Compiling occam to C with Tock

Adam Sampson

ats@offog.org

University of Kent

http://www.cs.kent.ac.uk/

Introduction

- We do most of our work with occam-π
- Big new project starting in a couple of months
- Existing compiler:
  - Derived from Inmos's original compiler
  - Poor straight-line code performance
  - Enormous codebase
  - Hard to maintain and extend
- . . . so we've been working on replacing it

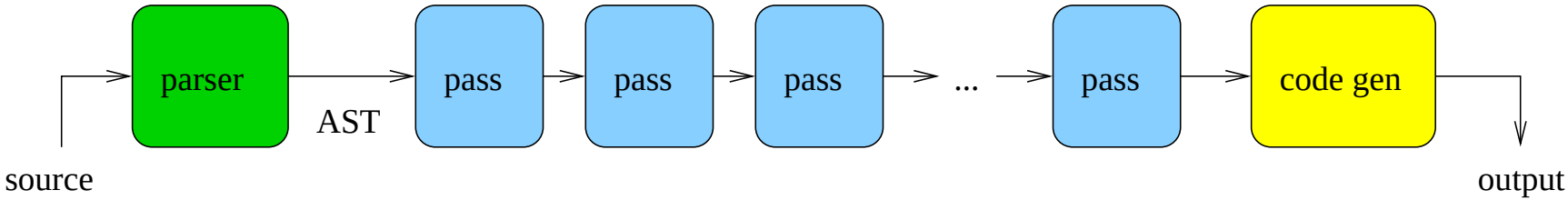
Previously on CPA...

- 2004: Jacobsen/Jadud, *The Transterpreter: A Transputer Interpreter*
  $\rightarrow$ The Transterpreter – portable occam runtime
- 2005: Barnes, *Interfacing C and occam-π*
  $\rightarrow$ CIF – C bindings to occam runtime
- 2006: Jacobsen/Dimmich/Jadud, *Native Code Generation using the Transterpreter*
  $\rightarrow$ 42 – nanopass occam compiler
- 2006: Barnes, *Compiling CSP*
  $\rightarrow$ NOCC – rewrite of occ21

Tock

- A new occam compiler (currently supports occam2.1 and some of occam-π)
- Generates efficient, portable C99 code
- Uses the existing KRoC runtime through CIF
- Implemented using Haskell
  - Lazy functional language, many users at Kent
  - Widely used for compiler implementation
  - Indentation-based, supports lightweight concurrency, . . .
- Designed to be easy to understand and extend

Nanopass compilation



- Parser turns source code into an AST
- Many small passes transform the AST
  - Simplifying, restructuring, annotating, checking. . .
  - Each pass does one thing only
- Output simply generated from the final AST
- Can be more complicated than this – e.g. usage checker

Parsing

- Uses *Parsec* – combinator-based parsing library
- Each production is a monadic function that returns an AST fragment for the thing it's matching (e.g. "a SEQ process", "an expression of type T")

```
sequence
    = do { sSEQ ; eol ; indent ;
           ps <- many1 process ; outdent ;
           return (Seq ps) }
```

- Operators provided to combine productions (e.g. "X or Y", "X then Y")

```
specifier = dataType <|> portType <|> ...
```

Parsing problems

- Parsing occam is slightly complicated
- Tokeniser must keep track of indentation
- Parser needs to check types to resolve ambiguities (e.g. in `c ! x`, is `x` a variable or a variant tag?)
- Parsec can do Prolog-style backtracking and cuts to handle ambiguous productions – it has “infinite lookahead”
- The syntax in the occam2.1 manual contains a number of errors

Passes

- Turn the occam AST into something closer to C
- Some of the passes in Tock:
  - Resolve user-defined types
  - Convert `FUNCTION`s to `PROC`s
  - Simplify array expressions
  - Wrap `PAR` processes in `PROC`s
  - Convert free names to arguments
  - Move nested `PROC`s to top level
- Different target languages would need different passes

How passes work

- Match patterns in the AST and apply transformations to them
- Uses Haskell's "Scrap Your Boilerplate" generic functions and pattern matching

```
cStyleNames = everywhere (mkT doName)
  where
    doName :: Name -> Name
    doName (Name s) =
      Name [if c == '.' then '_' else c
              | c <- s]
```

- Can use different traversal approaches as appropriate

Generating C code

- Output language is C99 – latest C standard
  - Inlining, better scoping, numeric types, better maths library. . .
- Tries to generate the same code a human would write
  - Compiler can do a better job of optimisation
  - Easier to debug with standard tools
- Better runtime error reporting than occ21

Example: occam code

```
PROC integrate (CHAN OF INT in, out)
  INT total:
  SEQ
    total := 0
    WHILE TRUE
      INT n:
      SEQ
        in ? n
        total := total + n
        out ! total
:
```

Example: Tock C code

```
void integrate_u6 (Process *me,
  Channel *in_u2, Channel *out_u3) {
    int total_u4;
    total_u4 = 0;
    while (true) {
        int n_u5;
        ChanInInt (in_u2, &n_u5);
        total_u4 = occam_add_int (total_u4,
          n_u5, "demo.occ:13:18");
        ChanOutInt (out_u3, total_u4);
    }
}
```

But you can't do *that in C!*

- CIF *mostly* hides the details of doing occam-style scheduling with C processes
- Don't need to worry about context switching
- Must allocate an appropriate amount of stack for each process
  - Analyse the output of the C compiler, looking for stack adjustment instructions

Whole-program compilation

- Tock translates the entire program to C at once, including libraries
- Allows better optimisation (e.g. inlining)
- Takes longer, though!
- Libraries should be parsed and checked ahead of time

A comparison with SPoC

- SPoC also generates C from occam
- Compiles in its own occam runtime
- Avoids stack usage entirely by putting local variables in structures
- Avoids context switching by compiling each `PROC` into a state machine
- . . . which makes the code hard to optimise
- Limited runtime checks

Example: SPoC C code

```
void P_integrate_accumulate
    (tSF_P_integrate_accumulate *FP) {
  while (true) {
    switch (FP->_Header.IP) {
    CASE(0): FP->total_55 = 0;
             GOTO(1);
    CASE(2): INPUT4(FP->in_53, &FP->n_56, 3);
    CASE(3): FP->total_55 =
               FP->total_55 + FP->n_56;
             OUTPUT4(FP->out_54,
               &FP->total_55, 4);
    CASE(4):
    CASE(1): if (true) GOTO(2);
             RETURN();
    ...
```

How much faster?

- Benchmark: compute 1000x1000 Mandelbrot set at double precision, convert to packed bitmap image, and compute checksum
- Exercises real and integer maths, but not communication

Compiler	Time per image (ms)
KRoC	3,889
SPoC	409
Tock	450

- Note that SPoC does no range/overflow checking!

Compiler size comparison

Compiler	Language	Lines of code
occ21 (KRoC)	C	150,000
NOCC	C	70,000
occ2c (SPoC)	C/GMD	24,000
Tock	Haskell	7,000

- Estimate Tock will be <15,000 lines for full occam-π support
- Tock should be more accessible for students and casual experimenters

Future plans

- Finish full occam-π implementation
- Better usage checking
- Precompiled library support
- Implement CIF on the Transterpreter runtime
  - More portable
  - Should be much faster than CCSP on uniprocessors
- Investigate alternative backends
  - C++CSP
  - ETC bytecode

One more thing. . .

- (Nothing to do with Tock)
- For years, we've been getting students on our parallelism course to write ASCII-art demos
- We have SDL bindings for occam-π already. . .

Occade

- occam-π module for writing simple graphical arcade games
- All based around the client-server pattern
- Features:
  - Sprites
  - Text
  - Background playfield
  - Collision detection
  - Input events
- Here's a demo. . .

The end

- Any questions?
- For more on Tock, see:
  `http://offog.org/tock`
- For more on Occade, see:
  `http://offog.org/occade`