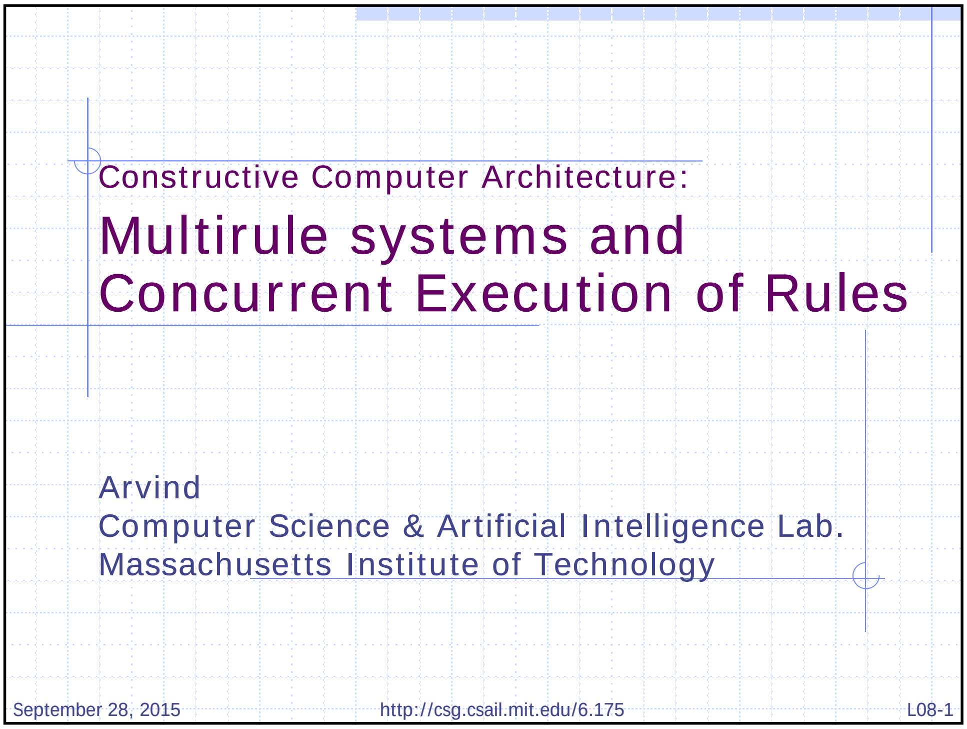
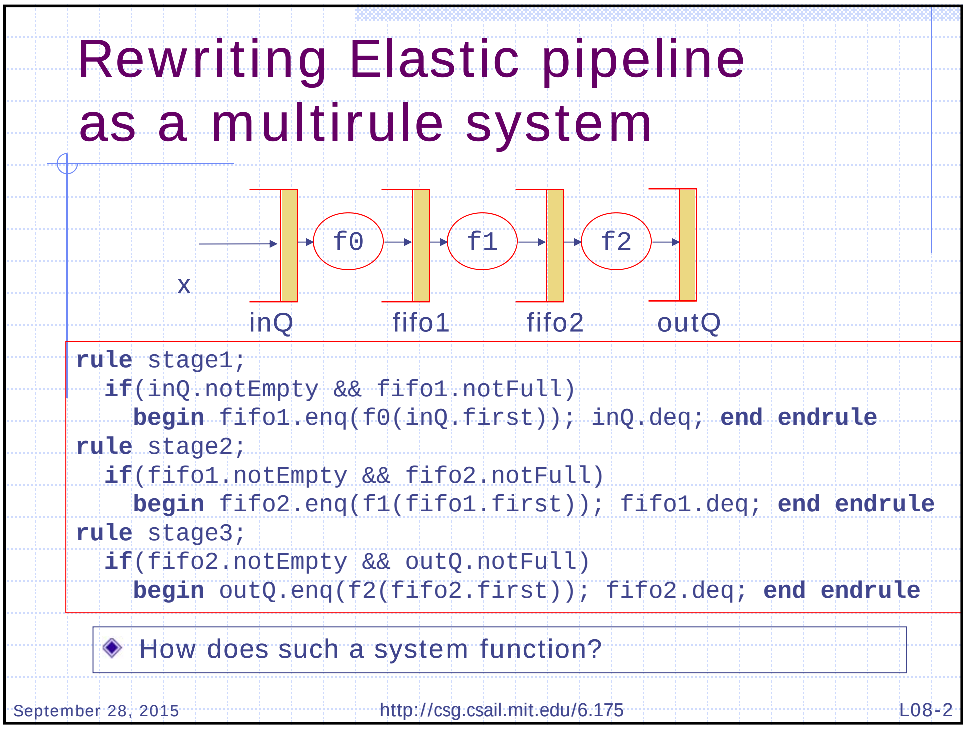


Constructive Computer Architecture:

Multirule systems and Concurrent Execution of Rules

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

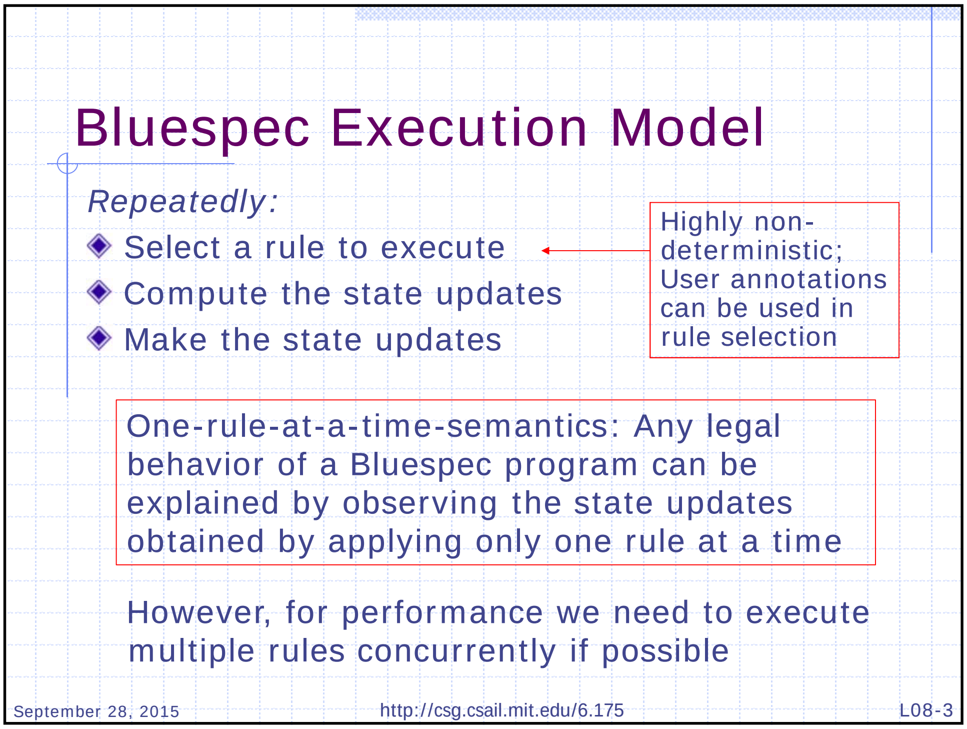


Rewriting Elastic pipeline as a multirule system

x
inQ
f0
fifo1
f1
fifo2
f2
outQ

```
rule stage1;
  if(inQ.notEmpty && fifo1.notFull)
    begin fifo1.enq(f0(inQ.first)); inQ.deq; end endrule
rule stage2;
  if(fifo1.notEmpty && fifo2.notFull)
    begin fifo2.enq(f1(fifo1.first)); fifo1.deq; end endrule
rule stage3;
  if(fifo2.notEmpty && outQ.notFull)
    begin outQ.enq(f2(fifo2.first)); fifo2.deq; end endrule
```

- How does such a system function?



Bluespec Execution Model

Repeatedly:

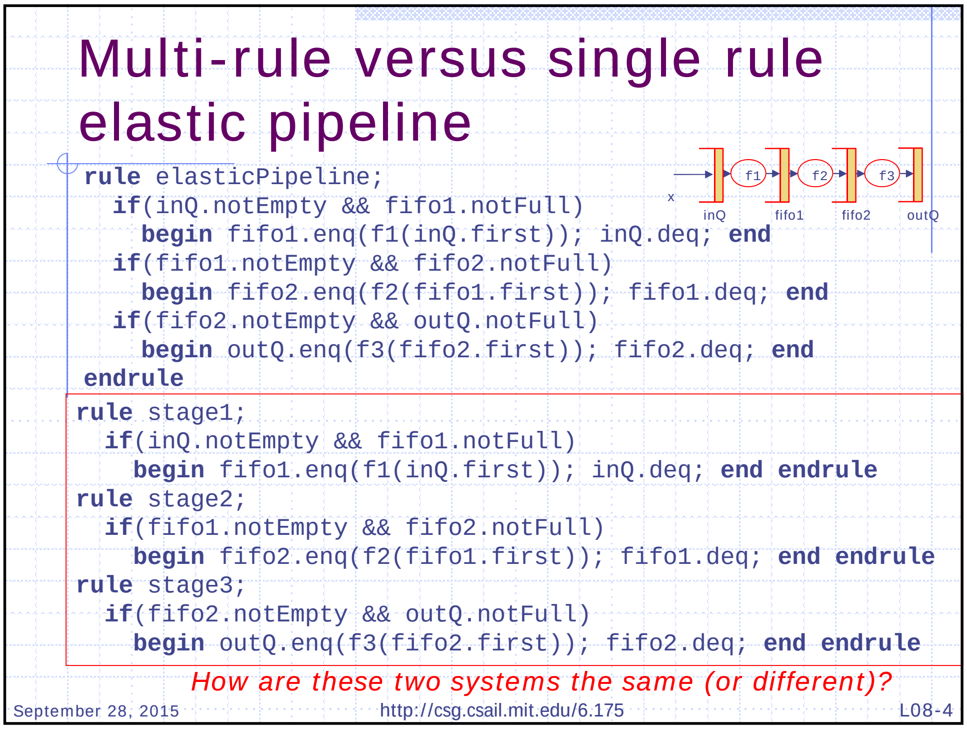
- Select a rule to execute
- Compute the state updates
- Make the state updates

Highly non-deterministic; User annotations can be used in rule selection

One-rule-at-a-time-semantics: Any legal behavior of a Bluespec program can be explained by observing the state updates obtained by applying only one rule at a time

However, for performance we need to execute multiple rules concurrently if possible

September 28, 2015 http://csg.csail.mit.edu/6.175 L08-3


x
f1
f2
f3
inQ
fifo1
fifo2
outQ

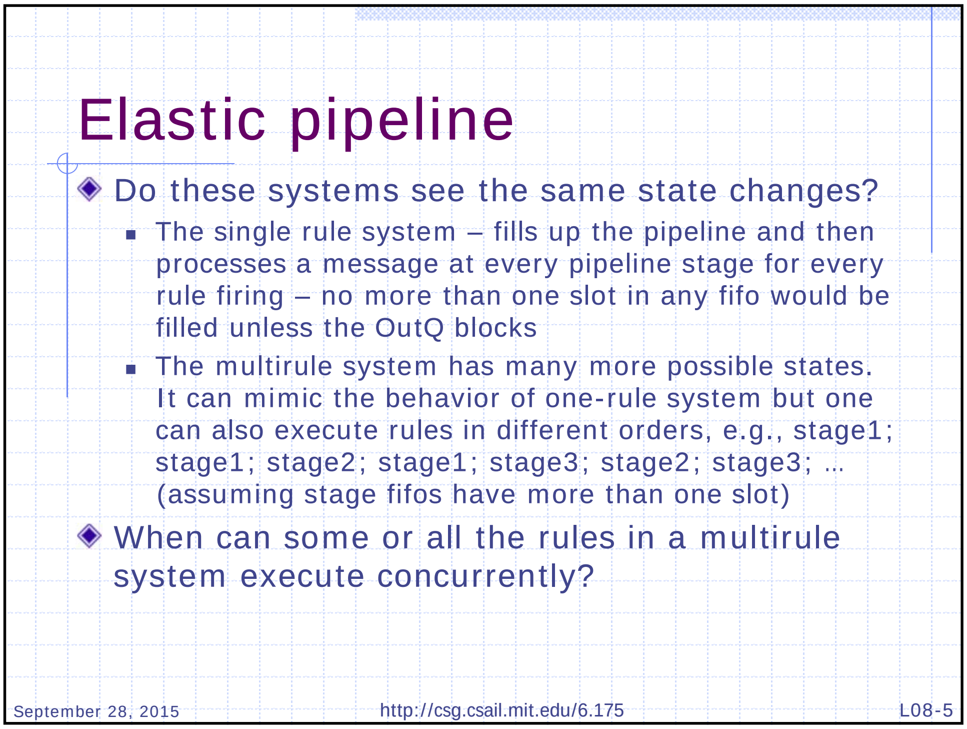
Multi-rule versus single rule elastic pipeline

```
rule elasticPipeline;
  if(inQ.notEmpty && fifo1.notFull)
    begin fifo1.enq(f1(inQ.first)); inQ.deq; end
  if(fifo1.notEmpty && fifo2.notFull)
    begin fifo2.enq(f2(fifo1.first)); fifo1.deq; end
  if(fifo2.notEmpty && outQ.notFull)
    begin outQ.enq(f3(fifo2.first)); fifo2.deq; end
endrule
```

```
rule stage1;
  if(inQ.notEmpty && fifo1.notFull)
    begin fifo1.enq(f1(inQ.first)); inQ.deq; end endrule
rule stage2;
  if(fifo1.notEmpty && fifo2.notFull)
    begin fifo2.enq(f2(fifo1.first)); fifo1.deq; end endrule
rule stage3;
  if(fifo2.notEmpty && outQ.notFull)
    begin outQ.enq(f3(fifo2.first)); fifo2.deq; end endrule
```

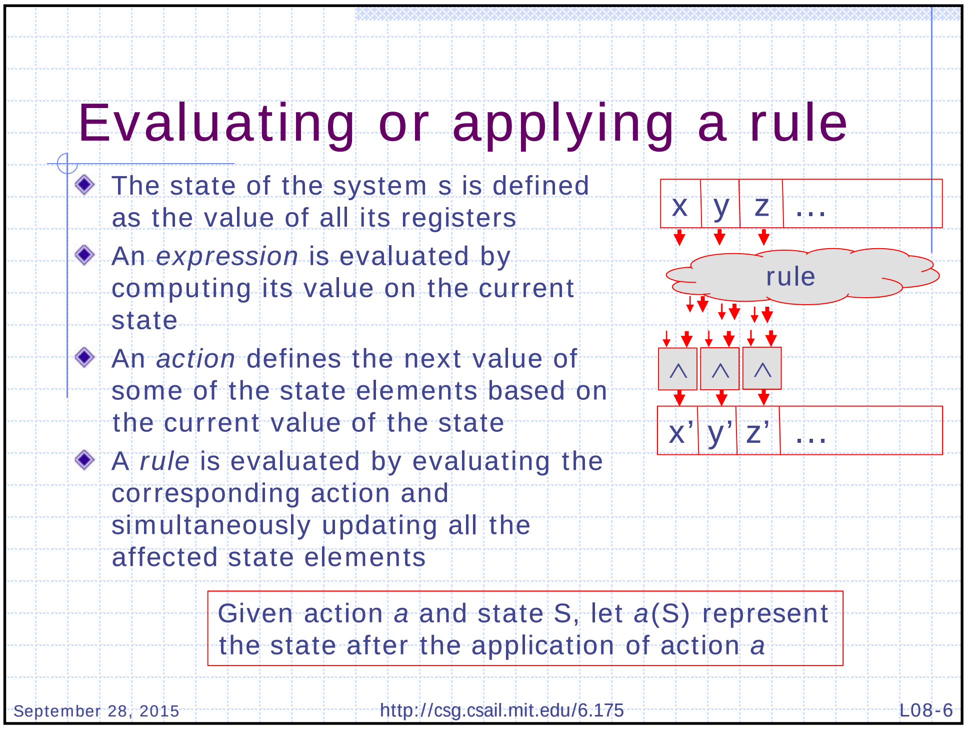
How are these two systems the same (or different)?

September 28, 2015 http://csg.csail.mit.edu/6.175 L08-4



Elastic pipeline

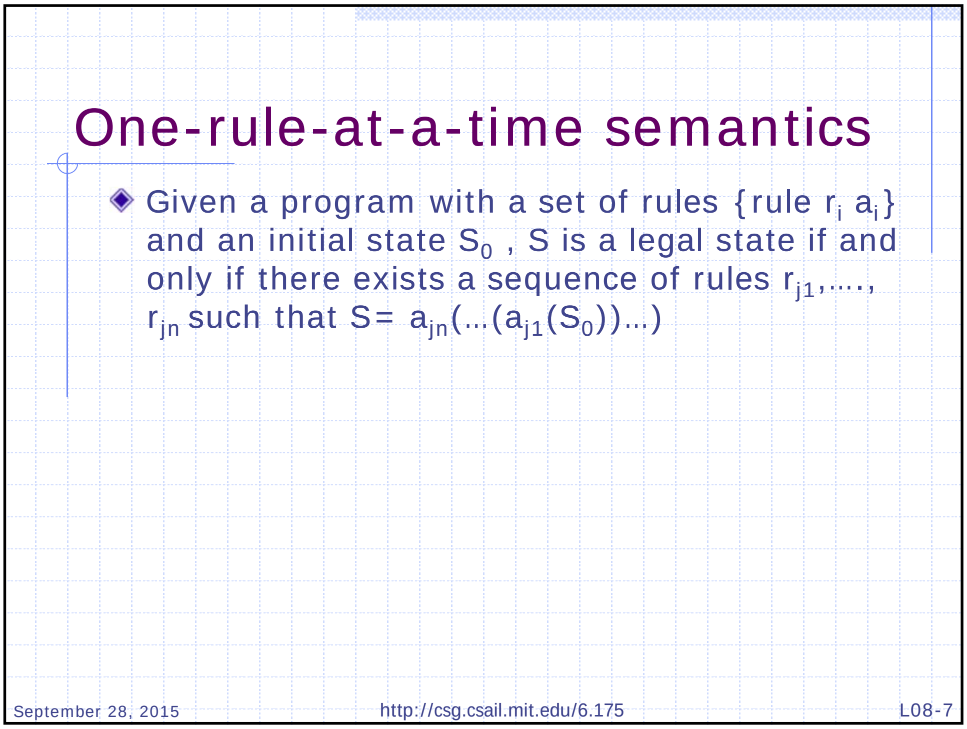
- Do these systems see the same state changes?
  - The single rule system – fills up the pipeline and then processes a message at every pipeline stage for every rule firing – no more than one slot in any fifo would be filled unless the OutQ blocks
  - The multirule system has many more possible states. It can mimic the behavior of one-rule system but one can also execute rules in different orders, e.g., stage1; stage1; stage2; stage1; stage3; stage2; stage3; … (assuming stage fifos have more than one slot)
- When can some or all the rules in a multirule system execute concurrently?



Evaluating or applying a rule

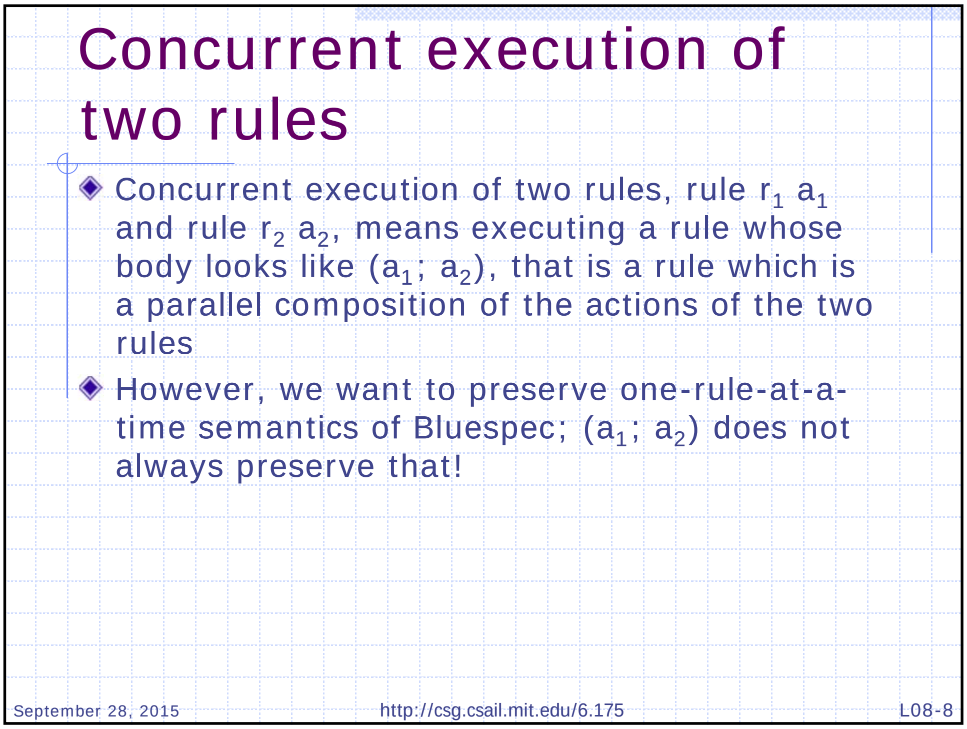
- The state of the system s is defined as the value of all its registers
- An *expression* is evaluated by computing its value on the current state
- An *action* defines the next value of some of the state elements based on the current value of the state
- A *rule* is evaluated by evaluating the corresponding action and simultaneously updating all the affected state elements

Given action *a* and state S, let *a*(S) represent the state after the application of action *a*



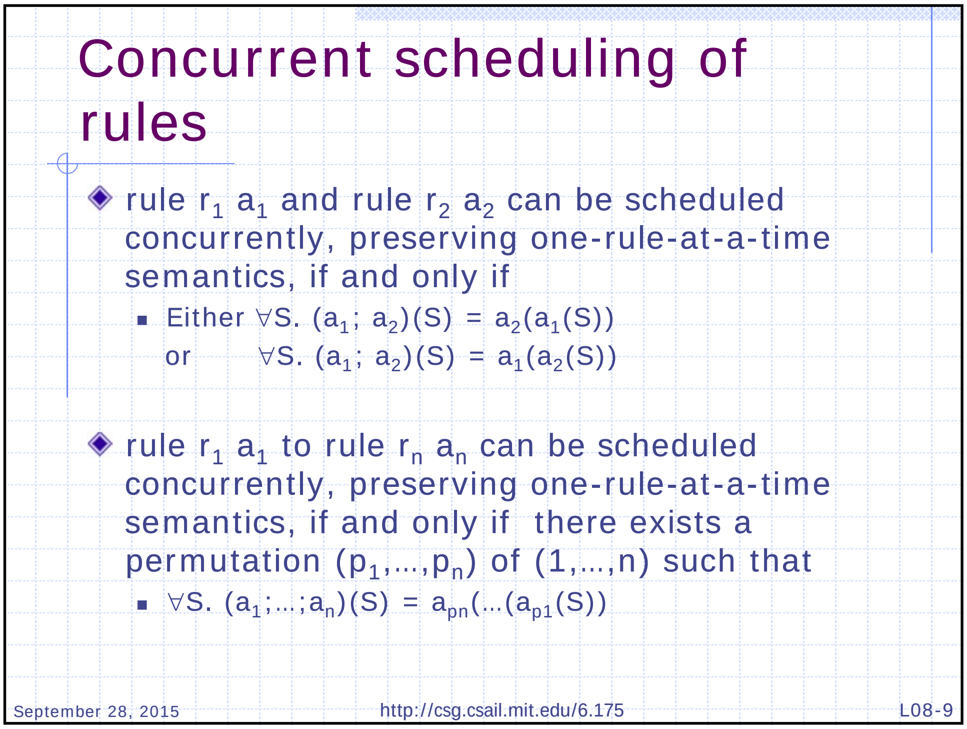
One-rule-at-a-time semantics

- Given a program with a set of rules {rule r_i a_i} and an initial state S_0 , S is a legal state if and only if there exists a sequence of rules r_{j1},…., r_{jn} such that S= a_{jn}(…(a_{j1}(S_0))…)



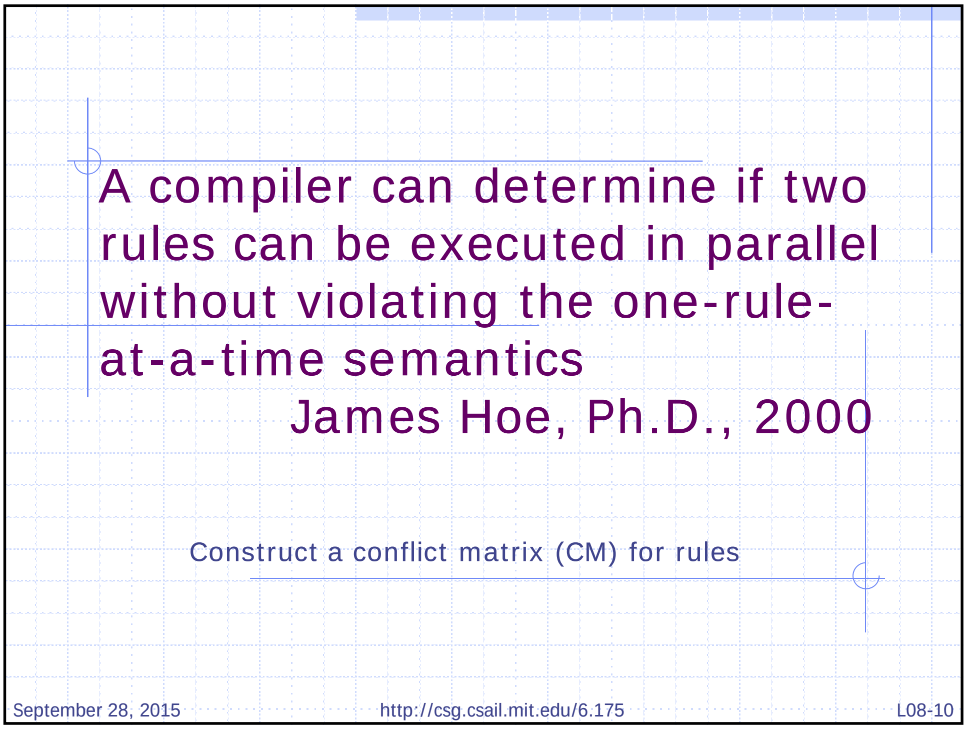
Concurrent execution of two rules

- Concurrent execution of two rules, rule r_1 a_1 and rule r_2 a_2, means executing a rule whose body looks like (a_1; a_2), that is a rule which is a parallel composition of the actions of the two rules
- However, we want to preserve one-rule-at-a-time semantics of Bluespec; (a_1; a_2) does not always preserve that!



Concurrent scheduling of rules

- rule r_1 a_1 and rule r_2 a_2 can be scheduled concurrently, preserving one-rule-at-a-time semantics, if and only if
  - Either $\forall S.\ (a_1;\ a_2)(S) = a_2(a_1(S))$
    or $\forall S.\ (a_1;\ a_2)(S) = a_1(a_2(S))$
- rule r_1 a_1 to rule r_n a_n can be scheduled concurrently, preserving one-rule-at-a-time semantics, if and only if there exists a permutation $(p_1,\ldots,p_n)$ of $(1,\ldots,n)$ such that
  - $\forall S.\ (a_1;\ldots;a_n)(S) = a_{pn}(\ldots(a_{p1}(S))$



A compiler can determine if two rules can be executed in parallel without violating the one-rule-at-a-time semantics

James Hoe, Ph.D., 2000

Construct a conflict matrix (CM) for rules

Extending CM to rules

- CM between two rules is computed exactly the same way as CM for the methods of a module
  - Given rule r1 a1 and rule r2 a2 such that

    mcalls(a1)={g11,g12...g1n}

    mcalls(a2)={g21,g22...g2m}
  - Compute
    - Conflict(x,y) = if x and y are methods of the same module then CM[x,y] else CF
    - CM[r1,r2] = conflict(g11,g21) $\cap$ conflict(g11,g22) $\cap$...

      $\cap$ conflict(g12,g21) $\cap$ conflict(g12,g22) $\cap$...

      ...

      $\cap$ conflict(g1n,g21) $\cap$ conflict(g12,g22) $\cap$...
  - Conflict relation is not transitive
    - r1 < r2, r2 < r3 does not imply r1 < r3

Using CMs for concurrent scheduling of rules

Two rules that are conflict free can be scheduled together without violating the one-rule-at-a-time semantics. In general, use the following theorem

Theorem: Given a set of rules {rule r_i a_i}, if there exists a permutation {p_1, p_2 ... p_n} of {1..n} such that

$\forall$ i < j. CM(a_{pi}, a_{pj}) is CF or <

then $\forall$ S. (a_1;…;a_n)(S) = a_{pn}(…(a_{p1}(S)).

Thus, rules r_1, r_2 ... r_n can be scheduled concurrently with the effect $\forall$ i, j. r_{pi} < r_{pj}

Example 1: Compiler Analysis

```
rule ra;
  if (z>10)
    x <= x+1;
endrule

rule rb;
  if (z>20)
   y <= y+2;
endrule
```

mcalls(ra) = {z.r, x.w, x.r}
mcalls(rb) = {z.r, y.w, y.r}

CM(ra, rb) =
conflict(z.r, z.r) ∩ conflict(z.r, y.w)
∩ conflict(z.r, y.r) ∩ conflict(x.w, z.r)
∩ conflict(x.w, y.w) ∩ conflict(x.w, y.r)
∩ conflict(x.r, z.r) ∩ conflict(x.r, y.w)
∩ Conflict(x.r, y.r)
= CF ∩ CF ∩ CF ∩ CF ... = CF

Rules ra and rb can be scheduled together without violating the one-rule-at-a-time-semantics. We say rules ra and rb are CF

Example 2: Compiler Analysis

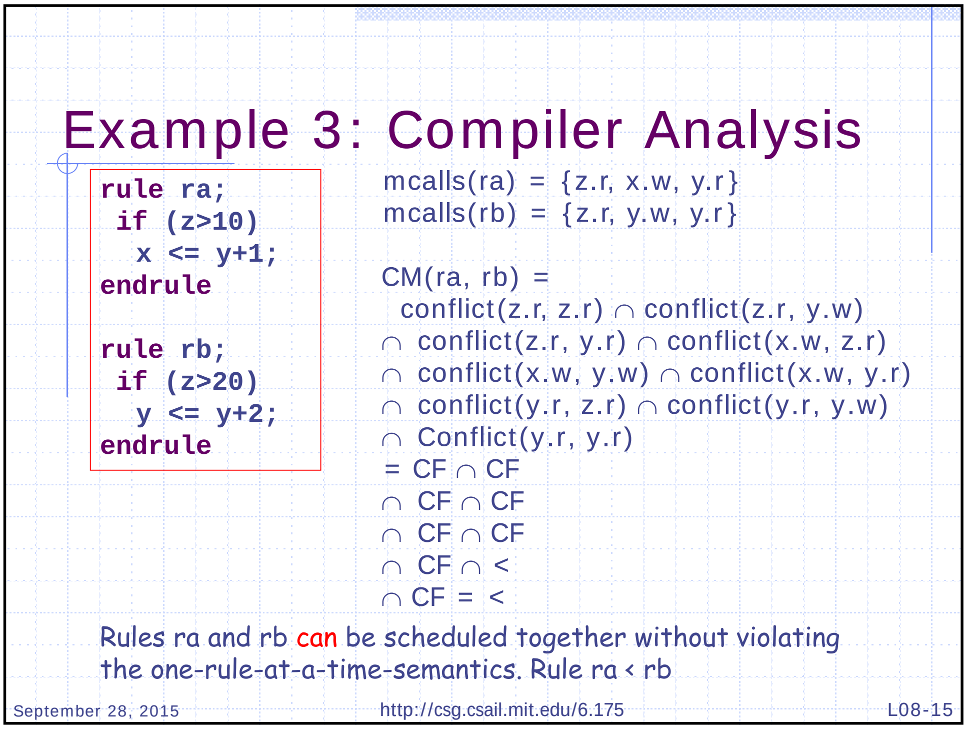
```
rule ra;
  if (z>10)
   x <= y+1;
endrule

rule rb;
  if (z>20)
   y <= x+2;
endrule
```

mcalls(ra) = {z.r, x.w, y.r}
mcalls(rb) = {z.r, y.w, x.r}

CM(ra, rb) =
conflict(z.r, z.r) ∩ conflict(z.r, y.w)
∩ conflict(z.r, x.r) ∩ conflict(x.w, z.r)
∩ conflict(x.w, y.w) ∩ conflict(x.w, x.r)
∩ conflict(y.r, z.r) ∩ conflict(y.r, y.w)
∩ Conflict(y.r, x.r)
= CF ∩ CF
∩ CF ∩ CF
∩ CF ∩ >
∩ CF ∩ <
∩ CF = C

Rules ra and rb cannot be scheduled together without violating the one-rule-at-a-time-semantics. Rules ra and rb are C



Example 3: Compiler Analysis

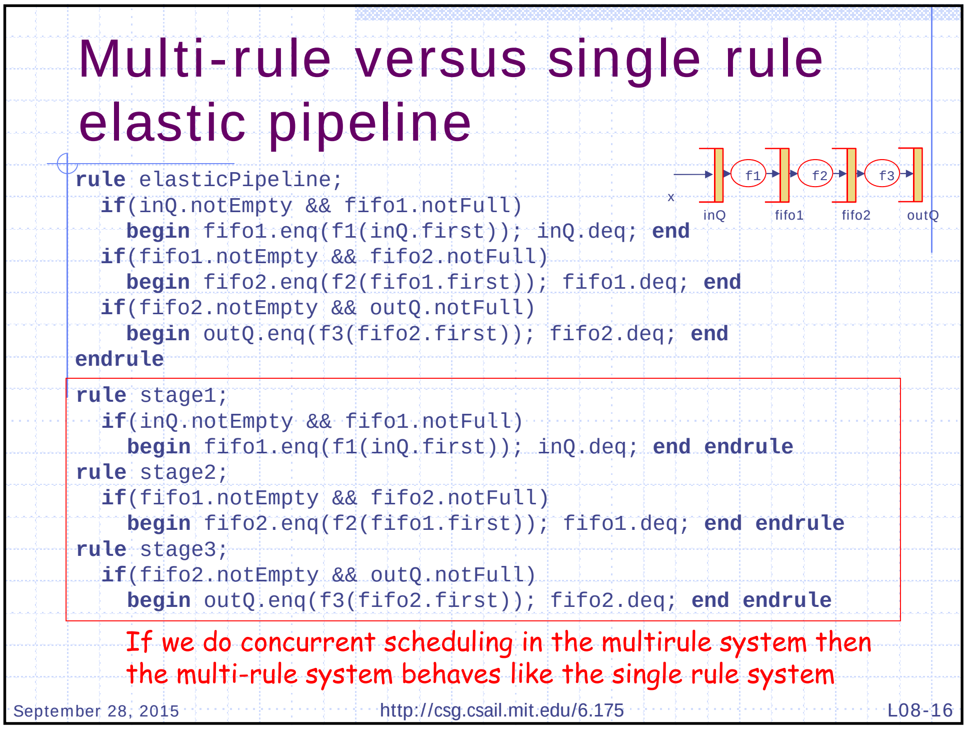
```
rule ra;
 if (z>10)
   x <= y+1;
endrule

rule rb;
 if (z>20)
   y <= y+2;
endrule
```

mcalls(ra) = {z.r, x.w, y.r}
mcalls(rb) = {z.r, y.w, y.r}

CM(ra, rb) =
 conflict(z.r, z.r) ∩ conflict(z.r, y.w)
∩ conflict(z.r, y.r) ∩ conflict(x.w, z.r)
∩ conflict(x.w, y.w) ∩ conflict(x.w, y.r)
∩ conflict(y.r, z.r) ∩ conflict(y.r, y.w)
∩ Conflict(y.r, y.r)
= CF ∩ CF
∩ CF ∩ CF
∩ CF ∩ CF
∩ CF ∩ <
∩ CF = <

Rules ra and rb can be scheduled together without violating the one-rule-at-a-time-semantics. Rule ra < rb



Multi-rule versus single rule elastic pipeline

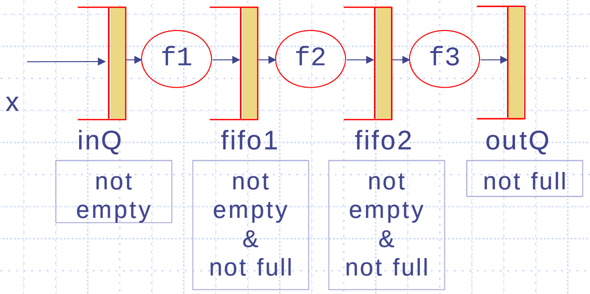
x
f1
f2
f3
inQ
fifo1
fifo2
outQ

```
rule elasticPipeline;
  if(inQ.notEmpty && fifo1.notFull)
    begin fifo1.enq(f1(inQ.first)); inQ.deq; end
  if(fifo1.notEmpty && fifo2.notFull)
    begin fifo2.enq(f2(fifo1.first)); fifo1.deq; end
  if(fifo2.notEmpty && outQ.notFull)
    begin outQ.enq(f3(fifo2.first)); fifo2.deq; end
endrule
```

```
rule stage1;
  if(inQ.notEmpty && fifo1.notFull)
    begin fifo1.enq(f1(inQ.first)); inQ.deq; end endrule
rule stage2;
  if(fifo1.notEmpty && fifo2.notFull)
    begin fifo2.enq(f2(fifo1.first)); fifo1.deq; end endrule
rule stage3;
  if(fifo2.notEmpty && outQ.notFull)
    begin outQ.enq(f3(fifo2.first)); fifo2.deq; end endrule
```

If we do concurrent scheduling in the multirule system then the multi-rule system behaves like the single rule system

Concurrency when the FIFOs do not permit concurrent enq and deq


x

f1

f2

f3

inQ

fifo1

fifo2

outQ

not empty

not empty & not full

not empty & not full

not full

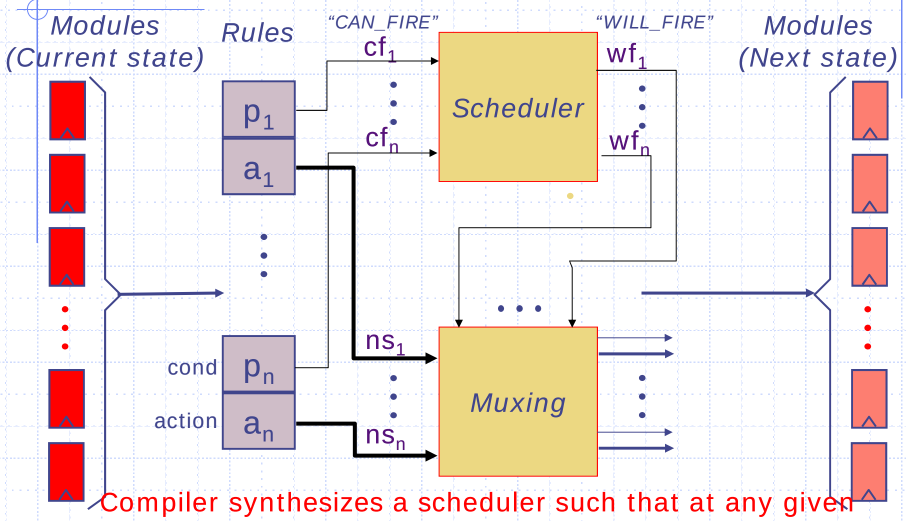
At best alternate stages in the pipeline will be able to fire concurrently

Practical scheduling concerns

- Rules often have a top level predicate or *guard*:
  - rule r1 if (p1); a1
- It does make sense to schedule such a rule for execution unless it’s predicate is true
- We can evaluate the guards of many* rules in parallel every (clock) cycle and then select for parallel execution only among those rules whose guards are true. Of course the selected rules must preserve one-rule-at-a-time semantics.

*Not all guards can be evaluated in parallel because of EHRs and method parameters

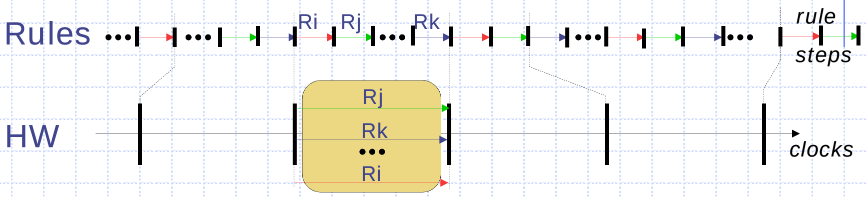
Scheduling and control logic



Compiler synthesizes a scheduler such that at any given time will-fire for only non-conflicting rules are true

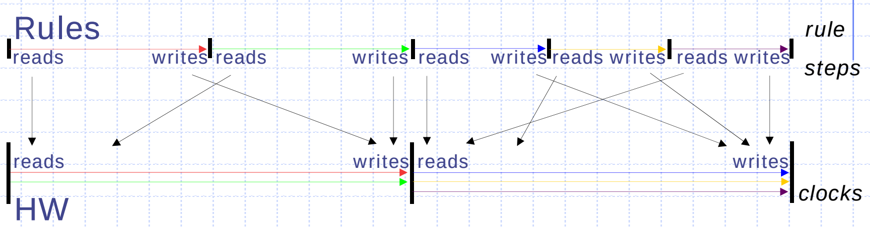
some insight into

Concurrent rule execution



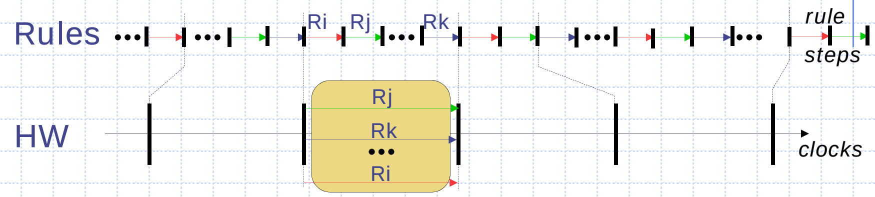
- There are more intermediate states in the rule semantics (a state after each rule step)
- In the HW, states change only at clock edges

Parallel execution reorders reads and writes



- In the rule semantics, each rule sees (reads) the effects (writes) of previous rules
- In the HW, rules only see the effects from previous clocks, and only affect subsequent clocks

Correctness



- The compiler will schedule rules concurrently only if the net state change is equivalent to sequential rule execution (which is what our theorem ensures)