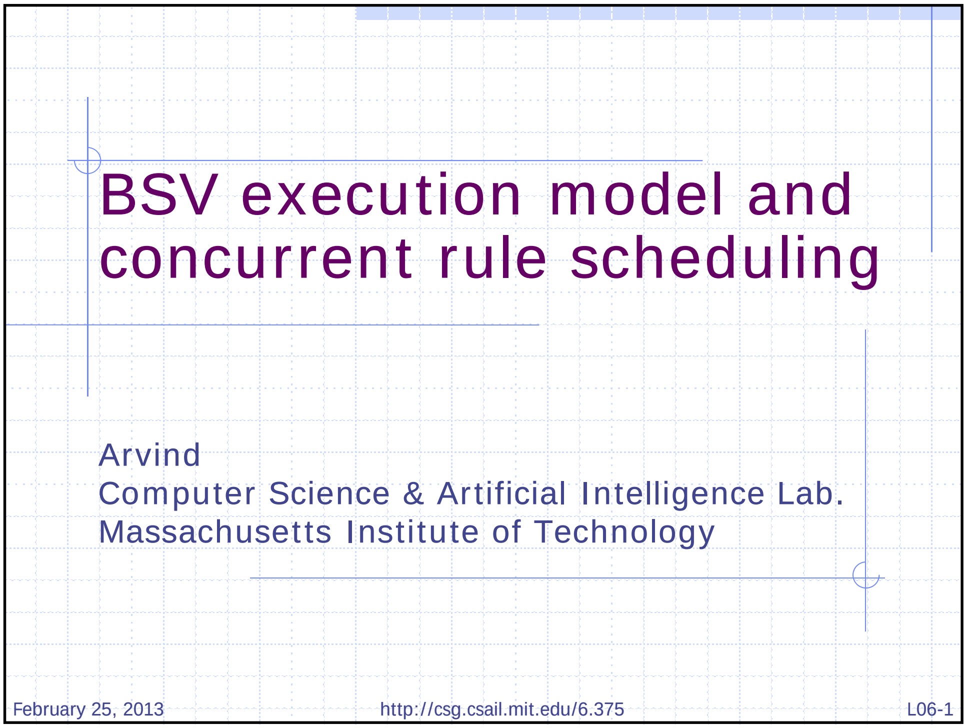
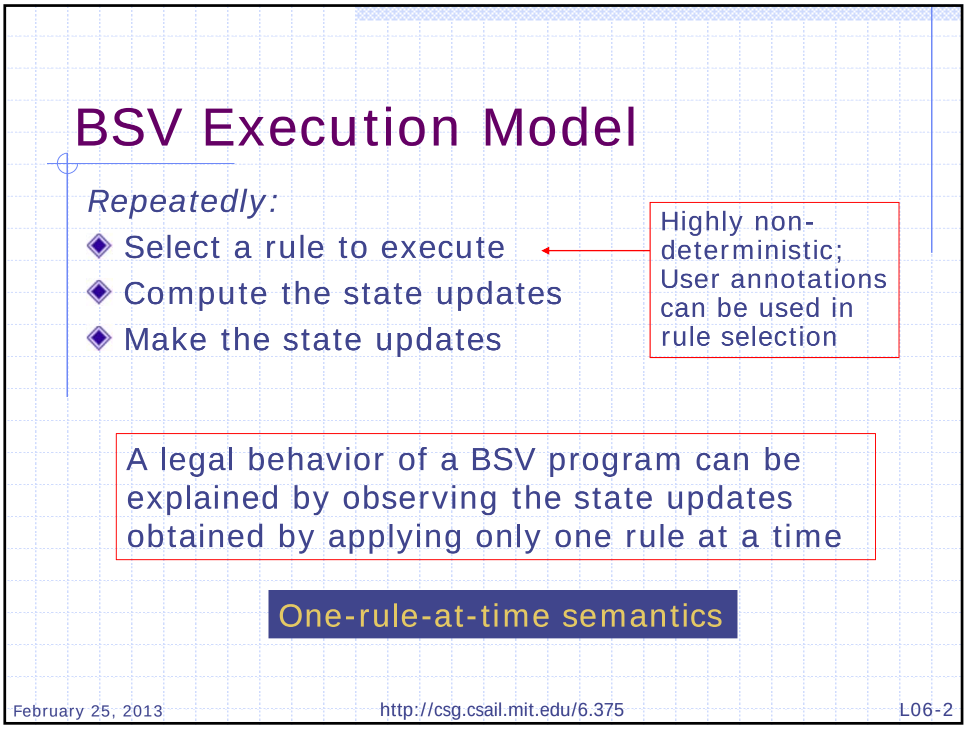


BSV execution model and concurrent rule scheduling

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



BSV Execution Model

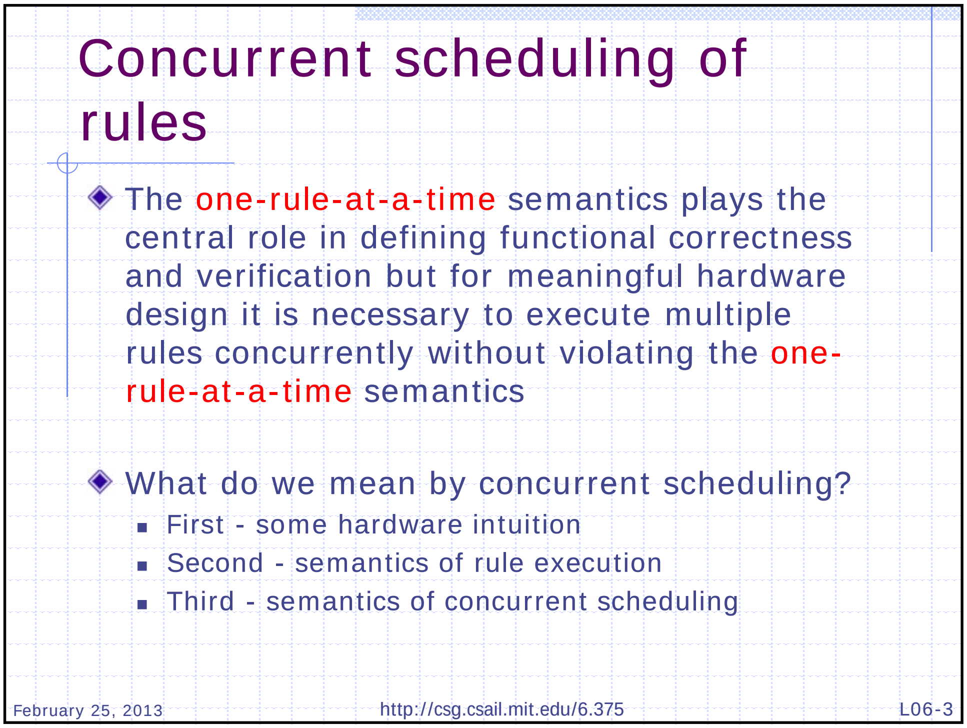
Repeatedly:

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

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

A legal behavior of a BSV program can be explained by observing the state updates obtained by applying only one rule at a time

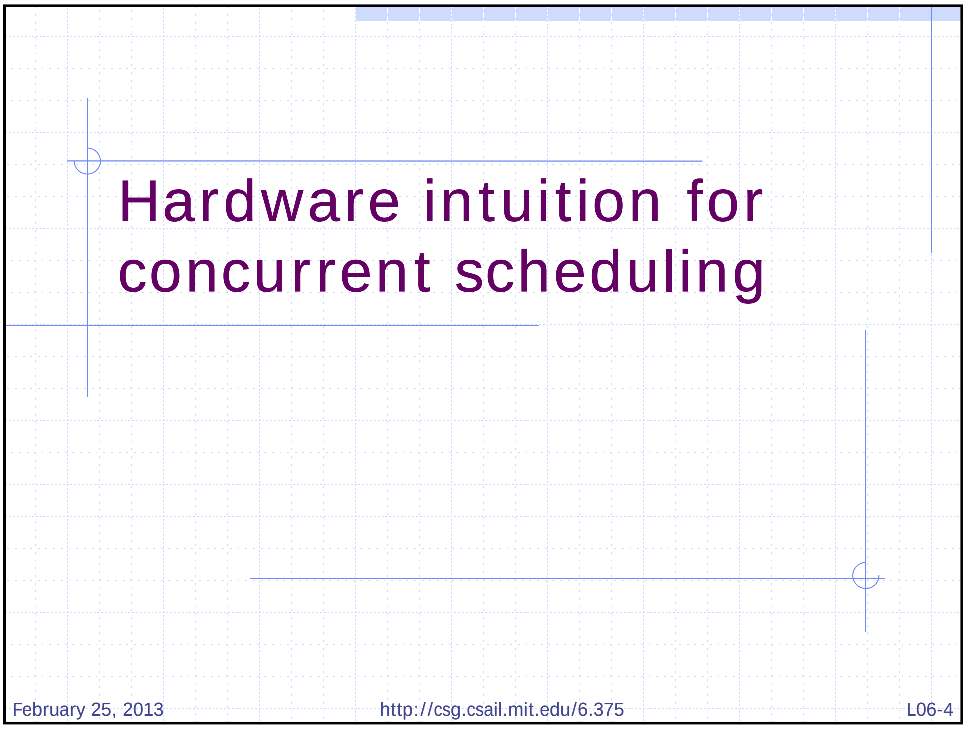
One-rule-at-time semantics



Concurrent scheduling of rules

- The one-rule-at-a-time semantics plays the central role in defining functional correctness and verification but for meaningful hardware design it is necessary to execute multiple rules concurrently without violating the one-rule-at-a-time semantics

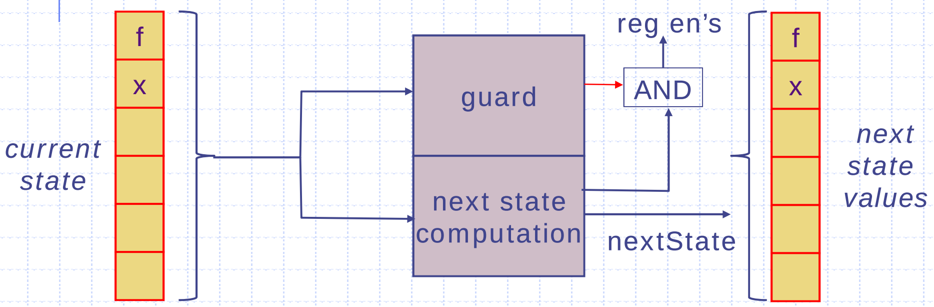
- What do we mean by concurrent scheduling?
  - First - some hardware intuition
  - Second - semantics of rule execution
  - Third - semantics of concurrent scheduling



Hardware intuition for concurrent scheduling

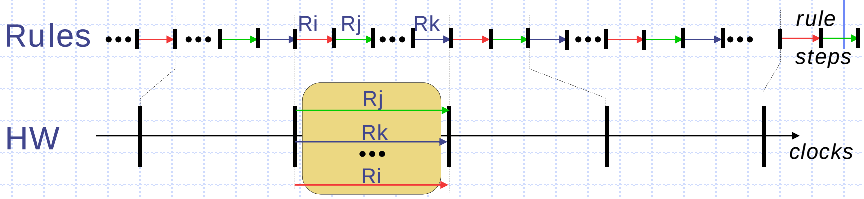
BSV Rule Execution

- A BSV program consists of state elements and rules, aka, Guarded Atomic Actions (GAA) that operate on the state elements
- Application of a rule modifies some state elements of the system in a deterministic manner

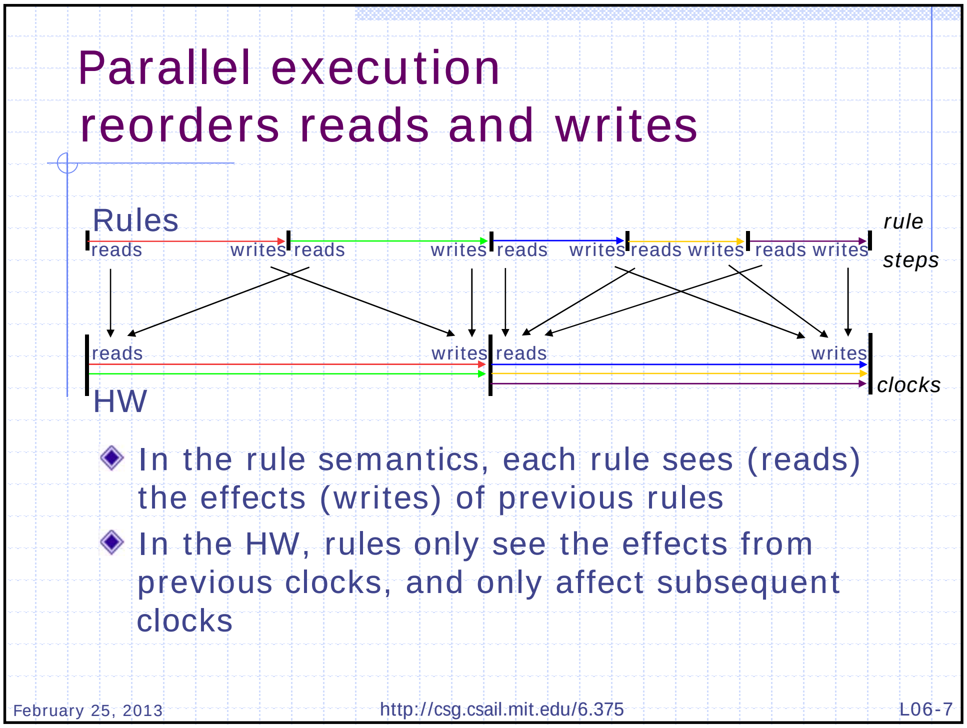


some insight into

Concurrent rule firing



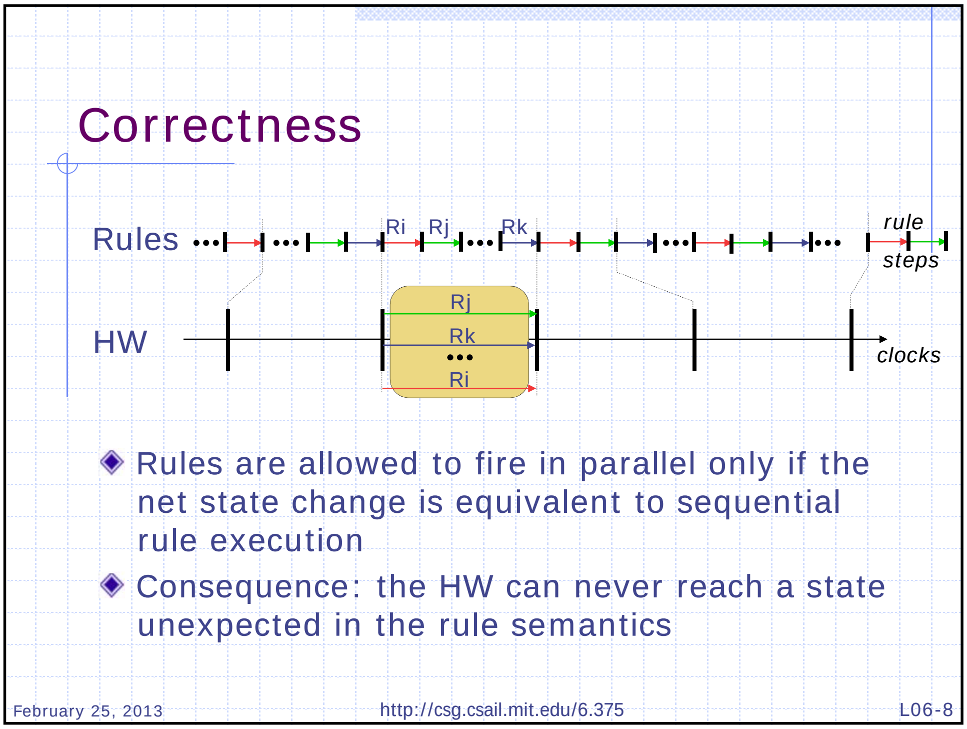
- There are more intermediate states in the rule semantics (a state after each rule step)
- In the HW, states change only at clock edges



Parallel execution reorders reads and writes

Rules
reads
writes
reads
writes
reads
writes
reads
writes
reads
writes
rule steps
reads
writes
reads
writes
HW
clocks

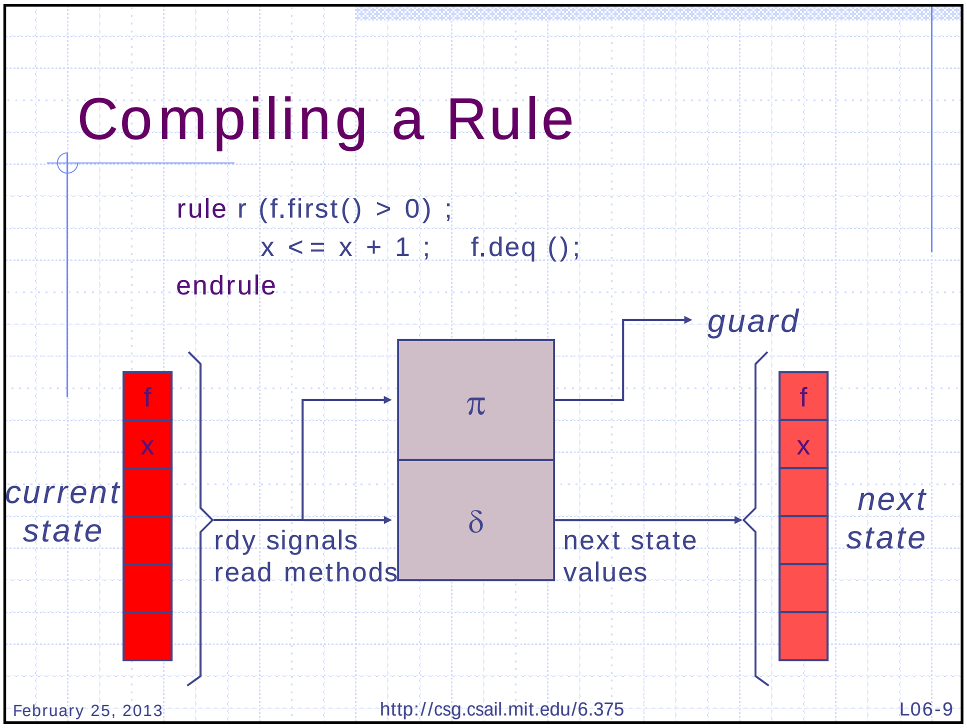
- In the rule semantics, each rule sees (reads) the effects (writes) of previous rules
- In the HW, rules only see the effects from previous clocks, and only affect subsequent clocks



Correctness

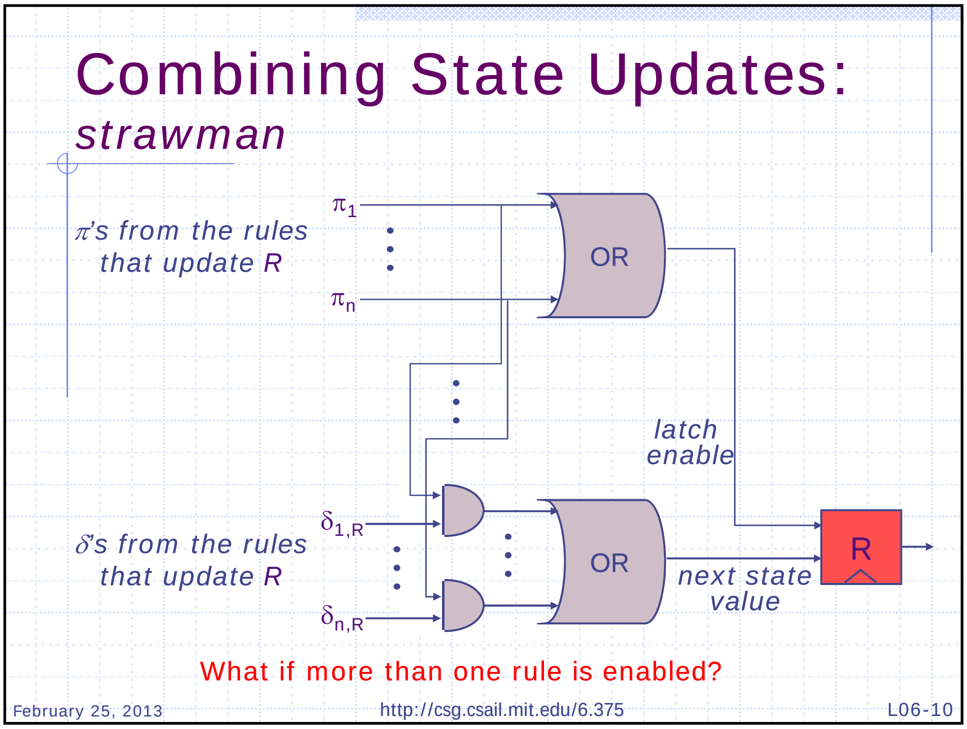
Rules
Ri
Rj
Rk
rule steps
HW
Rj
Rk
Ri
clocks

- Rules are allowed to fire in parallel only if the net state change is equivalent to sequential rule execution
- Consequence: the HW can never reach a state unexpected in the rule semantics



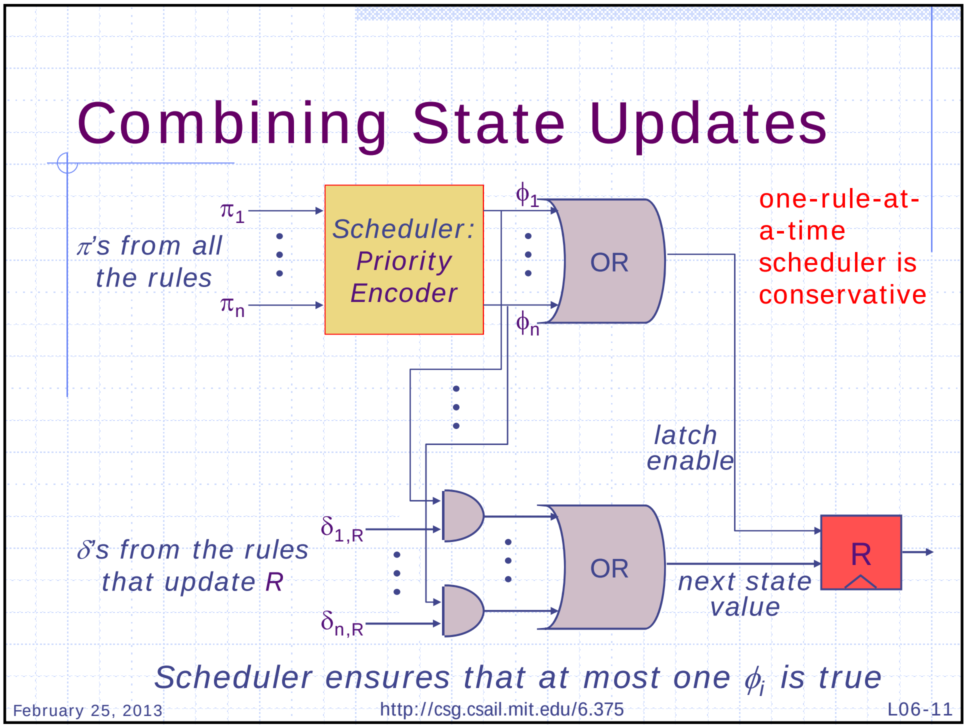
Compiling a Rule

```
rule r (f.first() > 0) ;
      x <= x + 1 ;    f.deq ();
endrule
```



Combining State Updates: *strawman*

What if more than one rule is enabled?

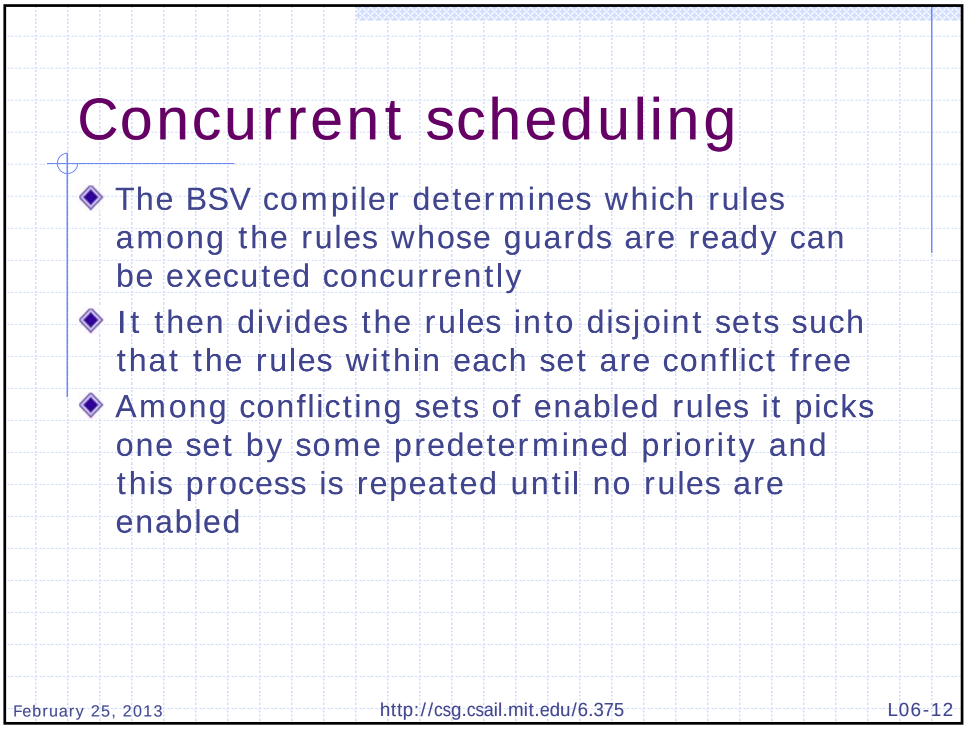


Combining State Updates

π_1
π's from all the rules
π_n
Scheduler: Priority Encoder
ϕ_1
ϕ_n
OR
latch enable
$\delta_{1,R}$
δ's from the rules that update R
$\delta_{n,R}$
OR
next state value
R

one-rule-at-a-time scheduler is conservative

Scheduler ensures that at most one ϕ_i is true



Concurrent scheduling

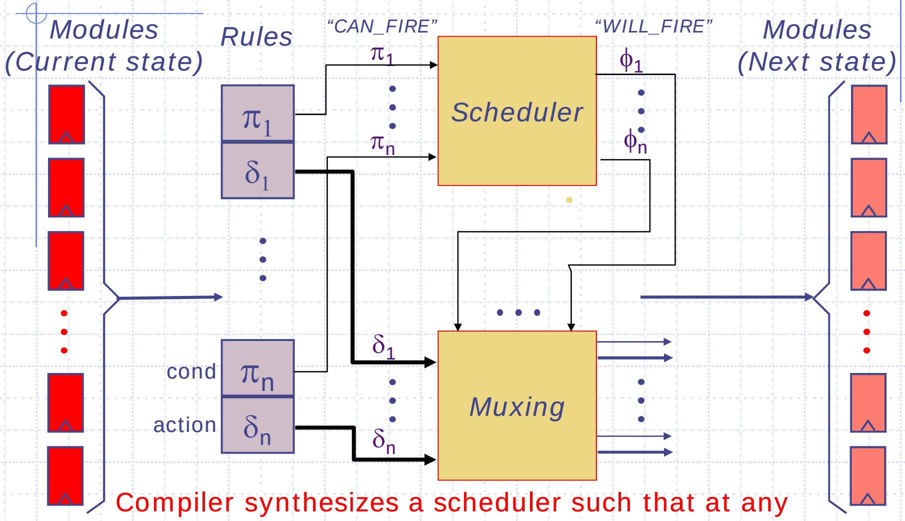
- The BSV compiler determines which rules among the rules whose guards are ready can be executed concurrently
- It then divides the rules into disjoint sets such that the rules within each set are conflict free
- Among conflicting sets of enabled rules it picks one set by some predetermined priority and this process is repeated until no rules are enabled

A compiler test for concurrent rule firing James Hoe, Ph.D., 2000

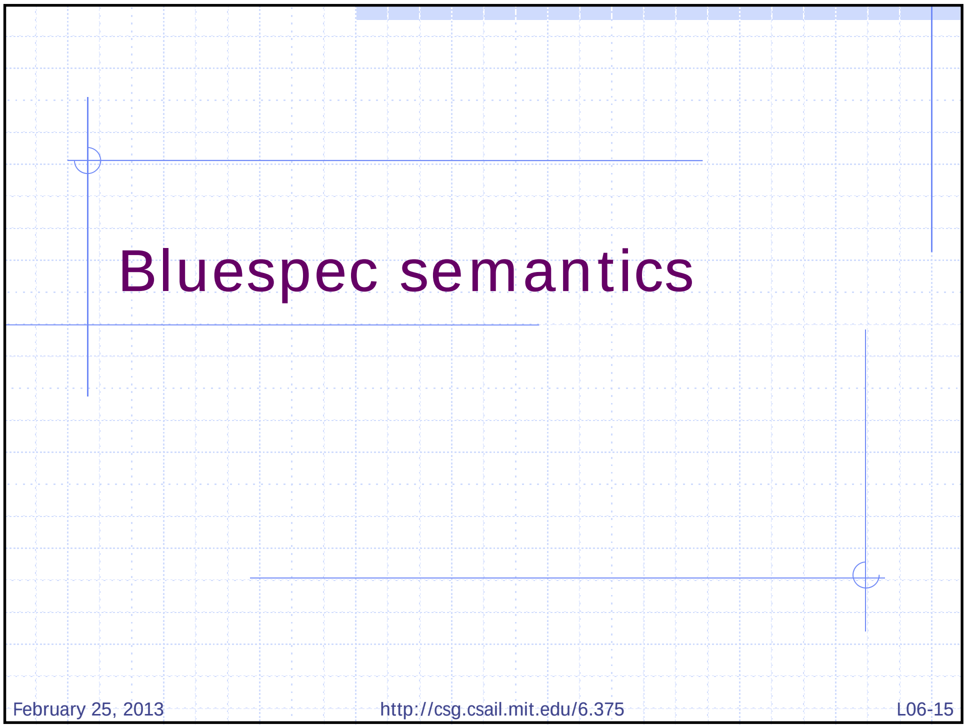
- Let RS(r) be the set of registers rule r may read
- Let WS(r) be the set of registers rule r may write

- Rules ra and rb are *conflict free* (CF) if
  $(RS(ra) \cap WS(rb) = \varphi) \wedge (RS(rb) \cap WS(ra) = \varphi) \wedge (WS(ra) \cap WS(rb) = \varphi)$
- Rules ra and rb are *sequentially composable* (SC) (ra<rb) if
  $(RS(rb) \cap WS(ra) = \varphi) \wedge (WS(ra) \cap WS(rb) = \varphi)$
- If Rules ra and rb *conflict* if they are not CF or SC

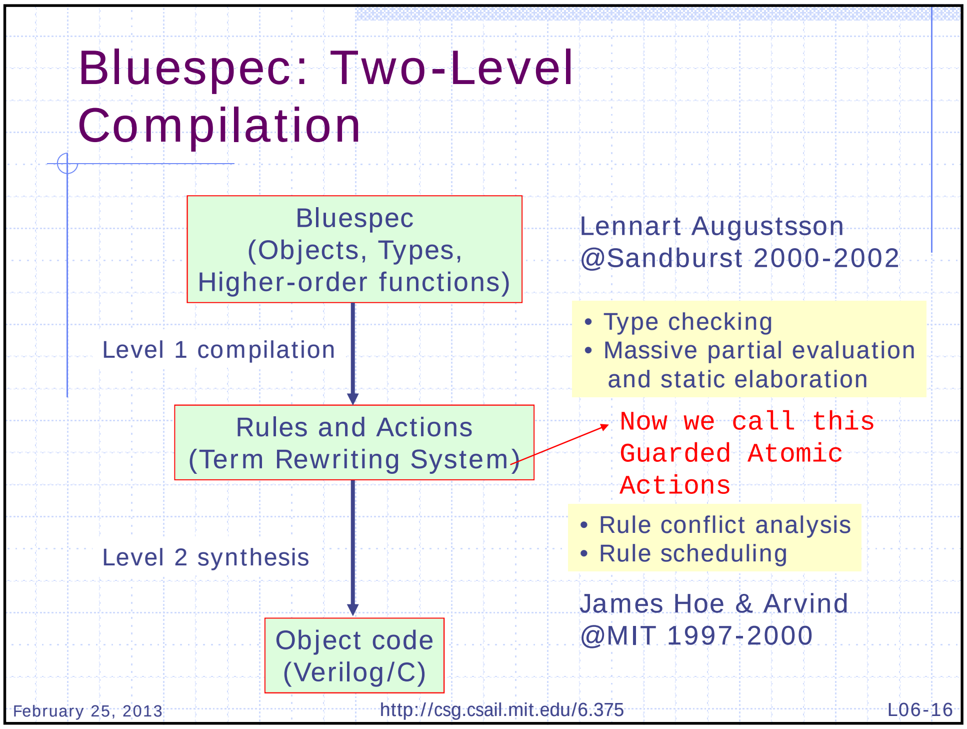
Scheduling and control logic



Compiler synthesizes a scheduler such that at any given time ϕ's for only non-conflicting rules are true



Bluespec semantics



Bluespec: Two-Level Compilation

Bluespec
(Objects, Types,
Higher-order functions)

Level 1 compilation

Rules and Actions
(Term Rewriting System)

Level 2 synthesis

Object code
(Verilog/C)

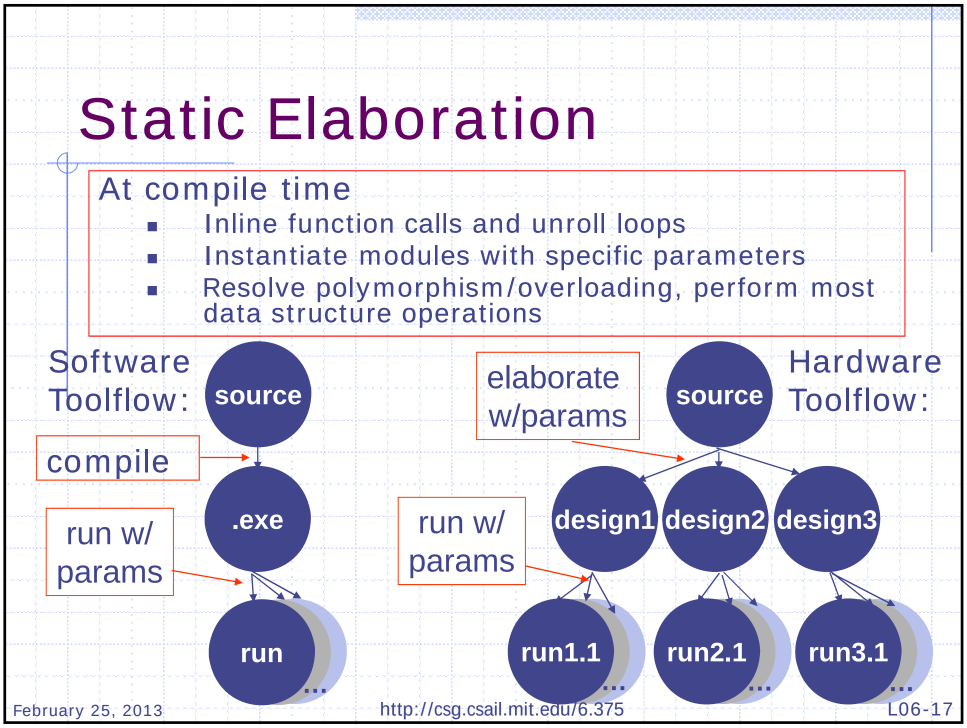
Lennart Augustsson
@Sandburst 2000-2002

- Type checking
- Massive partial evaluation and static elaboration

Now we call this Guarded Atomic Actions

- Rule conflict analysis
- Rule scheduling

James Hoe & Arvind
@MIT 1997-2000



Static Elaboration

At compile time

- Inline function calls and unroll loops
- Instantiate modules with specific parameters
- Resolve polymorphism/overloading, perform most data structure operations

Software Toolflow:

source

compile

.exe

run w/ params

run

…

elaborate w/params

source

Hardware Toolflow:

design1

design2

design3

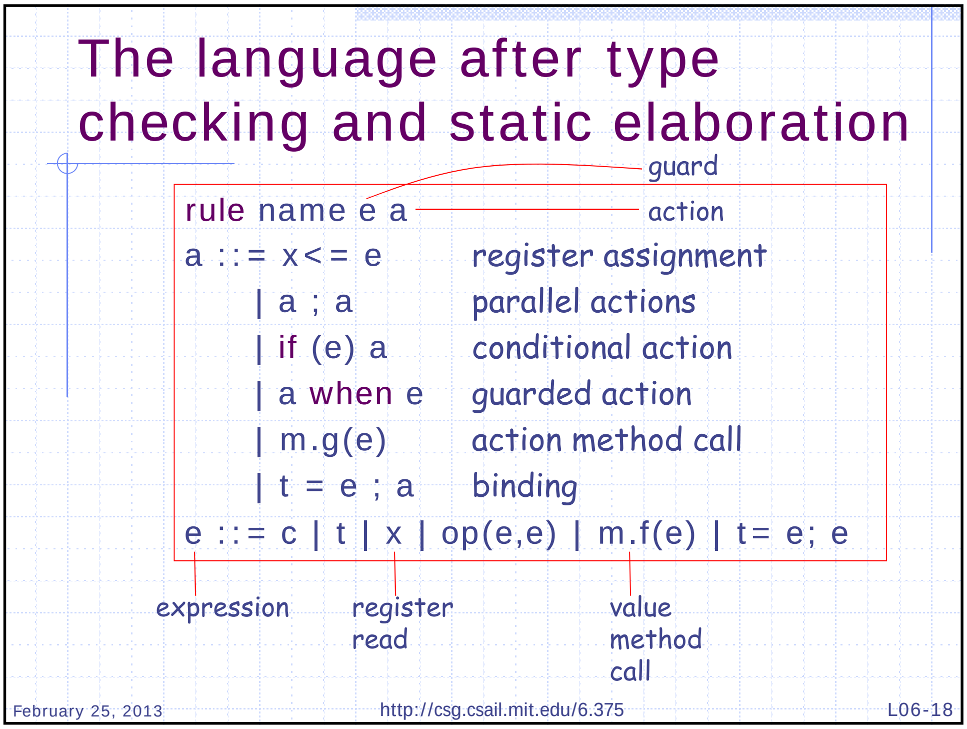
run w/ params

run1.1

run2.1

run3.1

…



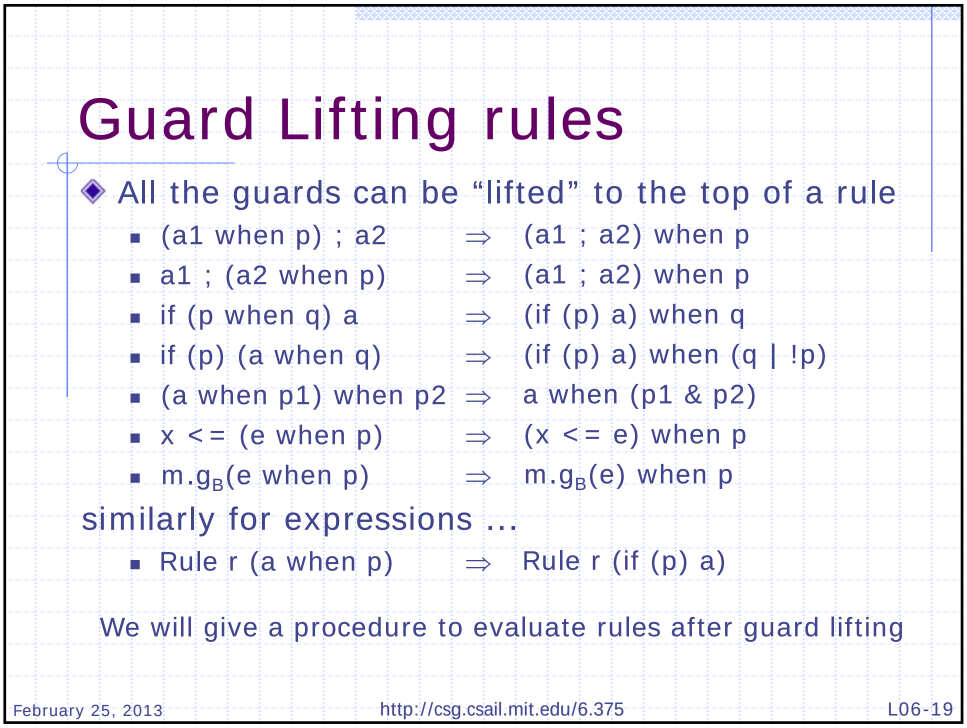
The language after type checking and static elaboration

```
rule name e a                    guard
                                 action
a ::= x<= e          register assignment
     | a ; a         parallel actions
     | if (e) a      conditional action
     | a when e      guarded action
     | m.g(e)        action method call
     | t = e ; a     binding
e ::= c | t | x | op(e,e) | m.f(e) | t= e; e
```

expression

register read

value method call



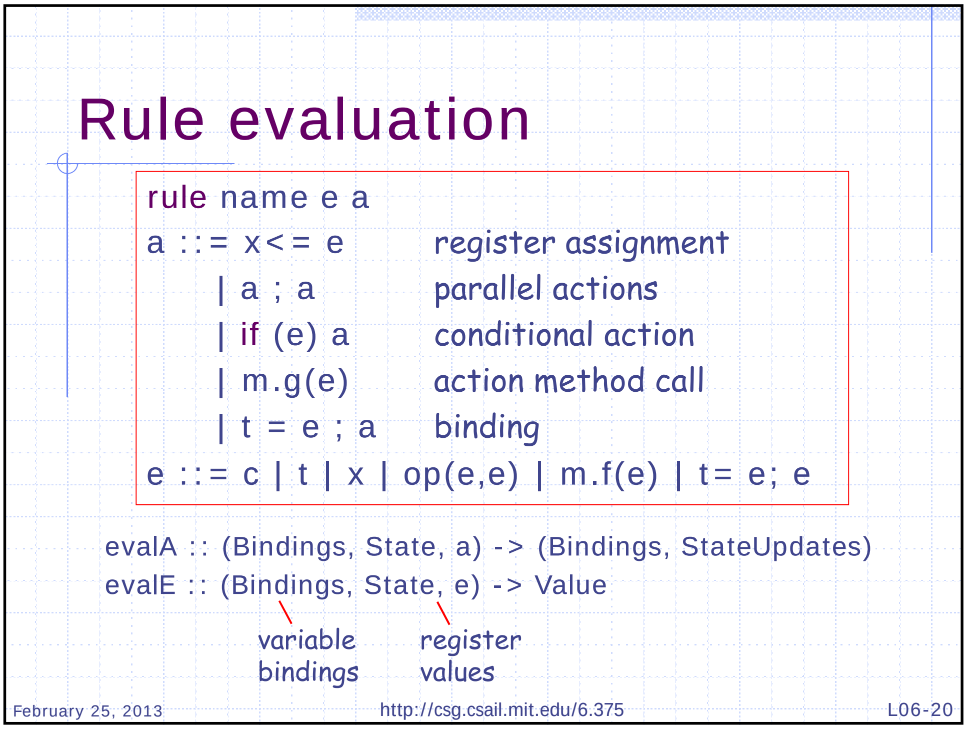
Guard Lifting rules

- All the guards can be “lifted” to the top of a rule
  - (a1 when p) ; a2 ⇒ (a1 ; a2) when p
  - a1 ; (a2 when p) ⇒ (a1 ; a2) when p
  - if (p when q) a ⇒ (if (p) a) when q
  - if (p) (a when q) ⇒ (if (p) a) when (q | !p)
  - (a when p1) when p2 ⇒ a when (p1 & p2)
  - x <= (e when p) ⇒ (x <= e) when p
  - $m.g_B$(e when p) ⇒ $m.g_B$(e) when p

similarly for expressions ...

- Rule r (a when p) ⇒ Rule r (if (p) a)

We will give a procedure to evaluate rules after guard lifting



Rule evaluation

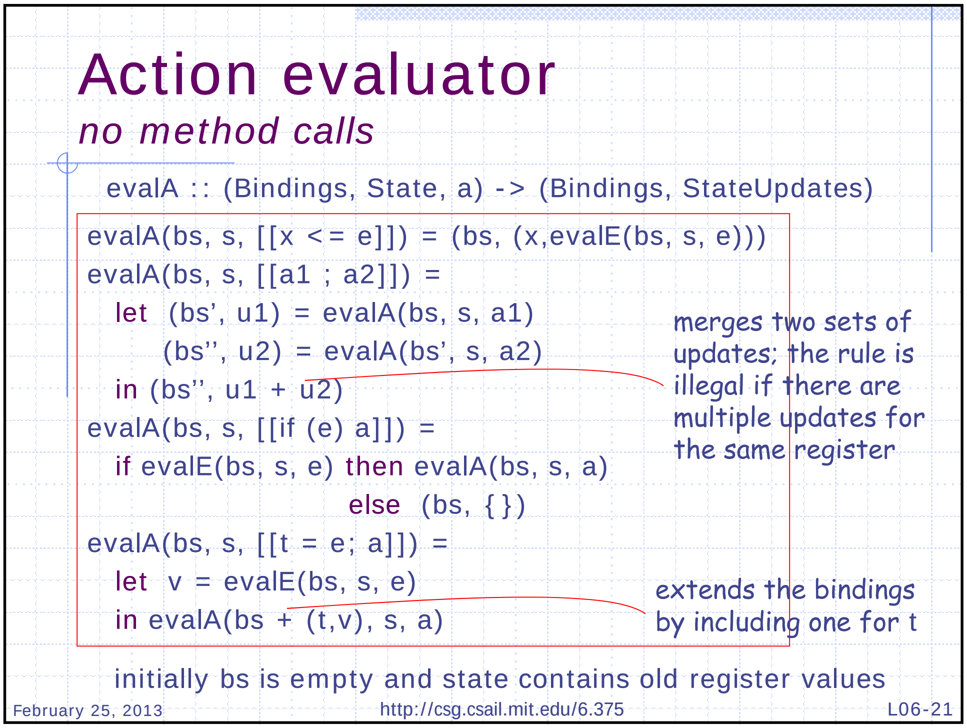
```
rule name e a
a ::= x<= e          register assignment
      | a ; a        parallel actions
      | if (e) a     conditional action
      | m.g(e)       action method call
      | t = e ; a    binding
e ::= c | t | x | op(e,e) | m.f(e) | t= e; e
```

evalA :: (Bindings, State, a) -> (Bindings, StateUpdates)

evalE :: (Bindings, State, e) -> Value

variable bindings

register values



Action evaluator

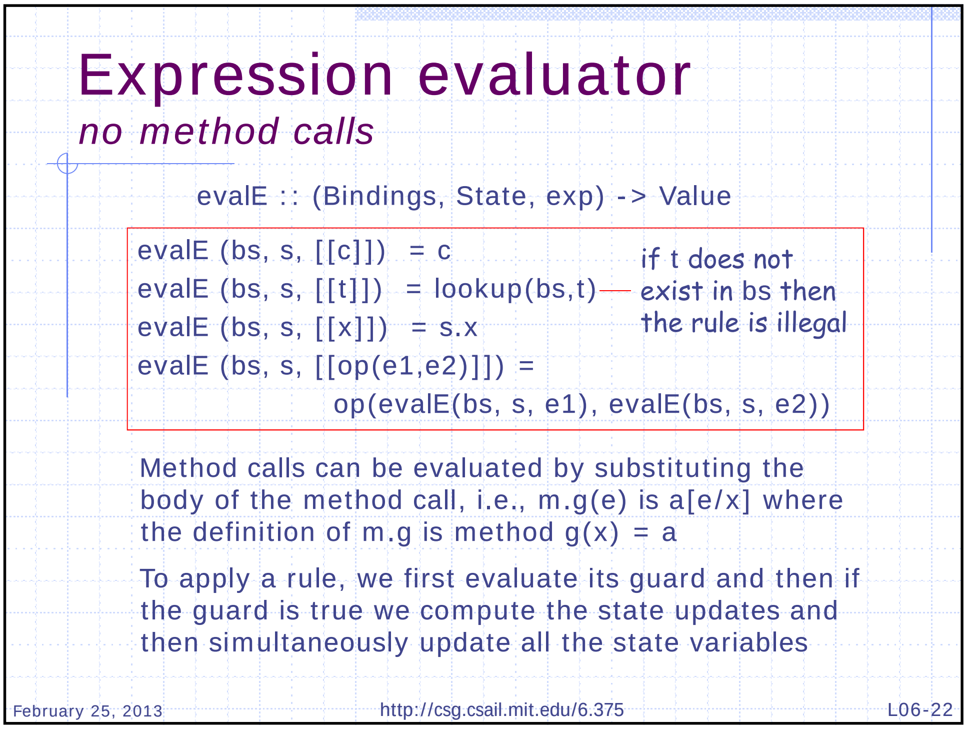
no method calls

```
  evalA :: (Bindings, State, a) -> (Bindings, StateUpdates)
evalA(bs, s, [[x <= e]]) = (bs, (x,evalE(bs, s, e)))
evalA(bs, s, [[a1 ; a2]]) =
   let  (bs', u1) = evalA(bs, s, a1)
        (bs'', u2) = evalA(bs', s, a2)
   in (bs'', u1 + u2)
evalA(bs, s, [[if (e) a]]) =
   if evalE(bs, s, e) then evalA(bs, s, a)
                        else  (bs, {})
evalA(bs, s, [[t = e; a]]) =
   let  v = evalE(bs, s, e)
   in evalA(bs + (t,v), s, a)
```

merges two sets of updates; the rule is illegal if there are multiple updates for the same register

extends the bindings by including one for t

initially bs is empty and state contains old register values



Expression evaluator

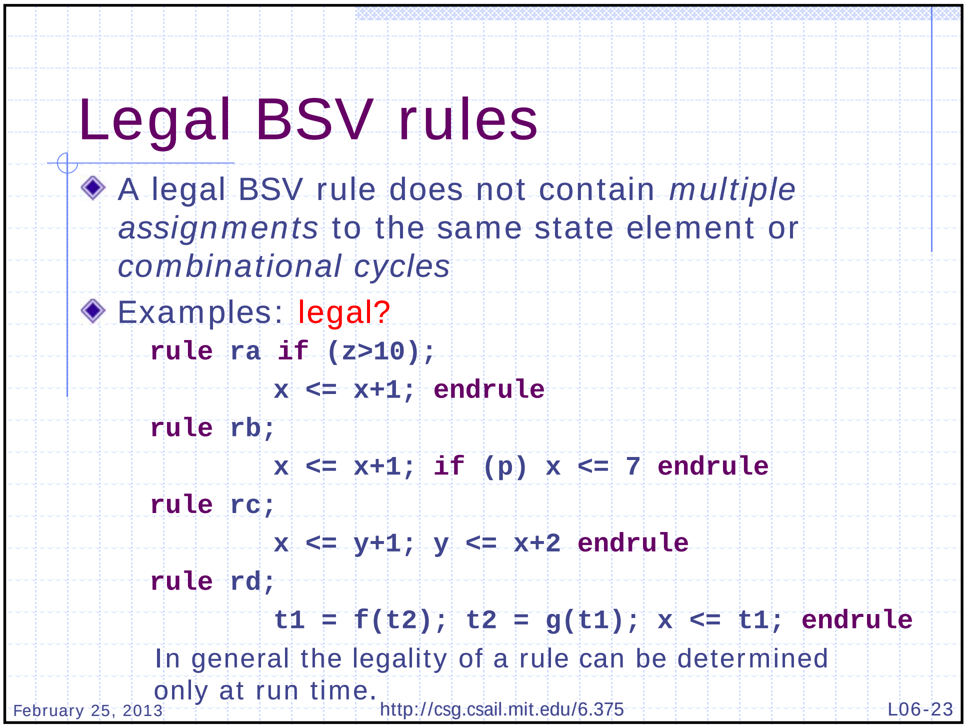
no method calls

```
      evalE :: (Bindings, State, exp) -> Value
evalE (bs, s, [[c]])  = c
evalE (bs, s, [[t]])  = lookup(bs,t)
evalE (bs, s, [[x]])  = s.x
evalE (bs, s, [[op(e1,e2)]]) =
                  op(evalE(bs, s, e1), evalE(bs, s, e2))
```

if t does not exist in bs then the rule is illegal

Method calls can be evaluated by substituting the body of the method call, i.e., m.g(e) is a[e/x] where the definition of m.g is method g(x) = a

To apply a rule, we first evaluate its guard and then if the guard is true we compute the state updates and then simultaneously update all the state variables

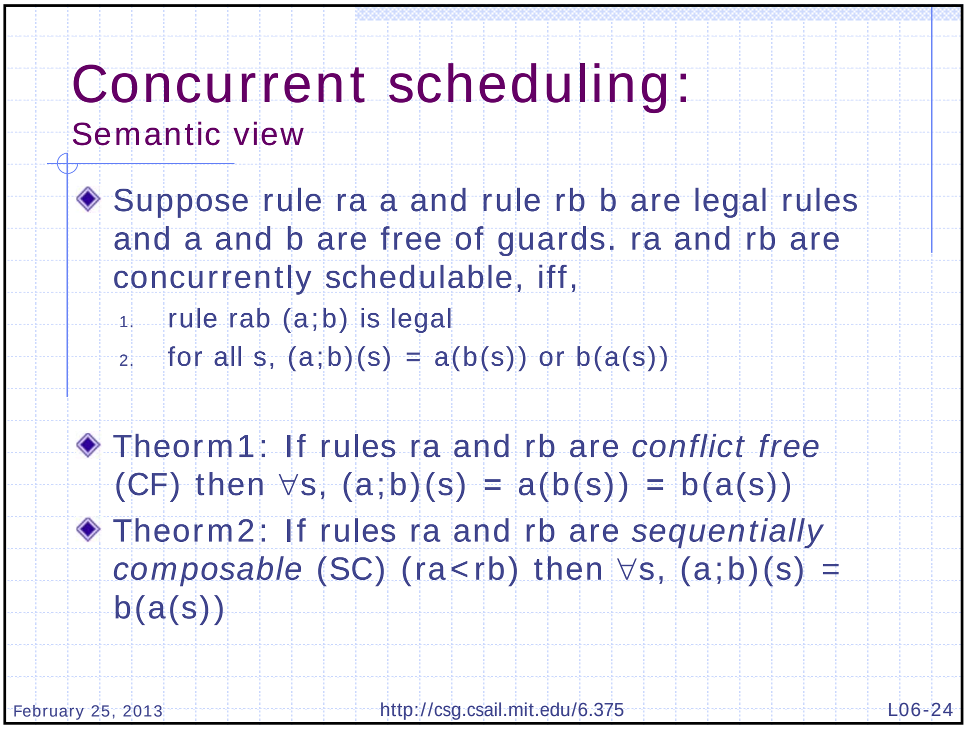


Legal BSV rules

- A legal BSV rule does not contain *multiple assignments* to the same state element or *combinational cycles*
- Examples: legal?

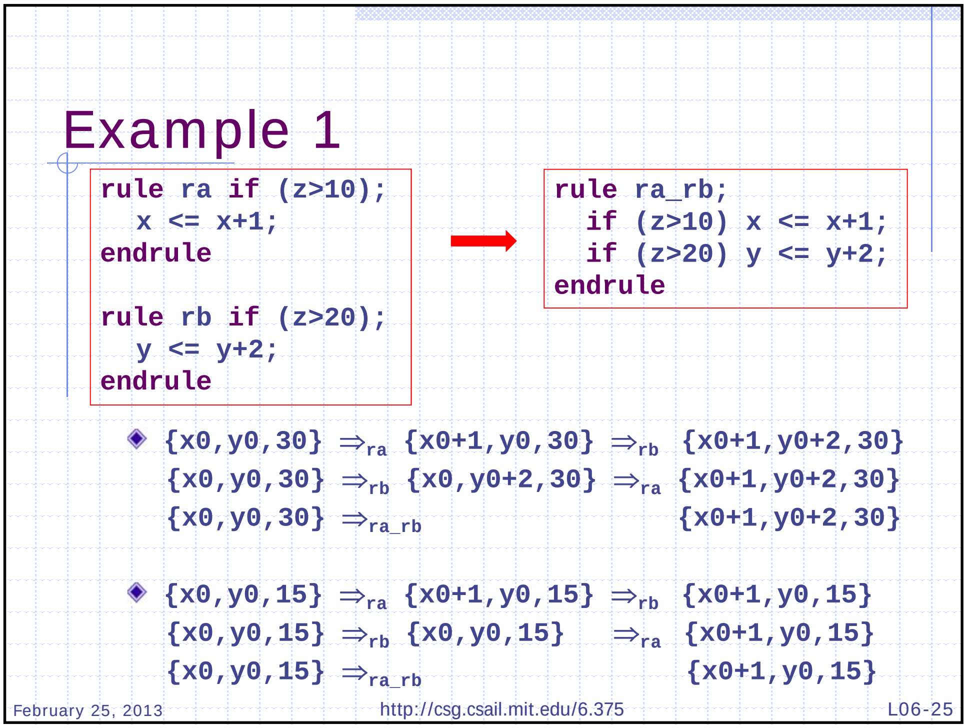
```
rule ra if (z>10);
        x <= x+1; endrule
rule rb;
        x <= x+1; if (p) x <= 7 endrule
rule rc;
        x <= y+1; y <= x+2 endrule
rule rd;
        t1 = f(t2); t2 = g(t1); x <= t1; endrule
```

In general the legality of a rule can be determined only at run time.



Concurrent scheduling:
Semantic view

- Suppose rule ra a and rule rb b are legal rules and a and b are free of guards. ra and rb are concurrently schedulable, iff,
  1. rule rab (a;b) is legal
  2. for all s, (a;b)(s) = a(b(s)) or b(a(s))
- Theorm1: If rules ra and rb are *conflict free* (CF) then ∀s, (a;b)(s) = a(b(s)) = b(a(s))
- Theorm2: If rules ra and rb are *sequentially composable* (SC) (ra<rb) then ∀s, (a;b)(s) = b(a(s))



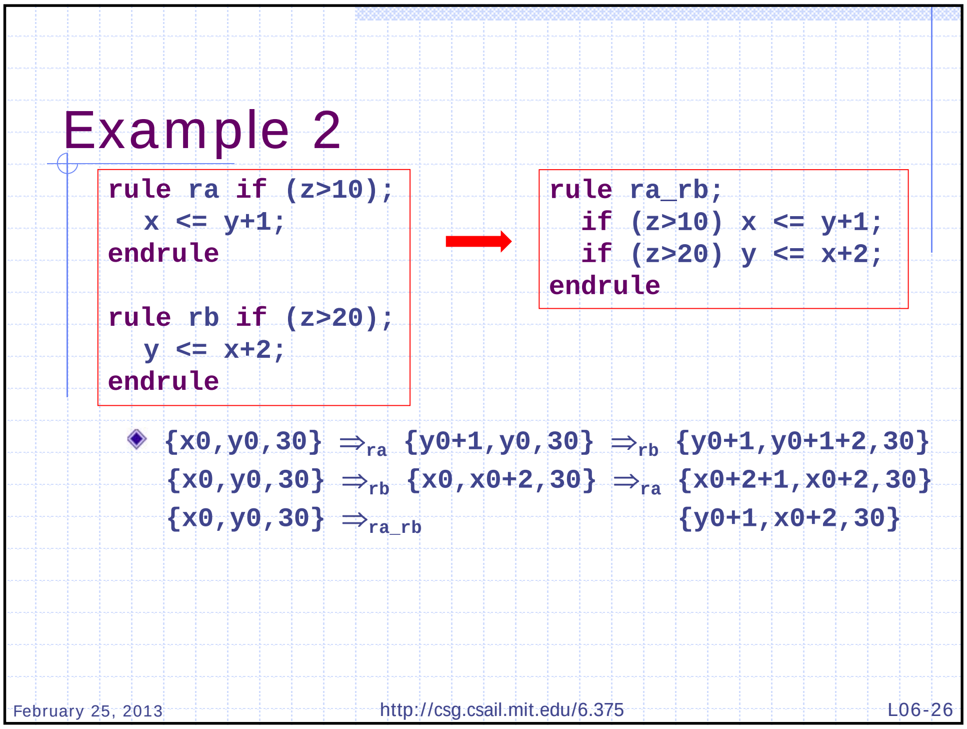
Example 1

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

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

```
rule ra_rb;
   if (z>10) x <= x+1;
   if (z>20) y <= y+2;
endrule
```

- $\{x0,y0,30\} \Rightarrow_{ra} \{x0+1,y0,30\} \Rightarrow_{rb} \{x0+1,y0+2,30\}$
  $\{x0,y0,30\} \Rightarrow_{rb} \{x0,y0+2,30\} \Rightarrow_{ra} \{x0+1,y0+2,30\}$
  $\{x0,y0,30\} \Rightarrow_{ra\_rb} \{x0+1,y0+2,30\}$
- $\{x0,y0,15\} \Rightarrow_{ra} \{x0+1,y0,15\} \Rightarrow_{rb} \{x0+1,y0,15\}$
  $\{x0,y0,15\} \Rightarrow_{rb} \{x0,y0,15\} \Rightarrow_{ra} \{x0+1,y0,15\}$
  $\{x0,y0,15\} \Rightarrow_{ra\_rb} \{x0+1,y0,15\}$



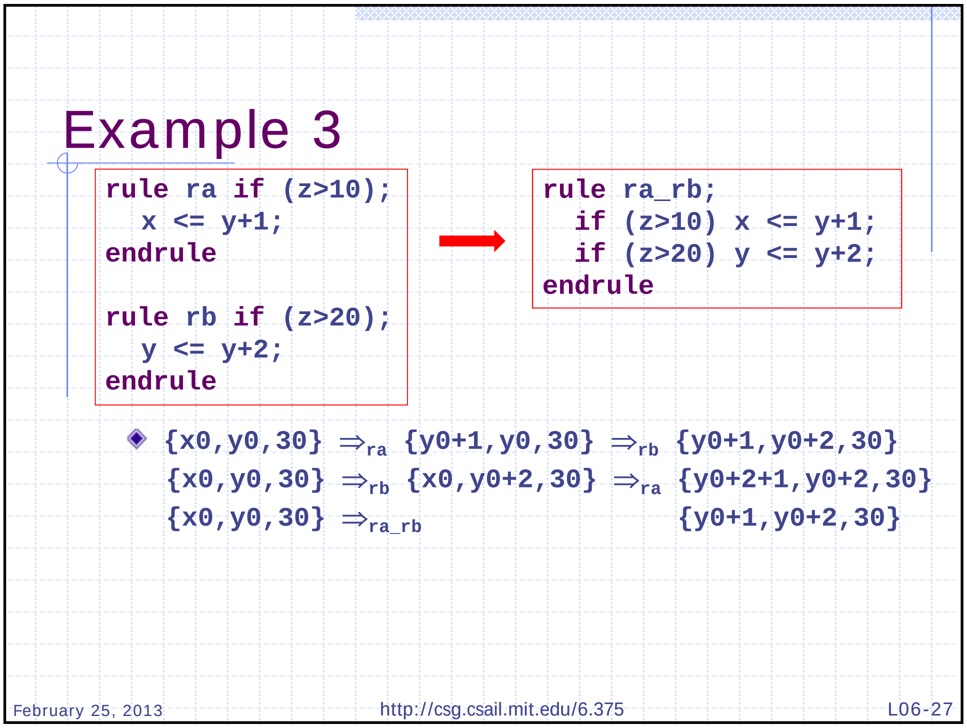
Example 2

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

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

```
rule ra_rb;
   if (z>10) x <= y+1;
   if (z>20) y <= x+2;
endrule
```

- $\{x0,y0,30\} \Rightarrow_{ra} \{y0+1,y0,30\} \Rightarrow_{rb} \{y0+1,y0+1+2,30\}$
  $\{x0,y0,30\} \Rightarrow_{rb} \{x0,x0+2,30\} \Rightarrow_{ra} \{x0+2+1,x0+2,30\}$
  $\{x0,y0,30\} \Rightarrow_{ra\_rb} \{y0+1,x0+2,30\}$



Example 3

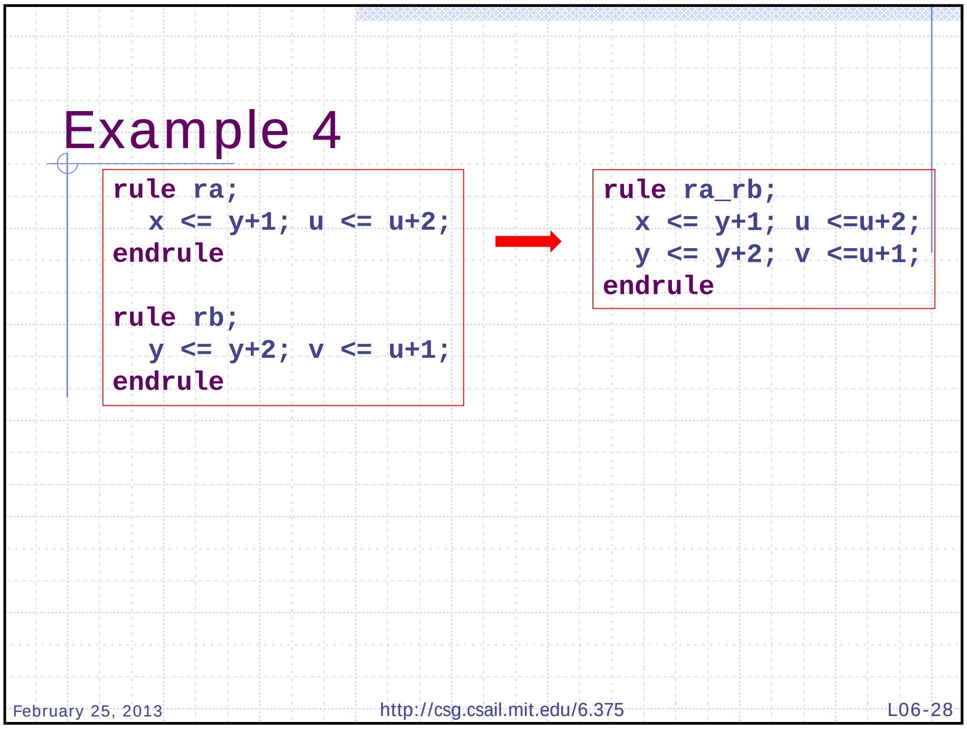
```
rule ra if (z>10);
  x <= y+1;
endrule

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

→

```
rule ra_rb;
  if (z>10) x <= y+1;
  if (z>20) y <= y+2;
endrule
```

- $\{x0,y0,30\} \Rightarrow_{ra} \{y0+1,y0,30\} \Rightarrow_{rb} \{y0+1,y0+2,30\}$
  $\{x0,y0,30\} \Rightarrow_{rb} \{x0,y0+2,30\} \Rightarrow_{ra} \{y0+2+1,y0+2,30\}$
  $\{x0,y0,30\} \Rightarrow_{ra\_rb} \{y0+1,y0+2,30\}$



Example 4

```
rule ra;
  x <= y+1; u <= u+2;
endrule

rule rb;
  y <= y+2; v <= u+1;
endrule
```

→

```
rule ra_rb;
  x <= y+1; u <=u+2;
  y <= y+2; v <=u+1;
endrule
```