The Semantics of Bluespec

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Nirav Dave, Mike Pellauer, Arvind
[MEMOCODE 2007]

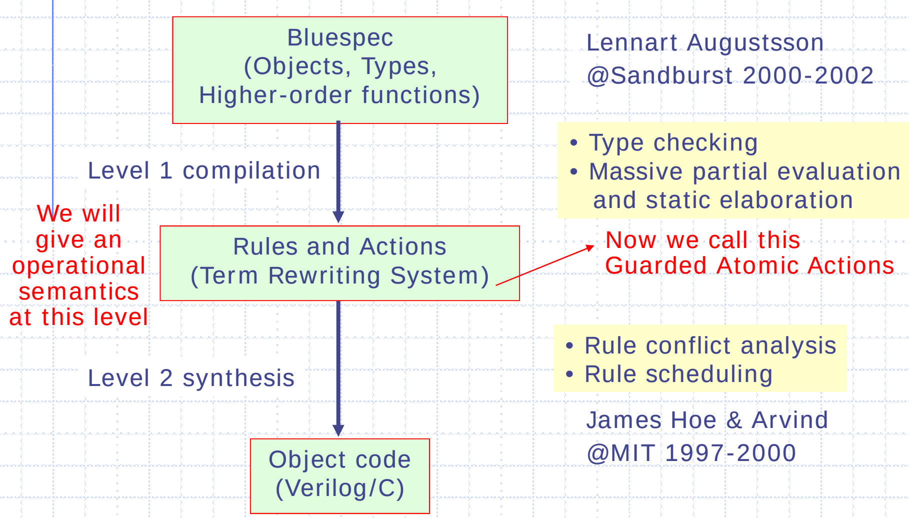
Why Semantics?

- The semantics helps in deciding
  - The meaning of a language construct – especially the corner cases
  - The correctness of the compiler, especially the correctness of optimizations
  - If the meaning of a construct is ambiguous or underspecified
- Characteristics of useful semantics
  - The "kernel language" should be small
  - The semantics should be concise, easy to understand and remember

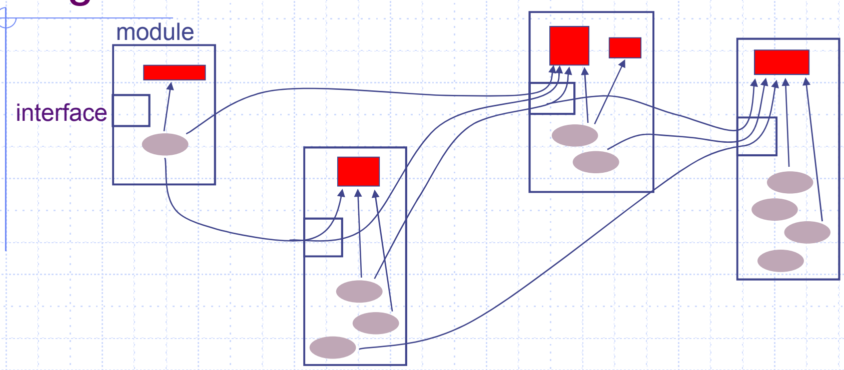
Indirect Semantics

- Most modern languages are big and their syntax contains many features for user convenience which interfere in understanding the deep semantics
- Consequently, the semantics are often given in two parts:
  1. A static translation from the source language into a simpler, smaller *kernel* language
  2. An Operational and sometimes the Denotational semantics of the *kernel* language

Bluespec: Two-Level Compilation



Bluespec: State and Rules organized into *modules*



All *state* (e.g., Registers, FIFOs, RAMs, ...) is explicit.

Behavior is expressed in terms of atomic actions on the state:

Rule: condition ➔ action

Rules can manipulate state in other modules only *via* their interfaces.

BTRS: A Language of GAAs

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

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

New

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

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

Action Connectives: Par vs. Seq

- Parallel compositions (a1|a2)
  - Neither action observes others' updates
  - Writes are disjoint (muliple writes to a register in a rule is an error)
  - Natural in hardware

(r1:= r2 | r2:= r1)   swaps r1 & r2

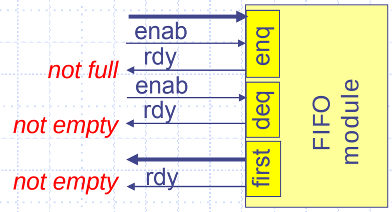
- Sequential Connective (a1;a2)
  - a2 observes a1's updates
  - Still atomic
  - Not in BSV due to implementation complications (requires modules with"derived" interfaces)

BSV Guards

- BSV allows guards only at the top level in rules
- However, it has implicit guards in method calls
  - For semantics these implicit guards are made explicit using "when"
  - BTRS allows guards everywhere without introducing any additional complexity

Guarded Interfaces: An Example FIFO

```
interface  FIFO#(type t);
  method Action enq(t);
  method Action deq();
  method t first()
endinterface
```



make implicit guards explicit

~~m.g(e)~~

$m.g_B(e)$ *when* $m.g_G$

Predicating Actions: Guards vs. Ifs

- Guards affect their surroundings

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

- The effect of an "if" is local

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

p1 has no effect on a2

Exercise: Understanding Guards

1. (a1 *when* p1) | (a2 *when* p2)
   ==> (a1 | a2) *when* (p1 && p2)

2. (*if* p *then* (a1 *when* q)) | a2
   ==> ((*if* p *then* a1)| a2) *when* (q || !p)

GAA Execution model

Repeatedly:

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

Hardware Implementation: Needs to restrict this behavior to be deterministic

To get "good" hardware:

- The compiler must schedule multiple rules in a single cycle
- But the effect must look like a serial execution of rules

Operational Semantics

- We will first give the semantics of a single rule, i.e., how it modifies the state of the system
  - Give a rule for how each language construct modifies the state
  - Rules must be compositional
- Then we will give the semantics of multiple rules

Semantics of a rule execution

- Specify which state elements the rule modifies
  - Let S represent the value of all the registers before the rule executes
  - Let U be the set of updates implied by the rule execution
  - Let B represent the let-bound values we encounter in execution

<S,U,B>

BTRS Action Rules: Register Assignment

reg-update

$$\frac{\langle S,U,B\rangle \vdash e \Rightarrow v,\ \ (v \mathrel{!=} NR)}{\langle S,U,B\rangle \vdash (r := e) \Rightarrow \{\}[v/r]}$$

NR represents the "not ready" value

BTRS Action Rules: Conditional Action

if-true

$$\frac{\langle S,U,B\rangle \vdash e \Rightarrow \text{true},\quad \langle S,U,B\rangle \vdash a \Rightarrow U'}{\langle S,U,B\rangle \vdash (\textit{if}\ e\ \textit{then}\ a) \Rightarrow U'}$$

if-false

$$\frac{\langle S,U,B\rangle \vdash e \Rightarrow \text{false}}{\langle S,U,B\rangle \vdash (\textit{if}\ e\ \textit{then}\ a) \Rightarrow U}$$

BTRS Action Rules:
Par and Seq action composition

par

<S,U,B> ├ a1 ⇒ U1, <S,U,B> ├ a2 ⇒ U2
———————————————————————
<S,U,B> ├ (a1 | a2) ⇒ pmerge(U1,U2)

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

seq

<S,U,B> ├ a1 ⇒ U1, <S,U1++U,B> ├ a2 ⇒ U2
———————————————————————
<S,U,B> ├ (a1 ; a2) ⇒ U2++U1

↑ Like a set union but U2 dominates if there are any duplicates

BTRS Action Rules:
Let and Method calls

a-let-sub

<S,U,B> ├ e ⇒ v, <S,U,B[v/t]> ├ a ⇒ U'
———————————————————————
<S,U,B> ├ ((t = e) *in* a) ⇒ U'

a-meth-call

<S,U,B> ├ e ⇒ v , (v != NR) ,
λx.a = lookup(m.g), <S,U,B[v/x]> ├ a ⇒ U'
———————————————————————
<S,U,B> ├ m.g(e) ⇒ U'

Guard Semantics

e-when-true

$$\frac{\langle S,U,B\rangle \vdash e1 \Rightarrow v, \quad \langle S,U,B\rangle \vdash e2 \Rightarrow \text{true}}{\langle S,U,B\rangle \vdash (e1\ \textit{when}\ e2) \Rightarrow v}$$

e-when-false

$$\frac{\langle S,U,B\rangle \vdash e1 \Rightarrow v, \quad \langle S,U,B\rangle \vdash e2 \Rightarrow \text{false}}{\langle S,U,B\rangle \vdash (e1\ \textit{when}\ e2) \Rightarrow \text{NR}}$$

a-when-ready

$$\frac{\langle S,U,B\rangle \vdash e \Rightarrow \text{true}, \quad \langle S,U,B\rangle \vdash a \Rightarrow U}{\langle S,U,B\rangle \vdash (a\ \textit{when}\ e) \Rightarrow U}$$

- If no rule applies, e.g., if e in (a when e) returns false or NR, then the system is stuck and the effect of the whole atomic action of which (a when e) is a part is "no action".

BTRS Expression Rules: register,constant,variables,ops

register read $\langle S,U,B\rangle \vdash r \Rightarrow (U{+}{+}S)\ (r)$

Constant $\langle S,U,B\rangle \vdash c \Rightarrow \underline{c}$

Variable $\langle S,U,B\rangle \vdash t \Rightarrow B(t)$

op

$$\frac{\langle S,U,B\rangle \vdash e1 \Rightarrow v1, \ v1 \mathrel{!=} \text{NR} \quad \langle S,U,B\rangle \vdash e2 \Rightarrow v2, \ v2 \mathrel{!=} \text{NR}}{\langle S,U,B\rangle \vdash (e1\ \textit{op}\ e2) \Rightarrow v1\ \underline{\textit{op}}\ v2}$$

BTRS Expression Rules: Conditional expression

tri-true

$$\frac{\langle S,U,B\rangle \vdash e1 \Rightarrow true, \quad \langle S,U,B\rangle \vdash e2 \Rightarrow v}{\langle S,U,B\rangle \vdash (e1 \ ?\ e2 : e3) \Rightarrow v}$$

tri-fasle

$$\frac{\langle S,U,B\rangle \vdash e1 \Rightarrow false, \quad \langle S,U,B\rangle \vdash e3 \Rightarrow v}{\langle S,U,B\rangle \vdash (e1 \ ?\ e2 : e3) \Rightarrow v}$$

BTRS Expression Rules: Let and Method calls

e-let-sub

$$\frac{\langle S,U,B\rangle \vdash e \Rightarrow v, \quad \langle S,U,B[v/t]\rangle \vdash e2 \Rightarrow v2}{\langle S,U,B\rangle \vdash ((t = e1)\ \mathit{in}\ e2) \Rightarrow v2}$$

e-meth-call

$$\frac{\langle S,U,B\rangle \vdash e \Rightarrow v\ ,\ (v\ != NR)\ , \quad \lambda x.eb = lookup(m.f), \quad \langle S,U,B[v/x]\rangle \vdash eb \Rightarrow v'}{\langle S,U,B\rangle \vdash m.f(e) \Rightarrow v'}$$

BTRS Specification Semantics

1. Pick a rule
2. Compute initial <S,U,B> (read all registers)
3. Apply the rule yielding U
4. If no failure then update all registers according to U
5. Go to step 1

Scheduling

Schedules specify which rules should be executed concurrently without violating one-rule-at-a-time semantics

A schedule can be specified by parallel and conditional composition or rules

Lifting When's to the top

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 ...

Final Takeaway

- Have fun designing systems with Bluespec

Thanks

Pizza time now!