Scheduling Primitives for Bluespec

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Work in progress

LPM in Bluespec

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)

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

Two-element FIFO does not solve the problem.

enq and deq cannot be enabled together!

Rule “recirculate” will never fire!

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
Extending the Bluespec language with the "sequential connective"

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 \( f(x) = e \)]

\[ e ::= r | c | t | Op(e, e) | e ? e : e | (t = e in e) | m.f(e) | e \text{ when } e \]
\[ a ::= r := e | \text{Conditional} \]
\[ a | a \quad \text{Composition} \]
\[ a ; a \quad \text{Sequential Composition} \]
\[ (t = e \text{ in } a) \]
\[ m.g(e) \quad \text{Method call} \]
\[ a \text{ when } e \quad \text{Guarded action} \]
Guards vs If’s

- Guards affect the surroundings
  \[(a_1 \text{ when } p_1) \mid a_2 \implies (a_1 \mid a_2) \text{ when } p_1\]

- Effect of an “if” is local
  \[(\text{if } p_1 \text{ then } a_1) \mid a_2 \implies \text{if } p_1 \text{ then } (a_1 \mid a_2) \text{ else } a_2\]

  \[p_1 \text{ has no effect on } a_2\]

Conditionals & Cases

\[\text{if } p \text{ then } a_1 \text{ else } a_2\]
\[\equiv \text{if } p \text{ then } a_1 \mid \text{if } \neg p \text{ then } a_2\]

Similarly for cases
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>Specification</th>
</tr>
</thead>
</table>
| reg-update | $\rho \vdash e \Rightarrow v$
  \[\rho \vdash (r := e) \Rightarrow \{r, v\}\] |
| if-true | $\rho \vdash e \Rightarrow \text{true}, \quad \rho \vdash a \Rightarrow U$
  \[\rho \vdash (\text{if } e \text{ then } a) \Rightarrow U\] |
| if-false | $\rho \vdash e \Rightarrow \text{false}$
  \[\rho \vdash (\text{if } e \text{ then } a) \Rightarrow \emptyset\] |
| par | $\rho \vdash a1 \Rightarrow U1, \quad \rho \vdash a2 \Rightarrow U2$
  \[\rho \vdash (a1 \mid a2) \Rightarrow \text{pmerge(U1,U2)}\] |
| seq | $\rho \vdash a1 \Rightarrow U1, \quad \text{update}(\rho, U1) \vdash a2 \Rightarrow U2$
  \[\rho \vdash (a1 ; a2) \Rightarrow \text{smerge(U1,U2)}\] |
BS Action Rules cont

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

Expression rules are similar

Guard Semantics

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

\[\text{If no rule applies then the system is stuck and the effect of the whole atomic action is “no action” (returns } \bot).\]

- For example, if \(e\) evaluates to false then not just \((a \text{ when } e)\) results in no updates but the whole atomic action of which \((a \text{ when } e)\) is a part results in no updates.
LPM in Bluespec

*using the sequential connective*

```plaintext
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)
```

Execution model

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

Highly non-deterministic
Current Implementation

sans guards

- All guards are evaluated in parallel and the compiler selects only among those rules whose guards are true.
  - Requires lifting Guards to the top
- Among the enabled rules the compiler schedules for concurrent execution all the rules that do not “conflict” with each other
- Among the enabled and conflicting rules the compiler selects a rule for execution based on some user specified static priority

Nondeterminism & Scheduling

- A Bluespec description admits nondeterminism but a compilation (with a scheduler) results in a deterministic implementation.
- Experience has shown that the user wants more control in the selection for both functionality and efficiency...
LPM: Scheduling issues

Bluespec scheduling is not good enough for this level of specificity.

Can a rule calling method “enter” execute concurrently with the “recirculate” rule? If not, who should have priority?

```
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)
```

LW: Lifting “whens” to the top

```
sans let blocks
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 ...
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

- A derived rule does not add new behaviors

\[ \text{rule } R_a \quad \text{S1} \quad \text{rule } R_b \quad \text{S2} \quad \text{rule } R_{a,b} \quad \text{S3} \]
Concurrent Scheduling & Rule composition

Rule R1 a1 when p1
Rule R2 a2 when p2

Suppose we want to schedule R1 and R2 concurrently.
The user writes:
S = par(R1,R2)
where the meaning of rule S is
Rule S ((if p1 then a1)|(if p2 then a2)) when p1 v p2

Parallel Scheduling can be expressed as a rule composition

This composition is NOT always valid because it may introduce new behaviors!
Is there a loss of some behaviors if S replaces R1 and R2?

Value forwarding & Sequential composition of rules

Rule R1 a1 when p1
Rule R2 a2 when p2

Suppose we want to schedule R1 and R2 concurrently and they conflict. Furthermore assume we want the effect to be R1 followed by R2
The user writes:
S = seq(R1,R2)
where the meaning of rule S is
Rule S ((if p1 then a1);(if p2 then a2)) when p1 v p2’
where p2’ is p2 after the effect of (if p1 then a1)

This composition is always valid!
S allows forwarding of values from a1 to a2
Rule Priorities

Rule R1 a1 when p1
Rule R2 a2 when p2

Suppose rules R1 and R2 conflict and we want to give priority to R1

The user writes:
R2’ = pri(R1,R2)
where the meaning of rule R2’ is
Rule R2’ (a2 when ¬p1 ^ p2)

R1 and R2’ are mutually exclusive and can be composed using par

Schedules

Grammar
S ::= R
| seq(S, S)
| par(S, S)
| pri(S, S)

1. The user must specify a schedule
2. A rule can occur multiple times in a schedule

Notice
par is commutative, i.e. par(R1,R2) = par(R2,R1)
Both par and seq are associative, i.e.,
par(R1,R2,R3) = par(R1,par(R2,R3)) = par(par(R1,R2),R3)
seq(R1,R2,R3) = seq(R1,seq(R2,R3)) = seq(seq(R1,R2),R3)
Sample Schedules

- `par(R1, pri(R1,R2))`
- `par(R1, seq(R2,R2))`
- `par(R1, pri(R1,seq(R2,R2)))`

Rule splitting

```plaintext
rule "foo"
  (if p(x) then a1 else a2)
```
can be split into the following two rules:

```plaintext
rule "foo-then"
  (if p(x) then a1)
```

```plaintext
rule "foo-else"
  (if !p(x) then a2)
```

One can provide greater control in scheduling by rule splitting.
LPM Revisited

Goal – A schedule which introduces no dead cycle while exiting

```plaintext
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)
```

LPM: Split the Internal rule

```plaintext
module lpm
rule "recirculate"
  (x = mem.res() in (y = fifo.first() in
   (if !done?(x) then (mem.deq(); mem.req(addr(x)))
    | (fifo.deq(); fifo.enq(y)))

rule "exit"
  (x = mem.res() in (y = fifo.first() in
   (if done?(x) then fifo.deq() | mem.deq() | outQ.enq(x)

action method enter(x) = mem.req(addr(x)) | fifo.enq(x)
```
LPM: Turn rules into methods

```
module lpm

action method recirculate () =
    (x = mem.res() in y = fifo.first() in
    (if !done?(x) then (mem.deq(); mem.req(addr(x)))
    | (fifo.deq(); fifo.enq(y)))

action method exit () =
    (x = mem.res() in y = fifo.first() in
    (if done?(x) then fifo.deq() | mem.deq() | outQ.enq(x)

action method enter(x) = mem.req(addr(x)) | fifo.enq(x)
```

Outside the module

Outside

rule "recirculate" lpm.recirculate()
rule "exit" lpm.exit()
rule "foo" ... lpm.enter(a)

LPM Schedule

- **Schedule-1**
  - \( r' = pri(enter, recirc) \)
  - \( S1 = par(r', exit, enter) \)

  **S1 is invalid**

- **Schedule-2**
  - \( r' = pri(enter, recirc) \)
  - \( ee = seq(exit, enter) \)
  - \( S2 = par(r', ee) \)

  enter has priority over recirculate; no dead cycles

- **Schedule-3**
  - \( e' = pri(recirc, enter) \)
  - \( ee = seq(exit, e') \)
  - \( S3 = par(recirc, ee) \)

  recirculate has priority over enter; no dead cycles

Assume enq/deq conflict

Difficult to pick between S2 and S3!