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

BS_0 : A simple language of Guarded Atomic Actions

a is an Action;

e is an Expression;

r is a register (state variable)

action a is guarded by predicate e

conditional action

compound atomic action

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

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

conditional expression

guarded expression

“;” is commutative and associative, i.e.

a1;a2 = a2;a1

a1;(a2;a3) = (a1;a2);a3

BS_0 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 r_1, …, ;

Rule R_1 a_1; … ; Rule R_m a_m ;

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

e ::= r | c | Op(r_a , r_b) | 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.  (a1 when p); a2
                  ==>  (a1; a2) when p
2.  (a when p1) when p2
                  ==> a when (p1 & p2)
3.  if p then (a when q)
                  ==> (if p then a) when (p&q | !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 BS_0

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.

$BS_1 = BS_0$ + Let blocks Introducing local names

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

Program ::=

Registers r_1, ..., ;

$t_1 = e_1$; ...; $t_n = e_n$;

Rule R_1 a_1; ... ; Rule R_m a_m ;

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

| ($t_1 = e_1$; ...; $t_n = e_n$; in a)

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

| t | ($t_1 = e_1$; ...; $t_n = e_n$; in e)

BS_1 Lifting rules

- Unique local names (t1, t2, ...) can be introduced anywhere for sharing

  e ==> (t = e in t)

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

  Some renaming of local variables may be required
- Substitution & when clauses
  - (t=e when p in e') ==> (t = e; t1 = p in [(t when t1)/t]e')
- Lifting rules for expressions are similar

Lifting lets to the top level

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

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

Some renaming of local variables may be required

BS_1: Canonical form

Program ::=

Registers r_1, ..., ;

t_1 = e_1; ...; t_n = e_n;

Rule R_1 ac_1; ... ; Rule R_m ac_m ;

ac ::= (t_1 = e_1; ...; t_n = e_n; in aw)

ac'::= (t_1 = e_1; ...; t_n = e_n; in a)

aw ::= a | a when e

a ::= r := e | if e then ac' | a ; a

ec ::= (t_1 = e_1; ...; t_n = e_n; in ew)

ec'::= (t_1 = e_1; ...; t_n = e_n; in e)

ew::= e | e when p

e ::= r | c | Op(r_a , r_b) | e ? ec' : ec'... | t

$BS_2 = BS_1$ + Modules

A program is a collection of (instantiated) modules m, m1,...;

A module is a collection of rules and interface methods

f, f1, f2, ... are names for "read methods"

g, g1, g2, ... are names for "action methods"

a ::= r := e | a when e | if e then a | a ; a | (t = e in a)
| m.g(e)

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

BS_2 Program

A program is a collection of instantiated modules:

Program ::= m_1 ; m_2 ; m_2 ; ...

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) ==>

  $(p = m.h_G \text{ in } m.h_B(e) \text{ when } p)$

BS_2 : Additional Lifting rules

- Only read methods can be named in a let block
  m.f(e) ==> (t=m.f(e) in t)
- m.g (t = e in e')   ==> (t = e in m.g(e'))
- m.g (e when p)   ==> m.g(e) 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?

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:= e1.p1 + … + en.pn

  where

  - e1, e2, … are expressions

  - p1, p2, … 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
- e1.p1 + … + en.pn 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 "+"