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.
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)

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
BS: A Language of Atomic Actions

A program is a collection of instantiated modules \( m_1 ; m_2 ; \ldots \)

Module ::= Module name

- [State variable \( r \)]
- [Rule \( R \) \( a \)]
- [Action method \( g(x) = a \)]
- [Read method \( f(x) = e \)]

\[
\begin{align*}
\text{a ::= r := e} & & \text{e ::= r | c | t} \\
| \text{if e then a} & & \text{Op(e, e)} \\
| \text{a | a} & & \text{e ? e : e} \\
| \text{a ; a} & & \text{(t = e in e)} \\
| \text{(t = e in a)} & & \text{m.f(e)} \\
| \text{m.g(e)} & & \text{e when e} \\
| \text{a when e} & & \text{Guarded action}
\end{align*}
\]
Execution model

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

Guards vs If’s

- Guards affect the surroundings
  \[(a1 \text{ when } p1) \mid a2 \implies (a1 \mid a2) \text{ when } p1\]

- Effect of an "if" is local
  \[(\text{if } p1 \text{ then } a1) \mid a2 \implies \text{if } p1 \text{ then } (a1 \mid a2) \text{ else } a2\]

\[p1 \text{ has no effect on } a2\]
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)

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 \( \rho \) 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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Pre-condition</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg-update</td>
<td>( \rho \models e \Rightarrow v )</td>
<td>( \rho \models (r := e) \Rightarrow {(r, v)} )</td>
</tr>
<tr>
<td>if-true</td>
<td>( \rho \models e \Rightarrow \text{true} ), ( \rho \models a \Rightarrow U )</td>
<td>( \rho \models (\text{if } e \text{ then } a) \Rightarrow U )</td>
</tr>
<tr>
<td>if-false</td>
<td>( \rho \models e \Rightarrow \text{false} )</td>
<td>( \rho \models (\text{if } e \text{ then } a) \Rightarrow \emptyset )</td>
</tr>
<tr>
<td>par</td>
<td>( \rho \models a_1 \Rightarrow U_1 ), ( \rho \models a_2 \Rightarrow U_2 )</td>
<td>( \rho \models (a_1 \mid a_2) \Rightarrow \text{pmerge}(U_1, U_2) )</td>
</tr>
<tr>
<td>seq</td>
<td>( \rho \models a_1 \Rightarrow U_1 ), update(( \rho ), U_1) \models a_2 \Rightarrow U_2</td>
<td>( \rho \models (a_1 ; a_2) \Rightarrow \text{smerge}(U_1, U_2) )</td>
</tr>
</tbody>
</table>

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

Like \text{pmerge} but U2 dominates U1
BS Action Rules \textit{cont}

\begin{align*}
\text{a-let-sub} & : \quad \rho \vdash e \Rightarrow v, \; \text{smerge}(\rho, \{(t,v)\}) \vdash a \Rightarrow U \quad (v \neq \bot) \\
& \quad \rho \vdash (t = e) \in a \Rightarrow U
\end{align*}

\begin{align*}
\text{a-meth-call} & : \quad \rho \vdash e \Rightarrow v, \quad \lambda x. a = \text{lookup}(m.g), \\
& \quad \text{smerge}(\rho, \{(x,v)\}) \vdash a \Rightarrow U \\
& \quad \rho \vdash m.g(e) \Rightarrow U
\end{align*}

Expression rules are similar

Guard Rules

\begin{align*}
\text{a-when-ready} & : \quad \rho \vdash e \Rightarrow \text{true}, \quad \rho \vdash a \Rightarrow U \\
& \quad \rho \vdash (a \text{ when } e) \Rightarrow U
\end{align*}

\begin{itemize}
\item If no rule applies then the system is stuck and the effect of the whole atomic action is “no action” (returns \(\bot\)).
\item For example, if \(e\) evaluates to false then not just \((a \text{ when } e)\) results in no updates but the whole atomic action of which \((a \text{ when } e)\) is a part results in no updates.
\end{itemize}
**Alternative let definition**

\[
\text{a-let-sub} \quad \rho \vdash e \Rightarrow v, \ smerge(\rho, (t,v)) \vdash a \Rightarrow U \\
\rho \vdash ((t = e); a) \Rightarrow U
\]

\[
\text{a-let-sub-bot} \quad \rho \vdash e \Rightarrow \bot, \ smerge(\rho, (t, \bot)) \vdash a \Rightarrow U \\
\rho \vdash ((t = e); a) \Rightarrow U
\]

*Let's 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
  
  \[(a1 \text{ when } p1) \mid a2 \implies (a1 \mid a2) \text{ when } p1\]

- A condition of a Conditional action only affects the actions within the scope of the conditional action
  
  \[(\text{if } p1 \text{ then } a1) \mid a2 \implies \text{if } p1 \text{ then } (a1 \mid a2) \text{ else } a2\]
  
  \[p1 \text{ has no effect on } a2 \ldots\]

- Mixing ifs and whens
  
  \[\text{if } p \text{ then } (a1 \text{ when } q1) \text{ else } (a2 \text{ when } q2)\]
  
  \[\equiv (\text{if } p \text{ then } a1 \text{ else } a2) \text{ when } (p \& q1 \mid \sim p \& q2)\]
Conditionals & Cases

\[
\text{if } p \text{ then } a1 \text{ else } a2 \\
\equiv \text{if } p \text{ then } a1 \mid \text{if } !p \text{ then } a2
\]

Similarly for cases

From specification to implementation

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

Highly non-deterministic

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. \((a1 \text{ when } p) | a2 = (a1 | a2) \text{ when } p\)
A2. \(a1 | (a2 \text{ when } p) = (a1 | a2) \text{ when } p\)
A3. \((a1 \text{ when } p); a2 = (a1 ; a2) \text{ when } p\)
A4. \(a1 ; (a2 \text{ when } p) = (a1 ; a2) \text{ when } p'\)
   where \(p'\) is \(p\) after the effect of \(a1\)
A5. \(\text{if } (p \text{ when } q) \text{ then } a = (\text{if } p \text{ then } a) \text{ when } q\)
A6. \(\text{if } p \text{ then } (a \text{ when } q) = (\text{if } p \text{ then } a) \text{ when } (q || !p)\)
A7. \((a \text{ when } p1) \text{ when } p2 = a \text{ when } (p1 && p2)\)
A8. \(r := (e \text{ when } p) = (r := e) \text{ when } p\)
A9. \(m.g(e \text{ when } p) = m.g(e) \text{ when } p\)
   similarly for expressions ...
Lifting “whens” to the top

- Closely related to the “when” axioms, e.g.,

\[
LW(a_1 \mid a_2) = (a_1' \mid a_2') \text{ when } (a_1_G \&\& a_2_G)
\]

where

\[
(a_1' \text{ when } a_1_G) = LW(a_1)
\]
\[
(a_2' \text{ when } a_2_G) = LW(a_2)
\]

- 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 \( (m.h_B(e') \text{ when } e_G \&\& m.h_G(e')) \)

where \( (e' \text{ when } e_G) = LW(e) \)
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
   \[ g_B = \lambda x. a \quad \text{and} \quad g_G = \lambda x. p \]
   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

\[ \text{A derived rule does not add new behaviors} \]

\[ S_1 \xrightarrow{\text{rule } R_a} S_2 \xrightarrow{\text{rule } R_b} S_3 \]

\[ S_1 \xrightarrow{r_{a,b}} S_3 \]
Rule composition

\[
\begin{align*}
\text{seq}(& \text{Rule R1 a1 when p1, Rule R2 when p2}) = \\
& \text{Rule seq}_R1_R2 (\text{if p1 then a1};(\text{if p2 then a2}) \\
\text{par}(& \text{Rule R1 a1 when p1, Rule R2 when p2}) = \\
& \text{Rule par}_R1_R2 (\text{if p1 then a1})|(\text{if p2 then a2}) \\
\text{pri}(& \text{Rule R1 a1 when p1, Rule R2 when p2}) = \\
& \text{Rule pri}_R1_R2 (a2 \text{ when } \neg p1 \land p2)
\end{align*}
\]

Scheduling Grammar

\[
\text{SC ::= R} \\
& | \text{seq(SC, SC)} \\
& | \text{par(SC, SC)} \\
& | \text{pri(SC, SC)}
\]