Bluespec-7: Semantics of Bluespec

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

BS₀ : A simple language of Guarded Atomic Actions

a is an Action;
e is an Expression;
r is a register (state variable)

a ::= r := e | a when e | if e then a | a ; a

e ::= r | c | Op(e , e ) | e ? e : e | e when e | ... 

";" is commutative and associative, i.e.
a₁;a₂ = a₂;a₁
a₁;(a₂;a₃) = (a₁;a₂);a₃

BS₀ Program

A program is a collection of registers and rules:
R, R₁, R₂, ... are names for rules;
r, r₁, r₂, ... are register names

Program ::= Registers r₁, ... ;
Rule R₁ a₁; ... ; Rule Rₘ aₘ ;

a ::= r := e | a when e | if e then a | a ; a

e ::= r | c | Op(rᵥ , rₑ ) | e ? e : e | e when e | ...
**Guards vs If’s**

- A guard on one action of a group of actions affects every action within the group:
  \[(a1 \text{ when } p1); (a2 \text{ when } p2) \Rightarrow (a1; a2) \text{ when } p1 \& p2\]
- A condition of a Conditional action only affects the actions within the scope of the conditional action:
  \[(\text{if } p1 \text{ then } a1); (\text{if } p2 \text{ then } a2)\]
  \[p1 \text{ has no effect on } a2 \ldots\]

**Canonicalizing BS\(_0\)** ignoring guards on expressions

Rules for canonicalization:

1. \[(a1 \text{ when } p); a2 \Rightarrow (a1; a2) \text{ when } p\]
2. \[(a \text{ when } p1) \text{ when } p2 \Rightarrow a \text{ when } (p1 \& p2)\]
3. \[\text{if } p \text{ then } (a \text{ when } q) \Rightarrow (\text{if } p \text{ then } a) \text{ when } (p \& q | !p)\]

**Conditionals & Cases**

- if \(p\) then \(a1\) else \(a2\)
  \[= \text{ if } p \text{ then } a1; \text{ if } !p \text{ then } a2\]

- Similarly for cases

**BS\(_0\) : Canonical form**

In the canonical form, expressions have no guards and an (or a compound) action has at most one guard and it occurs at the top level;

- \(ag\) is an Action with guard
- \(a\) is an Action without guard;
- \(e\) is an Expression;
- \(r\) is a register (state variable)

- \(ag ::= a \text{ when } e\)
- \(a ::= r := e \mid \text{if } e \text{ then } a \mid a ; a\)
- \(e ::= r \mid c \mid \text{Op}(e, e) \mid e ? e : e \mid \ldots\)
Rules for Canonicalizing BS₀

1. \((a₁ \text{ when } p); a₂\) \implies (a₁; a₂) when p
2.1 \((a \text{ when } p) \text{ when } q\) \implies a \text{ when } (p \& q)
2.2 \((e \text{ when } p) \text{ when } q\) \implies e \text{ when } (p \& q)
3.1 if \(p\) then \((a \text{ when } q)\) \implies (if \(p\) then \(a\)) \text{ when } (p \& \neg p | \neg p)
3.2 \(p \;?\; (e₁ \text{ when } q) : e₂\) \implies (p \;?\; e₁ : e₂) \text{ when } (p \& \neg p | \neg p)
4. \(r := (e \text{ when } q)\) \implies (r := e) \text{ when } q
5.1 Op(e₁ \text{ when } q, e₂) \implies Op(e₁,e₂) \text{ when } q
5.2 Op(e₁, e₂ \text{ when } q) \implies Op(e₁,e₂) \text{ when } q
3.3 \(p \;?\; e₁ : (e₂ \text{ when } q)\) \implies (p \;?\; e₁ : e₂) \text{ when } (p \& \neg p \& q)

Theorem: Canonical form for an action exists and is unique up to the boolean simplification of the guard expression.

BS₁ = BS₀ + Let blocks
Introducing local names

t, t₁, t₂, ... are identifiers (not registers)

Program ::= Registers \(r_1, ..., ;\)

t₁ = e₁; ...; tₙ = eₙ;
Rule R₁ a₁; ...; Rule Rₘ aₘ;

a ::= \(r := e \text{ when } a \text{ when } e \text{ when } a \text{ when } e \text{ when } a\)

\(|(t₁ = e₁; ...; tₙ = eₙ; \text{ in } a)|

e ::= r | c | Op(r₁, r₂) | e₁ \text{ when } e \text{ when } e \text{ when } e \text{ when } e \text{ when } e

| t | (t₁ = e₁; ...; tₙ = eₙ; \text{ in } e)

BS₁ Lifting rules

- Unique local names (t₁, t₂, ...) can be introduced anywhere for sharing
  
  \(e \implies (t = e \text{ in } t)\)

- Lifting rules for actions
  - \(r := (t = e \text{ in } e')\) \implies (t = e \text{ in } r := e')
  - a \text{ when } (t = e \text{ in } e') \implies (t = e \text{ in } (a \text{ when } e'))
  - (t = e \text{ in } (e' \text{ when } p)) \implies (t = e \text{ in } e') \text{ when } p
  - if \(t = e \text{ in } p\) then e₁ \implies (t = e \text{ in } (if \(p\) then e₁))
  - (t = e \text{ in } a₁; a₂) \implies (t = e \text{ in } (a₁; a₂))
  - (t₁ = e₁ in (t₂ = e₂ in a)) \implies (t₁ = e₁; t₂ = e₂ in a)

- Substitution \& when clauses
  - \(t = e \text{ when } p \text{ in } e'\) \implies (t = e; t₁ = p in [(t when t₁)/t]e')

- Lifting rules for expressions are similar

Lifting lets to the top level

- Registers \(r₁, ..., ;\)
  
  \(t₁ = e₁; ...; tₙ = eₙ;\)
  
  Rule R₁ a₁; ...; Rule Rₘ aₘ;

  \implies Registers \(r₁, ..., ;\)

  \(t₁ = e₁; ...; tₙ = eₙ;\)
  
  \(t₁ = e₁; ...; tₙ = eₙ;\)
  
  Rule R₁ a₁; ...; Rule Rₘ aₘ;

  Rule R₁ a₁

Some renaming of local variables may be required
**BS₁: Canonical form**

Program ::= 
  Registers \( r_1, \ldots, ; \) 
  \( t_1 = e_1; \ldots; t_n = e_n; \) 
  Rule \( R_1 \) ac₁; \ldots; Rule \( R_m \) acₘ ;

\( ac ::= (t_1 = e_1; \ldots; t_n = e_n; \text{ in } aw) \)
\( ac' ::= (t_1 = e_1; \ldots; t_n = e_n; \text{ in } a) \)
\( aw ::= a \mid a \text{ when } e \)
\( a ::= r := e \mid a \text{ when } e \mid \text{if } e \text{ then } a \mid a ; a \mid (t = e \text{ in } a) \mid m.g(e) \)

\( ec ::= (t_1 = e_1; \ldots; t_n = e_n; \text{ in } ew) \)
\( ec' ::= (t_1 = e_1; \ldots; t_n = e_n; \text{ in } e) \)
\( ew ::= e \mid e \text{ when } p \)
\( e ::= r \mid c \mid \text{Op}(r_a, r_b) \mid e \text{ when } e \mid t \mid (t = e \text{ in } e) \mid m.f(e) \)

---

**BS₂ = BS₁ + Modules**

A program is a collection of instantiated modules \( m, m_1, \ldots; \)
A module is a collection of rules and interface methods
\( f, f_1, f_2, \ldots \) are names for “read methods”
\( g, g_1, g_2, \ldots \) are names for “action methods”

\( a ::= r := e \mid a \text{ when } e \mid \text{if } e \text{ then } a \mid a ; a \mid (t = e \text{ in } a) \mid m.g(e) \)

\( e ::= r \mid c \mid \text{Op}(r_a, r_b) \mid e \text{ when } e \mid t \mid (t = e \text{ in } e) \mid m.f(e) \)

---

**BS₂ Program**

A program is a collection of instantiated modules:

Program ::= \( m_1 ; m_2 ; m_2 ; \ldots \)

Module ::= 
  Module name
  [Register \( r \)]
  [Rule \( R a \);]
  Interface

Interface ::= \[\text{action method}; \] [\text{read method}]
action method ::= \text{method } g (x) = a
read method ::= \text{method } f (x) = e

---

**Implicit conditions**

- Every method has two parts: guard and body. These will be designated by subscripts \( G \) and \( B \), respectively.

- Making guards explicit in every method call:
  \( m.h(e) \implies (p = m.h_G \text{ in } m.h_B(e) \text{ when } p) \)
**BS₂ : Additional Lifting rules**

- Only read methods can be named in a let block
  \[ m.f(e) \Rightarrow (t=m.f(e) \text{ in } t) \]

- \[ m.g (t = e \text{ in } e') \Rightarrow (t = e \text{ in } m.g(e')) \]

- \[ m.g (e \text{ when } p) \Rightarrow m.g(e) \text{ when } p \]

  - similar rules for read methods

---

**Some subtle issues**

- Does it matter if we first make the guards explicit and then lift or can we lift at any stage?

---

**BS₂ : Canonicalization procedure**

1. Make guards of method calls explicit
2. Lift *lets* to the top
3. Get rid of the *whens* from the *lets*
4. Lift *whens* to the top

---

**Getting ready for circuit generation**

- We need to collect multiple conditional assignments to one register in one expression, i.e.,
  
  \[ \ldots \text{if } p1 \text{ then } r:= e1; \]
  \[ \text{if } p2 \text{ then } r:= e2; \]
  \[ (r:= p3? e4: e5); \ldots \]
Notation for conditional assignment

- \( r:= e_1p_1 + \ldots + e_n p_n \)
  where
  - \( e_1, e_2, \ldots \) are expressions
  - \( p_1, p_2, \ldots \) are booleans

- \( e.p \) evaluates to \( e \) if \( p \) is true otherwise to False (zero’s)

- if \( p_i's \) are not pairwise mutually exclusive then the program is illegal

- \( e_1p_1 + \ldots + e_n p_n \) evaluates to some \( e_i \) or if all \( p_i \)'s are false then the value of \( r \) does not change

Collecting conditional assignments to a register

- Apply the following rules after the guards have been made explicit and the program has been canonicalized,

  1. if \( p \) then \( a \) \( \implies \) \( a \cdot p \)
  2. \( 1 \) \( (r := e) \cdot p \) \( \implies \) \( r := e \cdot p \)
  3. \( 2.1 \) \( m.g(e) \cdot p \) \( \implies \) \( m.g(e \cdot p) \)
  4. \( 3 \) \( (a1; a2) \cdot p \) \( \implies \) \( a1 \cdot p ; a2 \cdot p \)
  5. \( 4.1 \) \( r := e1 ; r := e2 \) \( \implies \) \( r := e1 + e2 \)
  6. \( 4.2 \) \( m.g(e1); m.g(e2) \) \( \implies \) \( m.g(e1 + e2) \)

**Theorem:** After applying the above rules to a Program in canonical form any action in it will be reduced the following form:
\[
\begin{align*}
  &r1 := e1; r2 := e2; \ldots \\
  &m.g(e); m1.g1(e1); \ldots \text{ where } e's \text{ may contain } "." \text{ and } "+"
\end{align*}
\]