Bluespec-7: Semantics of Bluespec

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

- Guarded actions versus Conditional actions
- Naming expressions: the "let" statement
- Modules and methods with implicit conditions
- Rule composition
BS₀ : A simple language of Guarded Atomic Actions

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

\[
a ::= r := e \mid a \text{ when } e \mid \text{if } e \text{ then } a \mid a ; a
\]

\[
e ::= r \mid c \mid \text{Op}(e, e) \mid e ? e : e \mid e \text{ when } e \mid ...
\]

“;” 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(rₐ , rₐ ) | 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 when p1); (a2 when p2)
  ==> (a1; a2) when p1 & p2

A condition of a Conditional action only affects the actions within the scope of the conditional action
(if p1 then a1); (if p2 then a2)
  p1 has no effect on a2 ...
Canonicalizing $BS_0$
ignoring guards on expressions

Rules for canonicalization:

1. $(a_1 \text{ when } p); a_2$
   $\implies (a_1; a_2) \text{ when } p$

2. $(a \text{ when } p_1) \text{ when } p_2$
   $\implies a \text{ when } (p_1 \& p_2)$

3. if $p$ then $(a \text{ when } q)$
   $\implies (\text{if } p \text{ then } a) \text{ when } (p\&q \mid !p)$
Conditionals & Cases

if p then a1 else a2
   = if p then a1; if !p 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;

move all the guards to 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 when e
a ::= r := e | if e then a | a ; a
e ::= r | c | Op(e , e ) | e ? e : e | ...
Rules for Canonicalizing BS₀

1. (a₁ when p); a₂ ==> (a₁; a₂) when p
2.1 (a when p) when q ==> a when (p & q)
2.2 (e when p) when q ==> e when (p & q)
3.1 if p then (a when q) ==> (if p then a) when (p & q | !p)
3.2 p ? (e₁ when q) : e₂ ==> (p ? e₁ : e₂) when (p & q | !p)
3.3 p ? e₁ : (e₂ when q) ==> (p ? e₁ : e₂) when (p | !p & q)
4  r := (e when q) ==> (r := e) when q
5.1 Op(e₁ when q, e₂) ==> Op(e₁,e₂) when q
5.2 Op(e₁, e₂ when q) ==> Op(e₁,e₂) when q
5.3 if (p when q) then a ==> (if p then a) when q
5.4 (p when q)? e₁ : e₂ ==> (p ? e₁ : e₂) when 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 r₁, ..., ;
    t₁ = e₁; ...; tₙ = eₙ;
    Rule R₁ a₁; ... ; Rule Rₘ aₘ ;

a ::= r := e | a when e | if e then a | a ; a
     | (t₁ = e₁; ...; tₙ = eₙ; in a)

e ::= r | c | Op(rₐ, r₋) | e ? e : e | e when e | ...
     | t | (t₁ = e₁; ...; tₙ = eₙ; 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
  - \( r := (t = e \text{ in } e') \implies (t = e \text{ in } r := e') \)
  - \( a \text{ when } (t = e \text{ in } e') \implies (t = e \text{ in } (a \text{ when } e')) \)
  - \( (t = e \text{ in } (e' \text{ when } p)) \implies (t = e \text{ in } e') \text{ when } p \)
  - \( \text{if } (t = e \text{ in } p) \text{ then } e_1 \implies (t = e \text{ in } (\text{if } p \text{ then } e_1)) \)
  - \( (t = e \text{ in } a_1); a_2 \implies (t = e \text{ in } (a_1; a_2)) \)
  - \( (t_1 = e_1 \text{ in } (t_2 = e_2 \text{ in } a)) \implies (t_1 = e_1; t_2 = e_2 \text{ in } a) \)

  Some renaming of local variables may be required

- Substitution & when clauses
  - \( (t = e \text{ when } p \text{ in } e') \implies (t = e; t_1 = p \text{ in } [(t \text{ when } t_1)/t]e') \)

- Lifting rules for expressions are similar
Lifting lets to the top level

Registers $r_1, \ldots, \;$
$t_1 = e_1; \ldots; t_n = e_n;$
Rule $R_1 a_1; \ldots ;$ Rule $R_m a_m ;$
Rule $R_i (t_{i1} = e_{i1}; \ldots; t_{in} = e_{in}; \text{ in } a_i)$

\[ \Rightarrow \]

Registers $r_1, \ldots, \;$
$t_1 = e_1; \ldots; t_n = e_n;$
$t_{i1} = e_{i1}; \ldots; t_{in} = e_{in};$
Rule $R_1 a_1; \ldots ;$ Rule $R_m a_m ;$
Rule $R_i a_i$

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

Program ::= 

\[
\text{Registers } r₁, \ldots, \\
\text{\hspace{1cm}} t₁ = e₁; \ldots; tₙ = eₙ; \\
\text{\hspace{1cm}} \text{Rule } R₁ \text{ ac₁; } \ldots \text{; Rule } Rₘ \text{ acₘ;}
\]

ac ::= (t₁ = e₁; \ldots; tₙ = eₙ; in aw)
ac' ::= (t₁ = e₁; \ldots; tₙ = eₙ; in a)
aw ::= a | a when e
a ::= r := e | if e then ac' | a ; a

ec ::= (t₁ = e₁; \ldots; tₙ = eₙ; in ew)
ec' ::= (t₁ = e₁; \ldots; tₙ = eₙ; in e)
ew ::= e | e when p
e ::= r | c | Op(rₐ , rₚ ) | e ? ec' : ec'... | t
**BS\(_2\) = BS\(_1\) + 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”

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

\[
e ::= \quad r \mid c \mid \text{Op}(r_a, r_b) \mid e ? e : e \mid \ldots \mid e \text{ when } e \\
   \quad \mid t \mid (t = e \text{ in } e) \mid m.f(e)
\]
BS$_2$ Program

A program is a collection of instantiated modules:

\[
\text{Program ::= } m_1 ; m_2 ; m_2 ; \ldots
\]

Module ::= 
- Module name
- [Register r]
- [Rule R a];
- Interface

Interface ::= [action method]; [read method]

action method ::= method g (x) = a
read method ::= 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:

\[
m.h(e) \implies (p = m.h_G \text{ in } m.h_B(e) \text{ when } p)
\]
BS$_2$ : Additional Lifting rules

- Only read methods can be named in a let block
  
m.f(e) ==> (t=m.f(e) in t)

- Similar rules for read methods
  
m.g (t = e in e’) ==> (t = e in m.g(e’))

- m.g (e when p) ==> m.g(e) when p
Some subtle issues

Does it matter if we first make the guards explicit and then lift or can we lift at any stage?
BS$_2$ : Canonicalization procedure

1. Make guards of method calls explicit

2. Lift $let$s to the top

3. Get rid of the $when$s from the $let$s

4. Lift $when$s to the top
Getting ready for circuit generation

We need to collect multiple conditional assignments to one register in one expression, i.e.,

... if p1 then r := e1;
if p2 then r := e2;
(r := p3? e4: e5); ...
Notation for conditional assignment

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

\( e_1.p_1 + ... + e_n.p_n \) 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  ==>  a . p
2.1 (r := e) . p  ==>  r := e . p
2.2 m.g(e) . p  ==>  m.g (e . p)
3. (a1; a2) . p  ==>  a1 . p ; a2 . p
4.1 r := e1 ; r := e2  ==>  r := e1 + e2
4.2 m.g(e1); m.g(e2)  ==>  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:
r1 := e1; r2 := e2; ....
m.g(e); m1.g1(e1); ... where e’s may contain “.” and “+”