The Semantics of Bluespec,
i.e., a Parallel Language with Guarded Atomic Actions and Modules

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

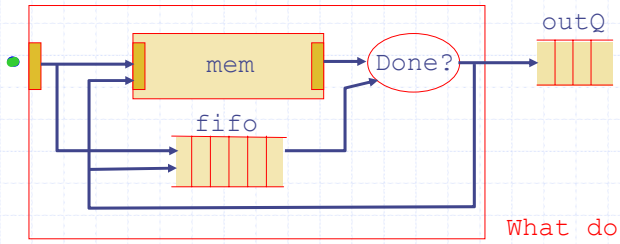
New work – just submitted to a conference
(with Nirav Dave and Michael Pellauer)

Is Bluespec a specification language or an implementation language?

Both – A Bluespec description admits non-determinism but a compilation (with a scheduler) results in a deterministic implementation.

Users want more control over the implementation than we have provided so far.

LPM in Bluespec



What do we expect when enq & deq are done together?

```
module lpm
rule "recirculate"
  x = mem.res(); y = fifo.first();
  if done?(x) then fifo.deq() ; mem.deq() ; outQ.enq(x)
              else mem.deq(); mem.req(addr(x));
                   fifo.deq(); fifo.enq(y)
action method enter(x) = mem.req(addr(x)) ; fifo.enq(x)
```


One Element FIFO

```
module mkFIFO1 (FIFO#(t));
  Reg#(t)    data  <- mkRegU();
  Reg#(Bool) full  <- mkReg(False);
  method Action enq(t x) if (!full);
    full <= True;     data <= x;
  endmethod
  method Action deq() if (full);
    full <= False;
  endmethod
  method t first() if (full);
    return (data);
  endmethod
endmodule
```

enq and deq cannot be enabled together!

Rule "recirculate" will never fire!

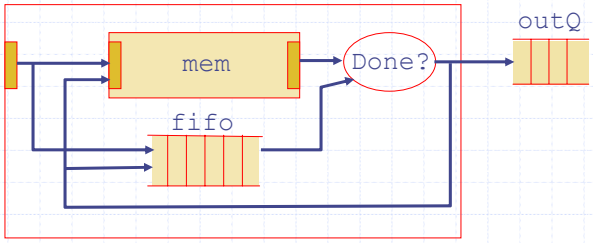
Two-Element FIFO

```
module mkFIFO2(FIFO#(t));
  Reg#(t)    data0 <- mkRegU;
  Reg#(t)    data1 <- mkRegU;
  Reg#(Bool) full0 <- mkReg(False);
  Reg#(Bool) full1 <- mkReg(False);

  method Action enq(t x) if (!full1);
    if (!full0) begin data0 <= x; full0 <= True; end
            else begin data1 <= x; full1 <= True; end
  endmethod

  method Action deq() if (full0);
    full0 <= full1;  data0 <= data1; full1 <= False;
  endmethod

  method t first() if (full0);
    return data0;
  endmethod
endmodule
```

enq and deq still conflict!

Even Nirav's clever solution on Quiz 2 will cause the rule to deadlock

What is missing?



rule "recirculate"
  x = mem.res(); y = fifo.first();
  if done?(x) ***then*** fifo.deq() ; mem.deq() ; outQ.enq(x)
  else mem.deq(); mem.req(addr(x));
  fifo.deq(); fifo.enq(y)

Need a way of
- saying deq happens before enq within the same rule
- building such a FIFO

The Bluespec Language

BS: A Language of Atomic Actions

A program is a collection of instantiated modules m_1 ; m_2 ; ...

Module ::= Module name
[State variable r]
[Rule R a]
[Action method g (x) = a]
[Read method f (x) = e]

```
a ::=   r := e
      | if e then a   <- Conditional action
      | a | a         <- Parallel Composition
      | a ; a         <- Sequential Composition
      | (t = e in a)
      | m.g(e)        <- Method call
      | a when e      <- Guarded action

e ::=   r | c | t
      | Op(e , e)
      | e ? e : e
      | (t = e in e)
      | m.f(e)
      | e when e
```

Execution model

Repeatedly:

- Select a rule to execute
- Compute the state updates
- Make the state updates

Highly non-deterministic

Primitives are provided to control the selection

Guards vs If's

- Guards affect the surroundings

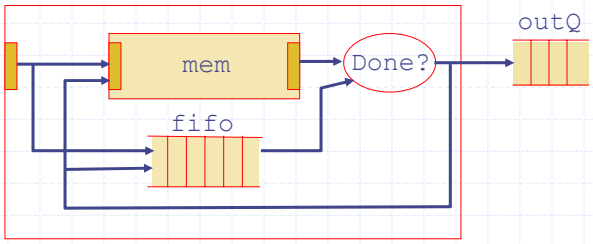
```
(a1 when p1)  | a2  ==>  (a1 | a2) when p1
```

- Effect of an "if" is local

```
(if p1 then a1) | a2 ==> if p1 then (a1 | a2) else a2
```

p1 has no effect on a2

LPM in Bluespec




```
module lpm
rule "recirculate"
  (x = mem.res() in (y = fifo.first() in
  (if done?(x) then fifo.deq() | mem.deq() | outQ.enq(x)
                 else  (mem.deq(); mem.req(addr(x)))
                     | (fifo.deq(); fifo.enq(y)))
action method  enter(x) = mem.req(addr(x)) | fifo.enq(x)
```

Notice

The plan

- Semantics of BS
- Lifting guards to the top
- Turning a BS specification into an implementation: Scheduling primitives
- Implementing the sequential connective

Semantics of a rule execution

- Specify which state elements the rule modifies
  - Let ρ represent the value of all the registers before the rule executes
  - Let U be the set of updates implied by the rule execution

BS Action Rules

reg-update
$$\frac{\rho \vdash e \Rightarrow v}{\rho \vdash (r := e) \Rightarrow \{(r, v)\}}$$

if-true
$$\frac{\rho \vdash e \Rightarrow \text{true}, \quad \rho \vdash a \Rightarrow U}{\rho \vdash (\textit{if } e \textit{ then } a) \Rightarrow U}$$

if-false
$$\frac{\rho \vdash e \Rightarrow \text{false}}{\rho \vdash (\textit{if } e \textit{ then } a) \Rightarrow \varnothing}$$

par
$$\frac{\rho \vdash a1 \Rightarrow U1, \quad \rho \vdash a2 \Rightarrow U2}{\rho \vdash (a1 \mid a2) \Rightarrow \text{pmerge}(U1,U2)}$$

Like a set union but blows up if there are any duplicates

seq
$$\frac{\rho \vdash a1 \Rightarrow U1, \quad \text{update}(\rho,U1) \vdash a2 \Rightarrow U2}{\rho \vdash (a1 \; ; \; a2) \Rightarrow \text{smerge}(U1,U2)}$$

Like pmerge but U2 dominates U1

BS Action Rules *cont*

a-let-sub

$$\frac{\rho \vdash e \Rightarrow v,\ \ smerge(\rho, \{(t,v)\}) \vdash a \Rightarrow U \quad (v\ !=\ \bot)}{\rho \vdash ((t = e)\ \textit{in}\ a) \Rightarrow U}$$

a-meth-call

$$\frac{\rho \vdash e \Rightarrow v, \quad \lambda x.a = lookup(m.g),\ smerge(\rho, \{(x,v)\}) \vdash a \Rightarrow U}{\rho \vdash m.g(e) \Rightarrow U}$$

Expression rules are similar

Guard Rules

a-when-ready

$$\frac{\rho \vdash e \Rightarrow true, \quad \rho \vdash a \Rightarrow U}{\rho \vdash (a\ \textit{when}\ e) \Rightarrow U}$$

- If no rule applies then the system is stuck and the effect of the whole atomic action is "no action" (returns $\bot$).
  - For example, if e evaluates to false then not just (a when e) results in no updates but the whole atomic action of which (a when e) is a part results in no updates.

Alternative let definition

a-let-sub
$$\frac{\rho \vdash e \Rightarrow v,\ \ smerge(\rho,(t,v)\})\vdash a \Rightarrow U}{\rho \vdash ((t = e);\ a) \Rightarrow U}$$

a-let-sub-bot
$$\frac{\rho \vdash e \Rightarrow \bot,\ \ smerge(\rho,(t,\bot)\})\vdash a \Rightarrow U}{\rho \vdash ((t = e);\ a) \Rightarrow U}$$

Lets don't need to be strict

Their expressions may not be ready as long as they aren't being used

BS: Guards vs If's

- A guard on one action of a parallel group of actions affects every action within the group

  (a1 when p1) | a2 ==> (a1 | a2) when p1
- A condition of a Conditional action only affects the actions within the scope of the conditional action

  (if p1 then a1) | a2 ==> if p1 then (a1 | a2) else a2

  p1 has no effect on a2 ...
- Mixing ifs and whens

  if p then (a1 when q1) else (a2 when q2)

  ≡ (if p then a1 else a2) when (p&q1 | ~p&q2)

Conditionals & Cases

if p then a1 else a2

≡ if p then a1 | if !p then a2

Similarly for cases

From specification to implementation

Repeatedly:

- Select a rule to execute ← Highly non-deterministic
- Compute the state updates
- Make the state updates

Simple fair scheduler: Does one rule at a time in a fixed static order

```
Introduce a counter cnt
Change (Rule R a when p) to
        (Rule R ((if cnt == i && p then a)
                | cnt := mod(cnt+1, n)))
```

- Totally sequential!
- Schedules rules that may not update any state

"When" lifting

- Suppose we lift all the guards (i.e., whens) to the top level of a rule and select a rule to schedule from only those rules whose guard is true.
  - All lifted guards can be evaluated in parallel if we have sufficient resources

"When" axioms

A1. (a1 when p) | a2 ≡ (a1 | a2) when p
A2. a1 | (a2 when p) ≡ (a1 | a2) when p

A3. (a1 when p) ; a2 ≡ (a1 ; a2) when p
A4. a1 ; (a2 when p) ≡ (a1 ; a2) when p'
where p' is p after the effect of a1

A5. if (p when q) then a ≡ (if p then a) when q
A6. if p then (a when q) ≡ (if p then a) when (q || !p)

A7. (a when p1) when p2 ≡ a when (p1 && p2)
A8. r := (e when p) ≡ (r := e) when p
A9. m.g(e when p) ≡ m.g(e) when p
similarly for expressions ...

Lifting "whens" to the top

- Closely related to the "when" axioms, e.g.,

  LW (a1 | a2) = (a1' | a2') when ($a1_G$ && $a2_G$)
  where (a1' when $a1_G$) = LW (a1)
  (a2' when $a2_G$) = LW (a2)

- Modules need special care because of guards (implicit conditions)

Implicit conditions

- Every method has two parts: guard and body. These will be designated by subscripts G and B, respectively
- Compiler splits every method into two methods, one of which is a value method corresponding to the guard part
- Replace each method call m.h(e) by
  ($m.h_B$(e') when e_G && $m.h_G$(e'))
  where (e' when e_G) = LW(e)

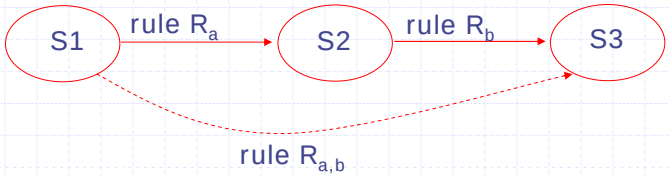
Complete when-lifting procedure

1. Apply LW procedure to each rule and method
2. Split each action method definition
   $g = \lambda x.(a \text{ when } p)$ into two methods
   $g_B = \lambda x.a$ and $g_G = \lambda x.p$
   Similarly for value methods.
3. For each rule of the form
   Rule R ((if p then a) when q)
   replace it with
   Rule R (a when p && q)

Repeat step 3 while applicable.

A property of rule-based systems

- A *derived* rule does not add new behaviors



Rule composition

seq(Rule R1 a1 when p1, Rule R2 when p2) =
Rule seq_R1_R2 (if p1 then a1);(if p2 then a2)

par(Rule R1 a1 when p1, Rule R2 when p2) =
Rule par_R1_R2 (if p1 then a1)|(if p2 then a2)

pri(Rule R1 a1 when p1, Rule R2 when p2) =
Rule pri_R1_R2 (a2 when ¬p1 ^ p2)

Scheduling Grammar

SC ::= R
| seq(SC, SC)
| par(SC, SC)
| pri(SC, SC)