OLL Compiler Project
–
Status at 201516

Barry M Cook

Independent project

Barry@hmh.f2s.com

OLL – What is it?

- The latest iteration of a ~20-year experiment
- A Compiler for a language supporting concurrency
- Targeting various platforms …
  - 1995: C
  - 2000: VHDL = Hardware (FPGA)
  - 2013: ARM
  - 2015: JVM, ARM, VHDL

Another Language?

Thesis:

- A large number of programming errors are caused by a mismatch between the problem being coded and the features of the language being used.
  - The coder loses sight of the problem when forcing it to fit a mismatched implementation language

Application Specific Languages

- Can include application specific shortcuts, rules, etc.
- Can optimise better for being constrained
- Are easier to use
- Reduce coding time
- Are more likely to produce error-free code

Application Areas

- For example:
  - Education
  - Business
  - Scientific
  - Engineering
    - Electronic
    - Chemical
    - ...
  - Web / internet
  - Embedded

Concurrency and Compilation

- Why concurrency?
  - Natural way to express algorithms
- Why a compiler?
  - Can choose syntax
  - Greater control than library
    - Error detection can be better
    - Better optimisation leads to faster code

My Application Area

- Embedded systems
  - Must keep running … No run-time errors
- Protocols
  - Between PC, Microcontroller, Hardware
  - Want ONE piece of code – not 2 or 3 which may not behave exactly the same way
- Time is very important
- ? Education ?
  - Concurrency is easy (etc.)

Main Goals

- Match my application area (easy to use)
- Prevent (i.e. detect the possibility at compile time)
  - Subscript-out-of-bounds
  - Numerical overflow
  - Deadlock
  - Mismatched units in calculations
  - ...

My Targets

- SAME source code for all targets (Hardware/Software “SameDesign”)
  - JVM
    - For user interaction / portability
      - Also easy to instrument / display operation
  - VHDL – for hardware (FPGA)
    - Naturally concurrent = fast
  - ARM (Cortex M3)
    - Widely used
- Need to take a subset of everything possible
    - e.g. Dynamic allocation is harder in hardware … omit or defer it

Personal Preferences

- I'm writing the compiler so I don't need to follow the usual conventions …
  - No reserved words
  - Source as lines – error containment
  - Subscript range checking at compile time
  - Create new data types
    - e.g. Fruit is Orange, Apple, Pear
  - Array slices
  - Array index other than integer
  - Units on values
  - Predictable numerical accuracy

More Personal Preferences

- SEQ by default
- Indented (like Python)
- One loop
- One choice
- Any bracket shape is OK (no need to remember which one to use)
  - `[{x + ([a+b] * [c+d])} / {x - ([a+b] * [c+d])}]`
  - `((x + ((a+b) * (c+d))) / (x - ((a+b) * (c+d))))`
- Deliberately chosen different words – reduce mis-choice
- Real numbers after Gustafson's UNUM's (?)

Reminder

- This is a personal voyage of discovery
- The result is intended to make MY life easier for what I do
- Things are still changing – I'm trying things and rejecting more than I keep
  - (Because I can) I'm changing the words / syntax / features to help with usage, optimisation AND making the compiler easier to write
    - What you see today might be different next month

Credits

- I've taken things from many existing languages – and rejected even more things from them
  - The most notable contributors, in alphabetical order, are
    - Ada, Algol60, Algol68, Assemblers (many), BASIC, BCPL, C, COBOL, FORTRAN, Java, occam, Pascal, Verilog, VHDL
  - Each of the above has at least one thing I like – and at least one thing that I dislike.

First example program

```
Program eg1 (* context information *)
  Let display := “Hello World!”
```

Hello World!

Second example program

```
Program eg2 (* context information *)
  SEQ i = 10..1
    Let display := i'"%d"; newline
  Let display := "BANG!"
```

10

9

8

...

1

BANG!

Timed example program

```
Program eg3 (* context information *)
  SEQ i = 10..1 @ 1s
    Let display := (i'“%d”; newline)
  Let display := “BANG!”
```

10

9

8

...

1

BANG!

UART Transmitter

```
SEQ @ baud rate
  LET tx := 0; data[0..7]; 1
```

An Example of Data Typing

- An integer is NOT an array of bits
- Conversion needs specification
  - LET integer := array'UNSIGNED

```
TYPE short       = 0..2^8-1
TYPE bit         = 0, 1
TYPE word index  = 0..15
TYPE word        = bit ( word index )
  FIELD ms(short) = (15..8)'UNSIGNED
  FIELD ls(short) = ( 7..0)'UNSIGNED

VAR X (word) := initial value
LET X.ms := 255 - X.ls
```

One-place buffer

```
PROC buffer1(IN input(type), OUT output(type))
  VAR  buffer(type)
  TYPE states         = empty, full
  VAR  state(states) := empty
  SEQ ..
    ALT
      WHEN state
        IS empty
          AWAIT buffer := input
            LET state := full
        IS full
          AWAIT output := buffer
            LET state := empty
```

One-place buffer

```
PROC buffer1(IN input(type), OUT output(type))
  VAR buffer(type)
  SEQ ..
    LET buffer := input
    LET output := buffer
```

Or

```
PROC buffer1(IN input(type), OUT output(type))
  SEQ ..
    LET output := input
```

Choice

```
WHEN x
  IS 1
    (*...*)
  IS 2,4,6
    (*...*)
  NOT 0..10
    (*...*)
  ELSE                (* 0,3,5,7,8,9,10 *)
    (*...*)
```

Exactly one path must selected

(Unless as a guard in an ALT, when it is also acceptable for no path to be selected)

Buffer (1 of 2)

```
PROC FIFO(IN input(type), OUT output(type),
          CONST size[1..] := 1000          )

  TYPE Buffer address = 0..buffer size
    ATTRIBUTE NEXT(Buffer address)
      WHEN $  (* this *)
        IS buffer size: RETURN 0
        ELSE          : RETURN $ + 1

  TYPE BUFFER         = type[Buffer address]

  VAR  buffer         ( BUFFER         )
  VAR  write address  ( Buffer address ) := 0
  VAR  read  address  ( Buffer address ) := 0
```

Buffer (2 of 2)

```
SEQ ..
  ALT
    WHEN read address
      NOT write address              (* not empty *)
        AWAIT output := buffer[read address]
          LET read address :=
                                read address'NEXT


    WHEN write address'NEXT
      NOT read address               (* not full *)
        AWAIT buffer[write address] := input
          LET write address :=
                                write address'NEXT
```

Commstime (1 of 2)

```
PROGRAM commstime @ context

  TYPE INT           = 0..2^30-1
    ATTRIBUTE NEXT(INT)
      WHEN $  (* this *)
        IS   INT'MAX: RETURN 0
        ELSE        : RETURN $ + 1

  CHAN to_delta    ( INT )
  CHAN to_inc      ( INT )
  CHAN to_prefix   ( INT )
  CHAN to_reporter ( INT )
```

Commstime (2 of 2)

```
PAR
  SEQ (* Prefix *)
    LET to_delta := 0
    SEQ ..
      LET to_delta := to_prefix

  SEQ .. (* Delta *)
    LET n := to_delta
    PAR
      LET to_inc      := n
      LET to_reporter := n

  SEQ .. (* Inc *)
    LET to_prefix := to_inc 'NEXT

  SEQ (* Reporter *)
    (* read from to_reporter and time things *)
    SKIP
```

Commstime - Performance

- Hand-compiled to Java then JDK to runnable (Java 1.7.0.03 on i7-4770 at 3.4GHz)
  - Java code similar to that described in “A Fast C Kernel for Portable occam Compilers” (Cook) at WoTUG-18 (1995)
  - 68 ns / iteration (~231 clock cycles)
  - 17 ns / communication (~58 clock cycles)
  - 8.5 ns / context switch (~29 clock cycles)
- ~350 times the speed of JCSP on the same machine

[We could use a few more benchmarks]

Summary

- Ongoing work, still a long way to go and taking far longer than I'd like
- Output for the 3 targets (JVM, ARM, FPGA) shown to be feasible
- (Performance) Results are encouraging

Barry@hmh.f2s.com