to handle
- Result: Tock's output is $\approx 8x$ faster than tranx86's for numeric code

Sharing backend code

- C and C++ structure is very similar
- Share functions for common constructs
 - Must support (mutual) recursion
 - Use "virtual function tables"
 - We're not reinventing OO, honest
 - Can override functions with stubs for unit testing



Sharing backend code

```

type GenOps = GenOps { ...
  genWhile :: GenOps -> Expression -> Statement -> CodeGenM (),
  genExpression :: GenOps -> Expression -> CodeGenM (),
  genStatement :: GenOps -> Statement -> CodeGenM (),
  ...
}

```

```

genWhileC :: GenOps -> Expression -> Statement -> CodeGenM ()
genWhileC ops exp body
  = do tell ["while("]
      call ops genExpression exp
      tell [")"]
      call ops genStatement body

```

Testing backend code

```
-- genWhile :: GenOps -> Expression -> Statement -> CodeGenM ()
-- genExpression :: GenOps -> Expression -> CodeGenM ()
-- genStatement :: GenOps -> Statement -> CodeGenM ()
```

```
testgenWhile
  = assertEquals "testgenWhile"
    "while(@@@)###"
    (runCodeGen (genWhileC whileOps undefined undefined))
where
  whileOps = GenOps { genExpression = (\_ _ -> tell ["@@@"]),
                    genStatement = (\_ _ -> tell ["###"]) }
```

Using undefined saves constructing a value,
and makes sure it's not evaluated!

Stack allocation

- When starting a process, must allocate stack for it
- But how much?
 - Can't tell directly, because we're not doing the code generation
 - Could use a size that's "always big enough" – but that's inefficient

Stack usage analysis

- Run the C/C++ compiler, then analyse the generated assembler code
 - Look for stack adjustment instructions
 - Compute an upper bound per function
 - Relink the program with the computed sizes inserted
- Some things are hard to analyse
 - Recursion, alloca, exceptions, virtual functions. . .
 - But we can control when we generate these (or avoid them entirely)

Reflections on Haskell

How're you gonna get your day's work done?

- Laziness isn't very useful
 - Nothing uses infinite data structures
 - Really want strict evaluation in most places
 - Force errors sooner!
 - Except: some utility in using “undefined” when testing

Monads in disguise

- Tock is a *practical* system, therefore...
- Monads, monads, monads
- Not a lot of pure functional code
- Heavy use of monad transformers

type PassM = ErrorT ErrorReport (StateT CompilerState IO)

type CodeGenM = WriterT [String] PassM

A fistful of dollars

- Use the `$` operator to avoid nested brackets
 - `liftIO $ putStr $ "quux is " ++ (fmt $ get quux)`
`liftIO (putStr ("quux is " ++ (fmt (get quux))))`
 - Adam got this from John Meacham's code...
- Define some helper operators for common tasks
 - e.g. applying a pure function to the result of a monadic operation
 - `(>>*) :: m a -> (a -> b) -> m b`
 - `v <- getFoo >>*fromMaybe`

Go Go Gadget Extensions!

- Several modules, obviously. . .
- forall – functions to apply operators to numeric types in evaluator
- Undecidable instances – typeclass synonyms

Debugging is hard

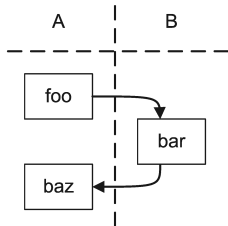
- Tracing does not generally do what you expect (owing to laziness)
- Existing tools often have limitations that make them unworkable with Tock code
 - e.g. no GHC extensions
- Unit tests do help to narrow down problems
- We often resort to printf debugging (that's why IO is in PassM)

Deciphering type errors

- GHC's errors are usually enormous and hard to understand
 - The line number is usually the most useful bit!
- Sometimes GHC can infer the right type, but a human can't
 - We use lots of **where** clauses
 - Would be nice to be able to ask GHCi about definitions inside them
- We often resort to trial and error

Going in circles

- Haskell disallows mutually recursive modules
 - A.foo calls B.bar, which calls A.baz
- But we want to do this quite often
 - e.g. evaluating constant expressions
- (Yes, there's the hs-boot mechanism. . . yuck!)
- Some of our modules are a bit arbitrary at the moment



A pain in the monads

- Monad transformers aren't available for all monads
 - e.g. Parsec, QuickCheck
 - You can't have a parser or test that does IO (safely!)
- Monad transformer typeclass instances (e.g. MonadState) aren't derived automatically

Future Work

- Support for more occam-pi features
- ETC backend, for Transterpreter
- Better usage checking
- Language experimentation
 - Rain
 - occam enhancements

Any questions?