Constructive Computer Architecture:

Hardware Compilation of Bluespec

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

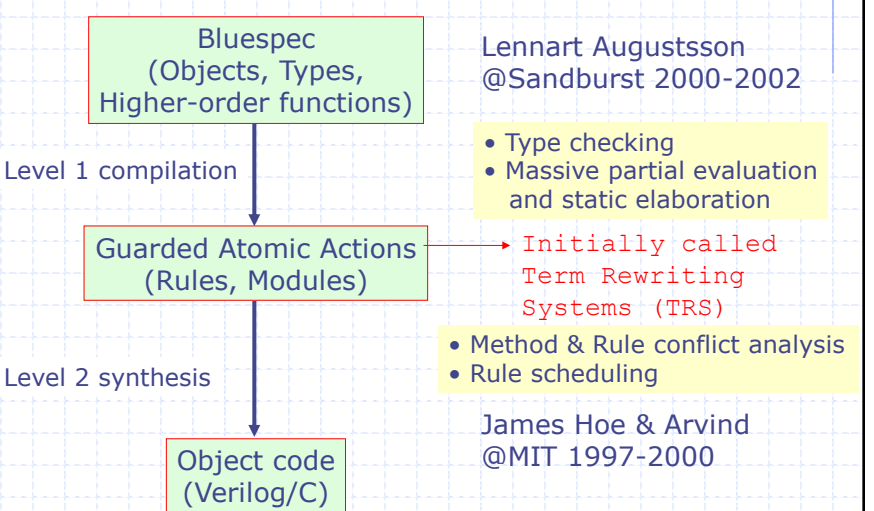
Contributors to the course material

- Arvind, Rishiyur S. Nikhil, Joel Emer, Muralidaran Vijayaraghavan
- Staff and students in 6.375 (Spring 2013), 6.S195 (Fall 2012), 6.S078 (Spring 2012)
  - Asif Khan, Richard Ruhler, Sang Woo Jun, Abhinav Agarwal, Myron King, Kermin Fleming, Ming Liu, Li-Shiuan Peh
- External
  - Prof Amey Karkare & students at IIT Kanpur
  - Prof Jihong Kim & students at Seoul Nation University
  - Prof Derek Chiou, University of Texas at Austin
  - Prof Yoav Etsion & students at Technion

Contents

- KBS syntax and well-formed programs
- Hardware representation
  - modules and method calls
- Generating Hardware
  - Linking
- The Scheduler

Bluespec: Two-Level Compilation



Lennart Augustsson
@Sandburst 2000-2002

- Type checking
- Massive partial evaluation and static elaboration

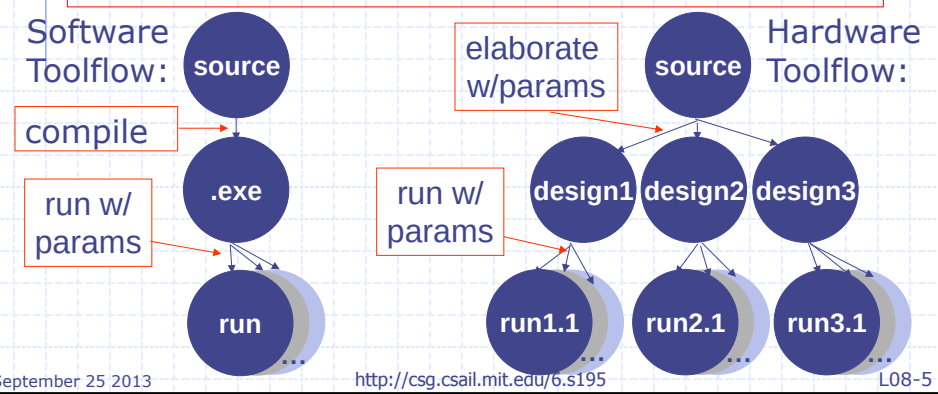
`Initially called Term Rewriting Systems (TRS)`

- Method & 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



Phase II compilation: From KBS1 to Circuits

We will assume that the type checking and static elaboration have been performed and all modules have been instantiated by the Phase I compiler

KBS0: A simple language for describing Sequential ckts

<a> is an action and <e> is an expression

```
<Program> ::= [rule <name> <a>]
              [x <- mkReg]          register instantiations
<a> ::= x.w(<e>)                    register assignment
      | <a> | <a>                   parallel actions
      | if (<e>) <a>                conditional action
      | let t = <e> in <a>          binding
<e> ::= c                           constants
      | t                           value of a binding
      | x.r                         register read
      | op(<e>,<e>)                 operators like And, Or, Not, +, ...
      | let t = <e> in <e>          binding
```

The names in the bindings (t ...) can be defined only once

KBS1: KBS0+Modules

```
<Program> ::= [<Module>]
<Module> := Module <name>           names; M, mkReg, mkFoo,...
        [x <- mkReg]                register instantiations; x,y..
        [m <- <mkM>]                module M instantiations; m,...
        [rule <name> <a>]           rules to describe behavior
        [valueMethod <name> (<id>*) = <e>]   } interface
        [actionMethod <name> (<id>*) = <a>]  } methods

<a> ::= KBS0 action
      | m.g(<e>)                    call to action method m.g
<e> ::= KBS0 expression
      | m.f(<e>)                    call to value method m.f
* Means zero or one occurrence
```

KBS1EHR: KBS1+EHRs

```
<Program> ::= [<Module>]
<Module> := Module <name>          names; M, mkReg, mkFoo,…
        [x <- mkReg]               register instantiations; x,y..
        [x <- mkEHR]               EHR instantiations; x,y..
        [m <- <mkM>]               module M instantiations; m,…
        [rule <name> <a>]          rules to describe behavior
        [valueMethod <name> (<id>*) = <e>]   } interface
        [actionMethod <name> (<id>*) = <a>]  } methods
<a> ::= KBS1 action
     | x.w0(<e>)   | x.w1 (<e>) …  write actions into EHRs
<e> ::= KBS1 expression
     | x.r0        | x.r1                    reading EHRs
* Means zero or one occurrence
```


Well-formed rules

- A program with double-write error:
  - x.w(5) | x.w(7)
- Either such errors have to be detected during the execution or such programs have to be rejected at compile time
- To avoid run-time errors the compiler accepts only those programs which follow two types of restrictions:
  1. A method (except for a zero-parameter value-method) can be called at most once by a rule
  2. Methods called by a rule must form a "partial order"

Single-call restriction and zero-parameter value methods

- Example
  - y.w(x.r + x.r)
  - This is a violation because x.r is called twice; however it can be transformed by the compiler into the following code
  - let t = x.r in y.w(t+t)
- We do not consider multiple calls to such methods as a violation

Single-call restriction and conditional method calls

- Example
  - if (p) x.w(y.r + 1) ; if (q) x.w(z.r) ;
  - This is a violation because x.w is called twice; however if the compiler can prove that p and q are mutually exclusive (e.g. q => !p) then only one of the calls will occur and there will be no violation
- Compiler associates a predicate with each method call and accepts multiple calls to a method if it can prove that the predicates are mutually exclusive

Syntax mandated Orderings

- if (e) a
  - mcalls(e) must precede the method calls in mcalls(a)
- m.g(e)
  - mcalls(e) must precede the method call m.g
- let t = e in a
  - mcalls(e) must precede the method calls in mcalls(a) if t is used in a

The compiler derives all the syntactic orderings and rejects a program if these orderings are violated by orderings imposed by the module definition

Examples of violations

- if (x.r1) x.w0(e)
  - Syntax mandated: x.r1 < x.w0
  - EHR mandated: x.w0 < x.r1

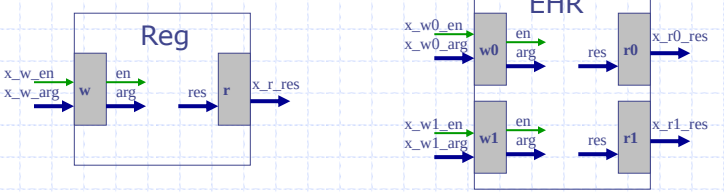
  contradiction!
- x.w0(y.r1) | y.w0(x.r1)
  - Syntax mandated: y.r1 < x.w0, x.r1 < y.w0
  - EHR mandated: x.w0 < x.r1, y.w0 < y.r1

  contradiction!

Hardware representation

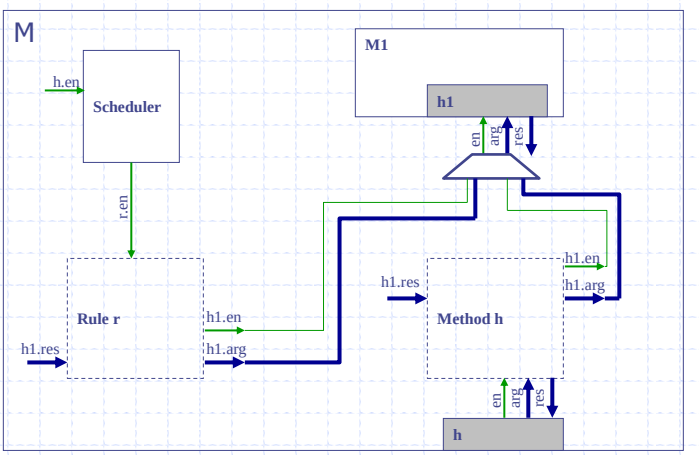
registers and EHRs

- A set of bindings can be thought of as a set of boxes which are connected by wires. A box represents an expression or the port of a module and wires are the variable names



Hardware representation

module

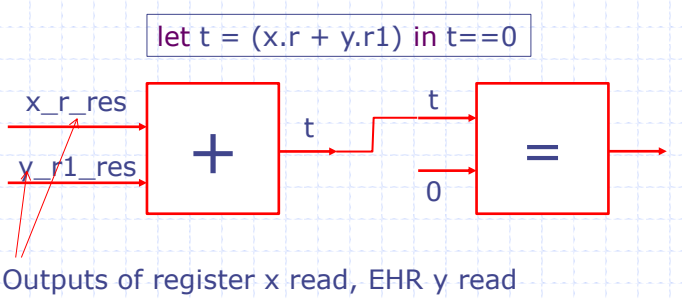


Method calls

- A method call h(e) involves three sets of wires
  - h_arg representing input argument (output of e)
  - h_en, when true means that the method is to be used
  - h_res representing the output of the method
- The compiler collects all the input arguments for each method call as a sum of predicated expressions:
  - h_arg = p1.e1+p2.e2+...
  - h_en = p1 || p2 ||...
- For each method h that can be called, the bindings are initialized with (h_arg, F.Bot) and (h_en, F)

Hardware Compilation:
Expressions

Expressions are structural and directly represent combinational circuits; some of their inputs are connected to x_r_res, m_g_res, …

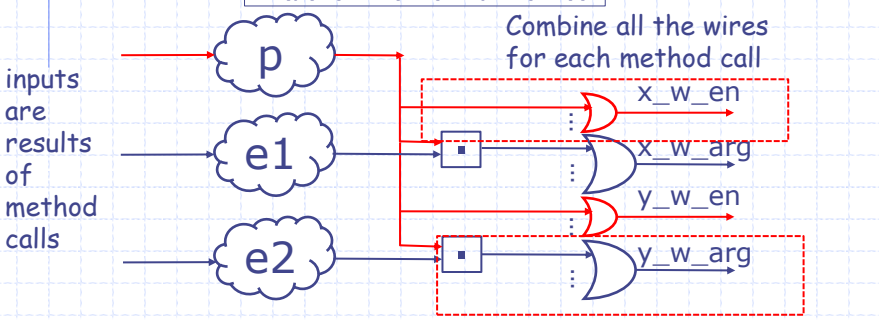


Hardware Compilation:

Actions

Actions use the circuits generated by compilation of expressions; their output wires are connected to x_w_en, x_w_arg, m_h_en, m_h_arg, …

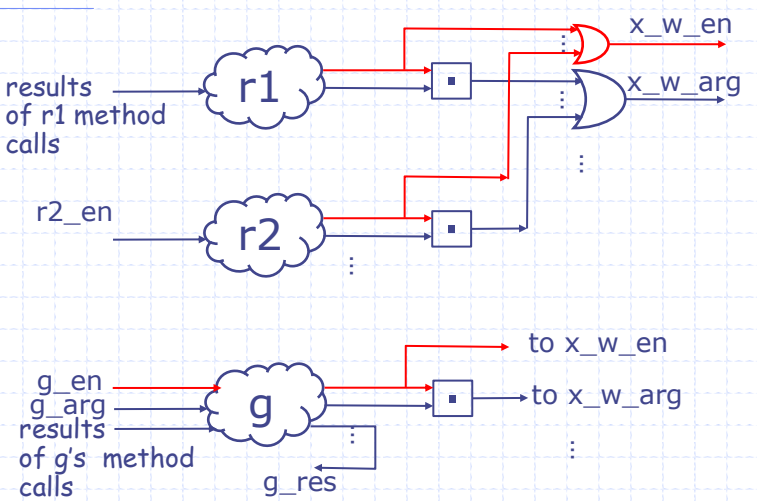
```
if (p) (x.w(e1) | y.w(e2))
```



m_g_arg = p1.e1 + p2.e2 + ...
m_g_en = p1 || p2 || ....

Hardware Compilation:

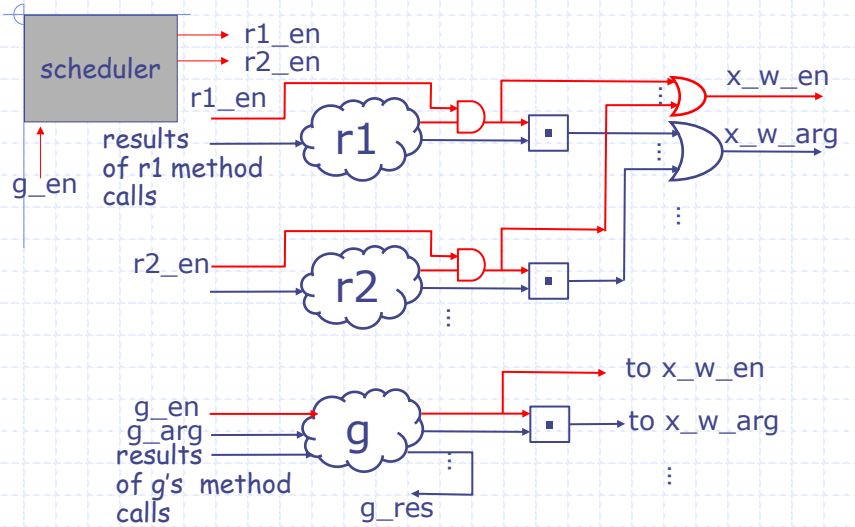
Rules and Methods



Scheduler

- A module may contain many rules but the Bluespec semantics dictates that all legal behaviors be derivable by executing only one rule at a time
- Based on the conflict information about each of the called methods, a scheduler is constructed by the compiler to decide which rule(s) can be execute concurrently and the schedule indicates it choice by setting r_en, the enable signal, of the chosen rule.
- The only dynamic input the scheduler needs is g_en for all of its defined methods g

Hardware Compilation:
Scheduler



The scheduler circuit

- Each specific scheduling strategy will result in a different scheduler circuit
- For functional correctness, it is important that the scheduling circuit enforces the following invariant
  - SC Invariant: Suppose r1, … rn are the rules of M being chosen to be scheduled in the current state, h1, … hk are the methods of M being called externally, then M preserves SC Invariant iff

    $\forall i.\forall (j > i).\forall x \in mcalls(ri), \forall y \in mcalls(rj).$

    $(\{<\} \subseteq Conflict(x, y))$
- Theorem: If every module obeys the SC Invariant, then the system will obey one-rule-at-a-time semantics

Detailed Hardware Compilation Procedure (optional)

Compiling hardware

- Expressions in KBS are structural and directly represent combinational circuits; some of their inputs are connected to x_r_res, m_g_res, …
- Actions use the circuits generated by compilation of expressions; their output wires are connected to x_w_en, x_w_arg, m_h_en, m_h_arg, …
- The compiler represents all connections as a set of bindings (next slide)
- The compiler collects the bindings by threading the bindings through all rules and methods

Syntax of bindings

```
B ::= [<b>]
b ::= <t> = <ei>
    | <h>_arg = <pe>
    | <h>_en = <ei>
ei ::= <c> | <t>
    | <op>(<ei>, <ei>)
    | <h>_res
pe ::= Bot
    | <be>.<ei> // be is a boolean ei
    | <be>.<pe>
    | <pe> + <pe>
```

Compiling Expressions

```
CE :: (Bindings, Predicate, Exp) -> (Bindings, Exp)
CE (bs,p,[[c]]) = (bs,c) ;
CE (bs,p,[[t]]) = (bs,t) ;
CE (bs,p,[[let t=e1 in e2]]) =
  { (bs1,e10) = CE(bs,p,[[e1]]);
    (bs2,e20) = CE((bs1[t]:=p.e10),p,[[e2]]) return (bs2,e20)};
CE (bs,p,[[op(e1,e2)]]) =
  { (bs1,e10) = CE(bs,p,[[e1]]);
    (bs2,e20) = CE(bs1,p,[[e2]])  return (bs2,op(e10,e20))};
CE (bs,p,[[h(e)]]) =
  { (bs1, e0) = CE(bs,p,[[e]]);
     bs2 = (bs1[h_arg]:=bs1[h_arg]+p.e0);
     bs3 = (bs2[h_en]:=bs2[h_en]+p.T)  return (bs3,h_res)};
CE (bs,p,[[h()]]) = ((bs[h_en]:=bs[h_en]||p), h_res);
```


Compiling Actions

```
CA :: (Bindings, Predicate, Action) -> Bindings
CA (bs,p,[[let t=e in a]]) =
  { (bs1,e0) = CE(bs,p,[[e]]);
   return CA((bs1[t]:=p.e0),p,[[a]])};
CA (bs,p,[[h(e)]]) =
  { (bs1,e0) = CE(bs,p,[[e]]);
     bs2 = (bs1[h_arg]:=bs1[h_arg]+p.e0);
  return (bs2[h_en]:=bs2[h_en]+p.T)};
CA (bs,p,[if (e) a]]) =
  { (bs1,e0) = CE(bs,p,[[e]])
  return CA((bs1[t]:=p.e0),t,[[a]])}; where t is fresh
CA (bs,p,[[a1 | a2]]) =
  { bs1 = CA(bs,p,[[a1]])   return CA(bs1,p,[[a2]])}
```

Compiling Rules and Methods

```
CR : (Bindings, Rule) -> Bindings
CR (bs,[[rule r a]]) = CA(bs,r_en,[a]])

CVM : (Bindings, Value-method) -> Bindings
CVM (bs, [[valueMethod  h(x)=e]]) =
  {  bs0 = (bs[x]:= h_arg);
     (bs1,e0) =  CE(bs0,h_en,([[e]]);
      bs2 = (bs1[h_res]:= e0);   return bs2};

CAM : (Bindings, Action-method) -> Bindings
CAM (bs,[[actionMethod  h(x)=a]]) =
    CA((bs[x]:=h_arg),h_e,[[a]]);
```


Compiling Modules and linking method calls

- The compiler produces a set of bindings for each module by starting with an empty set of bindings and then threading the bindings produced by each rule and method defined inside the module
- Modules are compiled inside out, that is, a module is compiled only after all the modules whose methods it calls have been compiled
- For each method call h(e) the compiler links (connects) the bindings of the caller module with the bindings of the modules whose methods are called by connecting the wires representing the formal parameter h_x and actual parameter h_arg of method h

One-rule-at-a-time semantics

- Rule execution ($\rightarrow$)

$$\frac{\text{Rule r a} \in \text{P} \qquad \langle S,\{\}\rangle \text{ |- a} \Rightarrow \text{ U}}{\text{P |- S} \rightarrow \text{update(S,U)}}$$

Where update(S,U)[x] = if (x,v) $\in$ U the v else S[x]

- Legal states: S is a legal state if and only if given an initial state S_0 , there exists a sequence of rules r_{j1},…., r_{jn} such that S= $r_{jn}(\ldots(r_{j1}(S_0))\ldots)$

$$\frac{\text{P |- } S_0 \rightarrow^* S}{S \in \text{LegalState(P,}S_0\text{)}}$$

where $\rightarrow^*$ is the transitive reflexive closure of $\rightarrow$